Version: 4.16

TouchGFX Documentation

Welcome to the official documentation of TouchGFX!

If you are new to this documentation and/or TouchGFX, we recommend that you read on to get an overview of what this documentation has to offer. The table of contents in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top right corner.

About this documentation

The main documentation for the site is organized into the following sections:

- **Introduction** - read surface-level information on TouchGFX and installation guide.
- **Basic Concepts** - introduction to key graphics concepts.
- **Development** - how to develop a TouchGFX application including, structure, workflow and tools.
- **Tutorials** - a collection of TouchGFX tutorials.

Target User

The TouchGFX documentation is targeting software developers with a basic skill-set within C++ and embedded GUI development on STM32. Newcomers to Embedded GUI Development are supported in the section Basic Concepts, while step by step guides and tutorials support everyone towards a smooth learning in TouchGFX development as well.

We would really like to improve this documentation in any way possible. If you don't understand something, or cannot find what you are looking for in the docs, help us make the documentation better by letting us know in the forum!

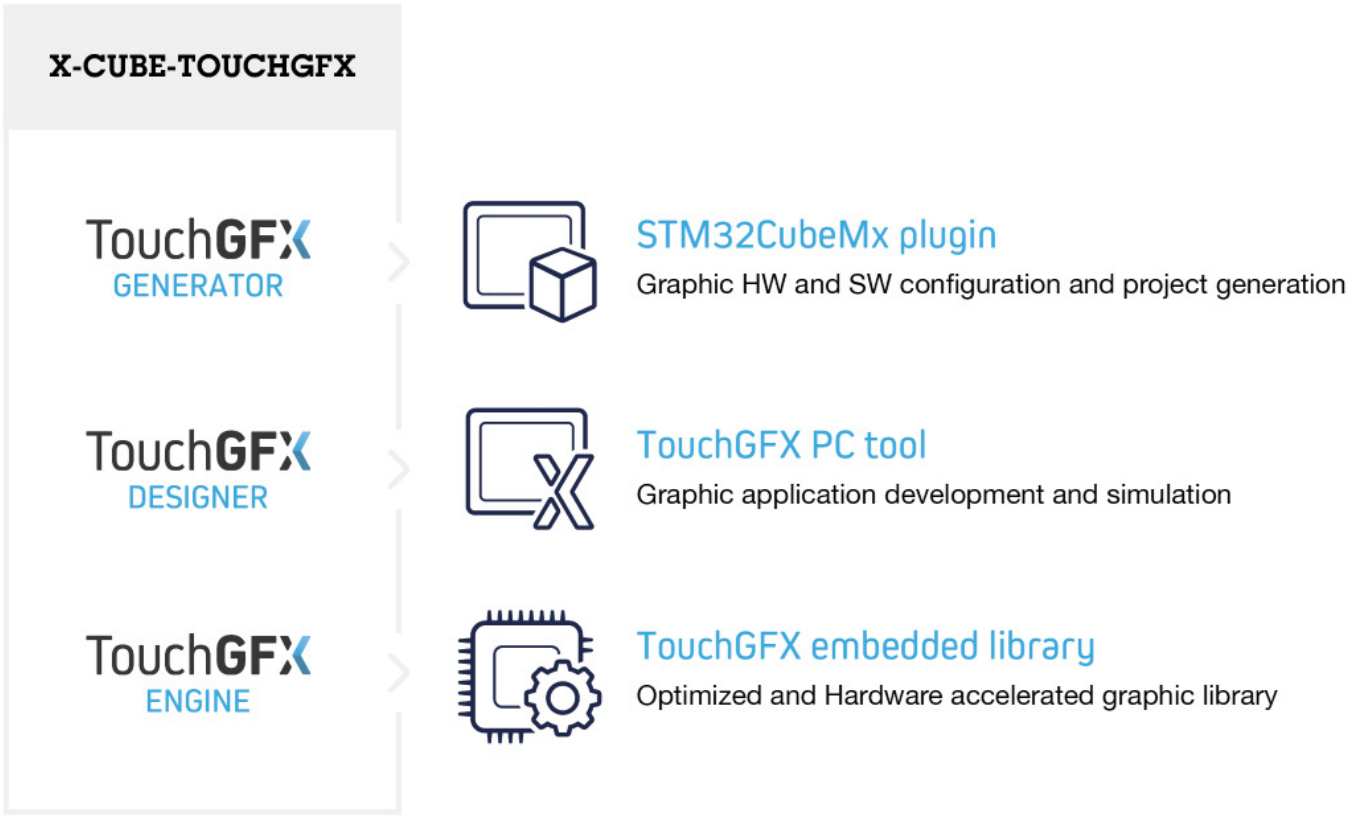
Version: 4.16

What is TouchGFX?

TouchGFX is delivered as one X-Cube package the *X-Cube-TouchGFX*.

With this you have all you need to do a full implementation of your GUI application for STM32 based hardware. TouchGFX consists of three main parts - two tools and one framework.

- **TouchGFX Designer:** An easy-to-use GUI builder in TouchGFX that lets you create the visual appearance of your TouchGFX application.
- **TouchGFX Generator:** A CubeMX plugin where the user can configure and generate a custom TouchGFX Abstraction Layer (AL) for their STM32-based hardware.
- **TouchGFX Engine:** The TouchGFX C++ framework that drives the UI application. Handles screen updates, user events and timing. The advanced TouchGFX technology is optimized for STM32 microcontrollers, giving you maximum performance with minimum CPU load and memory usage.



Version: 4.16

Installation

TouchGFX is distributed as an X-CUBE-TOUCHGFX zip file which has the following components inside:

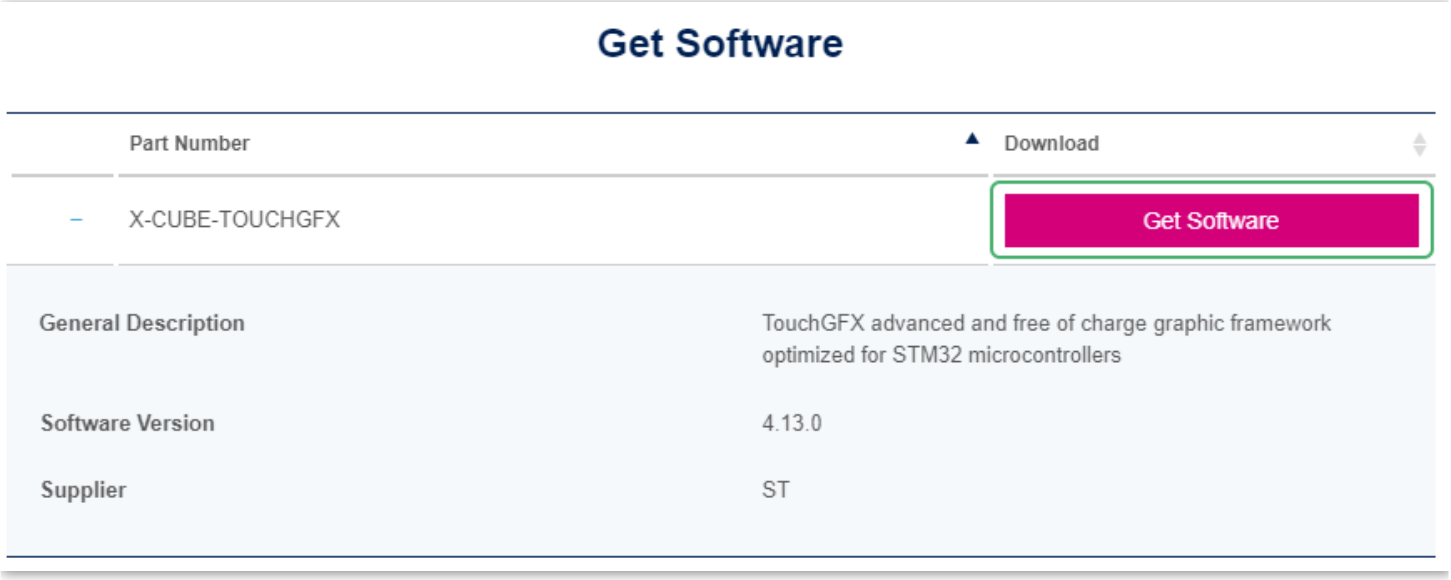
- **TouchGFX Designer** - Build a UI through a Windows-based GUI Builder
- **TouchGFX Generator** - Create a custom TouchGFX HAL through CubeMX
- **TouchGFX Engine** - The TouchGFX C++ framework that drives the UI application

Prototyping on STM32 Evaluation kits

If your intention is to simply try TouchGFX Designer and perhaps do some prototyping on STM32 Evaluation kits, refer to the section Installing TouchGFX Designer.

Installing TouchGFX Designer

Download X-CUBE-TOUCHGFX from the ST.com official website to anywhere on your hard drive and extract it.

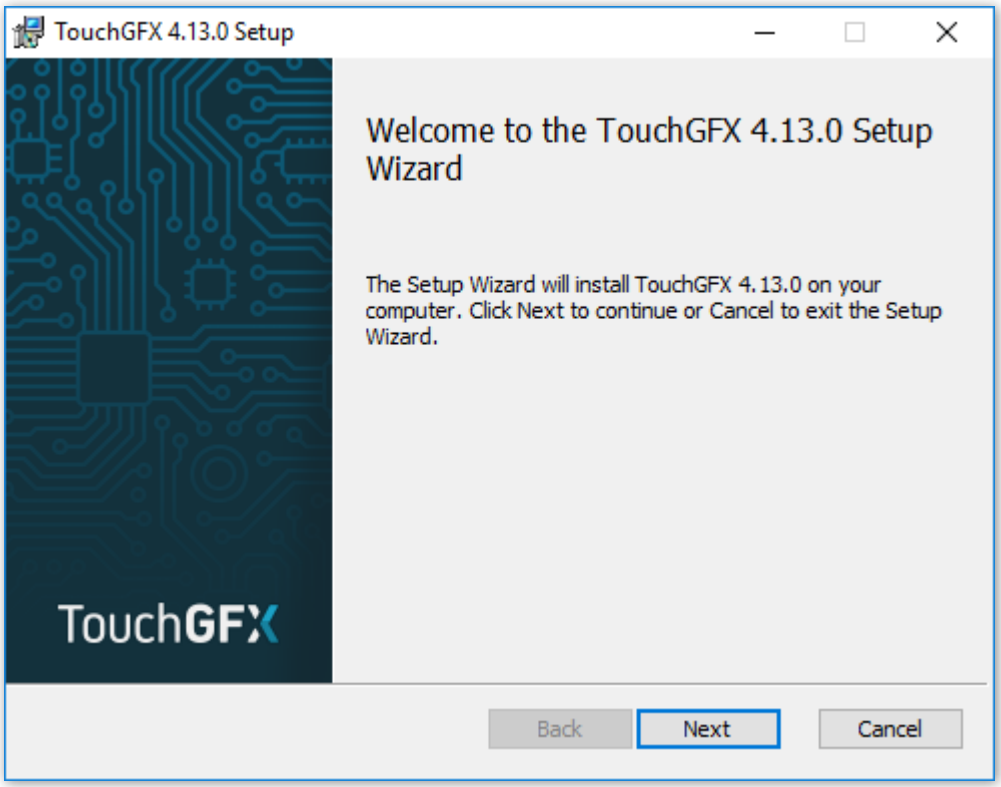


Downloading X-CUBE-TOUCHGFX from st.com

Inside the extracted folder, you will find the TouchGFX .msi installer in the following path:

```
Utilities\PC_Software\TouchGFXDesigner
```

Double-clicking the .msi file will bring up the installer. Follow the instructions to complete the installation process.



Installing TouchGFX Designer

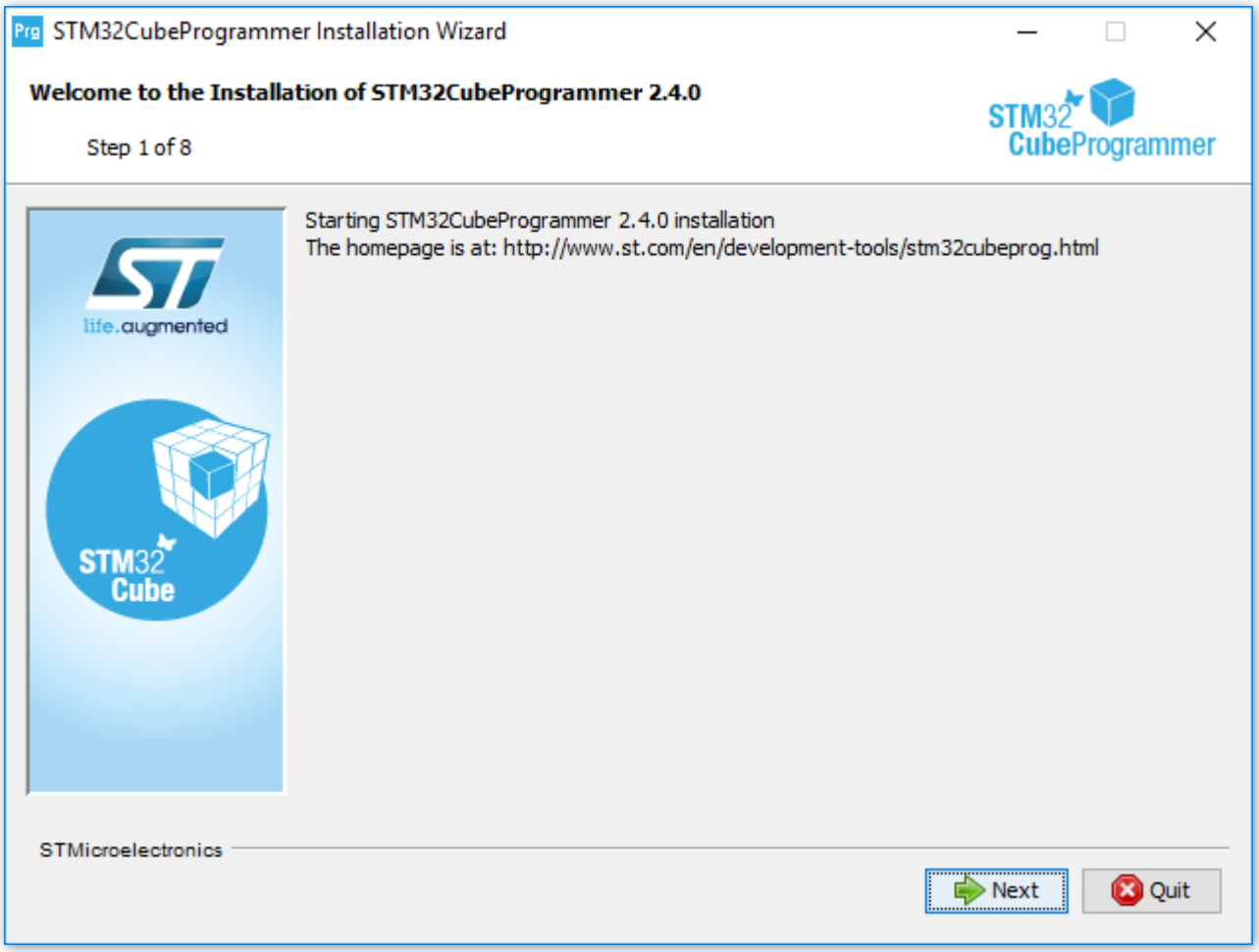
TIP

You need the following tools to be able to flash your board directly from within TouchGFX Designer:

- **STM32CubeProgrammer**
- **STM32 ST-LINK Utility**

Installing STM32CubeProgrammer

After downloading STM32CubeProgrammer from STM32CubeProgrammer download location, uncompress the downloaded `.zip` file and launch the `.exe` installer file. Then Follow the instructions to complete the installation process.

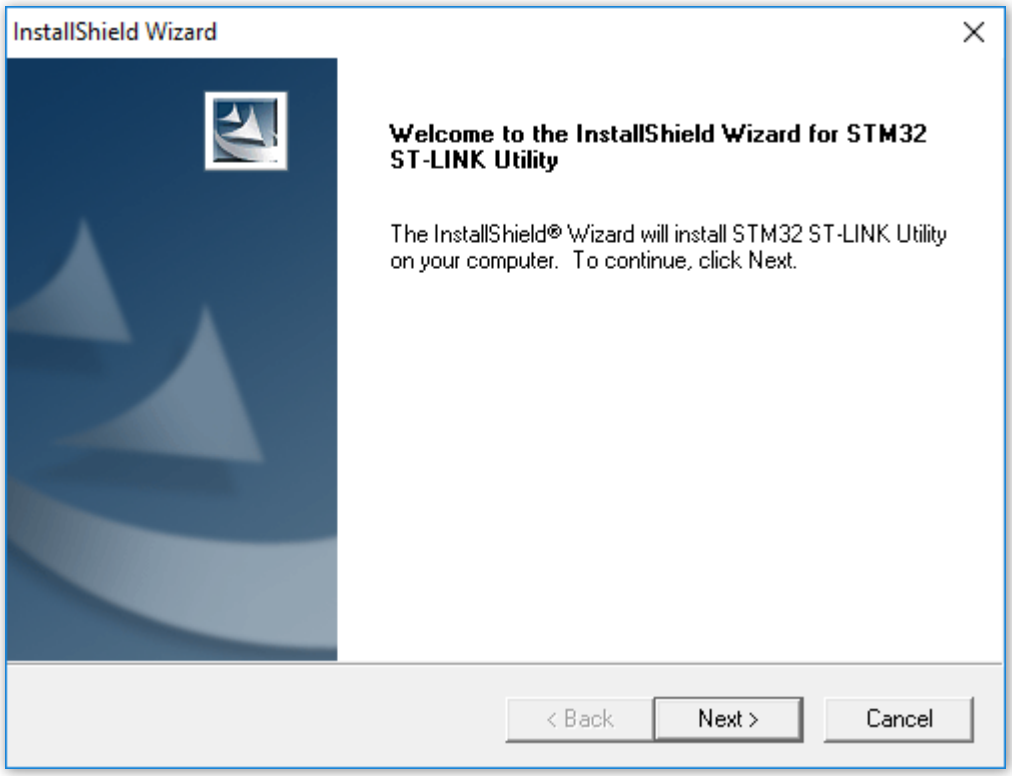


Installing TouchGFX Designer

> **CAUTION**
>
> **In order for the TouchGFX Designer and Makefiles to be able to use STM32CubeProgrammer for flashing target hardware it must be installed at the default install location:** `C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer`

Installing STM32 ST-LINK Utility

After downloading STM32 ST-LINK Utility from STM32 ST-LINK utility location, simply open the downloaded `.exe` file and follow the instructions to complete the installation process.



Installing STM32 ST-LINK Utility

⚠ CAUTION

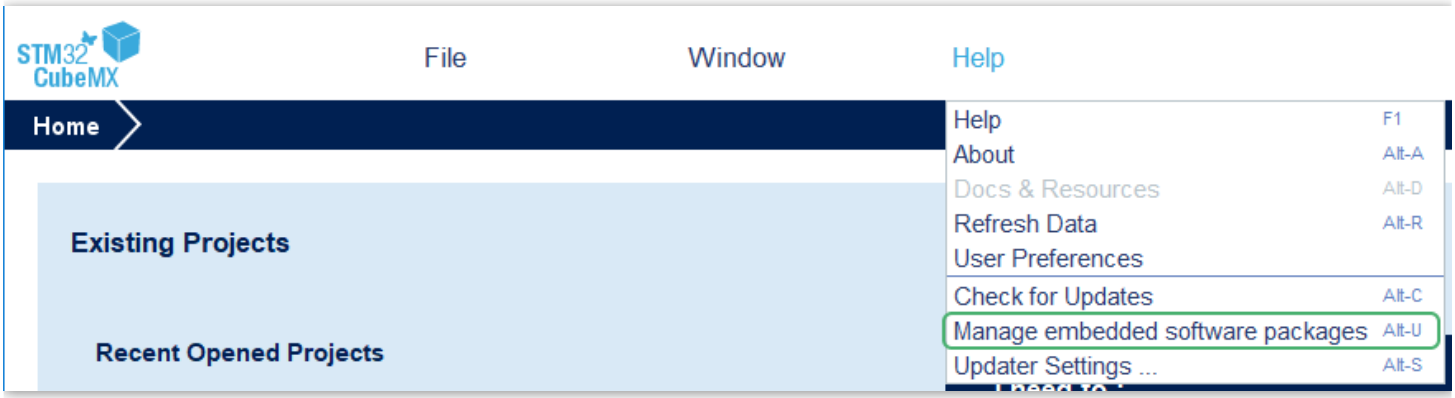
In order for the TouchGFX Designer and Makefiles to be able to use STM32 ST-LINK Utility for flashing target hardware it must be installed at the default install location: `C:\Program Files (x86)\STMicroelectronics\STM32 ST-LINK Utility`

Custom Product Development

If your intention is to run TouchGFX applications on either ST display kits or your own custom STM32 based platform, refer to the section Installing TouchGFX Generator in CubeMX.

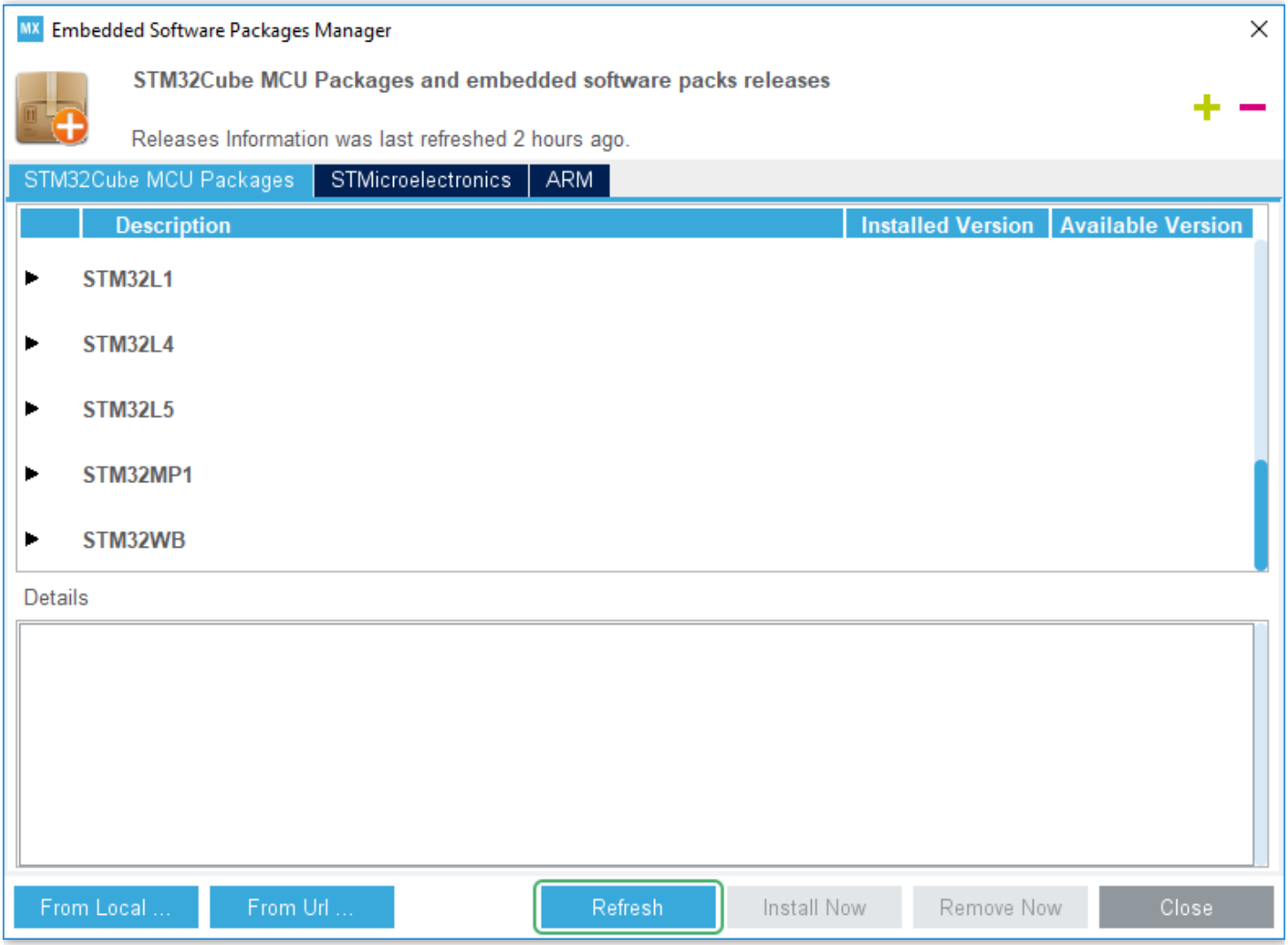
Installing TouchGFX Generator in CubeMX

To install X-CUBE-TOUCHGFX through CubeMX, start by selecting "Manage Embedded Software Packages" under "Help" (or press `ALT + U`).



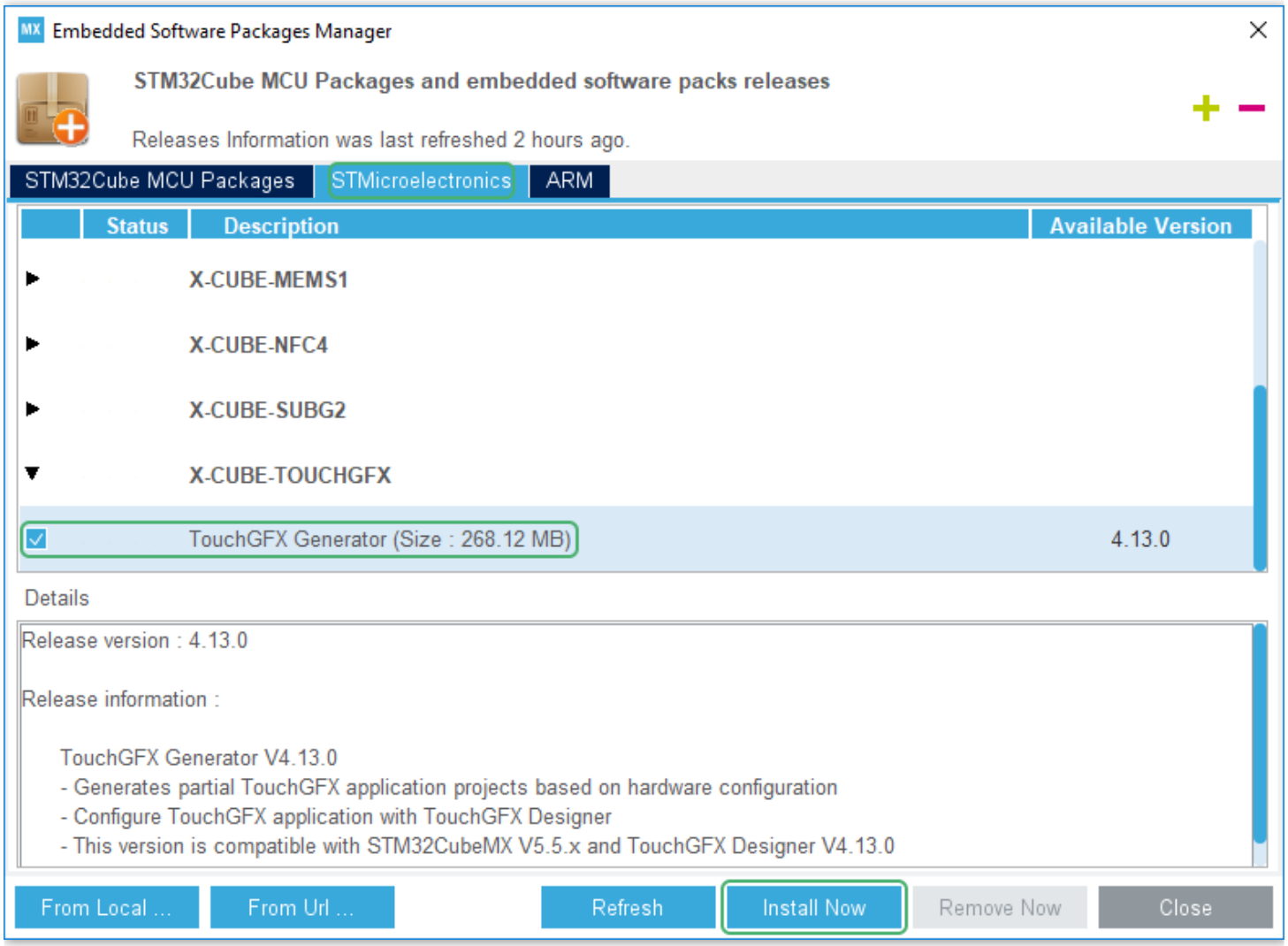
Help -> Manage embedded software packages

Click "Refresh" to get an updated list of available packages.



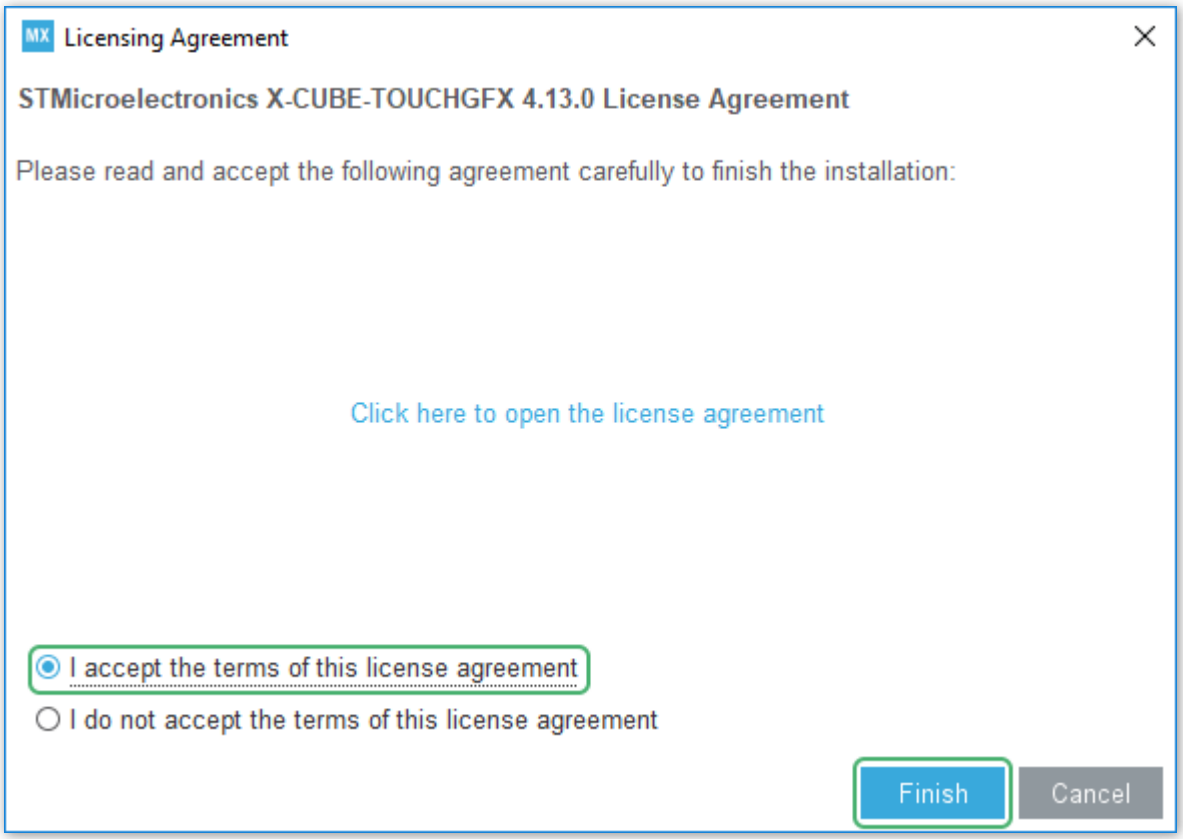
Refresh to update available packages

Go to the "STMicroelectronics" tab. Scroll until you find "X-CUBE-TOUCHGFX" and expand the node. Check the checkbox for "TouchGFX Generator" and click "Install Now". This will download the package and bring up the license agreement.



Finding the TouchGFX Generator package

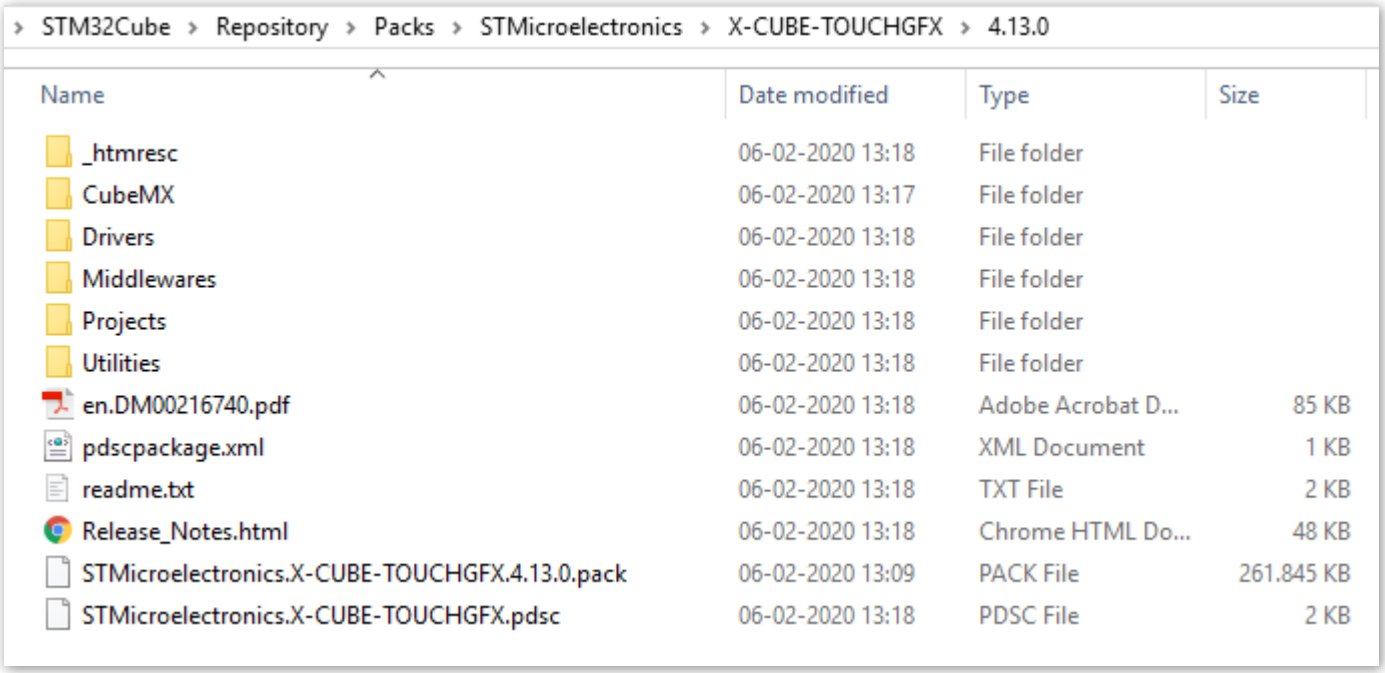
Read and accept the license agreement and click "Finish" to install TouchGFX Generator inside CubeMX.



Accepting the license agreement

The X-CUBE-TOUCHGFX distribution is now unpacked to the following path:

```
C:\Users\<user>\STM32Cube\Repository\Packs\STMicroelectronics\X-CUBE-TOUCHGFX\4.13.0
```



Location of the X-CUBE-TOUCHGFX package

Version: 4.16

Getting Started

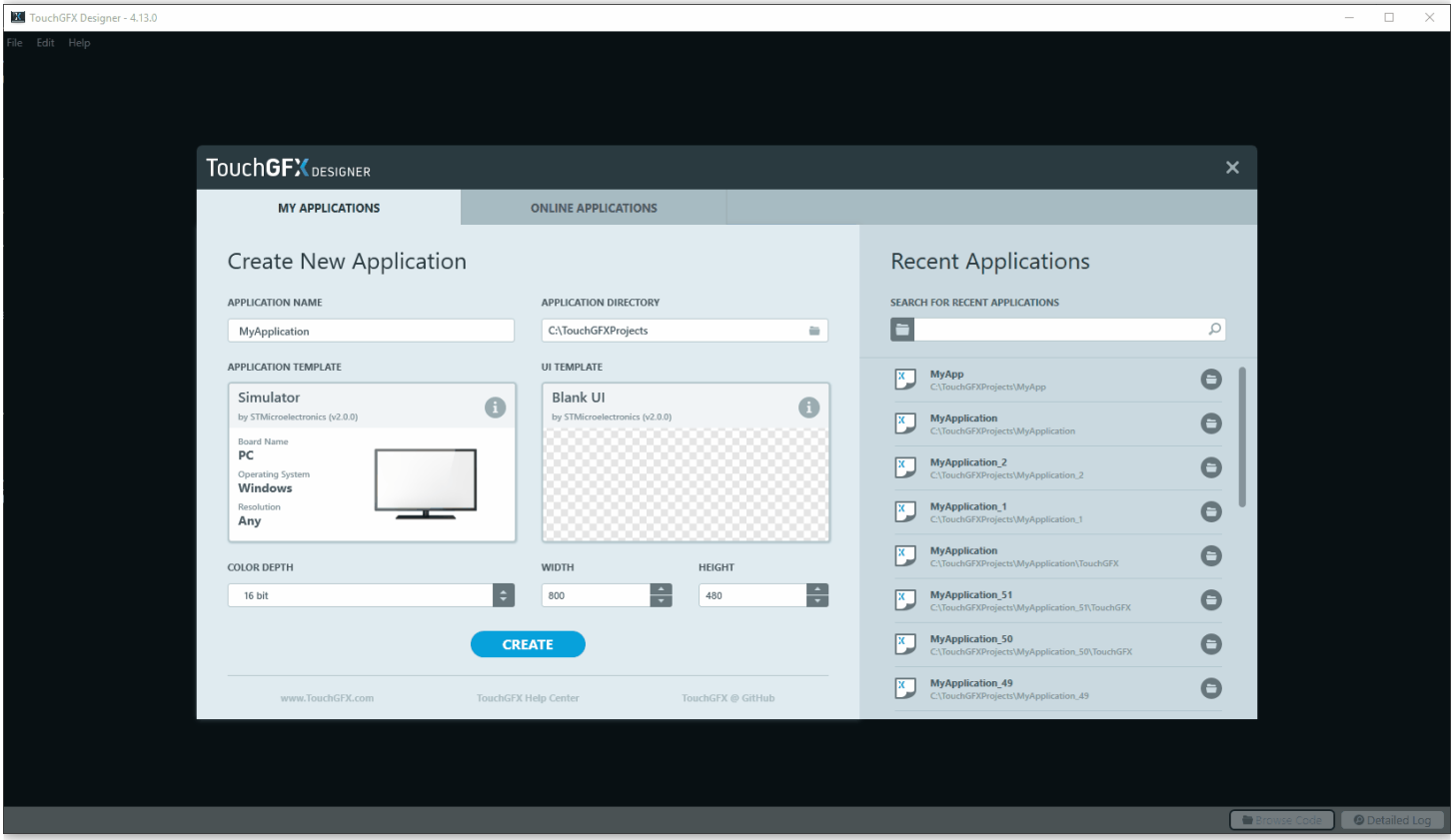
A prerequisite to get started using TouchGFX is to install the newest version of TouchGFX, which is described in the Installation section.

After this is done you are ready to begin your very first TouchGFX project. It is highly recommended that you read some key sections of this documentation to get familiar with the very basic of TouchGFX concepts. The most essential section is the Development Introduction which will give you an introduction to the software and hardware components needed for a complete TouchGFX project and the activities and tools involved in creating them. It also describes how to get started doing a fast prototype using premade components.

TouchGFX Designer Quick Start

Application Template + UI Template

If you are eager to start experimenting with TouchGFX, trying out an example UI project and maybe running it on an STM32 Evaluation Kit, you can go ahead and start the TouchGFX Designer.



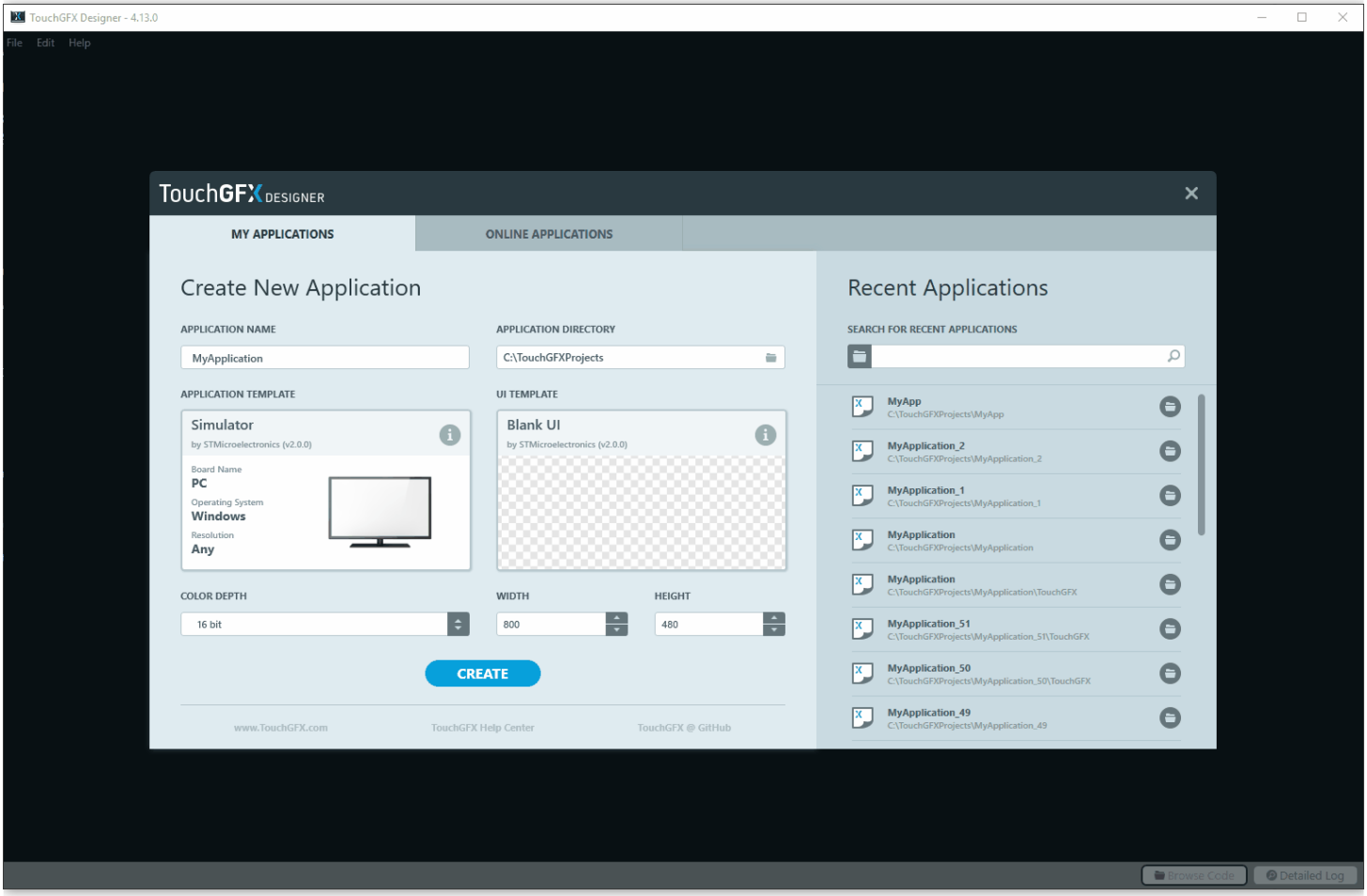
Quickly creating a project with an application template and a UI template

On the startup window in TouchGFX Designer you can select from a wide range of UI examples to start from. You can even combine this with board specific code for a range of STM32 Evaluation Kits, so you can try out the example on the embedded hardware right away.

If you do not have an STM32 Evaluation Kit, you can simply run the TouchGFX application on your PC. If you want to be guided through the first steps, you can have a look at the Tutorials chapter. Here you will be shown how to get a project up and running, ending with doing a rather complex UI application.

Online Applications

Aside from being able to combine an application template with a UI template, TouchGFX Designer also includes some out of the box demos for specific STM32 Evaluation Kits. These are located under the Online Applications tab.



Quickly creating a project with an online application

What's Next

At this point it is recommended that you browse through the documentation and read the chapters that look relevant to you. This will give you an overview of the documentation, so you know where to look for more information, and it will introduce you to key TouchGFX concepts.

The main chapters are:

- Basic Concepts: This chapter serves as background knowledge for the rest of the documentation. It introduces all the key graphics concepts that will be referenced later on. Depending on your current knowledge and which TouchGFX development activities you will do, you might want to skip some sections and consult them later if some details are unclear.
- Development: This chapter is the main chapter of the documentation. It explains the structure of a TouchGFX Project, the workflow and the tools involved in the entire project development cycle. Each step in the workflow has it own section and is described in details. Here you will find all you need to know on how to make your application run on your hardware and how you will do UI development, including descriptions of all features and components in the TouchGFX Framework.

Version: 4.16

Embedded Graphics

The term embedded graphics means many things.

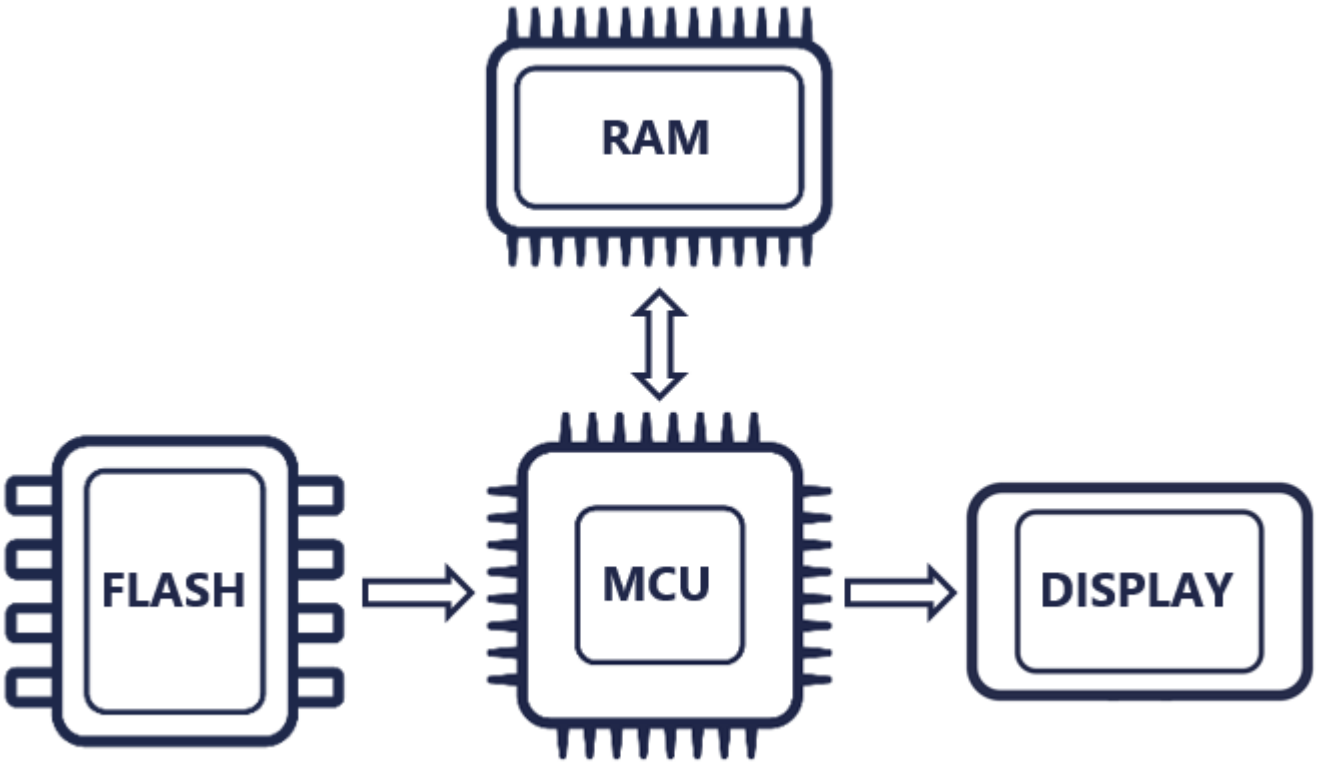
First of, the word embedded means different things to different people. To some an embedded system means a very dependable old 8 bit microcontroller, with no operating system, and virtually no RAM, ROM or GPIO. To others it might mean a modern day smart phone with gigabytes of everything.

Secondly, the word graphics has many interpretations. To some it means drawing your own pixels in your favourite painting program. To others again it means the 3D rendering software running on your gaming console.

To TouchGFX, embedded systems mean any system based on an STM32 microcontroller. And graphics means interactive applications with a user interface running at 60 frames per second.

The four main parts

In order to do graphics applications on such platforms, we consider four main components directly involved. Naturally many more components might be present in an embedded system, but these are less related to the display of graphics.

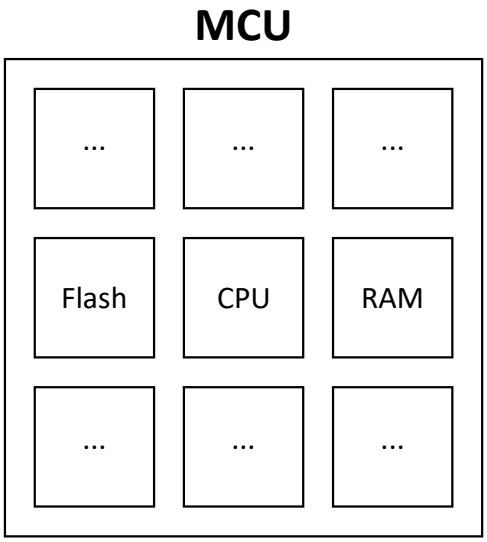


MCU, RAM, Flash and display

In short, TouchGFX, using the MCU, creates and updates an image in RAM, by assembling parts from flash. The assembled image is transferred to the display. This process is repeated as often as possible and needed.

MCU

The MCU is doing all the heavy lifting in this process. It reads images in flash and writes them to RAM. It calculates the resulting colors when merging a semi-transparent red text onto an image and saves these to RAM. It renders and stores all the pixels of a circle to RAM. It transfers the image from RAM to the display.



The MCU provides the CPU, internal flash, internal RAM and much more

STM32 MCUs have specific capabilities, like LTDC, Chrom-ART, Chrom-GRC, ..., that help in the realization of graphics.

FURTHER READING

See the **guide for selecting the right MCU** and the **board bring up guide for setting up the MCU**.

RAM

RAM is where the resulting calculated image (framebuffer) is stored. The RAM is being read and written by the MCU when updating the graphics. And read again when the resulting image is transferred to the display.

In many cases the resulting image is stored in RAM internal to the MCU. In cases where it is not feasible to have the resulting image in internal RAM, external RAM can be added to the setup.

FURTHER READING

See the **guide for selecting the right RAM** and the **board bring up guide for setting up the RAM**.

Flash

Flash is where all static data is placed. Images, fonts and texts. The flash is read by the MCU and the contents written to or combined with the RAM.

Most often an external flash is added to the setup, as the internal flash is seldomly large enough to hold all graphics assets. For very simple applications the internal flash is enough.

Ideally the flash is memory mapped, such that pixels can be read directly from the flash and written to RAM. Otherwise, when the flash is not memory mapped, the contents of the flash can be transferred to RAM and then read from there instead.

FURTHER READING

See the **guide for selecting the right flash** and the **board bring up guide for setting up the flash**.

Display

The display is where the image is actually displayed to the eyes of a person.

The calculated image (framebuffer) stored in RAM is sent by the MCU to the display at regular intervals.

FURTHER READING

See the **guide for selecting the right display** and the **board bring up guide for setting up the display**.

FURTHER READING

- See **Hardware selection** for details on the possible hardware choices.
- See **Board Bring Up** for details on setting up the board and components.

Version: 4.16

Color Formats

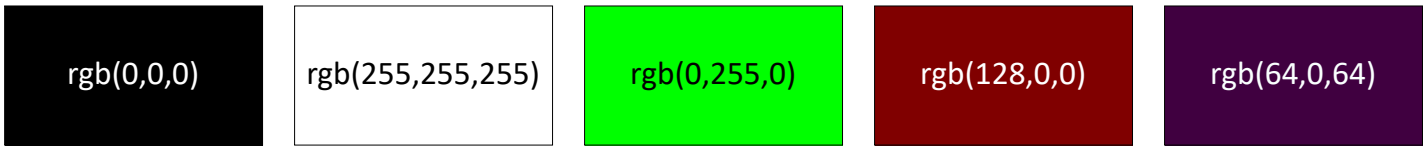
Colors are what is seen on the pixels of the display. These colors come from values stored in a framebuffer. Traditionally in graphics systems, there is a limited amount of possible colors that can be represented, used and displayed. This applies also to TouchGFX and TouchGFX applications.

The number of possible pixel colors of an application has an impact on many parts of an application. From the visual appearance of what is seen on the display to the memory consumption imposed by the framebuffer and the overall performance. This section will explain colors in TouchGFX in more detail and describe the color formats available in TouchGFX and highlight pros and cons.

Color

A color in TouchGFX is a triplet of red, green and blue components, known as an RGB color. Each component of the color ranges from 0 to 255. 0 means that the component is off, and 255 means that the component is at its maximum.

A completely black color is represented by the RGB color (0,0,0) and a completely white is (255,255,255). Bright green is (0,255,0), a semi bright red (128,0,0), a darkish purple (64,0,64) and so on.



Some RGB colors

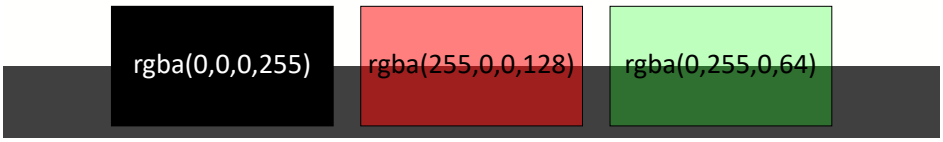
Grayscale

For grayscale applications all colors are gray, ranging from black to white and therefore the representation is the gray intensity instead of the RGB value. One can think of grayscale colors as RGB colors where R = G = B.

Opacity

In some circumstances we might consider colors to have a component describing the opacity of the color. The opacity ranges from 0 to 255, as the other components of the color. Colors with opacity are referred to as RGBA colors. The A stands for alpha, which is the classic name used for the opacity level.

A completely opaque black color here is (0,0,0,255), a somewhat transparent red is (255,0,0,128), and so on.



Some RGBA colors atop white and grey

When a color is not completely opaque it needs to be mixed together with the color already present. This mixing of colors is known as alpha blending.

Color depth

Color depth is the number of bits used to describe each color as stored in the framebuffer. We denote this number the bits per pixel, or in short bpp.

The more bits we use, the more colors we can describe.

A much used color depth is 24 bpp. As each bit can be either on or off, this implies that 2^{24} = 16777216 different colors can be represented.

Another, a little less used, color depth is 1 bpp. This color depth applies to black and white applications, and implies that only 2^1 = 2 different colors can be represented.

TouchGFX has built-in support for the following color depths:

- 32 bpp - 16777216 colors and corresponding opacity values
- 24 bpp - 16777216 colors
- 16 bpp - 65536 colors
- 6 bpp - 32 colors
- 4 bpp - 16 grayscale colors
- 2 bpp - 4 grayscale colors
- 1 bpp - 2 grayscale colors

A note on color component ranges. When working with less than 24 bpp color depth, each of the red, green and blue components does not directly range from 0 to 255. Instead such a component, say red in 16 bpp, ranges 0 to 31. We will think of the value 31 as representing the most red we can represent in 16 bpp, i.e. 255 in 24 bpp. One way of thinking of this is that colors of 16 bpp depth can only represent a subset of the colors possible in 24 bpp.

Formats

Having determined the amount of bits needed to represent colors, we investigate the contents of the bits in more detail. A color will have some bits describing the red component, some the green and some the blue.

Pixel color formats

Dependent on the color depth of the application, some particular color formats will be available.

RGB888

In TouchGFX, a color of 24 bpp color depth will have the color format RGB888. This means that 8 bits are used for each of the color components red, green and blue.

Representing such a color, say bright purple rgb(255,0,255), in code is done by assembling the components into a color value

```
uint32_t brightPurpleRGB888 = 255 << 16 | 0 << 8 | 255 << 0;
```

RGB565

For 16 bpp colors, TouchGFX uses the color format RGB565. That is 5 bits for red, 6 bits for green, 5 bits for blue. As we have 5 bits for red, fully lit is 31, and a bright red in code is

```
uint16_t brightPurpleRGB565 = 31 << 11 | 0 << 5 | 0 << 0;
```

RGBx2222, xRGB2222, BGRx2222, xBGR2222

For 6 bpp colors, TouchGFX supports 4 different formats, RGBx2222, xRGB2222, BGRx2222, xBGR222. 6 bit colors are stored as bytes and this is the reason for having the x in the forementioned formats. The color is padded with 2 bits, to fit into a byte. The reason for having both RGB and BGR is that some displays need this and we do not want to convert pixels before sending them to the display. Representing a bright yellow in RGBx2222 in code is

```
uint8_t brightYellowRGBx2222 = 3 << 6 | 3 << 4 | 0 << 2;
```

GRAY4, GRAY2, BW

For each of the grayscale color depths TouchGFX supports one corresponding color format. For 4 bpp the color format is denoted GRAY4, for 2 bpp it is GRAY2 and for 1 bpp it is denoted BW for black and white. In 4 bpp a completely white color is

```
uint8_t whiteGRAY4 = 15;
```

TouchGFX includes a helper function that will return the correct representation of a color in the correct color format.

```
#include <touchgfx/Color.hpp>
...
aColor = Color::getColorFrom24BitRGB(255,0,128);
```

Image formats

Images are an important part of most UI applications and images are filled with colors. In TouchGFX images are stored in memory and are filled with colors of a specific format. In many cases images are using one of the supported pixel color formats, but other image formats are also available. A pixel in an image of a particular image color format will be converted into the appropriate pixel format before being drawn

Image Color format	Description
ARGB8888	32 bits, 8 bits per component
L8_ARGB8888	8 bits indexed format, ARGB8888 palette
RGB888	24 bits, 8 bits per component.
L8_RGB888	8 bits indexed format, RGB888 palette
RGB666	24 bits, 6 bits per component
RGB565	16 bits, 5 bits red, 6 bits green, 5 bits blue
L8_RGB565	8 bits indexed format, RGB565 palette
ARGB2222	8 bits, 2 bits per component
ABGR2222	8 bits, 2 bits per component
RGBA2222	8 bits, 2 bits per component

Image Color format	Description
BGRA2222	8 bits, 2 bits per component
GRAY4	4 bits grayscale
GRAY2	2 bits grayscale
BW	1 bit grayscale
BW_RLE	1 bit grayscale run-length encoded

Some of these image formats, the L8 ones, are representing the image in question by a lookup table of colors (known as a CLUT) and indices into this table. The maximum number of possible colors in such an L8 image is 2^8-1 = 255. The L8 formats take up less space than their non-L8 counterparts, as an example a 100x100 image with 200 different colors, takes up 100*100*32 bits = 40000 bytes when stored in the ARGB8888 format, and only 100*100*8 bits + 200*32 bits = 10800 bytes when stored in the L8_ARGB8888 format.

The image format BW_RLE stores the colors as consecutive runs of black and white instead of storing single pixel colors. This can in many cases also be more space efficient.

The rest of the formats are the same as the pixel color formats above.

Text formats

Texts, or more accurately glyphs, are also stored in memory in a specific color format. The available text color formats in TouchGFX are

Text Color format	Description
A8	8 bits, opacity only
A4	4 bits, opacity only
A2	2 bits, opacity only
A1	1 bits, opacity only

Glyph formats are comparable to small images, where each color entry holds the level of opacity of each pixel. This enables applying the actual color, the red, green and blue components, at a later time, and enables drawing for instance the stored glyph 'A' in both a blue version and a red version.

The more bits used for each glyph the smoother and nicer it will typically appear.

Visual quality

When doing embedded graphics we want the highest visual quality, but at the same time we need to look at the amount of memory consumed.

Therefore it is many times desirable to go for an RGB565 color format instead of the richer RGB888 and in general we should go for the color format that provides us the most visual quality, while respecting our memory requirements.

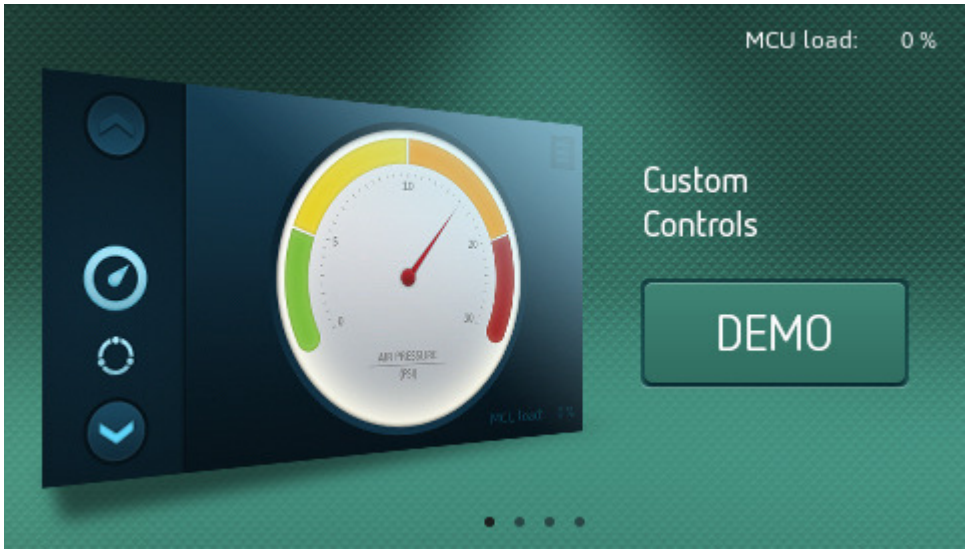
Dithering

TouchGFX employs what is known as dithering to improve the visual quality of images when representing these in different color formats.

Dithering is a well known technique for making images appear to have more colors than what is actually present. This is done by adding a bit of noise to the colors of the image.

One example - when converting an RGB888 image to an RGB565 image, instead of just chopping of the lower bits of each color component, the conversion process adds some noise to each of the resulting colors, the result being that the converted image looks richer and similar to the RGB888 original.

Explaining by images instead of words, we have an original RGB888 image, and a number of converted images. The converted images have the formats RGB565 with and without dithering, xRGB2222 with and without dithering, GRAY4 with and without dithering.



Original RGB888 image

Converted RGB565 images with and without dithering

Converted xRGB2222 images with and without dithering

Converted GRAY4 images with and without dithering

As can be seen dithering improves the perceived quality of images quite a bit. When looking closely at the RGB565 with and without dithering, it can be seen that the dithered version looks almost exactly like the original whereas the undithered one has some areas where color banding is apparent. This examplifies that in many cases 16 bit colors is more than enough to do great looking graphics.

When your graphics assets has big gradients, you might experience some color banding even in dithered images. Here are two examples. A bluish gradient ranging from RGB888 (64,190,222) to black and the converted RGB565 image with and without dithering.

Original RGB888 and converted RGB565 images with and without dithering

And another red gradient ranging from (255,0,0) to black.

Original RGB888 and converted RGB565 images with and without dithering

Looking closely, it can be seen that color banding is present in both the dithered and the undithered RGB565 versions. The red image has the most noticeable bands.

Always pay close attention to your resulting images and color formats and if needed alter your original images or choose another color format.

Performance

All the image formats discussed are optimized for "easiness" of drawing. This means that the images can more or be less copied to the framebuffer without much conversion taking place.

This is intentional and is one of the reasons TouchGFX can achieve fluent graphics on microcontrollers.

When designing a UI with TouchGFX one uses .png images and at compile time each of these images are converted into one of the efficient image formats detailed above. Read more on image formats and performance.

Alpha blending

At runtime the copying of image data is done either by a regular CPU copy operation or by using features of the MCU. If the image includes pixels that are not completely transparent or opaque, the pixels need to be alpha blended onto the background. In some STM32 MCUs this blending is supported by the hardware.

Other image formats

If you need to support other image formats at runtime, for example compressed image formats, such as .jpg or .png you can utilize the support of TouchGFX for dynamic bitmaps.

FURTHER READING

Wikipedia article on **color depth**.

Version: 4.16

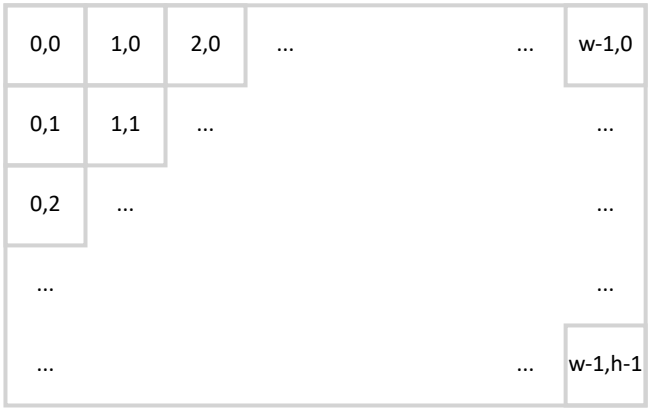
Framebuffer

A framebuffer is a piece of memory that is updated by the graphics engine to contain the next image to be shown on the display.

The framebuffer is a contiguous part of RAM of a given size.

Framebuffer memory

A framebuffer has an associated width and height. Therefore we typically think of a framebuffer as being a two dimensional part of memory, indexable by x, y coordinates.



2D framebuffer memory

A framebuffer has an associated color format. Each entry in the framebuffer will be a color in this color format. We will refer to each such entry in the framebuffer as a pixel.

We can update the color of a pixel at position x,y in the framebuffer by calculating the memory address of the pixel within the framebuffer and updating the stored color.

```
uint32_t pixelAddress = x + y * WIDTH;
framebuffer[ pixelAddress ] = newColor;
```

Similarly we can obtain the color of a pixel in the framebuffer and use this in calculations. For instance darkening the color of a pixel in the framebuffer (assuming we have a `darken` function available).

```
uint32_t pixelAddress = x + y * WIDTH;
framebuffer[ pixelAddress ] = darken( framebuffer[ pixelAddress ] );
```

Often the framebuffer memory is not written and read pixel by pixel as above, but by utilizing the underlying hardware capabilities of the system, e.g. the Chrom-ART DMA.

Colors

In TouchGFX the pixel color format of a framebuffer can be either:

- **Grayscale** 1, 2 or 4 bits per pixel (bpp) grayscale, or
- **High or true color** 16, 24 or 32 bpp color

The more bits per pixels used the more distinct colors can be represented by the framebuffer, moreover the more bits per pixels used the more memory will be consumed by the framebuffer.

Display

The contents of the framebuffer is what is ultimately transferred to and seen on the physical display. Therefore it is very common to have the same pixel width and height of the framebuffer and the display.

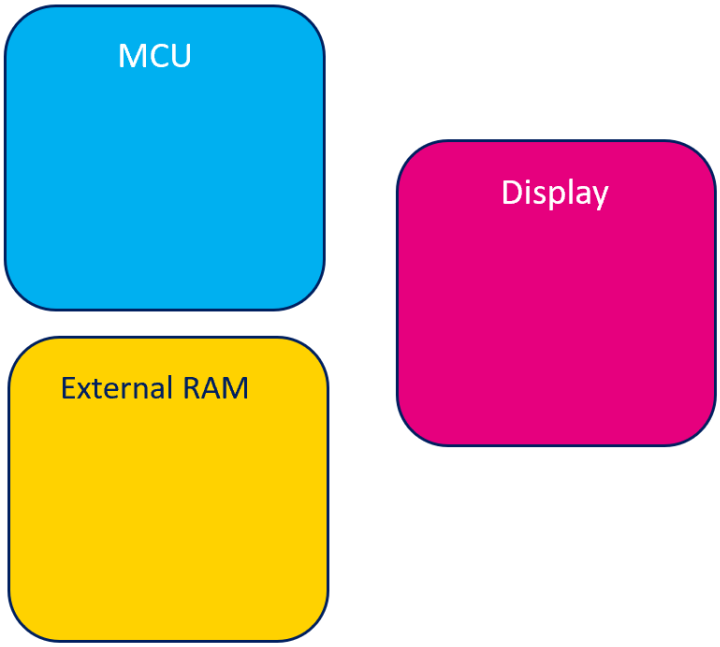
24 bpp framebuffer contents and resulting display

FURTHER READING

See the section on **display technologies** to learn more on different display types.

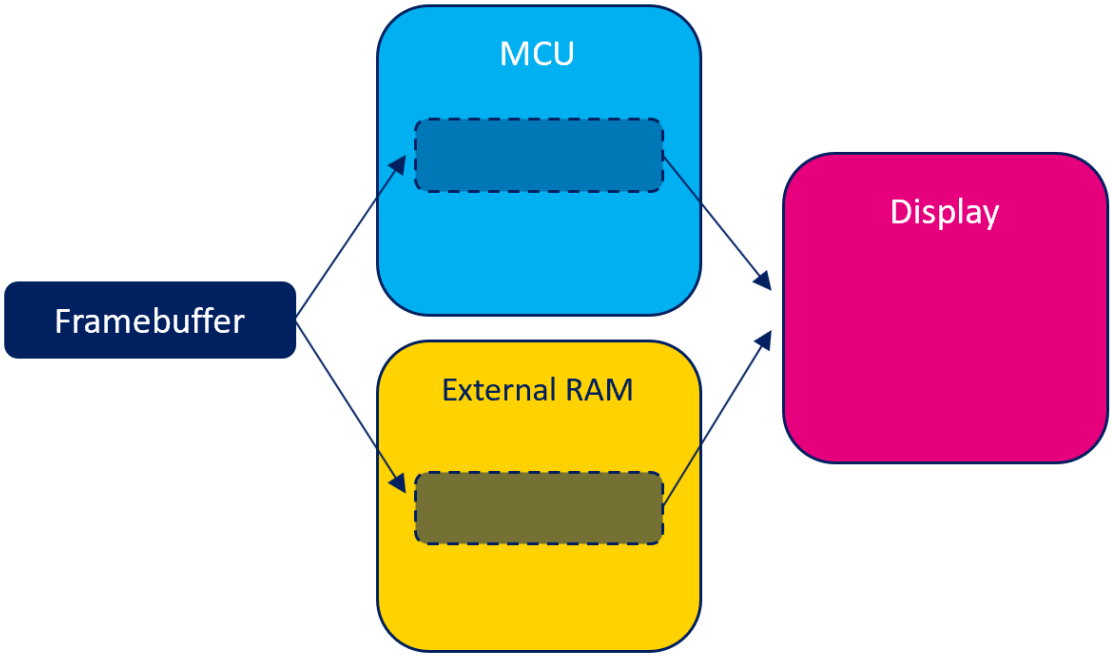
Location of framebuffer

One oversimplified view of a microcontroller based graphics system is here.



Oversimplified view of graphics system

A framebuffer can be placed either internally in the MCU or in external RAM.



Possible locations of framebuffer

Each possible location has potential benefits and drawbacks.

Internal RAM

Placing the framebuffer in RAM internal to the MCU makes the read and write access to the framebuffer as fast as possible. This means that your TouchGFX application will run as smooth as

possible. Conversely, internal RAM is a very scarce resource and one that is used by many parts of a system, therefore occupying a large part of this with a framebuffer might be infeasible.

If feasible, having a framebuffer in internal RAM, could reduce the overall cost of the system as no additional RAM is needed.

External RAM

If the system has external RAM, placing the framebuffer here is an alternative to placing it in internal RAM. The read and write access to external RAM will typically be slower than to the internal RAM, but the amount of external RAM will typically be much larger. Therefore this is sometimes the only viable solution.

The MCU might have capabilities, like caching, that makes access to external RAM faster. See the section on MCU for details.

Display with embedded RAM

Depending on the type of display in the system there might be memory embedded on the display. This memory holds the contents of the "physical" pixels of the display. Having this pixel memory in the display implies that the MCU can be idle while the display is still alive.

Placing a TouchGFX framebuffer in the RAM of the display is not possible, as the memory of the display is not memory mapped and is not intended nor suitable for random pixel reads or writes. Instead TouchGFX places the framebuffer in internal or external RAM and transfers this to the display RAM when appropriate.

Amount of framebuffers

TouchGFX can use one, multiple, or less than one framebuffer in the application. The amount of framebuffers might impact the visual appearance, performance and memory consumption of the application.

One framebuffer

One framebuffer is enough to hold precisely all pixels that are to be transferred to the display. One framebuffer (at least) is needed when the display has no display RAM on board. In this case one framebuffer is ideal when the complexity of the displayed graphics does not induce any visual artifacts.

More than one framebuffer

In TouchGFX, having multiple framebuffers means having two framebuffers. One framebuffer is used for writing the next resulting image, the other framebuffer is used for transferring to the display. This implies that no visual artifacts, e.g. tearing, will appear.

Less than one framebuffer

Having less than one framebuffer in general implies that

- less memory is consumed
- more drawing operations will be performed
- more transfers to display needed

In TouchGFX less than one framebuffer is denoted a partial framebuffer. The partial framebuffer scheme is available only for displays with display RAM.

Memory consumption

The amount of colors and the number of pixels in the framebuffer determines the memory consumed by the framebuffer.

In general the amount of memory used by a framebuffer is **width * height * color depth in bits / 8** bytes.

Resolution (pixels)	Colors (bpp)	Calculation	Memory consumed (byte)
800x480	16 bpp	800 * 480 * 16 / 8	768.000 B
480x272	24 bpp	480 * 272 * 24 / 8	391.680 B
100x100	8 bpp	100 * 100 * 8 / 8	10.000 B

When having more than one framebuffer the amount of memory consumed will be correspondingly larger. For example when having a double buffering scheme, using two framebuffers, will consume twice the amount of memory.

When having less than one framebuffer the amount of memory is explicitly allocated and controlled by the application. The memory consumption is therefore completely customizable, but be warned that using too little will harm the overall graphics performance.

FURTHER READING

- The STM32 **LTDC** display controller document has further details on framebuffers

Version: 4.16

Graphics Engine

TouchGFX graphics engine's main responsibility is drawing graphics on the display of an embedded device.

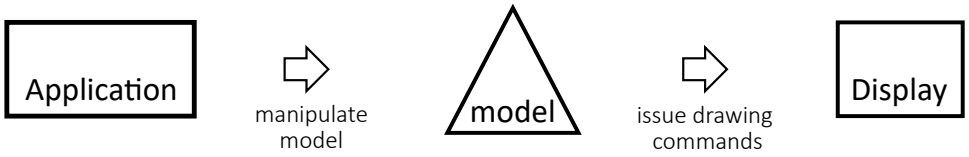
This section will give an overview of what kind of graphics engine TouchGFX is and provide some background on why it is this way.

Scene model

Graphics engines can be divided into two main categories.

- **Immediate mode graphics engines** provide an API that enables an application to directly draw things to the display. It is the responsibility of the application to ensure that the correct drawing operations are invoked at the right time.
- **Retained mode graphics engines** let the user manipulate an abstract model of the components being displayed. The engine takes care of translating this component model into the correct graphics drawing operations at the right times.

TouchGFX follows the retained mode graphics principles. In short this means that TouchGFX provides a model that can be manipulated by the user and TouchGFX then takes care of translating from this model into an optimized set of rendering method calls.



Retained mode graphics

The benefits of TouchGFX being retained are many. Primary ones are:

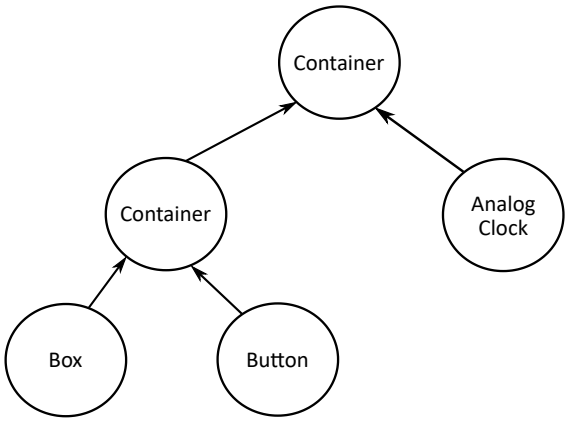
- **Ease of use:** A retained graphics engine is easy to use. The user adresses the configuration of components on screen, by invoking methods on the internal model and does not think in terms of actual drawing operations.
- **Performance:** TouchGFX analyses the scene model and optimizes the drawing calls needed to realize the model on screen. This includes deliberately not drawing hidden components, drawing and transferring only changed parts of components, managing framebuffers, and much more.
- **State management:** TouchGFX keeps track of which part of the scene model is active. This in turn makes it easier for the user to optimize the scene model contents.

The main drawback of TouchGFX adhering to the retained mode graphics scheme is:

- **Memory consumption:** Representing the scene model takes up some memory. TouchGFX reaches its performance levels, typically rendering 60 frames per second, by analyzing the scene model and optimizing the corresponding rendering done. Great effort has gone into reducing the amount of memory used by the scene model of TouchGFX. In typical applications the amount of memory for this model is well below one kilobyte.

Manipulate the model

The scene model consists of components.



A tree of widgets

Each of the components in the model has exactly one associated parent component. The parent component itself is also part of the scene model. Such a model is widely referred to as a tree.

We will often refer to a component in this tree as a UI component or a widget.

From the point of view of the application, the displayed graphics are updated by setting up and manipulating the widgets in the scene model. An example setup of a button, with position, images and added to the scene model, is:

```
myButton.setXY(100,50);
myButton.setBitmaps(Bitmap(BITMAP_BUTTON_RELEASED_ID), Bitmap(BITMAP_BUTTON_PRESSED_ID));
add(myButton);
```

Issue drawing commands

Ultimately, when rendering the scene model, TouchGFX will utilize its drawing API. This drawing API has methods for drawing graphics primitives, such as boxes, images, texts, lines, polygons, textured triangles, etc.

As an example, the Button widget in TouchGFX, when rendered, uses the drawPartialBitmap method for drawing images. The source code for the drawing of the button widget in TouchGFX looks close to:

touchgfx/widgets/button.cpp

```
void Button::draw(const Rect& invalidatedArea) const
{
    // calculate the part of the bitmap to draw
    api.drawPartialBitmap(bitmap, x, y, part, alpha);
}
```

inspect the actual source in `touchgfx/widgets/button.cpp` *for details.*

TouchGFX includes optimized implementations of the drawing API. The drawPartialBitmap method for instance utilizes the underlying hardware (Chrom-ART if available) to draw the bitmap.

One can utilize these API drawing methods when extending the scene model with new types of widgets. See the section on creating your own custom widget.

The implementation of the drawing API methods is platform specific and highly optimized for each specific MCU.

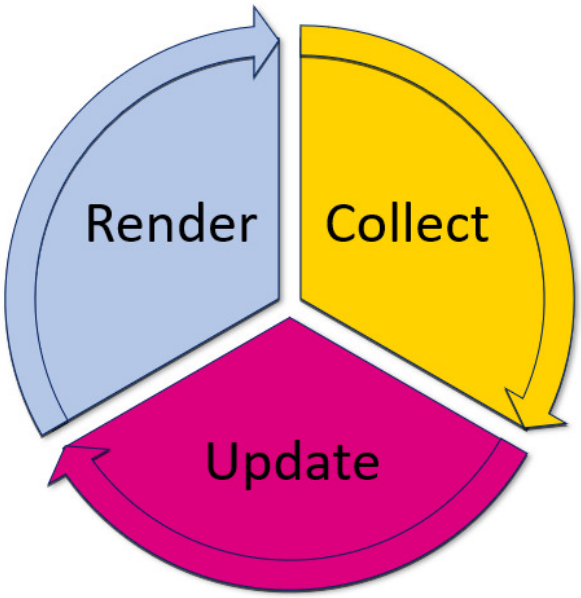
Main Loop

The workings of many game engines, graphics engines and in particular TouchGFX can be thought of as an infinite loop.

Within the main loop of TouchGFX there are three main activities:

- **Collect events:** Collect events from the touch screen, presses of physical buttons, messages from backend system, ...
- **Update scene model:** React to the collected events, updating the positions, animations, colors, images, ... of the model
- **Render scene model:** Redraw the parts of the model that has been updated and make them appear on the display

A diagrammatic version of the main loop is:



Main loop

Each execution of the main loop is denoted a tick of the application.

Platform adaptibility

As TouchGFX is designed for running on all STM32 embedded setups the above activities can be tailored.

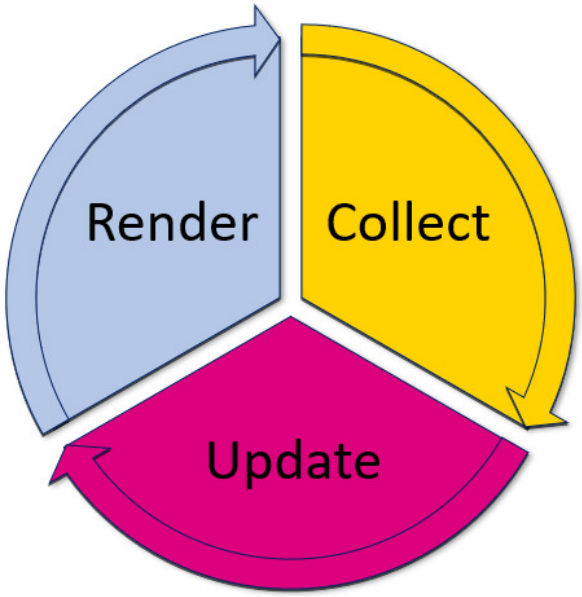
- The collection of events is handled by a dedicated abstraction layer. The tailoring of this layer is the subject of the section on TouchGFX AL Development.
- The updating of the model is completely up to the application. The details on how to do this update is the subject of UI Development.
- The rendering of graphics to the framebuffer is handled by TouchGFX and will in general not need to be customized. The transferring of the pixels in the framebuffer to the display is platform specific, see Board Bring Up and TouchGFX AL Development for how to customize this to specific platforms.

Read on to get more specifics on the main loop of TouchGFX.

Version: 4.16

Main Loop

In this section you will learn more about the workings of the graphics engine in TouchGFX and in particular the main loop. Recall that the main task for the graphics engine is to render the graphics (the ui model) of your application in to the framebuffer. This process happens over and over again to produce new frames on the display.



The graphics engine *collects* external events like display touches or button presses. These events are filtered and forwarded to the application. The application can use these events to update the ui model. E.g. by changing a button to its pressed state when the user touches the screen over the button, and later changing the button back to the released state when the user does not touch the screen anymore.

Finally the graphics engine renders the updated model to the framebuffer. This process loops forever.

After rendering a frame, the framebuffer is transferred to the display, where the user can see the graphics. The transfer to the display must be synchronized with the display to avoid disturbing glitches on the display. For some displays the transfers must happen regularly with a minimum time inverval. For other displays the transfer must happen when a signal is sent from the display.

The graphics engine implements this synchronisation by waiting for a "go" signal from the hardware abstraction layer. Read more about the hardware abstraction layer here

In pseudo code the main loop inside the TouchGFX graphics engine looks like this:

```
while(true) {
    collect();    // Collect events from outside
```

```
    update();     // Update the application ui model
    render();     // Render new updated graphics to the framebuffer
    wait();       // Wait for 'go' from display
}
```

The code is more involved in the real implementation, but the pseudo code above helps in understanding the main parts of the engine.

We will discuss these four stages in more detail below.

Collect

In this phase the graphics engine collects events from the outside environment. These events are typically touch events and buttons.

TouchGFX samples and propagates events to the application. The raw touch events are converted into more specific touch events:

- **Click:** The user pressed or released his finger from the display
- **Drag:** The user moved his finger on the display (while touching the display).
- **Gestures:** The user moved his finger fast in a direction and then released. This is called a *swipe* and is recognized by the graphics engine.

The events are forwarded to the ui elements (e.g. widgets) that are currently active.

The engine also forwards a *tick* event. This event represents the new frame (or the step in time), and is always send, also if there was no other external input. This event is used by applications to drive animations, or other timebased actions like changing to a pause screen after a specific time has elapsed.

Update

Here the graphics engine works together with the application to update the ui to reflect the collected events. The graphics engine knows which Screen is currently active and passes events to this object.

The basic principle is that the engine informs the application (i.e. the Screen and Widget objects in the ui model) about the events. In response, the application requests specific parts of the display to be redrawn. The application does not draw directly as response to the events, it changes the properties of Widgets and request redraws.

If for example a Click event occurs, the graphics engine searches the scene model of the Screen object to find the Widget that should receive the event. Some Widgets like Image and TextArea do not wish to receive Click events. They further have an empty event handler, so nothing happens.

Other Widgets like Button reacts to a Click event (pressed or released). The Button widget changes its state to show another image when pressed, and changes the state back again when the touch is released again.

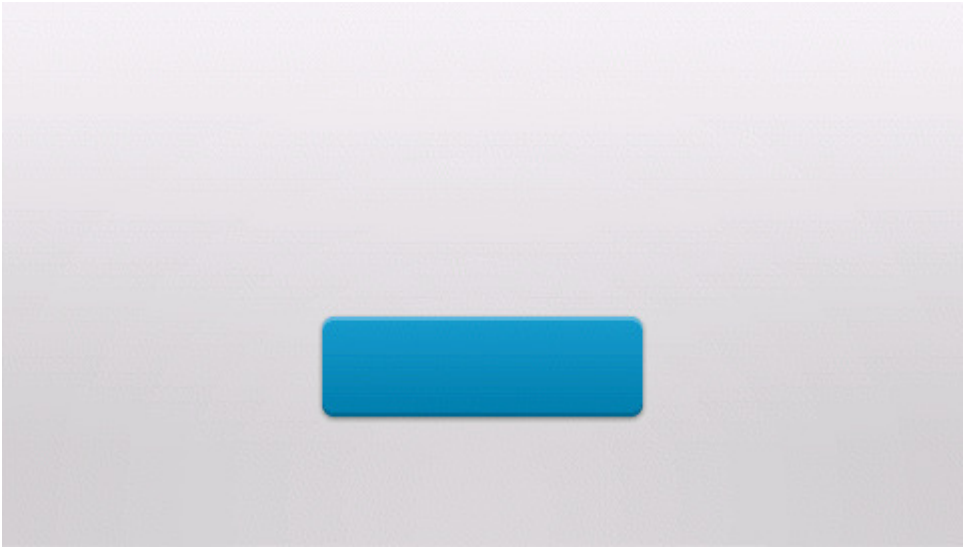


Image widget in the background with a Button widget in front

When a Widget like the Button changes its state, it must also be redrawn in the framebuffer. The Widget is responsible for communicating this back to the graphics engine as response to the event. The graphics engine does not itself redraw any Widgets based on the collected events. The Widgets keep track of their own internal state (for a Button, what image to draw), and instruct the graphics engine to redraw the part (a rectangle) of the display that is covered by the Widget.

The application itself can also react to the events. One of two ways are common:

- **Configure an interaction for a Widget in TouchGFX Designer** For example, we can configure an interaction to make another Widget visible when the Button is pressed. This interaction is executed after the Button has changed its state and requested a redraw of itself from the graphics engine. If you use the interaction to show another (invisible) Widget, the application should also request a redraw from the graphics engine.
- **React to events on the Screen** It is also possible to react to events in the Screen itself. The event handler are virtual functions on the Screen class. These functions can be reimplemented in the Screens in the application. This can e.g. be used to perform an action whenever the user touches the screen no matter where if the touch is on a Widget.

The Screen class has the following event handlers. These are called by the graphics engine when the corresponding external event has been collected:

framework/include/touchgfx/Screen.hpp

```
virtual void handleClickEvent(const ClickEvent& evt);

virtual void handleDragEvent(const DragEvent& evt);

virtual void handleGestureEvent(const GestureEvent& evt);

virtual void handleTickEvent();

virtual void handleKeyEvent(uint8_t key);
```

Any C++ code can be inserted in these event handlers. Typically applications update the state of some Widgets and/or call some application specific functions (business logic).

Time based updates

The handleTickEvent event handler is called in every frame. This allows the application to perform time based updates of the user interface. An example could be to fade a Widget away after 10 seconds. Assuming that we have 60 frames in a second, the code could look like:

```
void handleTickEvent() {
  tickCounter += 1;
  if (tickCounter == 600) {
    myWidget.startFadeAnimation(0, 20); // Fade to 0 = invisible in 20 frames
  }
}
```

The graphics engine also calls an event handler on the Model class. This event handler is typically used to perform repeated actions like checking message queues or sampling GPIO:

```
void Model::tick() {
  bool b = sampleGPIO_Input1(); // Sample polled IO
  if (b) {
    ...
  }
}
```

Requesting a redraw

As we discussed above with the Button example; the Widgets are responsible for requesting a redraw when their state changes. The mechanism behind this is called an *invalidated area*.

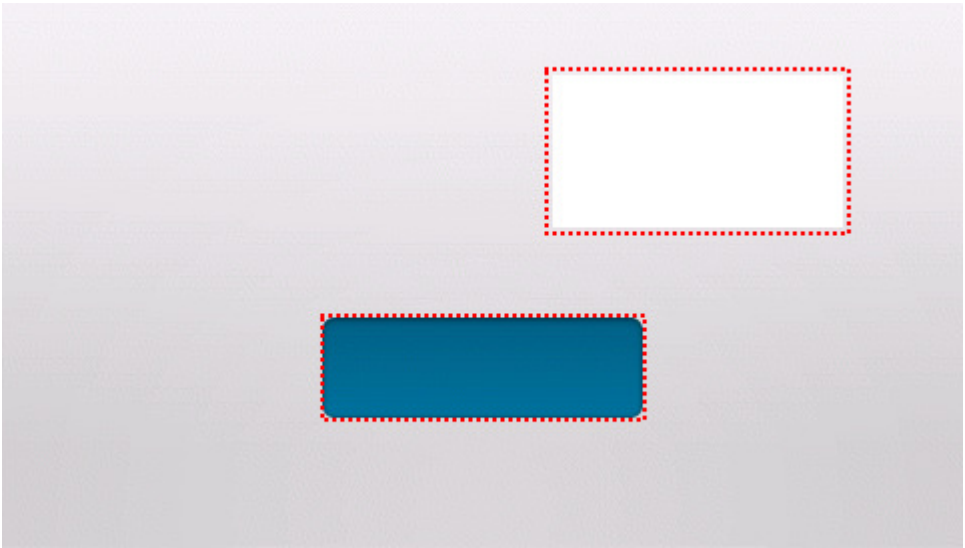
When the Button changes state (e.g. from released to pressed) and needs a redraw, the area covered by the Button Widget is an invalidated area. The graphics engine keeps a list of these invalidated areas requested for the frame. All the collected events (touch, button, tick) may result in one or more invalidated areas, so there can be many invalidated areas in every frame.

The event handlers on the Screen class can also request a redraw of an area. Here we change the color of a Box widget, box1, in frame 10 and request a redraw by calling the *invalidate* method on the Box:

```
void handleTickEvent() {
  tickCounter += 1;
  if (tickCounter == 10) {
    box1.setColor(Color::getColorFrom24BitRGB(0xFF, 0x00, 0x00)); // Set color to red
    box1.invalidate();                                            // Request redraw
  }
}
```

In this example the graphics engine will call the *handleTickEvent* handler in every frame. In the 10th frame, the application code requests a redraw of the area covered by *box1*. As a response to this the graphics engine will redraw that area in the framebuffer using the color saved in the *box1* widget.

In the user interface below we have a Button Widget and a Box Widget on top of a background image. If we insert an interaction on the Button to change the color of the Box when the Button is clicked we get two invalidated areas (indicated with red) when the user clicks the Button:



Two invalidated areas

The area of the Box is invalidated to get the new color drawn in the framebuffer. The Button also invalidates itself to get the released state drawn again.

Render

As we just discussed, the result of the update phase is a list of areas to redraw, the invalidated areas. The task for the render phase is basically to run through this list and draw the Widgets covering these areas into the framebuffer.

This phase is handled automatically by the graphics engine. The application has defined the scene model (the Widgets in the ui) and invalidated some area. The rest is handled by the engine.

The graphics engine handles the invalidated areas one-by-one. For each area the engine scans the scene model and collects a list of the widgets that is covered by the area (partly or in whole).

Given this list of Widgets the graphics engine calls the draw method on the Widgets. Starting with the widget in the background and ending with the foremost Widget.

The Widget's draw methods use the state of the Widget, e.g. the color, when drawing to the framebuffer. Any information that is needed to draw the Widget must be saved to the Widget during the update phase. Otherwise this information is not available in the rendering phase.

Wait

The TouchGFX graphics engine waits for a signal before updating and rendering the next frame. There are two reasons for waiting between the frames instead of just continuously rendering frames as fast as possible:

- The rendering is synchronized with the display. As mentioned above some displays requires that the framebuffer is transmitted repeatedly. While this transmission is ongoing, it is not adviseable to render arbitrarily to the framebuffer. The graphics engine therefore waits for a short time after the transmission is started before starting the rendering. Other displays send a signal to the microcontroller when the framebuffer should be transmitted. The graphics engine waits for that signal.

- Frames are rendered at a fixed rate. It is often beneficial for the application that frames are rendered at a fixed rate, as this makes it easier to create animations that lasts a specific time. For example, if you have a 60 Hz display, a two seconds animation should be programmed to complete in 120 frames.

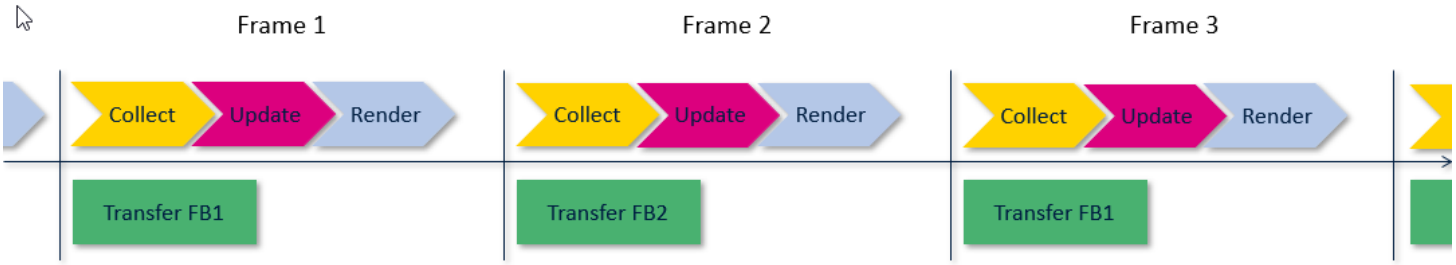
The time where the graphics engine is waiting is typically used by other lower priority processes in the application. In these cases the time is not wasted, as the lower priority processes should run at some point in time anyway.

Handling the framebuffers

Recall from the discussion previously that the graphics engine synchronizes with the display before the framebuffer is updated. After the rendering to the framebuffer the engine also needs to make sure that the display shows the updated framebuffer.

Two framebuffers

In the simplest setup we have two framebuffers available. The graphics engine alternates between the two framebuffers. While drawing a frame into a framebuffer, the other framebuffer is transferred to (and shown on) the display.

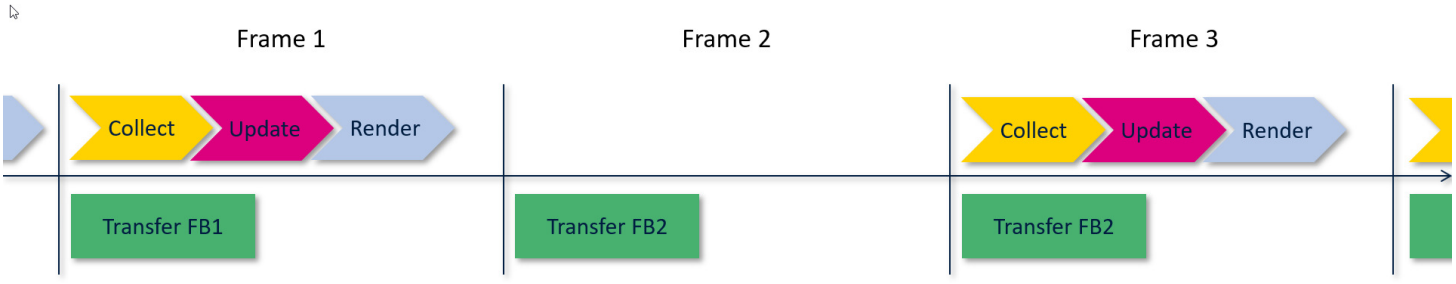


Double framebuffers

In this drawing we assume a parallel RGB display connected to the LTDC controller. This means that a framebuffer must be transmitted to the display in every frame. As we have two framebuffers the graphics engine can draw into one framebuffer while the other framebuffer is being transmitted. This scheme works very well and is preferred if possible.

As the graphics engine is drawing in every frame we also transmit a new framebuffer in all frames in the above drawing.

There will often be frames where the application is not updating anything. This implies that nothing is rendered. The same framebuffer is therefore transmitted again in the succeeding frame.

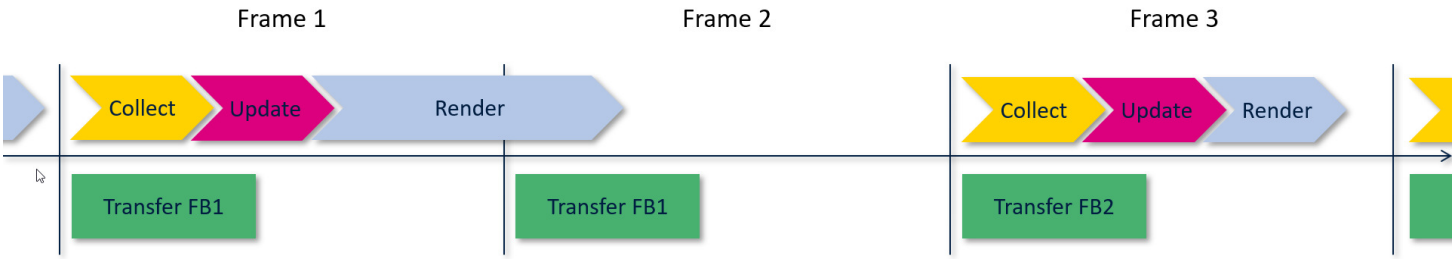


No update in frame 2

The application is not drawing anything in frame 2, so the graphics engine retransmits framebuffer 2 again in frame 3.

The typical parallel RGB display has a refresh rate around 60 Hz. This update frequence must be maintained by the microcontroller. This update frequency means that we have 16 ms to render a new

frame before the transmit begins again. In some cases the time to render a new frame is more than 16 ms. In this case the graphics engine just retransmit the same frame again (as before):



Long render time

The rendering of frame 1 takes more than 16 ms, so the frame 0 previously rendered to framebuffer 1 is retransmitted. The new frame in framebuffer 2 is transmitted in frame 3. When two framebuffers are available, the rendering time can be very long. The previous frame is retransmitted until the new frame is available.

One framebuffer

In some systems there is only memory for one framebuffer. If we have a parallel RGB display we are thus forced to transmit framebuffer 1 in every frame.

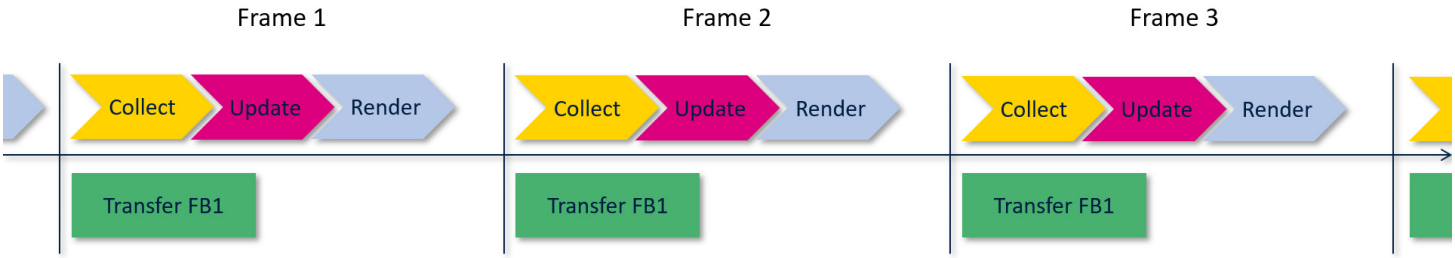
This can be problematic because the graphics engine is forced to draw into the same framebuffer that we are also transmitting to the display at the same time. If this is done without care there is a high risk that the display shows a frame that is a mix of the previous frame and the new.

One solution is to hold back the drawing until the transfer is complete and only draw in timeslot before the transfer starts again. This yields little time to draw as the transfer takes up a significant part of the overall frame time. Another drawback is that incomplete frames (tearing) might still occur if the drawing is not complete when the next transfer starts.

A more potential solution is to keep track of how much of the framebuffer is already transmitted and then limit the rendering to the appropriate part of the framebuffer. As the transfer progresses more and more of the framebuffer is available for the rendering algorithms.

The graphics engine contains algorithms that help the programmer to ensure that the drawing is performed correctly.

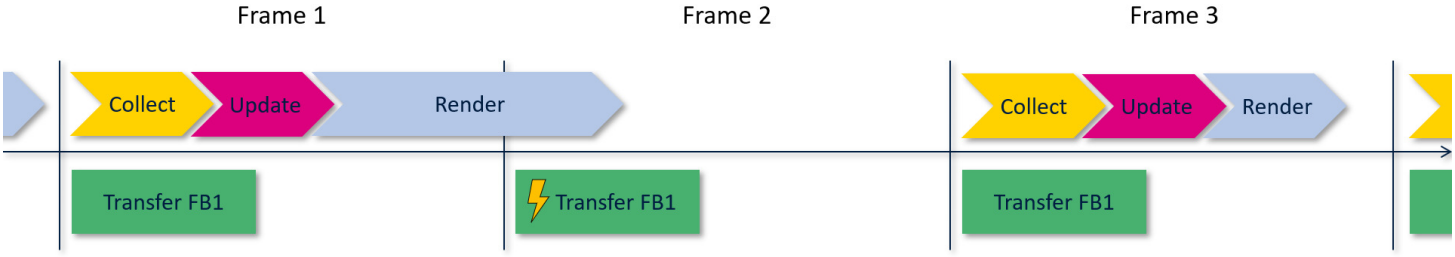
The application updates and renders the framebuffer in every frame:



A single framebuffer is retransmitted in every frame

The framebuffer is retransmitted unchanged if nothing is updated in a frame.

If the rendering time is longer than 16 ms the rendering has not finished when the retransmission starts again:



Long render time

In this situation the graphics engine must make sure that the part that is being transmitted is rendered completely. Otherwise the display will show the unfinished framebuffer.

In the next section we will discuss the rendering time for the individual Widgets. This will help the programmer to write applications of high performance.

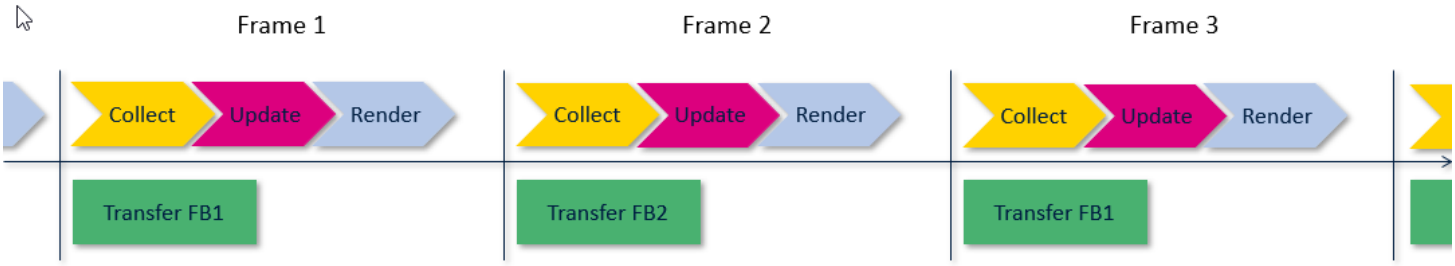
Version: 4.16

Performance

In this section we will discuss performance aspects of an embedded graphical user interface.

A high performance is here defined as getting a high frame rate while still obtaining the desired graphical effects and animations.

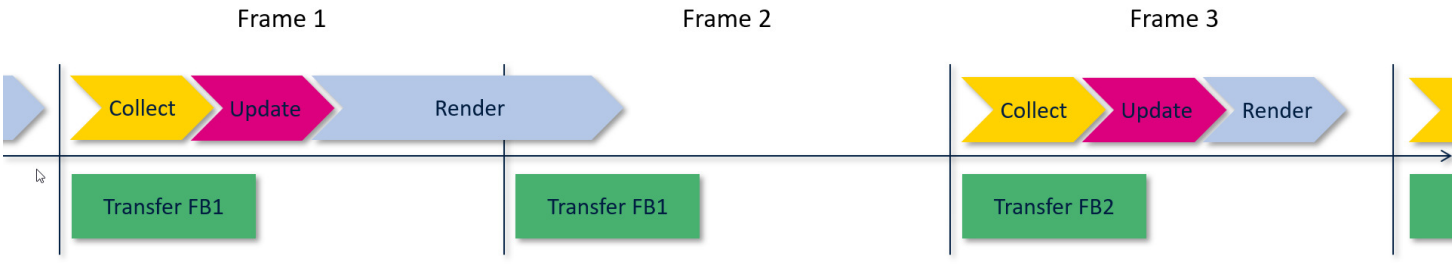
Let's recall from the previous section how the frame rate of the user interface is affected by the main loop. Assume again that there is a parallel RGB display attached to the LTDC and two framebuffers. The basic situation is illustrated below:



Double framebuffers

As the display is assumed refreshed 60 times each second there is approximately 16 ms between each refresh. The calculation is this: 1 s / 60 = 0.01667 s = 16.67 ms.

TouchGFX starts drawing frame 1 into framebuffer 2 at the time where the transfer of framebuffer 1 has started. If the rendering of frame 1 is finished before the next transfer starts we can transfer framebuffer 2. If not finished within 16.67 ms framebuffer 1 is transferred again and the display will appear unchanged:



Main loop time above 16.67 ms

This situation is denoted a lost frame.

The time for the collect and update phases are typically minuscule, e.g. less than 1 ms, and therefore more or less neglectable when considering the overall time taken of the main loop. Therefore, in the following and in general, when considering render time, this includes the collect and update phases.

If the rendering time in many frames exceeds the 16.67 ms time limit the frame rate on the display will be 30 frames per second (fps).

If the rendering generally is shorter than 16.67 ms, but in some frames longer than 16.67 ms, the frame rate may be close to 60 fps in average, but the animation may not appear fluent to the user. Depending on the application it can look like some steps in the animation are fast and some are slow. This is not desireable.

The rendering time can also be even longer. If it is just above 33 ms, the framerate will drop to 20 fps as we only have a new frame ready on every third transfer.

FPS	Max rendering time
60	16.67 ms
30	33.34 ms
20	50.00 ms
15	66.67 ms

The table shows the maximum rendering time (including the collect and update phases) that is available for a given framerate.

To achieve a good performance of a user interface it can be very beneficial to check and monitor the frame rate regularly. Two approaches can be used:

- Measure the rendering time
- Count the lost frames

Measuring the Rendering Time

The first approach of measuring the rendering time gives the most detailed information. The idea is basically to measure the time from the frame transfer to the end of the rendering phase. The graphics engine calls a function on the GPIO class when the collect phase starts and makes another call when the rendering phase ends. The application defines these function and can hook into them to peform measurements.

The measurements can be done in two ways:

- Use external timing device like an oscilloscope: To measure using an oscilloscope, the application should implement the `set(GPIO_ID)` and `clear(GPIO_ID)` methods from the `GPIO` interface.

The oscilloscope can then measure the rendering time as the time elapsed while the output is high.

- Use an internal timer: Another approach is to use an internal timer, like the sysTick timer. When the GPIO::set(RENDER_TIME) is called the application can save the value of the timer in a variable. When the clear call is made the application can read the timer again and subtract the previous value to get the render time. The speed of the timer will define the resolution of the measurement. The application must somehow make the render time visible. One way is to save the value in a global variable and maybe show the value on the screen in a TextArea. The value can also be checked with a debugger.

Counting the Lost Frames

The graphics engine counts the number of transfers that has occured during the last collect-update-rendering phase. The application can easily check this value to see if a frame was lost and the frame rate therefore lowered.

The count is available in the *HAL* class:

```
void handleTickEvent() {
  tickCounter += 1;
  if (HAL::getInstance()->getLCDRefreshCount() > 1) {
    //Alert programmer somehow
    ...
  }
}
```

Compensating for Lost Frames

When frames are lost and the framerate of one of our animations therefore lowered we can compensate to a certain degree. We can either:

- wait it out - let the animation go on, resulting in a longer animation duration, and possibly unsmooth animation.
- skip some frames - make sure that the overall animation does not take longer time than intended by skipping frames.

TouchGFX can be instructed to automaticallly skip some frames, when frames are lost. This can be accomplished by ticking animations more than once per actual frame. This can help in making animations more fluent when the render time is uneven.

HAL.hpp

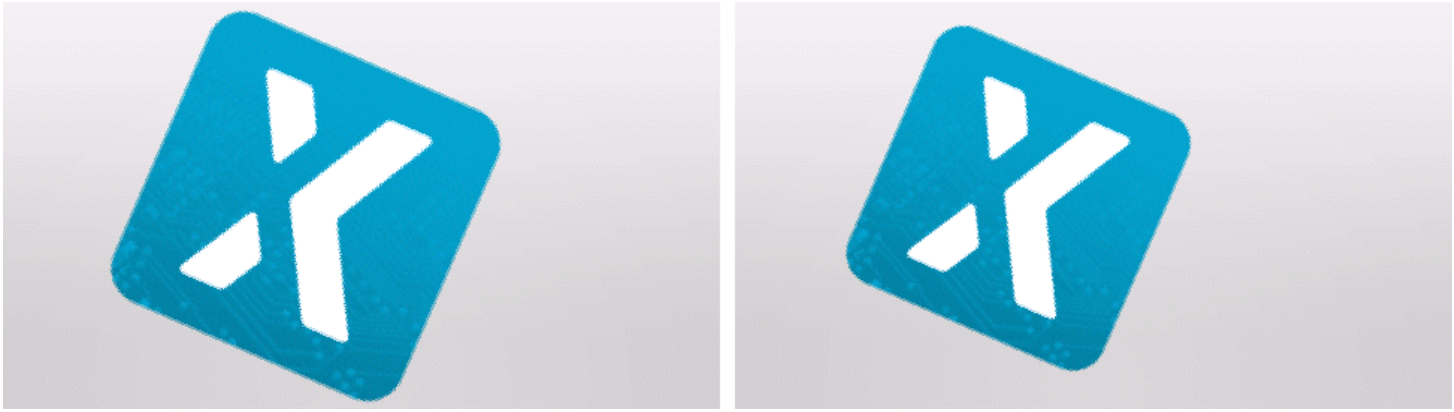
```
void setFrameRateCompensation(bool enabled)
```

What Affects the Rendering Time?

A number of different things affect the rendering time: The size of the updated parts, the use of layering, the complexity of the widgets, and the available hardware support for the rendering.

How Much of the Screen Is Updated?

The rendering time is generally proportional to the number of pixels that must be updated. So if an animation takes too long time to render, a possible fix is to reduce the area of the animation. For example, if you have a rotating image and the performance is not good enough, the performance can be improved by reducing the size of the image.



Reducing image size reduces the rendering time

Remember that the graphics engine redraws the areas that the application invalidated. This means that it is important to only invalidate the areas that actually requires a refresh.

The larger the invalidated areas, the longer the render time.

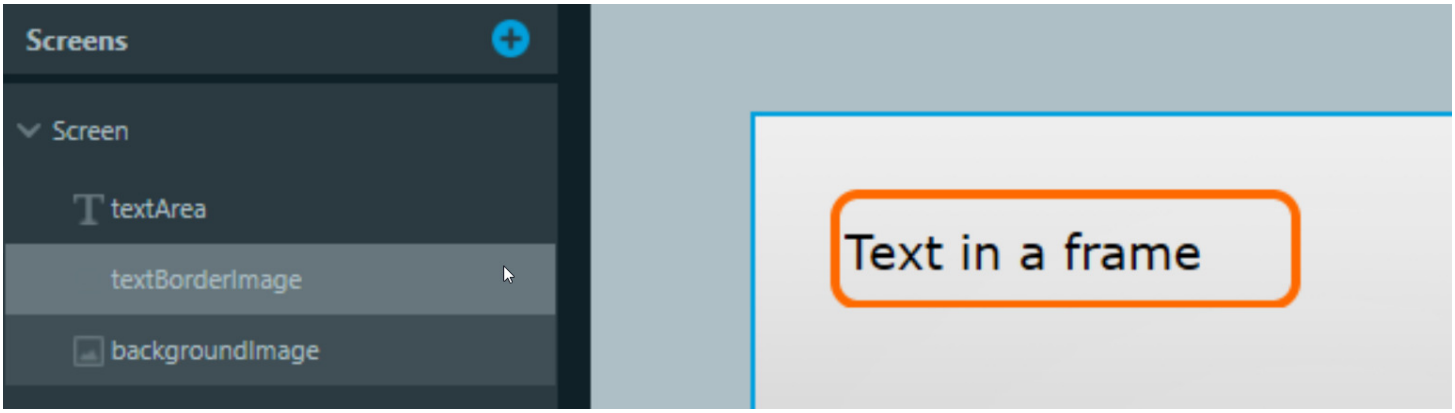
The Number of Layers in the Graphics

In a typical application the graphics will consist of different elements that are stacked upon each other. If one of the elements is updated all the elements must typically be redrawn.

A typical example of this is a background image, a frame, and some text:

Layering graphical elements

This user interface is created by putting a TextArea widget on top of an Image widget showing a transparent frame. Both on top of the background Image:



Layering graphical elements in TouchGFX Designer

This solution is used very often in application. It is a very easy solution with a high flexibility, as it is, for example, possible to change the frame at runtime or move the frame and the text on the background.

The problem regarding rendering time is that if the text is updated at runtime and needs to be redrawn, the graphics engine also needs to redraw the background and the frame; and then the new text. This increases the time to render the text considerably.

The more layers in an invalidated area, the longer the render time.

The Complexity of Rendering the Pixels

Not all pixels are equally difficult to render to the framebuffer. In all types of rendering the graphical engine must write the resulting pixel to the framebuffer. But the cost of calculating the pixel to write differs.

The fixed color, e.g. used in the Box Widget, has the lowest cost, as the calculation of a pixel is done once and reused for all the pixels. This means that we can get a very high performance by using a lot of Boxes. This is not recommended as the user interface will not be of high quality.

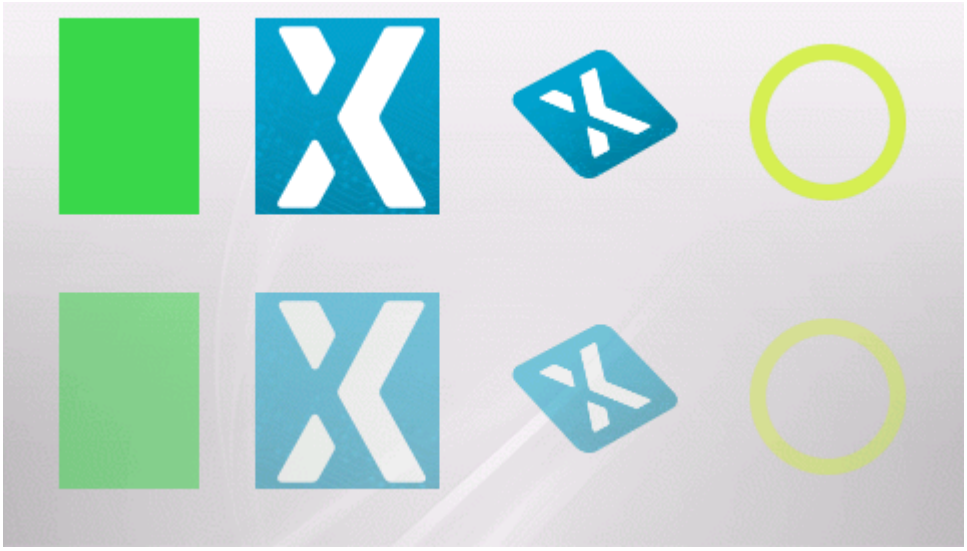
An image has the next lowest cost of pixel calculation since the pixels are stored in a ready to use format in the bitmaps. Calculating the pixel to write to the framebuffer is a matter of loading the color value from the correct place in the bitmap.

Text bears a cost comparable to images as the individual letters are actually represented as small images. In practice the cost is higher as the high number of small images gives rise to a considerable "start-stop" cost. For example the calculation of the position of the individual letters. In order for text to look as nice as possible, it is represented as small images with transparency, see the notes on transparency below.

Rotated or scaled images are more expensive. The task is again to load the pixel value from the bitmap, but this time it is a more time consuming calculation because the graphics engine has to incorporate the scaling and rotation.

Geometric elements like a circle are even more expensive. This time we cannot load the pixel color from a bitmap, but we have to calculate both the shape of the circle and the color of individual pixels in the circle.

Transparency adds to the cost of drawing an element. An element is transparent if some of the pixels are not solid. This increases the cost of drawing as the graphical engine first has to draw the element behind the transparent element (as we saw in the "text in a frame" section). Secondly the graphical engine then has to combine the background pixel with the pixel of the transparent element and write the result to the framebuffer. This calculation takes considerably more time than just writing the calculated pixel.



Box, Image, rotated Image, and circle. Solid elements in the first row. Transparent elements below.

Transparency always gives you an extra layer. But putting solid pixels on top of other solid pixels does not always increase the number of layers. The graphical engine tries to not draw pixels that are covered by other solid pixels, as this would be a waste of precious time.

The more expensive elements in the invalidated area, the longer the render time.

Remember that is only the elements that are part of the invalidated area, that adds to the rendering time. Elements outside of the invalidated areas do not impact the rendering time.

Read more about ui components and performance here.

Hardware Support for Rendering

Some STM32 microcontrollers contain a graphical accelerator called Chrom-ART (or DMA2D). This accelerator can reduce the rendering time. As the accelerator runs in parallel with the microcontroller core, the microcontroller is free to run other tasks while the accelerator renders graphics.

Chrom-ART is mainly useful for images and text. It is automatically used by the graphics engine when available.

When Should You Consider Rendering Time

Rendering time is not equally important all the time. You should pay attention to the rendering time when a slow frame rate is visible to the user. This is typically the case when you have an animation running on part of the screen (like a rotating icon) or when you move or slide something across the screen. If the update frequency is low it will appear step-wise instead of fluent to the user. If this is the case you should check the rendering time.

On the other hand, if you replace the whole screen with a new screen, it is normally not visible to the user if the frame rate dropped significantly during the change. This is because the user cannot see when the rendering started, but only when it finished.

These two rules mean that for animated elements (e.g. moving) you should use few layers and refrain from using complex elements and many layers. For other parts of the screen, this could be unproblematic.

Analog clock and a scroll list

In this example we have an analog clock on the left. The three clock hands are rendered by rotating small elongated images. This is normally fine as the hands do not move all the time. But if we wanted to move the clock around on the screen, they would be redrawn in every frame and that could be problematic, as drawing rotated images is typically time consuming.

On the right we have a scroll list. The user can move this list of numbers up and down, so we need a high framerate for the user interface to appear responsive. Therefore it is important that we consider the rendering time of the elements in the scroll list or reduce the size of the scroll list.

Tips To Get Good Performance

We end this section with a summary of the tips to obtain a good performance:

- **Do not redraw unchanged things** Make sure that you do not accidentially invalidate unnecessary parts of the display. This reduces the performance without any benifit.
- **Find balance between quality and speed** Reducing the complexity of the elements can improve the performance. A good balance between this and the performance is often the key.
- **Utilize hardware capabilities** The capability of a microcontroller with hardware acceleration (Chrom-ART) is often higher than a microcontroller without. Consider using a microcontroller with Chrom-ART.
- **Replace calculated graphics with images** The calculated circle is slower than an image of a circle. In general images can replace many static elements.
- **Adjust display refresh rate** As we discussed in the beginning of this section is the refresh rate a hard limit for the rendering time. If the rendering time exceeds the refresh rate, the frame rate drops. If your rendering time is just a little above the refresh rate, it may be possible to lower the refresh rate of the display to e.g. 55 Hz (corresponding to 18.2 ms), and keep the high frame rate.

Version: 4.16

Operating Systems

Introduction

In this section we will discuss the use of an operating system in graphical user interface applications.

Embedded devices are becoming more and more advanced. The majority of the systems are not only handling the graphical user interface, but often also complex control algorithms and tasks.

These tasks can for example be motor control, data aquisition, or security related tasks. Many modern devices contain communication protocol stacks like TCP/IP, for communication with data centers; or radio stacks like Bluetooth for communication with other local devices.

Interleaving other tasks with the user interface

In a simple device with the graphical user interface and only a few simple support tasks, like an egg timer, it is possible to structure the whole application around the user interface code. The application does very little besides the regular user interface updates, so the execution of the other tasks can with fair success be embedded into the user interface code.

As soon as the device contains more advanced functionality that "runs in the background" with separate timing requirements like regulating a motor, it quickly becomes difficult to integrate the two tasks in one while supporting the requirements.

As we discussed in the previous articles the graphics engine must keep drawing new frames to support a fluent user interface. If the graphics engine pauses this while running other tasks, the frame rate will decrease. Likewise, if the other tasks only run between the frames, in the idle time, then these tasks will suffer when the user interface is rendering complex scenes where there is less idle time. These effects makes it difficult to manually interleave the ui task with other complex tasks.

An example

Assume for the rest of this section that we are building a bluetooth speaker with a display. We have 3 major tasks: run the graphical user interface, feed music to the speaker, and handle the bluetooth stack for communication with other devices.

It is not difficult to see that an application architecture centered on the user interface is not good: Imagine e.g. that we blend the music code with the user interface and put the code for starting

playback in the eventhandler for a button in the user interface. Now the user interface is locked for the time it takes to start the music. Any animation running will be stopped meanwhile.

In general, the responsiveness of the user interface becomes dependant on the execution time of the music tasks (start, stop, next, etc.). This is a general problem, that we will come back to.

And what happens if we also want to be able to start music from Bluetooth? Should the user interface somehow be involved in that?

And how do we give priority to the music tasks, so that the music is without pauses? At the same time we also want the user interface to run with the highest performance when there is no music tasks to run.

All this can be solved by using an operating system with tasks, communication means, and synchronization.

RTOS

A real-time operating system is a small piece of software that supports applications with various services and distributes computing resources to the tasks in the application.

Using an RTOS allows you to structure your application in a number of independent, but coorporating tasks. These tasks are then executed concurrently by the RTOS when they have work to do and according to their priority.

We can even split a job into a high priority and a low priority task. Assume that we have to read bluetooth data from a buffer very fast when it arrives, and put it into a larger application buffer. The handling of the data can be postponed a little. This way we end up with two bluetooth tasks.

For our example we will start 4 tasks from main:

```
int main() {
  ...
  os_start_task(gui_task,      medium_priority);
  os_start_task(music_task,    low_priority);
  os_start_task(bt_comm_task,  high_priority);
  os_start_task(bt_appl_task,  low_priority);
  os_start_scheduler();
}
```

A similar split can be done with the music task: A high priority task to feed data to the speaker, and a low priority task to control what song is playing and sending notifications to the user interface.

The result using different priorities as above is that the bt_comm_task is running when there is data to handle and the user interface task runs otherwise. When the user interface task is waiting for the display, the two low priority task can run. The operating system scheduler will handle this time distribution for us.

In a typical TouchGFX application the user interface is waiting for the display in every frame, and it is also regularly waiting for the graphics accelerator, ChromArt, to finish drawing elements. This means that there will be many small pauses where the lower priority task can run. The operating system scheduler will automatically change the MCU to run these tasks when the higher priority tasks are waiting.

Task communication

When we use multiple tasks we also need a safe way of communicating between the tasks. One simple case is from the user interfaced to the music task. Here we need, among other cases, the music task to wait until the gui_task asks it to start playing a song. A simple way to implement that is to use a message queue. The music task sleeps until there is a message in the queue. The scheduler wakes the task when there is a message in the queue and when the higher priority tasks are not busy.

```
    ...
    music_task_input_queue = os_create_queue(10); //10 element queue
    ...
```

In the user interface, when "Play" is pressed, we send a message to the music task's queue:

```
void ScreenMusic::handlePlayPressed()
{
    os_send_message(music_task_input_queue, play_message);
}
```

The music task can wait for a message by reading the queue. This will block the task until a message arrives:

```
...
Message message;
os_receive_message(music_task_input_queue, &message);
```

After putting the message into the queue of the music task, the user interface is continuing to run and rendering the frame as fast as possible. We are not wasting time on handling the play message immediately. But, when the rendering is done and the ui task is waiting before rendering the next

frame, the scheduler will change the execution to the music task, which will handle the incoming messages.

Similary we can also give the user interface an input queue. The music task can then send a notification message e.g. when the song has ended. The user interface task should not wait for a message, but quickly check if a message is available without blocking, and read it in case.

This setup gives a very loose connection between the tasks in the system. We can actually test the music task without using the user interface, and we can also easily start music from the bluetooth task.

Handling interrupts

Some tasks needs to run as a response to an interrupts. In our example the bluetooth communication task is such an example. We want that task to run when the bluetooth chip has a new package for us. Assuming that we can get an interrupt in that case, we can send a message from the interrupt handler:

```
void BT_DataAvailable_Handler(void)
{
  os_send_message(bt_data_queue, data_available_message);
}
```

Other synchronization primitives than queues are also available. Semaphores and mutexes for example are found in many operating systems.

FreeRTOS

TouchGFX is tested with the FreeRTOS operating system during development. TouchGFX has very little requirements and can run on many other operating systems, but FreeRTOS is a good starting point unless you have some specific requirements.

FreeRTOS is a simple operating system that is free to use in commercial application. It is supplied in source code with the STM32 Cube firmware with ready to use examples for all STM32 microcontrollers.

See freertos.org for further information and license terms for FreeRTOS.

TouchGFX OS Wrappers

TouchGFX in its default configuration runs on FreeRTOS and uses a single message queue to synchronize with the display controller and a semaphore to guard the access to the framebuffer.

This is handled by the OSWrappers class defined in `touchgfx/os/OSWrappers.cpp`. This class has the following methods:

Method	Description
signalVSync()	This method should be called from the display driver when the display is ready for the next frame.
waitForVSync()	Called by the graphics engine to wait. Should not return until signalVSync is called.
isVSyncAvailable()	(Optional)Returns true if VSync has occured. Can be used to avoid blocking in the waitForVSync.
signalRenderingDone()	(Optional)Remove any outstanding VSync signals.
takeFrameBufferSemaphore()	Called by the graphics engine and the accelerator to gain direct access to the framebuffer
giveFrameBufferSemaphore()	Called to release the direct access again.

The default implementation uses a message queue to implement the VSync (frame) synchronization. The graphics engine task is sleeping until the next VSync arrives.

This OSWrapper class is generated by the TouchGFX Generator. Read more about the Generator here.

No RTOS

TouchGFX can also run without an operating system. In this case you must start the graphics enging main loop directly in your main:

```
int main()
{
    ...
    touchgfx::HAL::getInstance()->taskEntry();

    //never returns
}
```

Not using an RTOS does not lower the performance of TouchGFX. It may increase the MCU load and it will make it more difficult to run other tasks together with TouchGFX.

As described above you now need to drive any other task manually while the user interface is running in your main.

Model::tick

One way is to perform a task check in the Model class once in every frame:

Model.cpp

```
void Model::tick()
{
   //run other tasks here
   music_task_tick();
   bluetooth_task_tick();
}
```

Using this method all tasks will be executed once in every frame. The time consumed by the tasks will be added to the rendering time of the user interface. This is a simple and acceptable solution for simple systems, where all tasks can terminate quickly.

OSWrappers

Another method is to use the hooks in the OSWrappers class. As explained above the graphics engine calls method on this class when it needs to wait for events. You can use this to do other work while waiting for said events:

OSWrappers.cpp

```
static volatile uint8_t vsync_sem = 0;

void OSWrappers::signalVSync()
{
    vsync_sem = 1;
}

void OSWrappers::waitForVSync()
{
    vsync_sem = 0; //clear the flag, so we wait for the next vsync
    do {
        // Perform other work while waiting
        music_task_tick();
        bluetooth_task_tick();
    } while(!vsync_sem);
}
```

Using this method the idle task between the frame can be fully used by the other tasks, but the amount of time the tasks get will vary.

Another solution is to use the OSWrappers::isVSyncAvailable and OSWrappers::signalRenderingDone functions. This will allow the application to avoid having multiple while-loops. These functions are used by the TouchGFXGenerator when a No-operating-system configuration is selected.

It is important that the tasks can divide their work in to small steps of maybe 1 millisecond. Otherwise it will hurt the user interface performance.

Version: 4.16

Memory Usage

Introduction

In this section we will discuss the memory usage of a TouchGFX application. A typical TouchGFX application uses 4 types of memory, but this will depend on the hardware used:

Memory Type	Usage
Internal RAM	Internal RAM is used for configuration data like coordinates and colors of all the Widgets. A few objects for the current screen is allocated here. The operating system memory including the runtime stack of the UI task is also in internal RAM. All data for other software components like filesystems and display drivers is also placed in internal RAM.
Internal Flash	Internal flash is used for program code for the application, the TouchGFX library, and other libraries used.
External RAM	External RAM is typically used for framebuffers and maybe a bitmap cache.
External Flash	External flash is used to store images, fonts and texts.

Static Memory Allocation

TouchGFX only uses static memory allocation. This means that all memory is preallocated. No memory is allocated by TouchGFX at runtime. This ensures that you will never run out of memory, if the application could fit into memory at start.

Screens and Widgets

In TouchGFX the user interface is created by developing a number of C++ classes. The classes are created by TouchGFX Designer when you design the screens. For each screen designed in TouchGFX Designer you automatically get a number of classes (the MVP architecture).
When you show a screen on the display objects of the classes is automatically allocated by TouchGFX in internal RAM.

When you change from one screen to another screen, the objects allocated for the previous screen are not used anymore, only the objects for the new screen. Therefore the new objects are allocated in the

place in internal RAM where old objects were allocated (the old objects are overwritten). The internal RAM only holds objects for one screen at one point in time.

Based on the classes defined it is possible for the C++ compiler to calculate the size of the largest screen classes, and reserve memory for those classes.

The memory usage in internal RAM thus does not depend on the number of screens in the application, but on the size of the largest screen.

The memory set aside for these objects is called the FrontendHeap.

TouchGFX

Application code

The application code is normally placed in the internal flash. The application code consists of the program code you write, the code generated by TouchGFX Designer, code from the TouchGFX library and other libraries you use.

The amount of application code will of course increase when you write more code and add more screens to your application. The amount of code taken from libraries increases the first time you use a feature. For example, the first time you add a Button to a screen, the Button code from the TouchGFX library is included in your application which therefore grows. The second time you add a Button to the same or another screen, no additional code is taken from the TouchGFX library, and the application only grows by the amount of code you write or TouchGFXDesigner generates.

Assets

Assets like images, texts, and fonts are converted to c++ files and linked into your application. The data for the assets are normally put in the external flash, but can also be put in internal flash. This is controlled by the linker script.

When you add an image, the application size will grow proportionally to the size of the image.

When you add texts the application will grow two bytes for each character in the text. If you use the same string of character twice it is only included once.

Only the characters used by the application are taken from the font files. This means that if you only use the upper case letters A-Z in your application, the lower case letters a-z in the font are not included in your application. If you later add texts that use these letters, the font data in your application will grow in size.

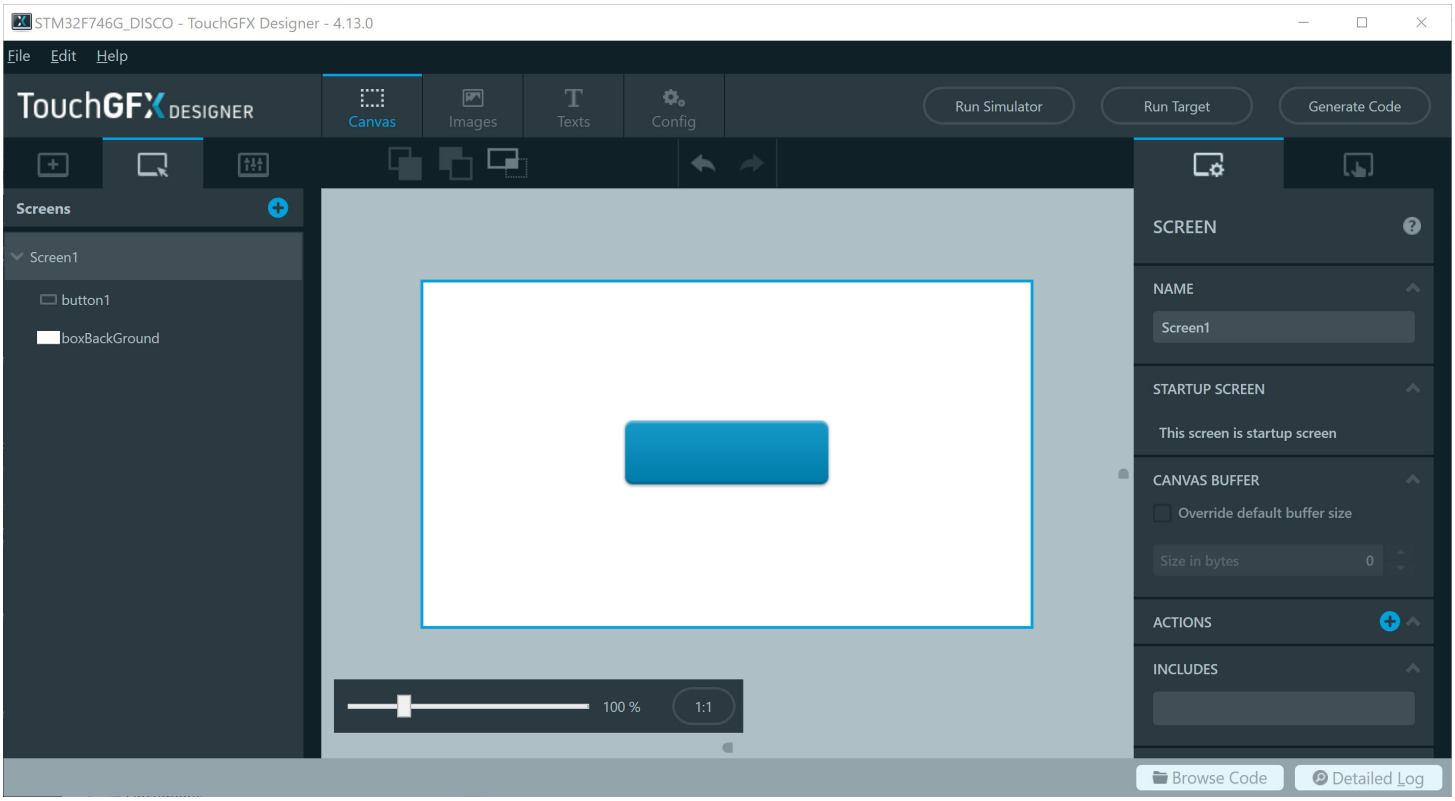
The size of the characters in flash depends on the selected font size. In you increase the font size, the application size will increase.

Checking memory usage

Memory usage of a specific application can be found by examining the map file generated by the linker.

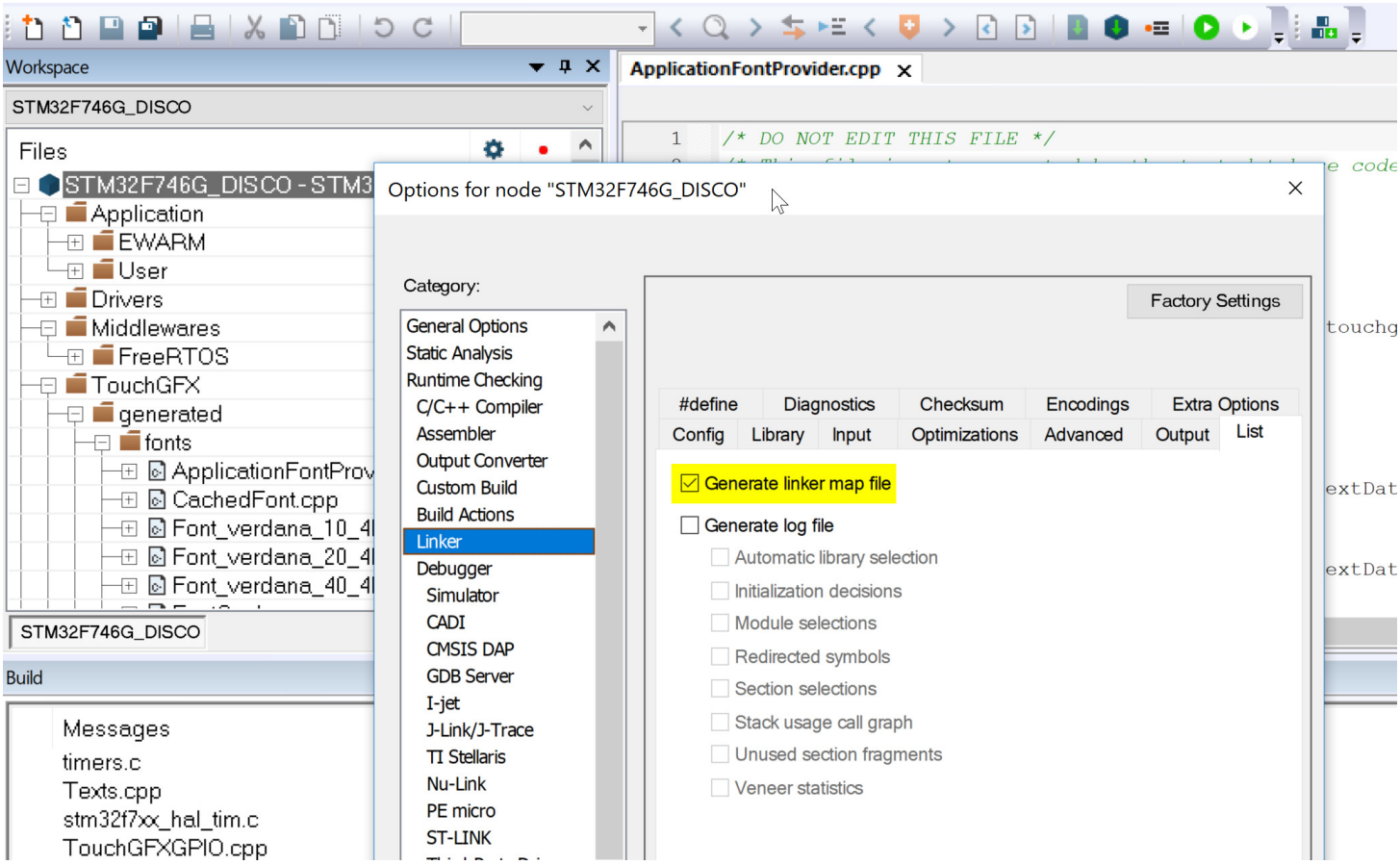
Here we examine a map file generated by the IAR Embedded Workbench. Other compilers produce similar map files.

We start by creating an empty project in TouchGFX Designer for the STM32F746Discovery evaluation kit:



STM32F746 project with a Box and a Button

After opening the project in IAR, we check in the properties that IAR generates a .MAP file:



Generate a linker map file

After compilation in IAR we can check the linker map file, STM32F746G_DISCO.map, found in the *EWARM/STM32F746G_DISCO/List* folder.

The IAR linker map files contain a nice summary. Look for *MODULE SUMMARY*:

```
******************************************************************************
*** MODULE SUMMARY
***

    Module                                  ro code  ro data  rw data
    ------                                  -------  -------  -------
command line/config:
    ----------------------------------------------------------------
    Total:

C:\TouchGFXProjectsDocumentation\STM32F746MemoryUsage\EWARM\STM32F746G_DISCO\Obj: [1]
    ApplicationFontProvider.o                                    20
    BitmapDatabase.o                             12              40
    Blue_Buttons_Round_Edge_small.o                          40'800
    Blue_Buttons_Round_Edge_small_pressed.o                  40'800
    Font_verdana_10_4bpp_0.o                                     24
    Font_verdana_20_4bpp_0.o                                     72
    Font_verdana_40_4bpp_0.o                                    280
    FrontendApplication.o                                46      60
    FrontendApplicationBase.o                           706     816
    GeneratedFont.o                                      84      84
```

```
    Kerning_verdana_10_4bpp.o                                4
    Kerning_verdana_20_4bpp.o                                4
    Kerning_verdana_40_4bpp.o                                4
    Model.o                                 10
    OSWrappers.o                           156        1        9
    STM32DMA.o                             898      176
    STM32TouchController.o                 162       24        4
...
    heap_4.o                               444             32'792
...
touchgfx_core.a: [7]
    AbstractButton.o                       136
    AbstractPartition.o                      8
    Application.o                        2'218      290       28
    Bitmap.o                             1'064      604       36
    Box.o                                  108      104
    Button.o                               276      308
    ConstFont.o                             62
    Container.o                            510      396
    DMA.o                                  558      252
    DisplayTransformation.o                192
    Drawable.o                             418
    FontManager.o                           12                4
    Gestures.o                             364       60
    HAL.o                                1'758      544       18
    LCD24bpp.o                           2'732    1'604       80
    Screen.o                             1'924      124
    TouchCalibration.o                     252               76
    TypedText.o                                              14
    ------------------------------------------------------------
    Total:                              12'728    4'286      256

    Gaps                                                4        3
    Linker created                                     36    2'560
----------------------------------------------------------------------
    Grand Total:                        38'676   88'973   42'731
```

This table has three columns of numbers. *ro code* and *ro data* is read-only and is placed in flash. *rw data* is non-const read-write variables and which are placed in RAM.

The rows in the table are divided into 7 blocks. The first block is all the .cpp files in the project. The next six blocks are the libraries used in the project (.a files). The last one is the TouchGFX library.

We can see that the TouchGFX library (the "touchgfx_core.a: [7]" section) adds 12.728 bytes of code to the application (and 4.286 bytes of constant data).

Internal RAM

To find the total internal RAM usage we look in the *Grand Total* row in the bottom of the Module Summary table. The third column is the internal RAM. This means that the project uses 42.731 bytes of internal RAM. Looking at the total for the TouchGFX library we see that 256 bytes are used by the TouchGFX library [7]. 32.792 bytes are used by heap_4.o. This is the dynamic memory heap reserved for FREERTOS. 32Kb is the default value, but the heap size can be configured in CubeMX. A typical TouchGFX program uses a few Kb from this heap, mainly to allocate a stack for the user interface task.

By searching for the FrontendHeap, we can find the size of the screen objects:

```
FrontendHeap::getInstance()::instance
                        0x2000'95d0   0x240  Data  Gb  TouchGFXConfiguration.o [1]
```

The objects required for the user interface occupies 0x240 bytes = 576 bytes.

Internal Flash

We see from the *Grand Total* row that this application uses 38.676 bytes code + 88.973 bytes data. Only some of this is the internal flash. At least the two images for the Button is in external flash.

To find out how much code and data that is going into the internal flash we start by checking the *PLACEMENT SUMMARY* (a few details removed):

```
*******************************************************************************
*** PLACEMENT SUMMARY
***

"A0":  place at address 0x800'0000 { ro section .intvec };
"P1":  place in [from 0x800'0000 to 0x80f'ffff] { ro };
"P2":  place in [from 0x2000'0000 to 0x2004'ffff] { rw };
"P3":  place in [from 0x9000'0000 to 0x90ff'ffff] {
          section ExtFlashSection, section FontFlashSection,
          section TextFlashSection };
```

The internal flash is starting at address 0x08000000. It is covered by the two regions "A0" and "P1".

Looking a bit further in the map file we can see what is placed in these regions:

```
  Section               Kind         Address      Size  Object
  -------               ----         -------      ----  ------
"A0":                                            0x1c8
  .intvec               ro code   0x800'0000     0x1c8  startup_stm32f746xx.o [1]
                                - 0x800'01c8     0x1c8

"P1":                                            0xb05d
```

```
  .text                 ro code   0x800'01c8     0x9b8  main.o [1]
  .text                 ro code   0x800'0b80      0x14  memset.o [5]
...
  .text                 ro code   0x800'b17a       0x2  AbstractButton.o [7]
  .rodata               const     0x800'b17c       0x1  unwind_debug.o [6]
  .rodata               const     0x800'b17d       0x0  zero_init3.o [5]
  .rodata               const     0x800'b17d       0x0  lz77_init_single.o [5]
  Initializer bytes     const     0x800'b17d      0xa8  <for P2-1>
                              - 0x800'b225    0xb05d
```

This means that 0x1c8 bytes = 456 bytes are used by "A0", and 0xb05d bytes = 45.149 bytes by "P1". The total usage of the internal flash is thus 45.605 bytes.

External Flash

The external flash is the "P3" region (starting at address 0x90000000). Here is the content of that region:

```
"P3":                                       0x1'4076
  ExtFlashSection       const    0x9000'0000    0x9f60  Blue_Buttons_Round_Edge_small.o
  ExtFlashSection       const    0x9000'9f60    0x9f60  Blue_Buttons_Round_Edge_small_pre
  FontFlashSection      const    0x9001'3ec0     0x118  Font_verdana_40_4bpp_0.o [1]
  FontFlashSection      const    0x9001'3fd8      0x48  Font_verdana_20_4bpp_0.o [1]
  FontFlashSection      const    0x9001'4020      0x18  Font_verdana_10_4bpp_0.o [1]
  FontFlashSection      const    0x9001'4038      0x10  Table_verdana_10_4bpp.o [1]
  FontFlashSection      const    0x9001'4048      0x10  Table_verdana_20_4bpp.o [1]
  FontFlashSection      const    0x9001'4058      0x10  Table_verdana_40_4bpp.o [1]
  FontFlashSection      const    0x9001'4068       0x4  Kerning_verdana_10_4bpp.o [1]
  FontFlashSection      const    0x9001'406c       0x4  Kerning_verdana_20_4bpp.o [1]
  FontFlashSection      const    0x9001'4070       0x4  Kerning_verdana_40_4bpp.o [1]
  TextFlashSection      const    0x9001'4074       0x2  Texts.o [1]
                              - 0x9001'4076  0x1'4076
```

We see that the total usage of the external flash is 0x14076 bytes = 82.038 bytes. The majority of that is used by the two images for the Button (two times 0x9f60 bytes = 40.800 bytes). The rest of the data is for 3 fonts. They don't use much space in this example as they only contain the '?' character, because we do not use any texts in this example.

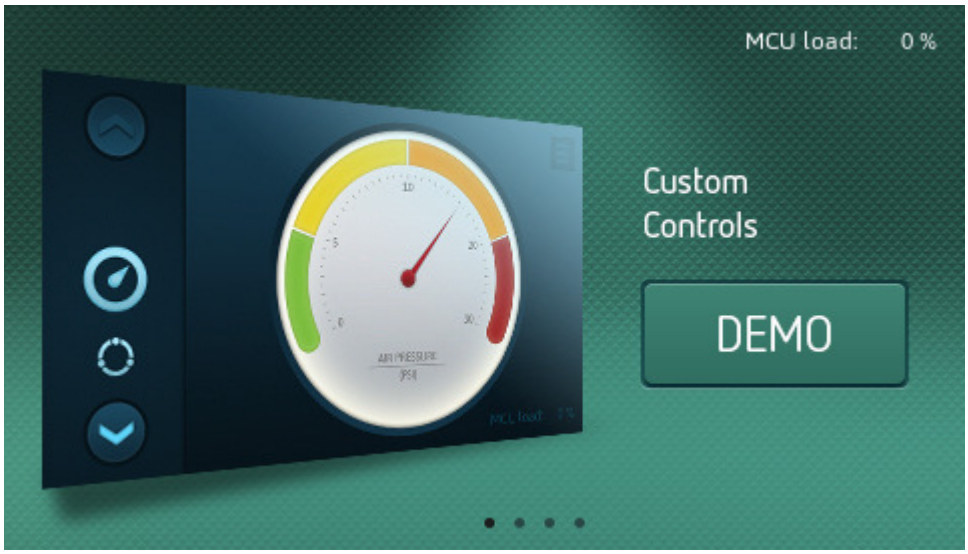
Summary

The only thing placed in external RAM is the framebuffers. These cannot be found in the linker script as they are not defined as variables in the application. The resolution is 480x272 pixels in 24 bit. We have two framebuffers to the total usage is 480 * 272 * 3 * 2 = 786.360 bytes.

Memory Type	Usage
Internal RAM	42.731 bytes
TouchGFX Screen objects	576 bytes
Internal Flash	45605 bytes
TouchGFX Framework	12.728 bytes code
External RAM	786.360 bytes
External Flash	82.028 bytes

Demo 1

To give another example here are the numbers for the TouchGFX Demo1 which can be found in TouchGFX Designer. It contains 5 screens and more than 100 images:



STM32F746 Demo 1

Summary

Memory Type	Usage
Internal RAM	51.387 bytes
TouchGFX Screen objects	10.772 bytes
Internal Flash	187.768 bytes

Memory Type	Usage
TouchGFX Framework Code	85.174 bytes code
External RAM	786.360 bytes
External Flash	5.281.812 bytes

Version: 4.16

Development Introduction

Main Activities

A TouchGFX project involves a set of activities that you will be addressing during the development phase. The effort in each of them are dependent on what the goal of your project is. If you are doing a UI prototype you can use premade code for major parts of the application and thus speed up your project development by skipping most of these activities. If you are doing a full project based on a custom made board, you will be addressing each of these activities in your project.

A TouchGFX projects main activities

Main Components

Your TouchGFX project is made up of five main software and hardware components. Each of the activities will generate one of the main component for your TouchGFX project. The TouchGFX Engine is not an output of any main activity, this is the starting point for your TouchGFX project and is available when you have downloaded and installed.

A TouchGFX projects main components

The following sections will give an overview of each of the activities and components. Each of the activities are further described in full details in this chapters remaining sections.

Hardware Selection

This activity is the initial activity in your TouchGFX project. Selecting the hardware on which your application will run. Deciding on which hardware components you need and what influences these have on your TouchGFX application. When you are done with this step you have a Display Board available for your TouchGFX project.

Prototyping

If you are doing a UI prototype an STM32 Evaluation Kit will be the perfect choice to get up and running quickly. Here there are no considerations to be done about the hardware components, how to connect to the board or similar issues. In this case the Hardware Selection activity is only a matter of selecting an available STM32 Evaluation kit, which is the best match with your final product in terms of MCU performance, memory setup and display size.

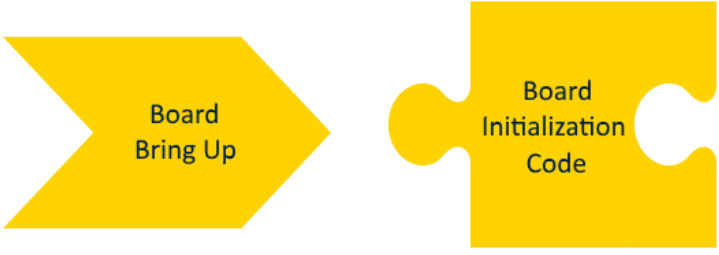
Custom Hardware

If you are creating your own hardware solution there are a lot of choices to be made and issues to consider. The section Hardware Selection will help you in this task, answering a lot of questions and describe how your choices influence a TouchGFX application.

Often in your project you will not have the final hardware ready before late in the project. In this case it is very common to select an STM32 Evaluation Kit that resembles your final board and use this in the first steps of UI Development. If you do not have such board, you can also start out by just using the TouchGFX Simulator that runs on your PC.

A full description of this step can be found in the Hardware Selection section.

Board Bring Up



This activity is a central task to enable TouchGFX to be executed on your board. The output component is called Board Initialization Code which is a general initialization code that setup your MCU and all peripherals, preparing it for application execution. This initialization code is independent of TouchGFX, it is only handling pure hardware setup.

CubeMX

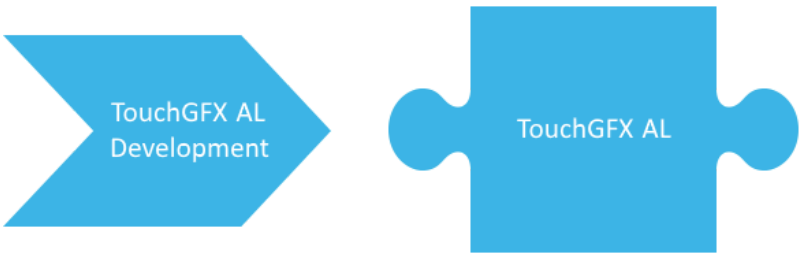
The main tool for this activity is CubeMX. It helps you configuring the MCU and generate general startup code. For peripherals, such as external RAM and Display, you will add initialization code and specific peripheral drivers yourself. It is also possible to do the entire Board Initialization Code without use of CubeMX, but is not recommended unless you have expert knowledge on STM32 and board bring up.

Application Templates (ATs)

If you are doing a UI Prototype or just want to try out TouchGFX you can base your application on an existing Application Template (AT) for one of the standard STM32 Evaluation Kits available in TouchGFX Designer. These include all the Board Initialization Code needed. The ATs are based on a CubeMX configuration, so it is possible for you to modify the configuration if you want to experiment or add access to more peripherals.

A full description of this step can be found in the Board Bring Up section.

TouchGFX AL Development



This activity is key in making the TouchGFX Engine run on top of your fully initialized Display Board (Display Board + Board Initialization Code). The output component is called TouchGFX Abstraction Layer (AL) and is a software layer that is an abstraction of your hardware and enables the TouchGFX Engine to run on your board.

TouchGFX Generator

The main tool in this activity is TouchGFX Generator which is a CubeMX plugin that allows you to configure and generate most of the TouchGFX AL code. You will most probably also write some part of

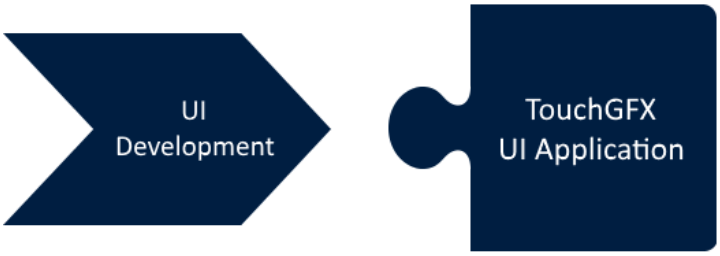
the TouchGFX AL in code by yourself. The TouchGFX Generator will assist you in this step by creating empty functions for you to implement.

It is important to notice that for your TouchGFX AL to work it is important that your Board Initialization Code is done correctly and that the MCU, External RAM, Display and so on is configured correctly.

If you are doing a UI Prototype or just want to try out TouchGFX you can base your application on an existing Application Template (AT) for one of the standard STM32 Evaluation Kits available in TouchGFX Designer. This includes all the TouchGFX AL code you need. The ATs are based on a CubeMX and TouchGFX Generator configuration, so it is possible for you to modify the configuration if you want to experiment later on.

A full description of this step can be found in the TouchGFX AL Development section.

UI Development



This activity is where you probably will spend most of your project development time. Here you will create the User Interface code that will make up the visible part of your TouchGFX project, the component which is called the TouchGFX UI Application.

TouchGFX Designer

The main tools in this activity are TouchGFX Designer and your favorite IDE or text editor. In TouchGFX Designer you will setup, design and create the screens in your application and generate main parts of the UI Application as C++ code. For the application logic (handling events, communicating with the non-UI part of the system) you will use an IDE or text editor to write C++ code, that coexists and interacts with the generated code from TouchGFX Designer.

Application Templates

If you are doing a UI Prototype or just want to try out TouchGFX and do not want to spend time doing the other activities, you can either base your application on the PC based TouchGFX Simulator or you can use one of the existing Application Template (AT) for one of the standard STM32 Evaluation Kits. In any case you are ready to start developing your UI Application right away.

UI templates

If you just want something to run or want to be inspired you can select one of the TouchGFX demos or examples which can be found as UI Templates when creating a new project in TouchGFX Designer. IF you do so, nothing has to be done, just compile, flash and run.

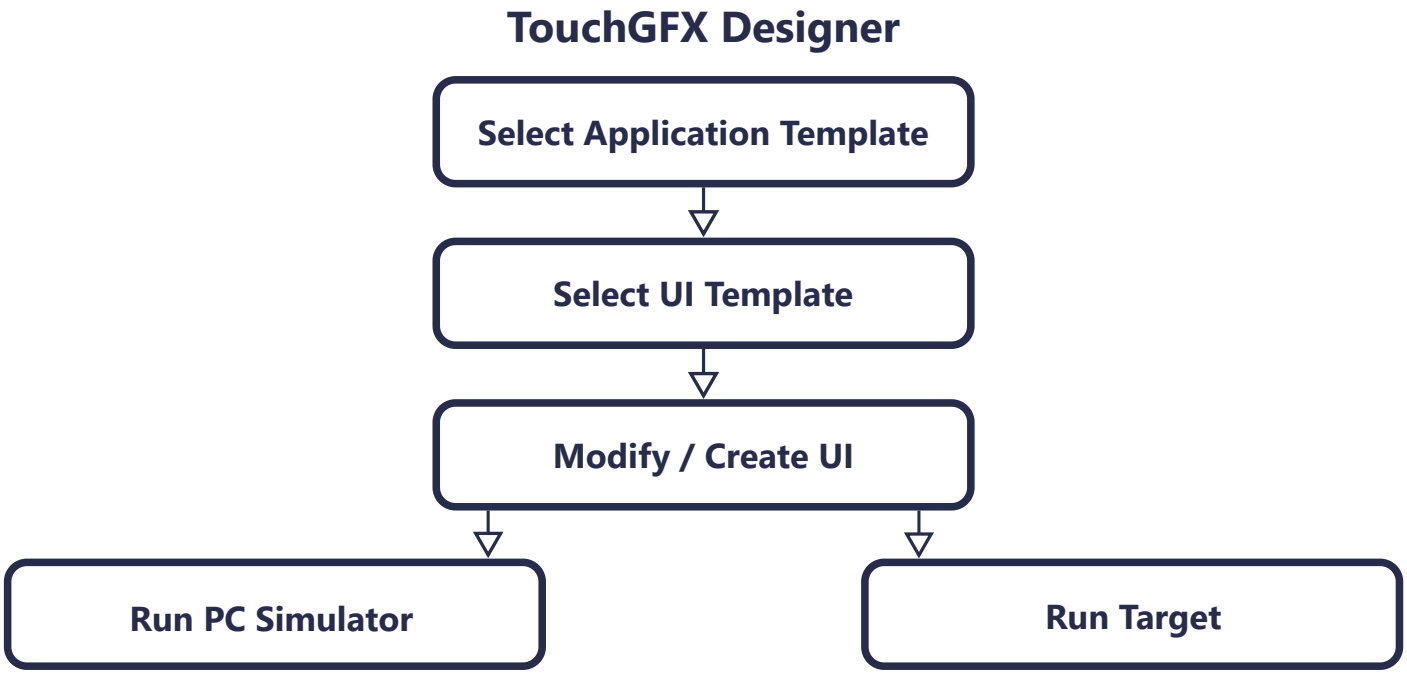
Custom Hardware

If you have already done all the other activities and thus have a running board ready for a TouchGFX UI Application you can either start from scratch or select one of the examples or demo. If the resolutions of your custom board and the example match then they should run on your custom board as well.

A full description of this step can be found in the UI Development section.

Workflow

As you can see TouchGFX development involves a lot of activites and tools. It is, however, important to notice that you do not need to do them all at once, and you do not necessarily need your Display Board, Board Initialization Code and TouchGFX AL before starting your UI development. This can be done using STM32 Evaluation Kits or the TouchGFX Simulator.



TouchGFX Designer workflow

Generated Code and User Code

In each of the three software activities, Board Bring Up, TouchGFX AL Development and UI Development, you will use tools that generate code for you. Common for these tools is that they do not generate all the code you need, you will be adding user written code to the project as well. For all three tools you can go back and forth between using the tool and writing code. The generated code and the user code are independent and can be updated separately.

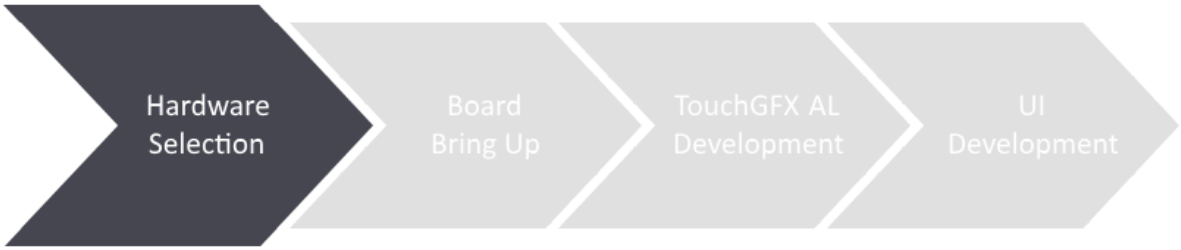
Sometimes you will also be switching back and forth between the activities and thus using different tools. This is often the case when doing TouchGFX AL Development, where you have a very simple TouchGFX UI to test your TouchGFX AL. There is, however, no problem shifting between activities, as the tools, the generate code and your user code coexists and can be updated without any problem.

Change of Compiler/IDE

During the activities you will have to compile your code for your target board. The compilers supported by TouchGFX are IAR, ARMCC, ARMCLANG and GCC(CubeIDE). The toolchain selected for your TouchGFX project is configured in CubeMX, so if you want to change it you should open the TouchGFX project .ioc file in CubeMX and change the toolchain setting. When selecting an Application Template (AT) in TouchGFX Designer it will have one toolchain preselected, so you will only find a project file for one of the toolchains. As the AT comes with an .ioc as well, you can open it and change the toolchain to match your needs.

Version: 4.16

Hardware Selection Introduction



Project activity

Project component

There are many parameters to consider and evaluate when choosing the hardware platform for running a graphical user interface. This article attempts to address considerations about the MCU, display, external memories, UI performance, etc.

It is recommended to read the section on preliminary considerations before hardware components, as there are several parameters and decisions which will impact the choice of hardware.

- Preliminary Considerations - contains several pointers to considerations you should take into account before moving on finding the right hardware.
- Hardware Components - contains information on the different components that makes up a hardware solution and what impact they have on a TouchGFX application.

Version: 4.16

Preliminary Considerations

The purpose of this section is to give some pointers to what considerations should be accounted for before deciding upon hardware. Every product is different and as such has different criteria and requirements, so let the following serve as a source of inspiration for what questions you should ask yourself before making a decision.

Topics being covered in the following are related to the appearance of the display, the needs of memory in the system, the desired performance of the UI and the physical design of the product.

Display Resolution

Displays come in many different resolutions and aspect ratios. In general, TouchGFX is not dependent on any of these parameters. The display resolution is one of the major factors when selecting suitable hardware and, a higher resolution often equals more pixels, and therefore more data to render and transfer.

STM32 microcontrollers generally support up to XGA resolutions (1024*768) in 16/24 bpp, and also support non-standard resolutions like wide or round displays. For resolutions above XGA one must typically compromise on color depth, frames per second, ...

Below is 3 examples of standard resolutions:

Display resolution examples

Pixel density should also be considered, as a larger display size warrants a higher resolution to be perceived as sharp, though higher pixel density often correlates with higher cost.

Some of the questions you should ask yourself when picking a resolution for your application is:

- **What is the end-user target segment?** Often consumers demand higher pixel density while some industrial applications can compromise this for lower cost or easier integration.
- **Are you going to be using a lot of small text in your application?** Large blocks of small text are usually a lot more readable on higher resolution displays due to greater pixel density.
- **Are you generally going to be showing a lot of different elements on a screen at a time?** Larger displays allow showing more elements, or making certain elements more clear, as more inches are available.

Color Depth

Second major factor is the color depth (bits per pixel) which dictates the amount of information which can be stored per pixel in an image, which thereby means how many different colors you are able to assign to a single pixel.

1 bit per pixel and 24 bits per pixel applications

Displays are supporting different color depths, and running a 16bpp GUI application on a 24bpp display is possible, but there will be an impact the other way around running a 24bpp application on a display only capable of showing 16 bit colors.

Displaying complex images with a lot of nuances in color demands a higher color depth to be as close to the source image as possible. The chosen color depth has an impact on the amount of memory needed.

Do not underestimate what you can achieve on lower color depths, as a lot of modern UI design philosophy revolves around flattened and less color intensive applications (for example Google's

Material Design). TouchGFX can help in making complex images useful on lower color depths, by applying one of a set of dithering algorithms. Below you can see some examples of what you can achieve at lower color depths:

Low color depth application examples

Some of the questions you should ask yourself when picking a color depth for your application is:

- **Do you need to display real life images?** If using real life images or multi-layered composed images, it is recommended to use 24 bpp pixels both in the application and the display, as 16bpp in some cases is insufficient in showing all needed colors. 16bpp can in many cases be sufficient enough and is still one of the industry standards.
- **Is grey scale colors or simple 6/8 bpp perhaps all you really need to convey what your application needs?** Perhaps your application does not need sprawling colors to convey its functionality properly and as such lower color depths can be chosen. This is also decreasing the framebuffer size and thereby the RAM needs.
- **Do you have a limitation on RAM and/or flash?** Limiting the color depth will decrease the size of both bitmaps and framebuffer (RAM) needs.

Framebuffer Size Calculation

A framebuffer is the location where pixel data for a frame is stored, rendered and transferred to the display. The size of the framebuffer is important as a higher pixel amount and higher color depth calls for a higher throughput on RAM and display interface.

The size in bytes of a framebuffer is calculated by:

```
display width * display height * (bits per pixel / 8)
```

As an example, an 800x480 application with a color depth of 16bpp and a single framebuffer would need a framebuffer allocated with a size of:

```
800 * 480 * (16 / 8) = 768.000 bytes (768.000/1024 = 750Kbytes)
```

So when you decide on a resolution and color depth, be sure you have enough RAM to support it. Some applications requires 2 framebuffers, so in the above example the needed RAM is 750 Kilobytes * 2 = 1500 Kilobytes.

Framebuffer calculations

FURTHER READING

Note that the memory needed to support the framebuffer also heavily relies on the chosen framebuffer strategy (single, double, partial) which you can read more about in the Framebuffer strategy article **Framebuffer**. Some STM32 microcontrollers supports up to HVGA resolutions running only internal RAM, for a very cost-effective solution.

Display

Interface

It is possible to select displays with different display interfaces (such as SPI, LTDC, MIPI-DSI), which all have different impacts on number of pins needed, bandwidth, supported resolutions and potentially also the amount of external RAM needed. Read more about this and the pros and cons of each in the Display section.

Size

The physical size of the display is also important to consider. Larger displays are generally easier to operate and easier to give precise touch commands, but also require a larger resolution to be easy on the eyes, thus impacting the need for more memory and throughput. If the information on the display is being shown 1-2 meters away, the text, icons etc. needs to be large enough.

Touch

There are two main types of touch displays:

Capacitive

Capacitive touch displays have much higher touch sensitivity which is important if the application requires more advanced touch operations such as dragging, swiping etc. and is also the most used in modern devices due to this fact.

However, they are also more expensive and can often not operated with gloves, so if this is important, perhaps resistive is the solution.

Resistive

This cheaper alternative is much less sensitive and has poor visibility in sunlight, but can be operated with gloves as it is less sensitive to unintended interactions, and is generally more resistant.

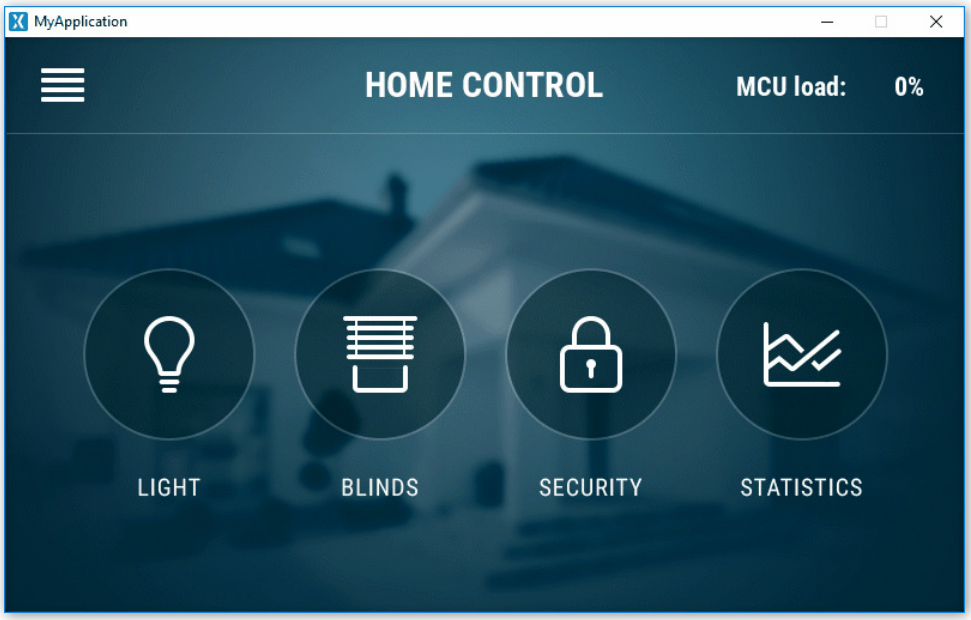
So if all your touch operations consist of simple button presses, perhaps a resistive display is enough. The STM32F429-DISCO board is using a Resistive touch display.

Animations

Running complex animations, like full screen transitions, rotation, and scaling, can have a significant impact on performance if the throughput and calculation power of the hardware is not sufficient.

Some of the questions you need to ask yourself when deciding on the level of animation in your application are:

- **Do you need high speed full screen transitions?** Full screen transitions need to render the full framebuffer and are as such dependent on enough MCU cycles, and fast enough access and transfer of pixel data. The needed system performance also depends on resolution and color depth. High resolution full screen transition are mostly recommended to be used on STM32 high performance products. Some transitions require additional storage, and might therefore result in a larger amount of memory needed.
- **Do you need complex texture mapper animations like rotation and scaling?** Animating a texture mapper can be quite intensive on the system when it comes to calculations and transferring bitmaps and as such generally needs higher MHz, and high memory throughput.



Animations

Mechanical Design Requirements

Physical casing requirements of a product will vary greatly and can have an impact on the hardware chosen. Home appliances will have other requirements to hazardous industrial usage and therefore, some of the questions you should ask yourself when unveiling the physical limitations could be:

- **Is your product required to be very small?** An example could be a smart watch, which will have a limited casing size, which is limiting the size of the PCB, and therefore choosing the correct components is important. STM32 is offering a wide variety of MCU packages, like a WLCSP package.
- **Is your product going to be subject to extreme temperatures?** Capacitive displays can under perform during extraordinary heat or extraordinary cold. So perhaps if you're installing your product in for example cold storage, a resistive display will be a better user experience. The STM32 product portfolio offers microcontrollers with ambient temperature range up to 85, 105 and 125 degrees.
- **Does your product need to be very resistant to outdoor environmental factors such as water or dust?** Different technologies offer different quality and features, and adding a cover lens for protection purposes could be one option.
- **Is visibility in high sunlight important?** Displays varies in candela and lumen, and the higher lumen and candela of the display the higher readability of the display. Adding a special cover lens can also improve this. Or using a another display technology which may also offer reflective features.

Frames Per Second

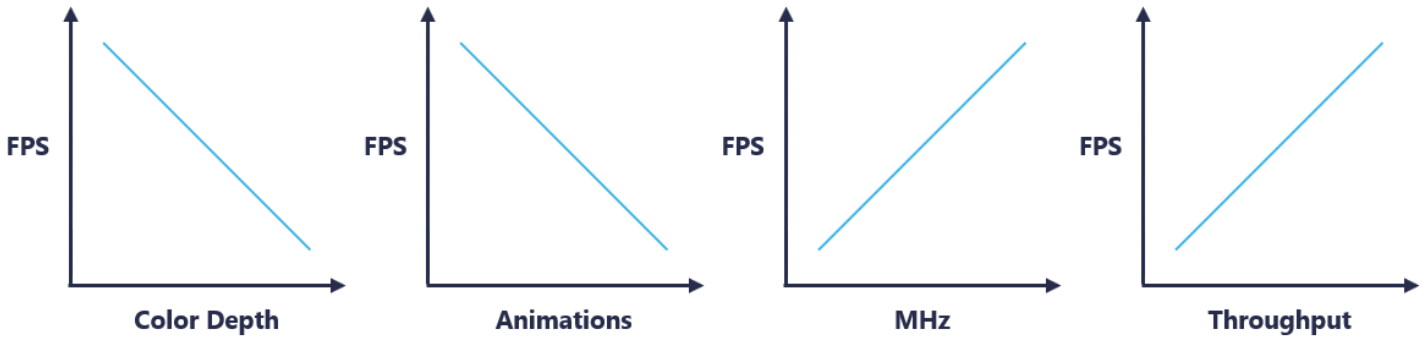
A higher and stable Frames Per Second (FPS) count is often preferable since this makes an application seem much smoother.

Sometimes FPS is less important. For instance in a static GUI with minimum display animation updates. In such cases lower cost hardware might be suitable.

But having a pretty and smooth application with a lot of flashy animations can also be a huge selling point depending on the targeted user segment, so as with anything related to hardware selection, it is all about meeting the end-users expectations, and providing a good user experience.

The overall performance of a graphical user interface comes down to the performance at system level, accounting for components like the MCU, RAM, Flash, display, interfaces throughput, and also hardware capabilities like the STM32 Chrom-ART.

The figures below paint a very generalized picture of the impact of some different parameters. To select the right hardware, these parameters need to be considered. Also taking into consideration that the STM32 Chrom-ART is offloading the MCU, and thereby in some cases decreases the importance of a high MCU frequency.



The impact on FPS of different parameters

Version: 4.16

MCU

The microcontroller unit (MCU) is at the core of any embedded solution and there are a wide variety of options in both costs and features.

When selecting an MCU for graphics, one should consider the supported display interfaces, the MCU package, size and the achievable graphics performance which depends on two main points:

Image composition

- The availability of graphics accelerators integrated in the MCU.
- The availability of cache memory in the system.

Memory access and bandwidth

- The clock frequency and the subsystem bus frequency.
- The access to the internal flash and RAM memories.

It is also important to consider the other aspects of the application (motor control, wireless, etc.), which are running in addition to the graphics. These can influence the choice of MCU.

This page will go through the different MCU options and which parameters should be considered when deciding on the STM32 MCU you should select for your GUI driven application.

MCU PORTFOLIO FOR GRAPHICS

STM32 SERIES	FREQUENCY	HARDWARE ACCELERATION	DISPLAY INTERFACES	SUPPORTED RESOLUTIONS
STM32G0 (CM0+)	64 MHz		SPI	Up to 320*240
STM32L4 (CM4)	80 MHz	Chrom-ART ™	Parallel SPI	Up to 480*272
STM32L4+ (CM4)	120 MHz	Chrom-ART ™ Chrom-GRC™	Parallel LCD TFT MIPI-DSI	Up to 450*450
STM32F4 (CM4)	180 MHz	Chrom-ART ™	Parallel LCD TFT MIPI-DSI	Up to 800*480
STM32F7 (CM7)	216 MHz	Chrom-ART ™ Hardware JPEG Codec	Parallel LCD TFT MIPI-DSI	Up to 1024*768
STM32H7 (CM7)	480 MHz	Chrom-ART ™ Hardware JPEG Codec	Parallel LCD TFT MIPI-DSI	Up to 1024*768

STM32 MCU

FURTHER READING

- For a more complete overview of all product lines, peripherals, prices etc., **the ST MCU Finder is available here**.

Frequency

The core frequency has a major impact on the performance of a graphical application in terms of screen refresh, fluidity of screens and animations.

It impacts the amount of data that can be transferred from an internal or external memory to the display framebuffer and also the calculations and animations possible.

The higher the frequency, the more data it is possible to transfer within a given timeframe and the more complex animations can be made.

The core frequencies of the STM32 products is up to **480MHz**.

> **NOTE**
>
> **The higher the frequency, the greater the power consumption.**

Graphic Subsystem Frequency

It is important to differentiate the core CPU frequency from the graphic subsystem frequency. The graphic subsystem frequency includes the frequency of the internal busses, the frequency of the graphics accelerator as well as the access speed of the internal and external memories.

The graphic subsystem frequency also has a major impact on the overall graphic performance.

Example

An example of assessing the theoretical core and subsystem performance when running from internal RAM on an STM32H7 can be seen next:

- The CPU core is running at **480MHz**.
- The 64-bit AXI bus frequency at **240MHz**.
- The LCD-TFT display controller (LTDC) uses the 64-bit AXI bus, and does 8 transfers in 10 cycles.
- The internal RAM poses no significant latency, i.e. 0 wait states.

The bandwidth of the internal RAM when accessed by the LTDC peripheral is then:

- Bandwidth = 240 MHz x 8/10 x 8 bytes = **1.536Mbytes/s**.

With such bandwidth, the internal RAM can ensure 1000 frames per second (fps) for 800x480 resolution at 32bpp color depth. Typically one would limit the transfer to the display (by adjusting pixel clock, porches, ...) to 60 frames per second, so the bandwidth of the LTDC and internal RAM is not a bottleneck.

Embedded Hardware Acceleration Features

Different STM32 MCUs have different built-in hardware acceleration features that help in achieving high performing graphics applications.

Chrom-ART

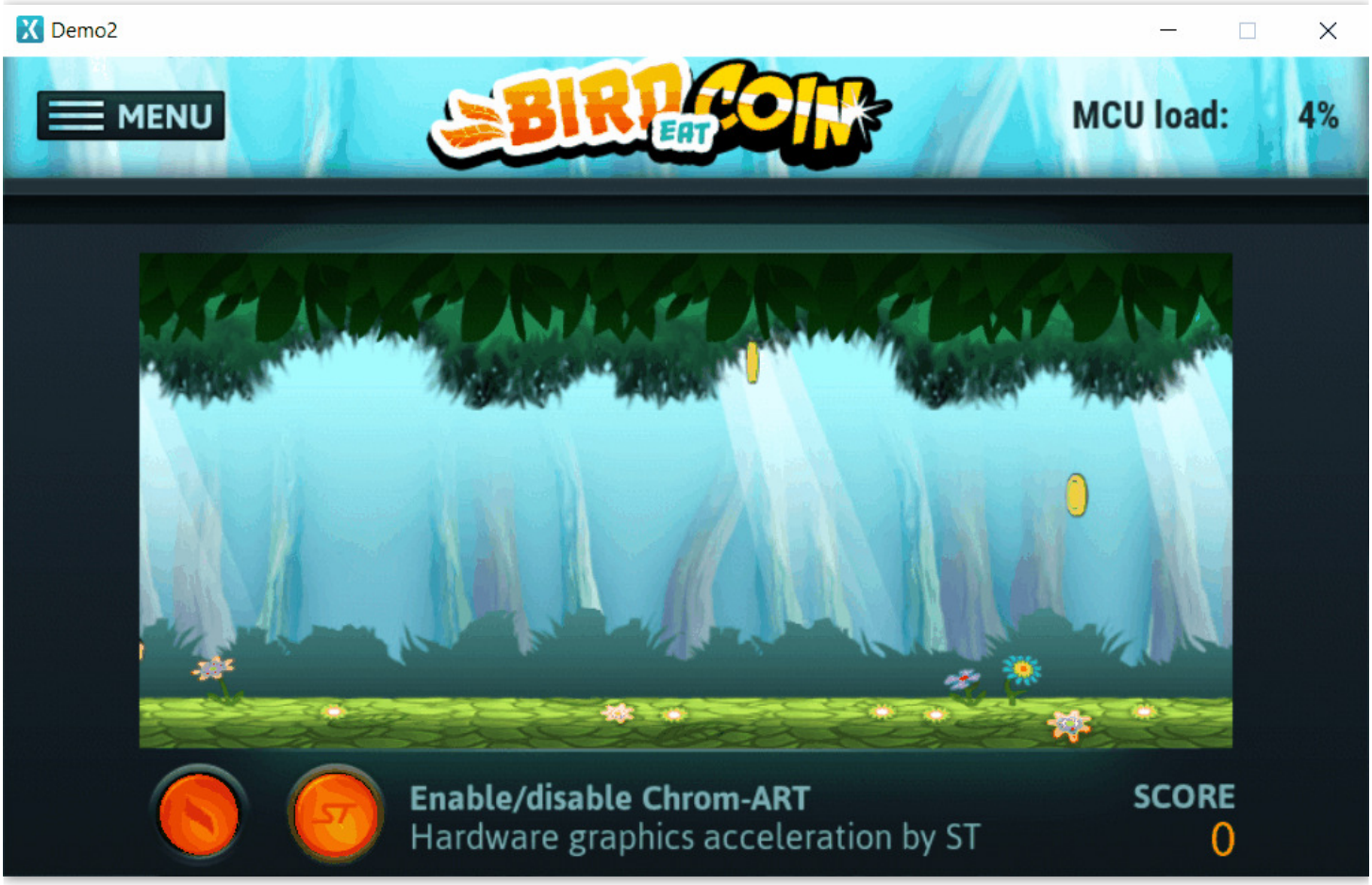
Chrom-ART is an advanced DMA that aids in doing graphical operations. It is also known as DMA2D.

The Chrom-ART accelerator, integrated in many STM32 platforms, is able to manipulate and transfer images without CPU load. It has the capability to accelerate the majority of the graphic operations, such as color filling, image copying, blending, and pixel format conversions.

The Chrom-ART accelerator is able to perform blending of two layers and convert the initial pixel formats to the desired output pixel format and transfer the result to the memory destination in only one operation.

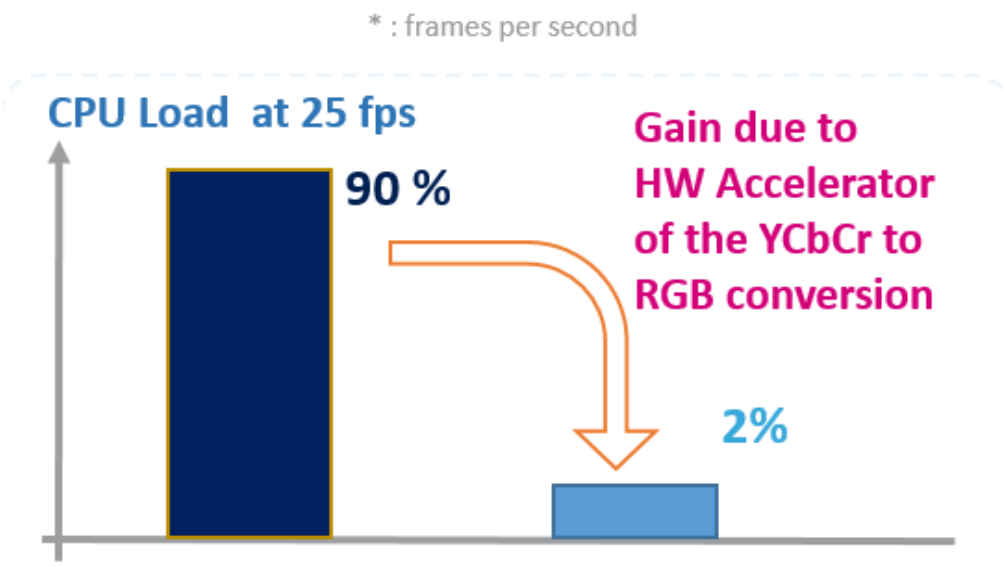
The Chrom-ART accelerator also supports color formats with color look up tables (CLUT). This can help with saving memory.

Example of an application running on the **STM32F496-EVAL** board where the CPU load is decreased from **82%** to **4%** when the Chrom-ART is enabled:



Bird-Eat-Coin Chrom-ART example

In addition, the capability to convert from **YCbCr** format to **RGB** format is added with STM32H7 products to the Chrom-ART peripheral. This feature, combined with the JPEG hardware codec can offload the CPU when encoding and decoding JPEG images.



YCbCr to RGB Hardware performance

The Chrom-ART accelerator, with the features listed above, offers a huge advantage for graphical applications. If available in the chosen MCU, TouchGFX handles all Chrom-ART features and redirects all possible drawing operations to the Chrom-ART peripheral instead of the CPU.

The Chrom-ART peripheral is available with high performance STM32 families.

FURTHER READING

- Refer to AN4943 application note for more information:, **Chrom-ART Hardware acceleration**.

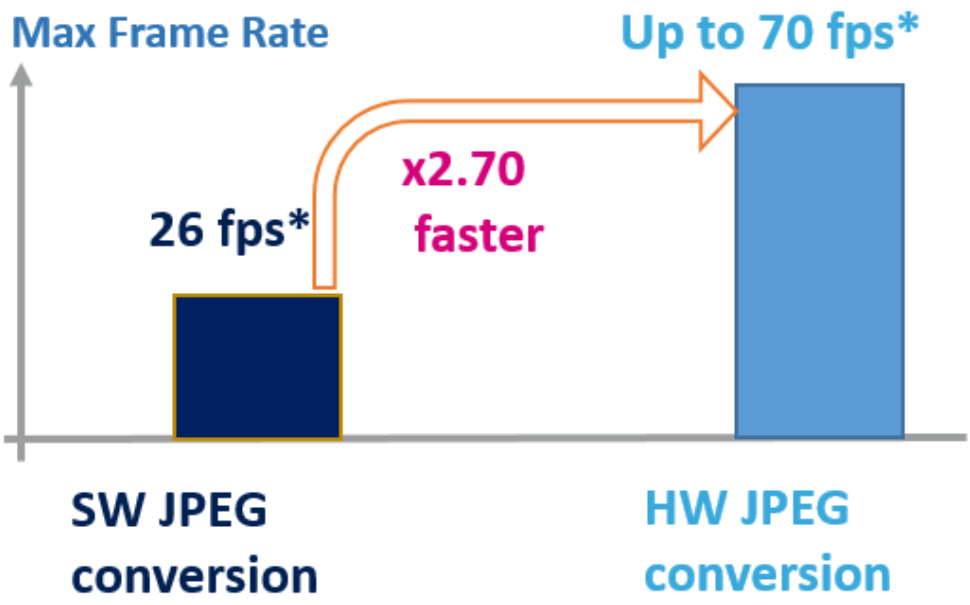
JPEG Hardware Codec

The **STM32H7** and **STM32F7** families provide a hardware JPEG codec to encode and decode images and videos.

This feature is important if the UI application needs to play a video file or display JPEG images.

JPEG images generally take up less memory. The JPEG hardware codec ensures that the images can be decoded at runtime without CPU overload.

Some TouchGFX demos utilizes the JPEG hardware codec, offloading the CPU while playing an MJPEG video.



Hardware JPEG codec performance

FURTHER READING

- Refer to AN4996 application note for more information: **Hardware JPEG codec**.

Chrom-GRC

The STM32 Chrom-GRC™ (GFXMMU) is a peripheral in some STM32 microcontrollers that aims to efficiently support the emerging trend towards non-rectangular displays.

The Chrom-GRC™ peripheral enables applications to reduce the amount of RAM needed for storing the framebuffer when addressing non-rectangular displays.

In the case of a round display, the peripheral reduces the memory requirements by **20%**.

The Chrom-GRC™ peripheral is not mandatory when controlling non-square screens, but it is recommended.

Memory optimization with Chrom-GRC peripheral

FURTHER READING

- Refer to AN5051 application note for more information: **Graphic memory optimization**.

Internal Flash

A graphical user interface application using bitmap resources needs non-volatile memory to store the data. The execution from and access to internal flash is in some cases up to two times faster than external flash.

As the internal flash is limited in size, in many cases it is often used for storing the TouchGFX framework, screen definitions and UI logic while the bitmap data is stored in external flash.

The portfolio of STM32 products used for graphic applications is between **a few Kbytes** and **a few Mega bytes** of internal flash memory.

External memory may be required when the amount of bitmap data does not fit within internal flash.

FURTHER READING

Refer to **External Memories** for more details.

TouchGFX flash memory requirement:

- Framework: **60kbytes** to **100kbytes**.
- Screen definition and GUI logic: **1** to **100Kbytes**.

These numbers depend on the framework features used and the size and complexity of the application.

Internal RAM

Internal RAM can be used for storing the framebuffer(s), when the size of these fit within the available memory. Alternatively one might add external memory to the setup.

Calculating the size of a framebuffer depends on the width, height and color depth. For example, a display with HVGA resolution (480x320) and 16 bit colors, the memory needed for one framebuffer is:

Size of 1 framebuffer = **480 x 320 x 2** = **307.200 bytes**

The STM32 products used for graphic applications ranges from **a few Kbytes** and **a few Mega Bytes** of internal RAM.

> **FURTHER READING**
>
> Refer to the **External Memories** section for more details on framebuffers in external memory.

TouchGFX RAM requirement:

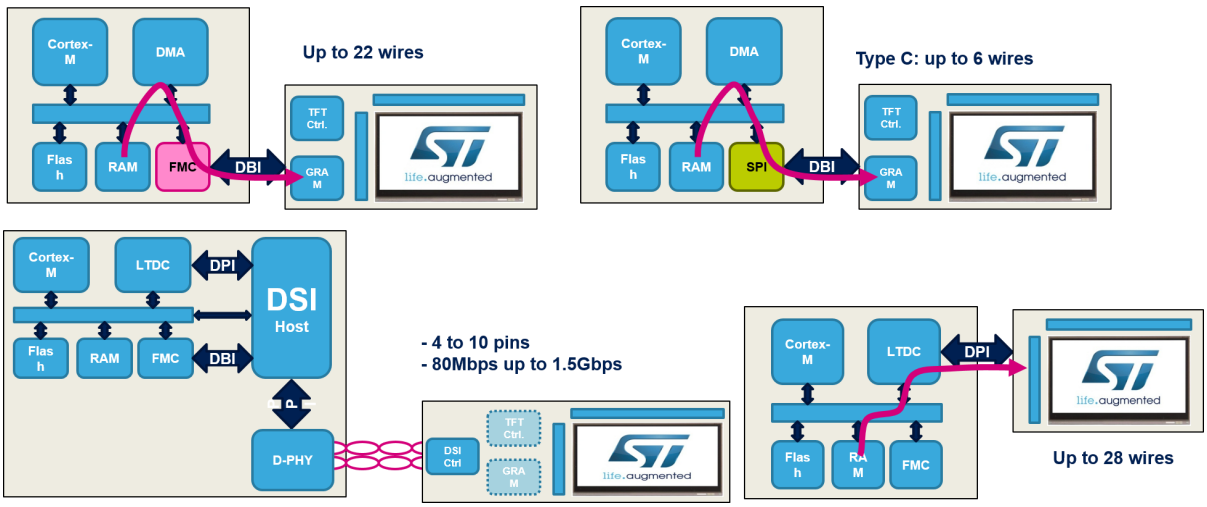
- Framework: **10Kbytes** to **30Kbytes**
- Widgets: **1Kbytes** to **15Kbytes**

Memory requirements may vary from application to application.

LCD Controller

The choice of the MCU also depends on the display interface that will be used and the resolution. The 800x480 resolution for example can only be achieved with an efficient interface in terms of data transfer speed. RGB-TFT and MPI-DSI interfaces are often used for higher resolutions, as the bandwidth is in many cases higher than SPI or parallel 8080/6800. Small resolution displays often embed controller and GRAM and so can be connected through simple SPI or 8080/6800 interfaces.

High resolution displays (WQVGA and above) often don't embed controller and GRAM, therefore the controller needs to be at the microcontroller side. On STM32 MCUs embedding RGB-TFT and MIPI DSI interfaces the controller is present.



The picture shows 4 examples of different display interfaces with/without GRAM and display controller.

FURTHER READING

Refer to the **Display** section for more information.

Packages & I/O

The number of I/Os needed is dependent on the chosen display and external memories. Running a parallel display with parallel RAM/flash can require a high number of I/Os resulting in a larger package.

Memory Interfacing

When internal flash and RAM in the microcontroller is not sufficient, choosing the right MCU with the most suitable external memory interface becomes important. The STM32 products provide different memory controller peripherals to interface with the NOR, NAND, SRAM, SDRAM, LPSDR SDRAM, and PSRAM memories.

Flexible Memory Controller & Flexible Static Memory Controller (FMC/FSMC)

In addition to the support of the static RAM, the FMC adds dynamic RAM support (SDRAM) to the FSMC. The flexible memory controller (FMC) with its high external access speed and 8, 16 and especially 32 bit data bus, allows for higher throughout from and to external RAM and hence better support of higher resolution. The FMC has an independent chip select for each memory bank. The FMC can control an external flash memory for the data and an external RAM memory for the framebuffer and heap extension for the graphical stack.

Serial Memory Interface

Depending on the STM32 product, the serial memory interface is embedded and allows interfacing with single, double, quad, octo, and hyperBus flash memories alongside QSPI, PSRAM, OPI PSRAM, and Hyper RAM memories. The serial high speed memory interface can control up to **256 Mbytes** when in memory mapped mode and **4Gbytes** in indirect mode.

Compared to parallel interfaces, the serial memory interface permits the connection of a lower cost external flash memory to small packages and reduces the number of used pins.

> **FURTHER READING**
>
> Refer to AN4760 application note for more information: **Quad-SPI interface on STM32 microcontrollers**.

STM32 Value Line products

For price optimization, STM32H7 and STM32F7 platforms offer value line products with limited amount of internal flash. With these products, the graphic resources will be stored in the external flash.

Cortex®-M Cores

STM32 MCUs comes in different ARC Cortex®-M architectures. Below are the most used cores for running graphics on STM32.

Cortex®-M0+

The **Cortex®-M0+** is characterized by its simple architecture and low price. It is recommended for smaller static graphic applications, running at lower resolutions.

Cortex®-M4

The **Cortex®-M4** contains more functionalities than the **M0+** and accelerates calculations. It includes a DSP instruction set and a single precision FPU unit. These instructions offload the CPU and increases the speed of calculations.

Cortex®-M7

The **Cortex®-M7** contains a more complex architecture but also a DSP instruction set, and comes with a more efficient FPU unit with double precision and a level1 cache memory with up to **16KB** for

data and instructions. The cache memory gives the possibility of having data and instructions close to the calculation unit in order to optimize the fetch time.

Feature overview

Feature	Cortex-M0+	Cortex-M4	Cortex-M7
DMIPS/MHz range	0.95-1.36	1.25-1.95	2.14-3.23
Core Mark®/MHz	2.46	3.42	5.01
Digital Signal Processing (DSP) extension	No	Yes	Yes
Floating Point Hardware	No	Yes (SP)	Yes (SP + DP)
Built-in-caches	No	No	Yes (option 4-64KB), I-Cachen D-Cache
Bus Protocol	AHB Lite,Fast I/O	AHB Lite, APB	AXI4, AHB Lite, APB, TCM
Dual Core Lock-Step Support	No	No	Yes

Level 1 cache:

The STM32H7 and STM32F7 families include up to **16 Kbytes** of L1-Cache both for instructions and data. An L1-Cache stores a set of data or instructions near the CPU, so the CPU does not have to keep fetching the same data that is repeatedly used.

FURTHER READING

Refer to AN4839 application note for more information: **Level 1 Cache**.

Dual core

The STM32H7 series includes the dual-core line:

Arm® Cortex®-M7 and Cortex®-M4 cores can respectively run up to 480 MHz and 240 MHz enabling more processing and application partitioning. Dual-core STM32H7 product lines are available with an embedded SMPS for improved dynamic power efficiency.

The second Cortex®-M4 can offload heavy calculations to open up the M7 core for the drawing/graphic operations.

NOTE

The TouchGFX Generator tool is not available as additional software for STM32H7 dual-cores

Bus architecture

The majority of STM32 microcontrollers provide a **32-bits multi-AHB** bus matrix interconnecting all the masters (CPU, DMAs, etc.) and the slaves (flash memory, RAM, FSMC, AHB and APB peripherals). This ensures seamless and efficient operations even when several high-speed peripherals work simultaneously.

In addition to multi-AHB interconnect, some STM32 (Cortex®-M7) products embed **64-bit** AXI to expand bandwidth. This yields the best compromise between performance and power consumption.

Price

The size of the internal flash, internal RAM, and number of pins available in the package influence the price of the MCU. Considering the requirements of the interface, resolution, performance, etc., the user can ultimately find suitable MCUs and estimate price.

FURTHER READING

- See **STM32 32-bit Arm Cortex MCUs** for available STM32 microcontrollers.

Version: 4.16

Display

Products are getting richer with enhanced user experiences, embedding newer larger displays, and replacing older segment displays with low and high color displays.

This chapter focuses on some considerations that should be included when selecting the right display for your embedded GUI product.

Different types of displays

NOTE

Generally, TouchGFX runs on any kind of display, and is not dependent on display technologies, interfaces, viewing angles, brightness etc.

Examples of Displays

Selecting the right display technology can be complicated as key factors in each display are different. The following chapter is high-level addressing the different technologies, and can hopefully help you in the right direction.

Each kind of display consists of rows and columns of pixels, which can be driven in different ways, having internal and/or external display controller and RAM for framebuffers. In some technologies,

each pixel needs to be updated frequently compared to other technologies where this is not necessary, as updates only happens when something changes in the GUI.

There is a vast amount of different display technologies. Some of the most used display technologies are described in the following.

LCD-TFT

TFT stands for thin-film-transistor and is a variant of LCD displays with an active matrix. LCD-TFTs are widely used in embedded products as they are available in many different resolutions, sizes, interfaces, price ranges, etc.

Some variants of TFT-LCDs are TN and IPS panels. Examples of IPS TFT-LCDs, is the STM32F769 DISCO and STM32H747 DISCO, both running a 800*480 MIPI-DSI TFT IPS LCD display. Examples of TFT-LCD TN displays are the STM32F746G DISCO and STM32H7B3I-DK. Both technologies come in different qualities, but some differences can be the color presentation and viewing angles, where IPS panels often are the best.

LCD-TFT layers example

MIP

MIP means memory in pixels, which uses a pixel technology which only needs power/data when something changes on the screen. MIP displays are low power and runs low to full color GUIs.

ePaper/eInk

eInk displays are low color displays, ideal for applications with low power consumption needs, wide viewing angles, and easy readability. TouchGFX Implementer SDATAWAY demonstrates an eInk display running an TouchGFX application on a STM32F412 here: https://www.touchgfx.com/cases/e-ink/

E-Ink

Display Interface Overview

The display is connected to the MCU via different types of interfaces. The display interfaces vary on different parameters, and the section below addresses the graphics related parameters like number of pins needed, max bandwidth supporting different resolutions.

TouchGFX can use any display interface, and STM32 microcontrollers offer a wide range of display interfaces connecting to Motorola 6800, Intel 8080, SPI, RGB-TFT, and MIPI-DSI.

Interface	# of pins	Target resolutions	Max bandwidth	Benefits	Disadvar
SPI	4*	Up to 480*272	16 MHz	Simple hardware interface, faster than I2C,	
Parallel 8080/6800 (FMC)	8/16*	Up to 480*272			
RGB-TFT (LTDC)	8/18/24*	Up to 1280*800		High performance, low cost	High pin parallel commun can cause issues, ca require h clock frec

Interface	# of pins	Target resolutions	Max bandwidth	Benefits	Disadvar
MIPI-DSI (LTDC)	4/10	Up to 1280*800	80Mbps-1.5Gbps	High performance, low pin count,	Complex protocols drivers
LVDS**		1366*768		Low EMC/interference, high speed	Bridge ne

- *Additional pins can be needed for: touch, power, controls signals etc.
- ** a bridge is needed for interfacing with a LVDS display.

Brightness and Backlight

Brightness is often measured in candela/m^2. Backlights can be the most power consuming part of the display. In sunlight one would need around 600 cd/M2. Often higher brightness increases the temperature, minimizing the lifetime of the LEDs.

Viewing Position and Color Inversion

When embedding a display into a product, it is important to anticipate and know which viewing positions the user can have. In some displays from certain viewing positions, a color inversion can happen. This means that installing the display in the right position, allowing the user to operate and experience the GUI while seeing the right colors designed by the graphics designer, can be tricky.

The color inversion can happen on TN panels. Adding a SWV film can help increasing viewing angles.

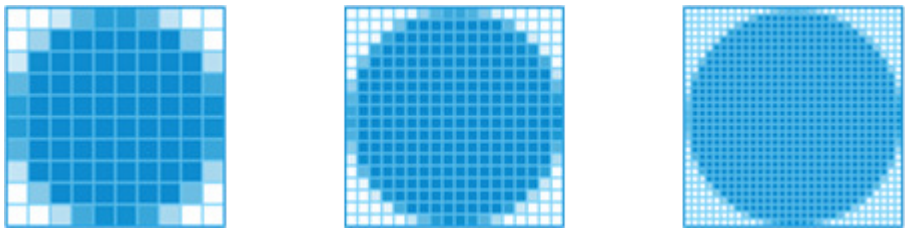
Resulting colors from different viewing position

Display Lifetime

The lifetime is defined as the time until the display reaches half brightness at 25 degrees. If your product has a long life cycle, then this parameter must be taken into consideration.

Pixel density

Pixel density defines how many pixels are shown per inch or square inch. Choosing the right pixel density can depend on the expectations from the end user, environment, design needs etc. Putting this into perspective, a high-end mobile phone runs a 6.1” 2340x1080 with a pixel density per square inch of 178,500, while a commonly used 5” TFT display running 800x480 has 34.816 Pixels per square inch.



Low, medium and high pixel density

Some standard resolutions, display sizes and pixel densities measured in pixels per square inch (PPI^2):

QVGA 320*240	2.4″ (27,777 PPI^2)	3.5″ (13,061 PPI^2)
WQVGA 480*272	4,3″ (16,462 PPI^2)	5″ (12,175 PPI^2)
HVGA 480*320	3.5″ (27,167 PPI^2)	
VGA 640*480	5,7″ (19,698 PPI^2)	6.4 (15,625 PPI^2)
WVGA 800*480	4″ (54,400 PPI^2)	5″ (34,816 PPI^2)
WSVGA 1024*600	7″ (28,746 PPI^2)	10,1″ (13,808 PPI^2)

For some applications it can be difficult seeing any difference, unless the display is being looked at very closely. Examples of pixels densities are: STM32F476DISCO with 16,462 PPI^2 and STM32F769DISCO with 54,400 PPI^2.

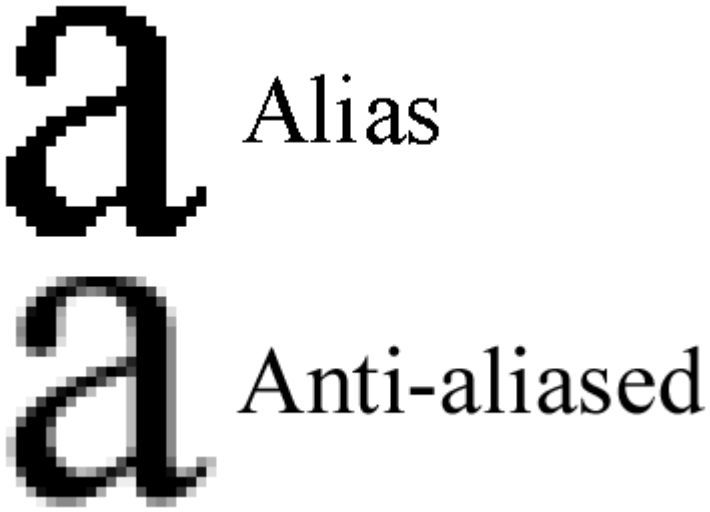
The example above of different pixel densities can in some cases impact the dynamic color range and anti-aliasing:

Dynamic color range

The dynamic color range is the ratio between two contrasting colors, like black and white. In the example above, the blue and white contains different levels of white and blue. The image on the left has lower pixel density, and the picture on the right has more pixels to show all the colors represented, creating a smoother transition between different colors and edges.

Anti-aliasing

When the pixel density is too low, a staircase effect can appear. Using anti-aliasing in the application can smooth out these staircase edges in an image. When looking at the first two blue circles, the staircase effect appears, as the pixel density does not allow the display to represent enough pixels to have a high enough color range enabling high enough anti-aliasing.



Anti-aliasing

Environment

When deciding which display to use, the environment is a vital part to consider. Some questions to ask yourself are:

- **Is the display in direct sunlight?**
- **Is it being used in rugged environments where it needs to be impact resistent?**
- **Is it being handled by one wearing gloves?**
- **Does it need vandal proofing?**
- **Is it being operated with physical buttons only?**

Answering these questions will give you an better idea of which touch technology to select or even if touch is required.

> **NOTE**
>
> **TouchGFX runs on both touch and non-touch displays, and the TouchGFX GUI can be controlled by buttons, hand and voice gestures also.**

Touch / Non-touch displays

There are different touch technologies available in the market today and some examples are: resistive, capacitive (surface, projected), SAW touch, infrared touch. This section will only address some of these technologies:

Capacitive Touch

This is one of the most popular touch technologies. It comes in two sensing technologies:

- Self capacitance is for single finger touch
- Mutual capacitance allowing multi touch but more challenged when exposed to water/moist (TouchGFX does not support multi touch).

Most STM32 DISCO boards are using capacitive touch, some examples are the STM32H7B3I DISCO, STM32H750 DISCO, STM32F746G DISCO.

Resistive Touch

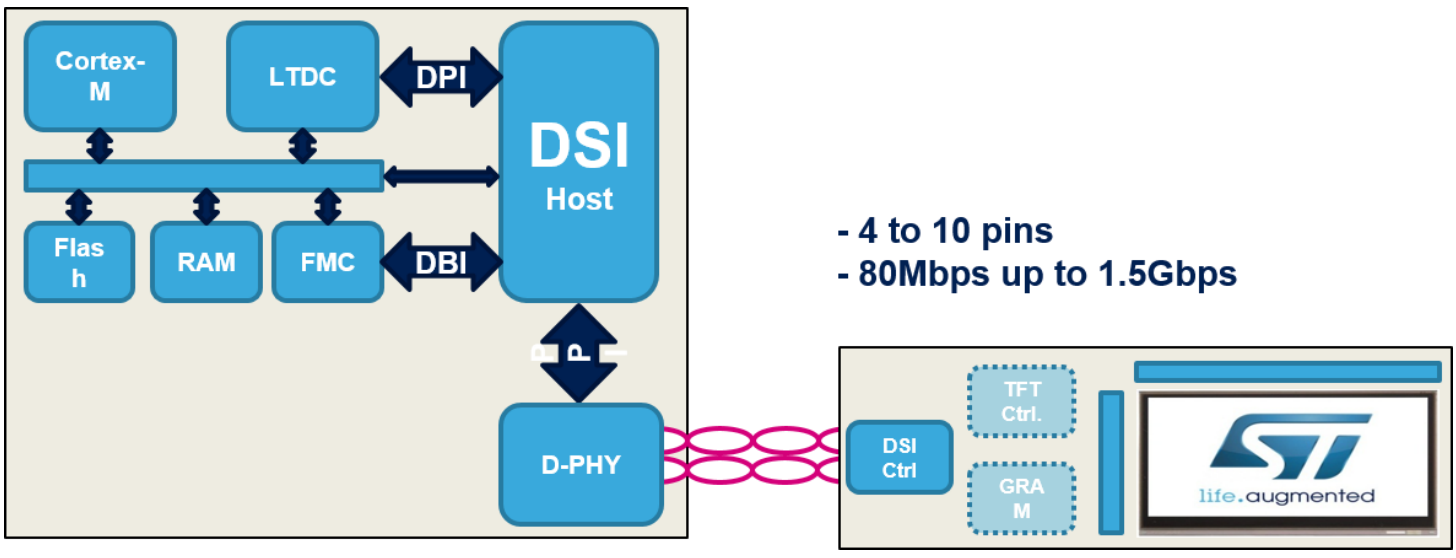
Resistive touch is a simple technology activated by mechanical pressure, and only requires ADC - or a simple touch controller. It is often low price due to maturity. The surface is more protected to scratches and tearing, so more difficult to vandal proof. It also has lower sunlight readability. The STM32F429 DISCO board uses resistive touch, available with a TouchGFX application.

Non-touch

Often if the GUI is being controlled by buttons, just displaying images/video or controlled externally by another device, then adding touch to the product might not even be relevant. By not adding a touch layer to the display, this will decrease the price.

Displays with RAM

Displays with either Motorla 6800, Intel 8080, SPI, or MIPI-DSI interfaces usually embed RAM (GRAM), which has the size of 1 full framebuffer. These types of displays can connect to the MCU via SPI, FMC or DSI-host(LTDC). A second RAM (framebuffer) is required externally to the display RAM and this can be in the MCU or in external RAM.

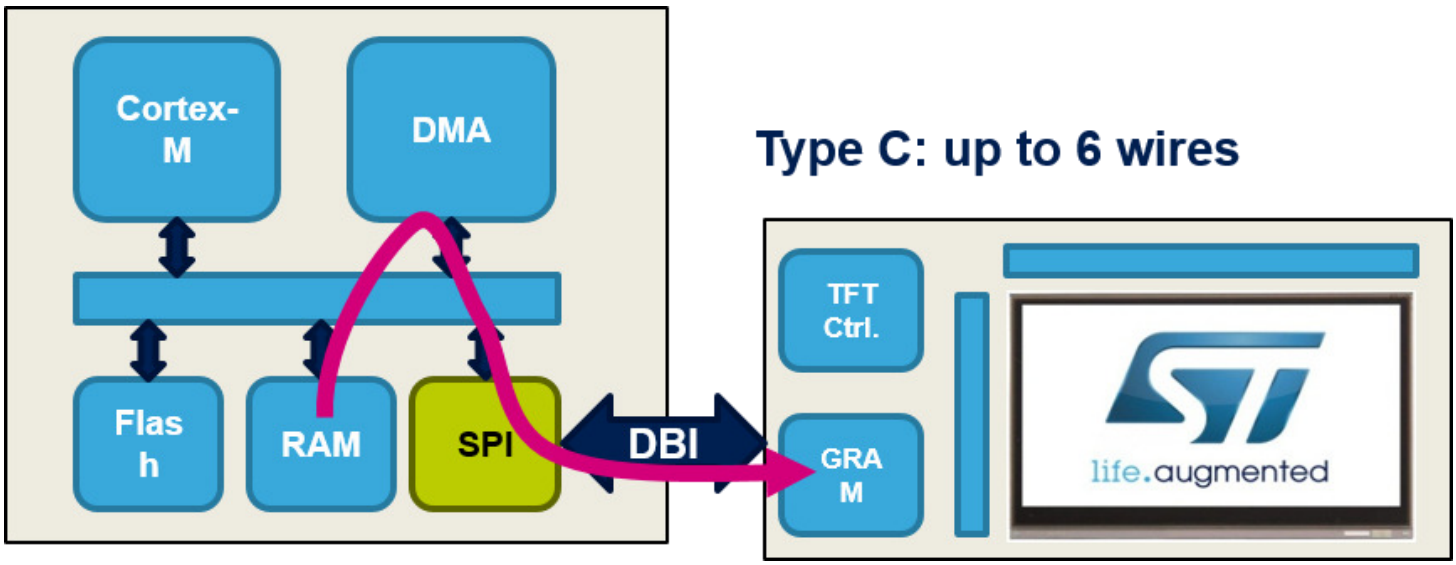


MIPI-DSI display

In some cases the need of external RAM (external to the MCU) for storing the framebuffer is not needed, and thereby the available internal RAM in the MCU is used. If the MCU RAM is lower than 1

full framebuffer size, using the TouchGFX partial framebuffer feature is an option, allowing a very small framebuffer footprint.

> **FURTHER READING**
>
> Read more in the **Partial Framebuffer section**



SPI display

Non-square pixels / Pixel aspect ratio

The most common pixel shape is square, but some displays use non-square pixels. Pixel ratio is the ratio between the width of a pixel and the height of a pixel. The aspect ratio using a square pixel with 100 pixel width and 100 pixel height is therefore 1/1. But non-square pixels result in a different pixel aspect ratio. If a graphics designer does not take this into account, the displayed bitmaps may be stretched like the example below.

Stretched bitmaps

Cover lense

As the display is the face of your embedded graphical user interface product, adding a cover lens could improve the look and feel. The cover lense can improve the design, scratch resistance, impact

strength, colors, etc.

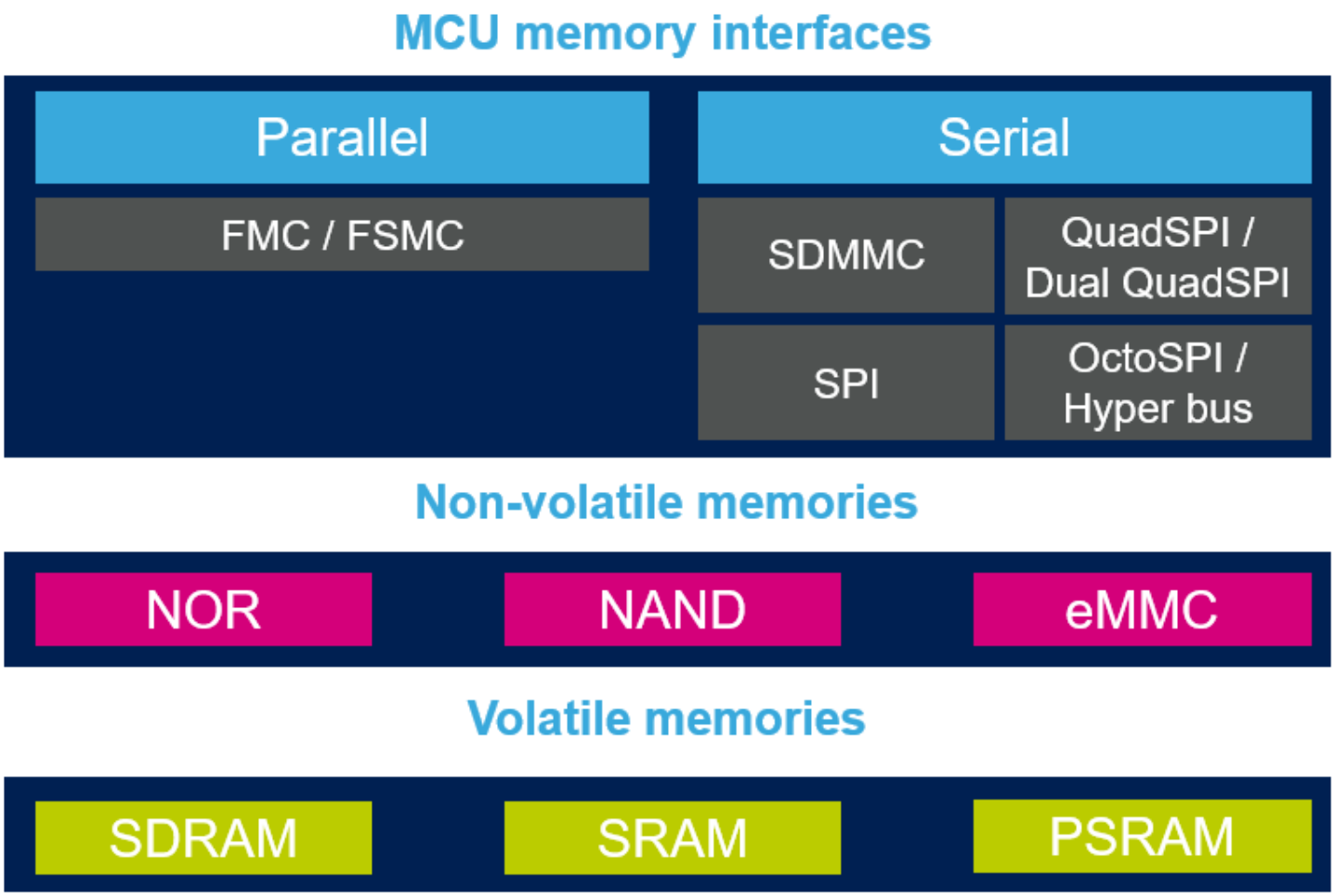
Version: 4.16

External Memories

This chapter focuses on helping you choose the external memories for your embedded graphical user interface. Before reading this chapter, it is recommended to read the Preliminary Considerations and MCU so you are aware of some of the dependencies which are important when choosing the right external memories.

Running a TouchGFX GUI application sometimes requires external memories for storing the framebuffer(s), bitmaps, fonts, translations, etc. A TouchGFX GUI is not dependent on external memories to run, but needs either internal RAM (in MCU) or external RAM for storing the framebuffer(s), and internal and/or external flash for storing data.

The overview below shows some external memories which can be used with an STM32 MCU. Some of the different memory examples are available with both serial and parallel interfaces.



Memory overview

The different STM32 microcontrollers come with different external memory interfaces, allowing to connect different external memories.

Non-volatile Memories

In a GUI application, the non-volatile storage (flash) is mainly used for storing some or all graphical data assets, such as bitmaps, fonts, translations, and TouchGFX application code. The non-volatile memories are supported by the STM32 products and can be connected with different types of MCU interfaces using either parallel or serial memories and different configurations.

Non-volatile memories

The choice of the non-volatile storage depends on:

- Density
- Performance
- Type of the interface (parallel/serial)
- Bill of Material

NOR Flash

The NOR flash is a non-volatile memory that allows random access to any area in the memory.

NOR flash ranges typically between **128 Mbits** to **2 Gbits**.

For example, for 480x320 resolution and 16 bits per pixel as color depth, the user interface needs ~300Kbytes for a full screen background image. This does not take into account the additional bitmaps needed for buttons, sliders, icons, fonts used, number of languages, etc. A 256 Mbits (32 MB) NOR flash can store up to ~100 unique full screen images, and less when adding the rest of the graphical assets needed.

The NOR flash can be used in **memory mapped** mode where the external flash is seen as an internal memory for read operations. This mode allows the system masters (such as DMA, LTDC, DMA2D, GFXMMU or SDMMC) to access the memory autonomously even in low-power mode when the CPU is stopped, which is ideal for mobile and wearable applications.

The NOR flash memory is available with different interface options:

- Parallel NOR flash (with x8 or x16 interfaces)
- Serial NOR flash (single, dual, quad and octo data lines for serial memories, and hyperbus flash)

Serial NOR Flash Memories

Serial NOR flash memory is widely used as storage in graphical applications.

This type of memory has benefits such as:

- High frequency
- Simplifying and reducing the printed circuit board (PCB) area
- Memory mapped mode up to **256Mbytes** of addressable area
- Number of needed pins is between **4** to **12** pins

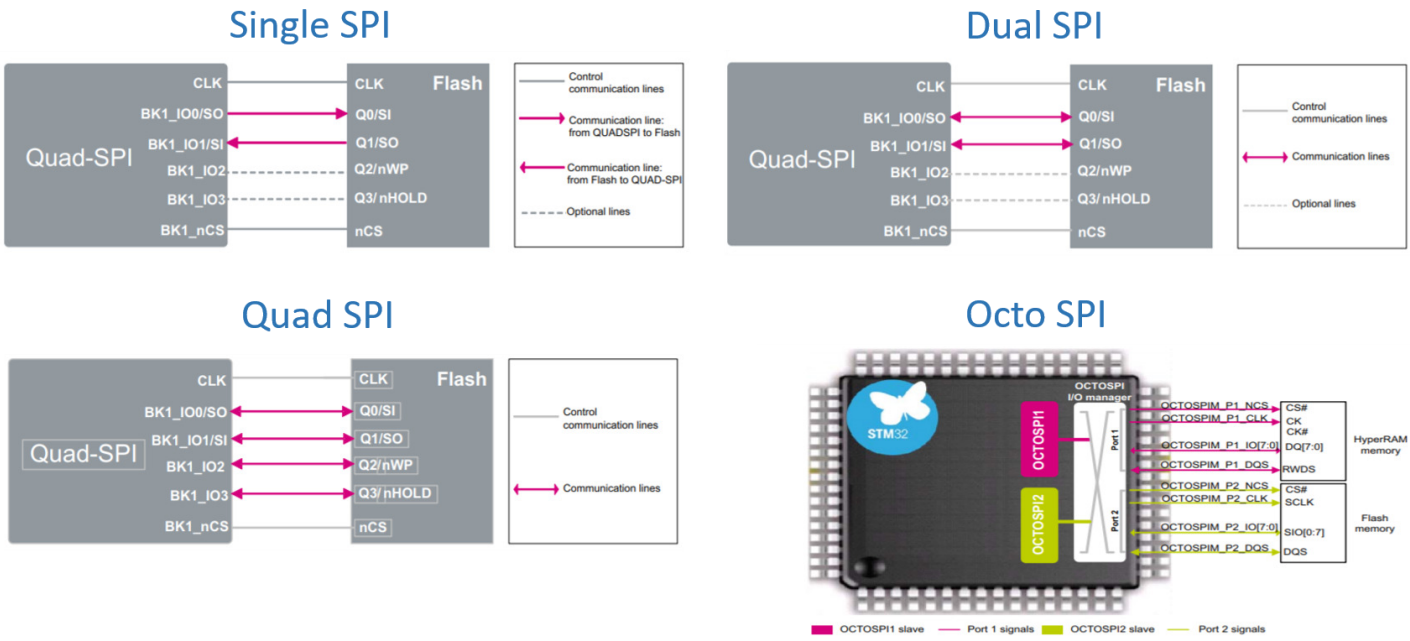
NOR Single, Dual, Quad, Octo Flash Memories

The NOR flash memories are available with different data lines configurations.

- Single
- Dual
- Quad
- Octo

Switching to the serial NOR flash memories with more data lines enhances the performance and the bandwidth of the memory interface, but also requires more pins to interface with the STM32 products.

Below is an overview of the different SPI memories depending on the number of data lines:



Serial interface overview

Parallel NOR Flash Memories

Parallel NOR flash memory has the same advantages as the serial flash memory in term of performance and configuration. The parallel NOR flash can be configured in memory mapped mode and can be accessed as if it was an internal memory. The differences between the parallel and serial NOR flash is the number of pins and the complexity of the printed circuit board (PCB).

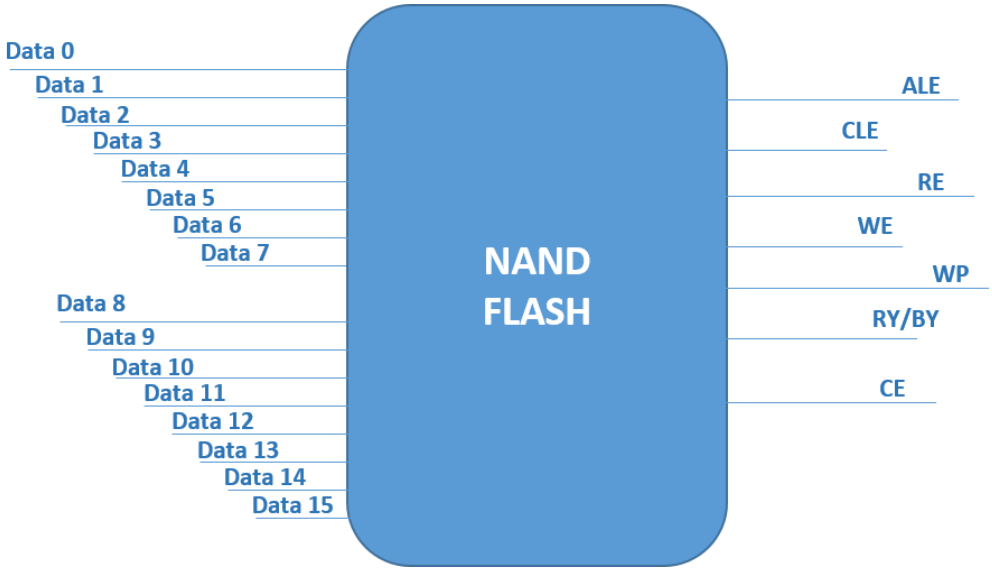
Up to **47 pins** are needed for the NOR flash memory with up to 24 pins for allocation and 16 pins for data.

NAND Flash Memories

NAND flash memory is ideal for graphical applications requiring a high volume of graphical assets and **faster write and erase operations**. The NAND flash memories cannnot be configured in a memory mapped mode and as a consequence, the NAND flash memories are **not recommended for code execution**.

NAND flash ranges between **1 Gbit** to **512 Gbits**.

Using a cache in RAM is often necessary when using NAND flash. This enables moving the currently used graphics assets to RAM and drawing them from the cache.



NAND flash memory

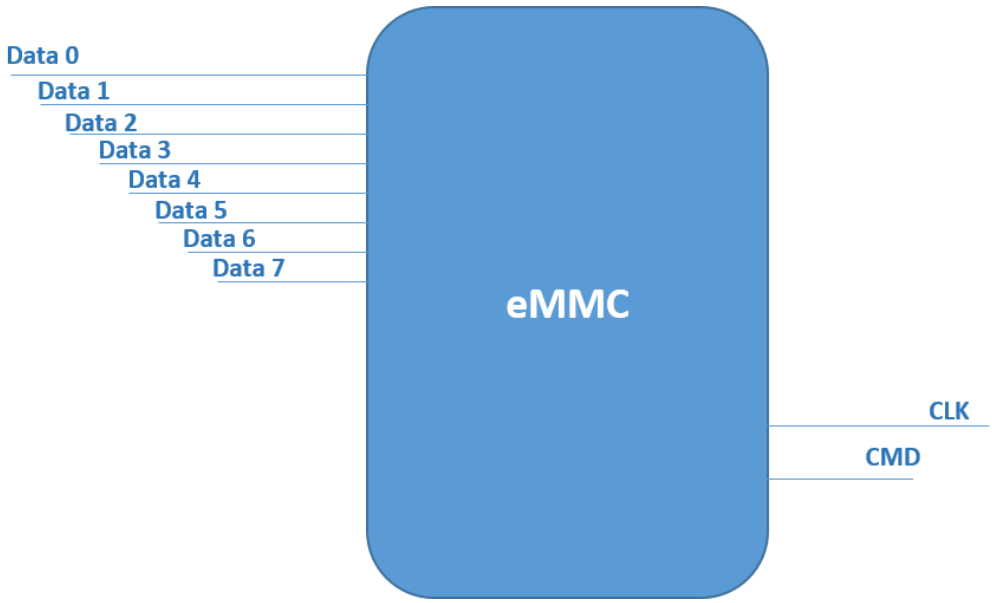
eMMC Memories

eMMC (Embedded Multi Media Card), established by the MMC Association, is equivalent to a NAND flash in addition to a master integrated controller. One obvious advantage of eMMC is the integration of a controller in the package that provides standard interfaces and management for the flash memory, allowing manufacturers to concentrate on other parts of product development and shorten the time to market.

The eMMC flash ranges between **2 Gbits** to **128 Gbits**.

The eMMC has relatively lower random read performance compared to NAND and NOR. eMMC can require adding a cache to overcome slow random read.

Up to **10 pins** are needed for the eMMC flash memory with 8 pins for data and 2 pins for control.



eMMC memory

Volatile Memories

External volatile memory is mainly used for storing the framebuffer(s), if the internal MCU RAM is insufficient, and in some cases to cache assets from non-memory mapped flash. This section focuses on SRAM, SDRAM and PSRAM as they are commonly used in embedded systems running a GUI. But there are other available variants, and the memory manufactures are using different naming schemes for their memories for example "hyper RAM", "IoT RAM", "octal RAM". Common for most of them is that it is possible to find an STM32 MCU which supports it.

Volatile Memories

When selecting the right external RAM, we recommend having the following in mind:

- Density
- Performance
- Power consumption
- Interface / pin size

- Framebuffer strategy

SRAM

SRAM is a static random-access memory which retains the bit data as long as the power is supplied. Generally SRAM provides faster access, but can be more expensive than DRAM and it comes in smaller densities. SRAM typically has a faster access time compared to DRAM and is therefore more suitable for GUIs needing more animations, scaling, rotation etc. SRAM comes in both synchronous and asynchronous modes, where the synchronous modes offers higher bandwidth capabilities, but also a more complex interface.

> **NOTE**
>
> **Also available as a non-volatile random-access memory called nvSRAM which also has the ability to store and recall data.**

SDRAM

SDRAM is a dynamic random-access memory and stores each bit of data on capacitors, which requires less physical space to store the same amount of data compared to SRAM. As it requires constant refresh in order to keep the data, it requires more power compared to SRAM.

SDRAM densities typically come in 16 Mbits up to 512 Mbits, available in 8, 16, and 32 bit interfaces, running frequencies between 100-200 MHz.

A suitable SDRAM for storing two framebuffers running a 24bpp 800*480 resolution would be a 32 Mbits SDRAM as a double framebuffer strategy requires ~18Mbits of RAM.

PSRAM

PSRAM is pseudo static random access memory, with an internal structure of a DRAM (control logic) with an SRAM interface. It typically comes in 8-256 Mbits densities. PSRAM compared to traditional SDRAM and SRAM has the advantages of higher speed and lower power consumption.

Additional memories

New octal RAM and Hyper RAM memories use serial 8 bit interfaces in a single and dual data rate mode, offering high throughput speed and good integration.

Selection of External RAM Density

If your strategy is to place the framebuffer(s) in external RAM, this table gives you an overview of different RAM densities available in the market.

It also provides you with an overview of needed RAM for running double framebuffer setup in 1, 2, 4, 8, 16, and 24 bits per pixel (dividing by 2 gives you the required density for a single framebuffer).

In some cases the single framebuffer setup is sufficient and in some STM32 microcontrollers, you have enough internal RAM for placing the framebuffer(s).

SDRAM & OctoSPI Densities				
16Mbit	32Mbit	64Mbit	128Mbit	256Mbit
2.000KB	4.000KB	8.000KB	16.000KB	32.000KB

SDRAM and OctoSPI Densities

Double frame buffering	480*272	640*480	800*480	1024*600	1024*768
1bpp	32 Kbytes	75 Kbytes	94 Kbytes	150Kbytes	192 Kbytes
2bpp	64 Kbytes	150 Kbytes	188 Kbytes	300 Kbytes	384 Kbytes
4bpp	128 Kbytes	300 Kbytes	375 Kbytes	600 Kbytes	769 Kbytes
8bpp	261 Kbytes	614 Kbytes	768 Kbytes	1.228 Kbytes	1.572 Kbytes
16bpp	522 Kbytes	1.228 Kbytes	1.536 Kbytes	2.457 Kbytes	3.145 Kbytes
24bpp	783 Kbytes	1.843 Kbytes	2.304 Kbytes	3.686 Kbytes	4.718 Kbytes

Required RAM for double framebuffer setup

Version: 4.16

Board Bring Up Introduction

Project activity

This chapter will help you through the board bring up phase of starting TouchGFX programming on a new platform. Bringing up the board means making sure that all necessary parts of the board and corresponding drivers are working correctly before TouchGFX is added to the mix.

If you already have a working board with a display, many of the activities in this phase will be easy. If you have a completely new custom made board, you should expect that this phase will take some days to complete. The work does normally pay off as an unstable platform makes it difficult to write good applications. A stable and proven platform on the other hand allows you to concentrate on the application.

This chapter is for you if are a developer with the task of ensuring that your hardware and low-level software components on the board are working as expected. This chapter is not for you if you are concerned only with developing the actual UI of your application.

In bringing up your board, you should have a thorough understanding of the components and peripherals on your board, the connections between all these, the protocols they communicate via and the driver code available and/or needed for each one.

The next chapter TouchGFX AL Development discusses how to create the abstraction layer that allows TouchGFX to run on top of your hardware and drivers.

Tools of the trade

Some important tools when bringing up an STM32 based board are listed in the table below.

Tool	Description
STM32CubeMX	An easy to use tool for configuration of the MCU and generation of initialization source code for a project and internal peripherals.

Tool	Description
STM32Cube Firmware Package	The Cube Firmware for your MCU family contains many example projects and applications that show how to use various peripherals.
Vendor datasheets	The datasheets for your external devices, like the display or flash, contain important information such as timing and voltage for correct initialisation of both MCU (e.g. through CubeMX) and the external device.
Vendor driver code	To save time you should request example code for your external devices from the vendor. The driver code often needs to be ported to your STM32 MCU, but this is often simpler than writing driver code from scratch.

All the work done in board bring up phase is not dependent on TouchGFX and should not involve any TouchGFX code. Instead the work and the resulting code will serve as a solid foundation for developing the TouchGFX Abstraction Layer.

The primary goal is to make sure that your hardware and lower-level software is in fact working in accordance with your expectations for the final application.

Verification of Functionality

The code produced during the board bring up, in form of one or more test projects will serve two purposes:

Abstraction Layer	The board initialization code is the foundation upon which we will build the TouchGFX AL and ultimately the working UI application
Test code	The verification code written accompanying the board initialization code, will be the ultimate place to go to if/when things do not act as expected. During the board bring up phase you will create a number of small verification programs ensuring that the board and each particular component are working as expected. These verification programs will be of great value when progressing, they can be revisited and enhanced if/when things become uncertain.

Due to the above two reasons it can be very beneficial to save the verification programs systematically. This will allow you to use the test programs again later. For example to test new revisions of your hardware or to find the root cause, when your larger applications are unexpectedly failing. It is also recommended to make systematic notes of any measurements, e.g. memory bandwidth, done during development.

Board Initialisation Code is the bases for the TouchGFX AL

Overall Process

Since we cannot know the concrete makeup of your hardware the following **How To** guide will be a general step-by-step guide to bring up and prepare boards to run TouchGFX.

Each step of the guide will be concerned with one particular part of your hardware and/or software components and the bring up of this. One example of a custom component part of your hardware could be the touch controller. The overall goal is to communicate with the touch controller to get information on any touches on the display. The specific commands to send to the touch controller depends on the specific controller you are using on your hardware, so the guide cannot provide the complete driver. For this you need to combine the guide with the information in your touch controller datasheet.

> **NOTE**
>
> **When reading and performing the step-by-step guide for your custom board bring up, we recommend that you:**
>
> - **do one step at the time**
> - **verify each step thoroughly before moving on**
> - **use the guide as a means for debugging, when something is not working as intended, or go back and revisit previous steps to make sure you did not break those steps**
> - **do not be alarmed if you experience unexpected behaviour - bringing up a board is a non trivial task**

Each step in the guide will follow the following structure:

- **Motivation**
  This part will explain the step and motivate why the step is an important step in preparing for running TouchGFX on your hardware.
- **Goal**
  The goal part lists the goals for this step. A list of verification points details the specific tests that you should perform. These verification points ensures that your software implements the requirements needed to running TouchGFX sucessfully on your hardware.
- **Prerequisites**
  Here we list items that are required to perform the tasks.

- **Do**
  This part explains as concrete as possible how to write the software required to configure and use the hardware. For some steps it is not possible to be very precise as the software depends a lot on the hardware you use. In that case this part lists the steps on a higher level and you must find the details relevant for your hardware yourself.

The individual how-to steps are:

Step	Content
Create Project	Create an empty project in CubeMX
CPU Running	Ensure that the MCU is running at the desired speed
Framebuffer in internal RAM	Allocate a framebuffer in internal RAM and transmit it to the display
External RAM	Enable the external RAM
Framebuffer in external RAM	Move the framebuffer to external RAM and transmit it to the display
External addressable flash	Enable external memory-mapped flash
External block mode flash	Enable external block-mode flash
Hardware acceleration	Enable the Chrom-ART graphics accelerator
Touch controller	Setup communication to the touch controller
Physical buttons	Configure access to physical buttons
Flash loader	Develop a way to write data to the external flash

Version: 4.16

1. Create Project

Motivation

In this section we will use CubeMX to generate a working project for a specific MCU. This project will be the basis for the rest of the steps in this how-to guide.

We will refine the project using CubeMX in coming steps and write and integrate code to make all required peripherals work.

This project will be long lived and should be kept. You should now decide on a strategy to keep the different versions available, so that you can go back and run them again. Either on new hardware or just to recheck the hardware.

Maybe you need many small test programs. In that case this project is a good starting point.

Goal

The goal is to create a project in CubeMX that can be flashed to your board and executed. If you have an IDE with a debugger (e.g STM32CubeIDE or IAR Embedded Workbench) you should also check that you can debug and step through your project on the MCU.

If you do not have a debugger you should find a way to print out debug statements from various places in your project. E.g. on a serial port.

Verification

Here are the verification points for this section:

Verification Point	Rationale
Project opens in IDE	The project was generated correctly by CubeMX and can be used as starting point for further board bring up development.
Project compiles	The project is setup correctly with drivers and include paths. We can write more code and recompile the project repeatedly.
Breakpoints are hit	The project can be debugged and stops at breakpoints. We can examine the project state and investigate errors.

Prerequisites

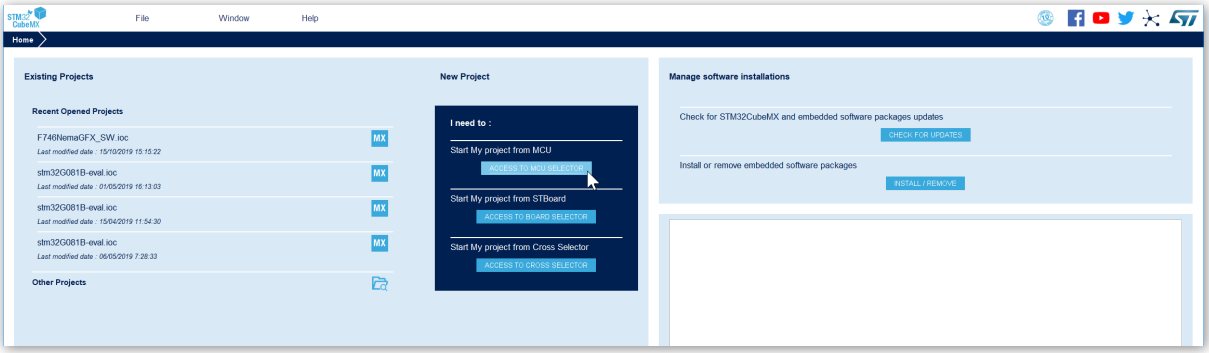
The following are the prerequisites for this step:

- STM32 based board
- Programming / debugging interface - ST-LINK, JLINK or similar
- CubeMX installed
- IDE installed - STM32CubeIDE, IAR Embedded Workbench, Keil uVision or similar

Do

We will now go through the steps of creating a new project in CubeMX. In this example we will use the STM32F429 MCU. You should of course select the MCU on your hardware.

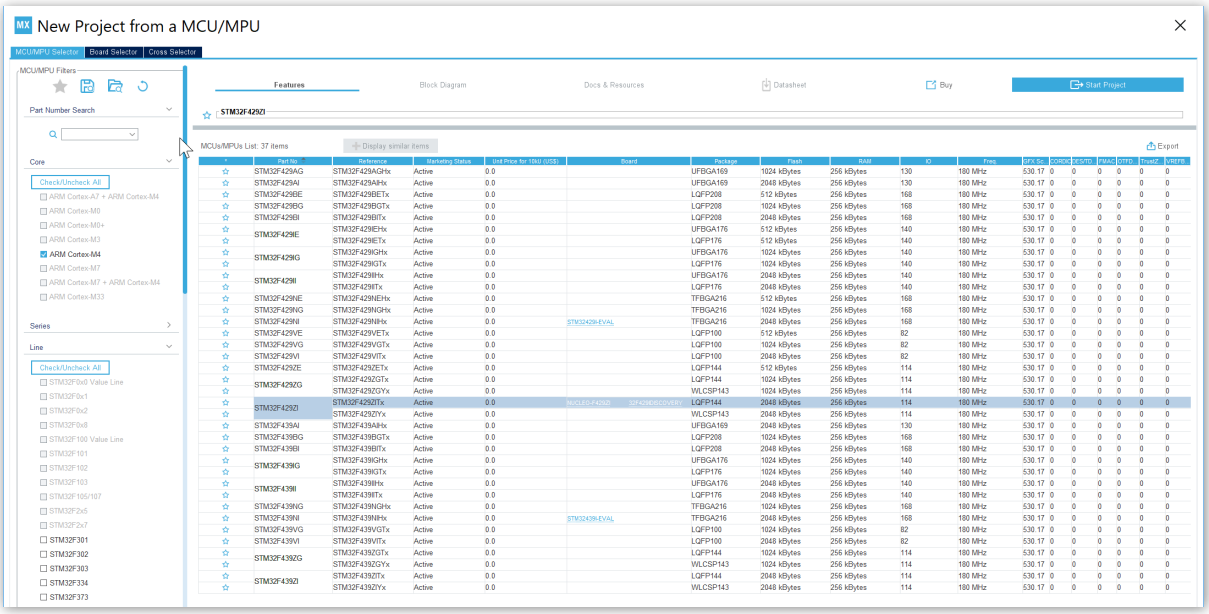
In CubeMX click "ACCESS TO MCU SELECTOR" in the "Start My Project from MCU":



Create new Project

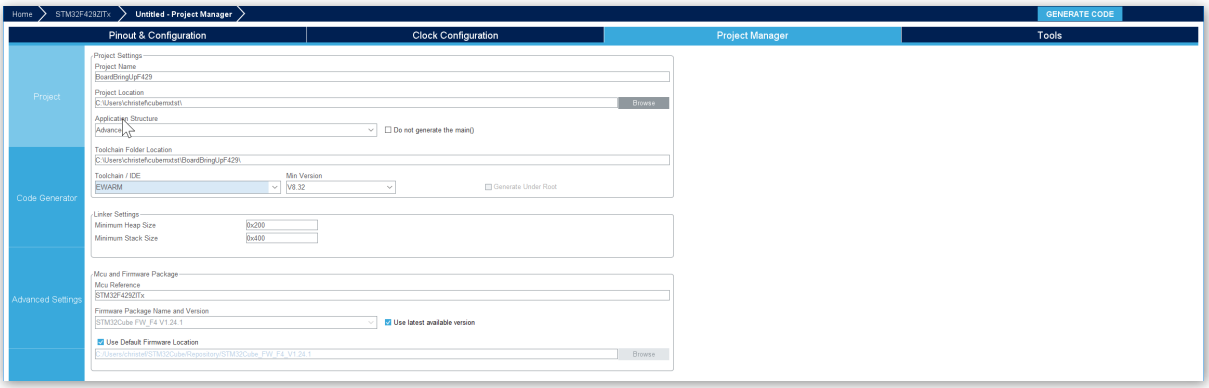
It is also possible to start a new project based on a STM32 evaluation kit, e.g. the STM32F429Discovery board. You can/should do this if your hardware design is based on one of the evaluation kits.

We then select the relevant MCU. Here we select the STM32F429ZIT6U:



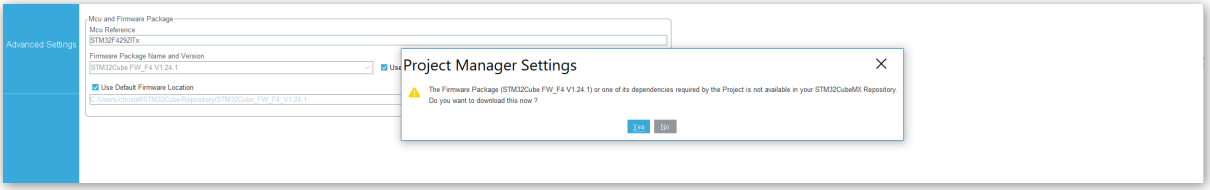
Select the MCU

Change to the "Project Manager" tab, and give your project a name. You can of course also select a new project location. Under "Application Structure", select Advanced. Under "Toolchain / IDE" you must select your IDE. For this example we select IAR:



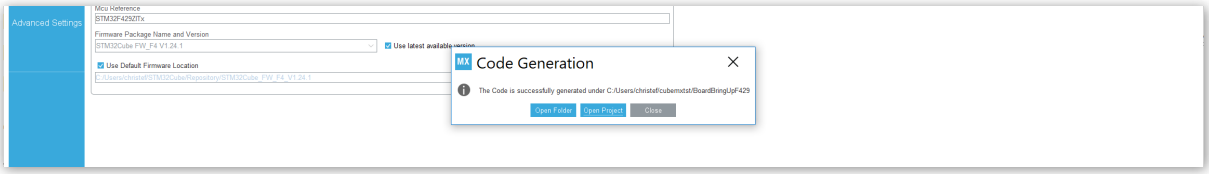
Set project name and IDE

Now click the "Generate Code" button in the upper right corner. If this is your first project for the selected MCU family (F4/F7/H7) CubeMX automatically proposes to download the relevant Cube Firmware package. Accept that to get the latest version for later use.

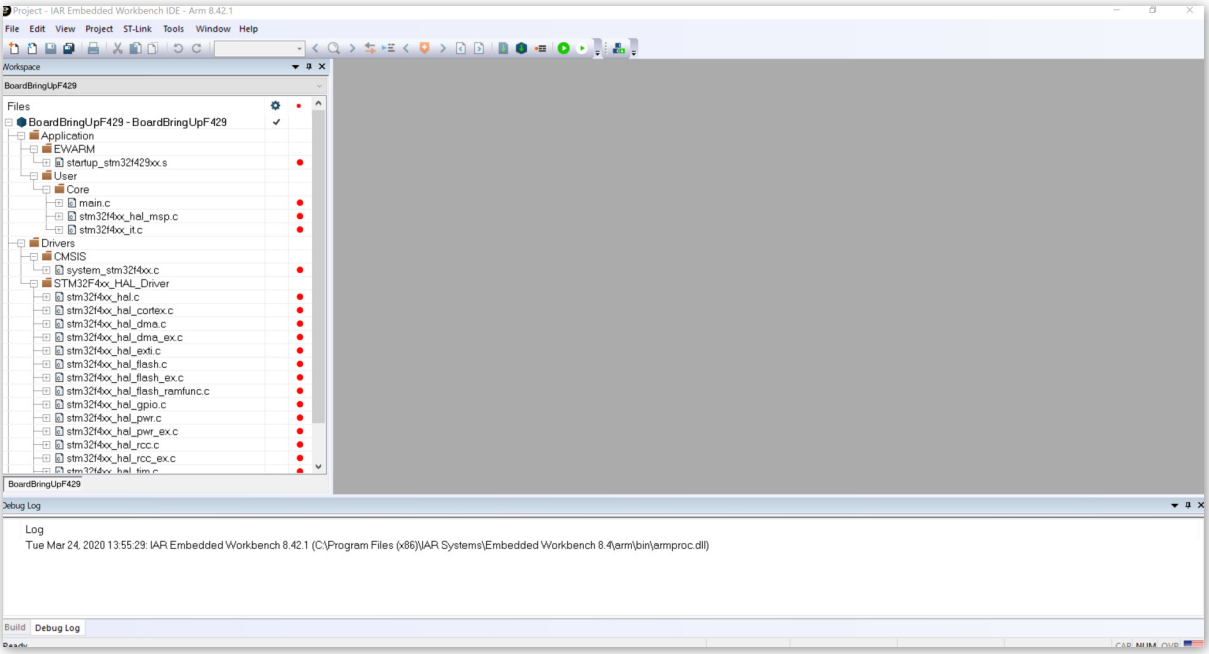


CubeMX can download Cube Firmware

Click "Open Project" to open the project in your IDE:



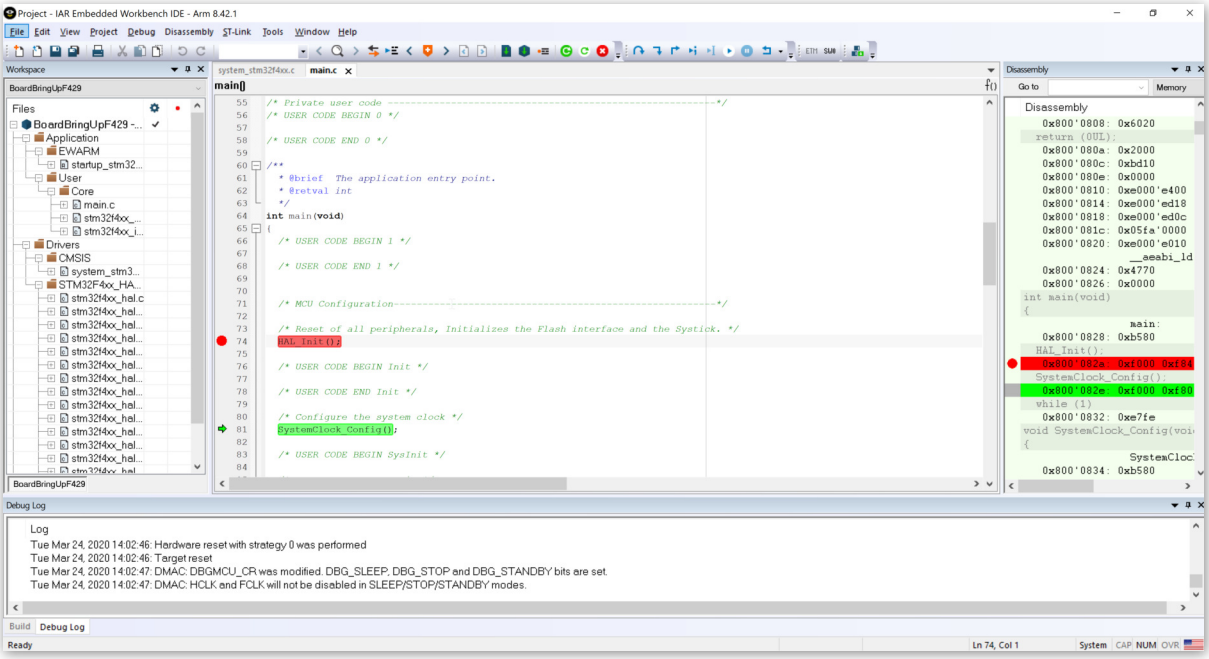
The project is generated



The project is opened in IAR Embedded Workbench

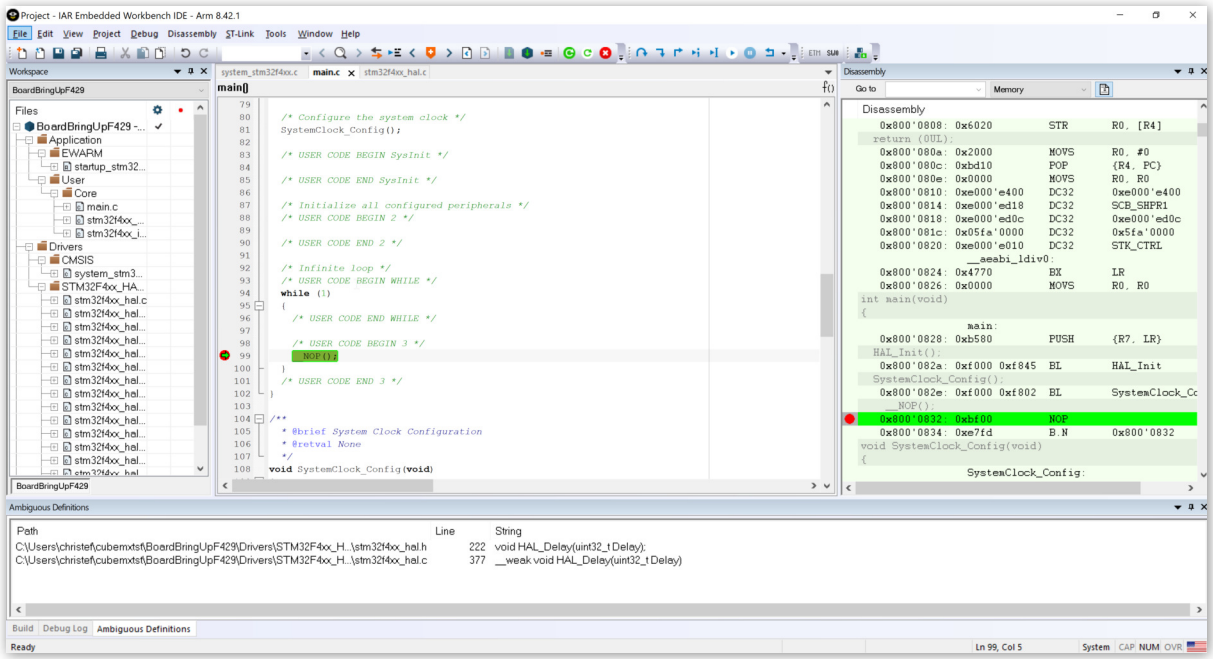
The project generated by CubeMX contains startup code specific to the MCU, interrupt table, system initialisation code, and HAL drivers for all the peripherals in the MCU.

The project can now be compiled and debugged from your IDE. In IAR we click Project->Make to compile the project, and Project->Download and Debug to debug the project:



Debugging the project in IAR Embedded Workbench

The IAR project is setup to use the STLink debugger. If you are using something else, then change the Debugger properties for the project in your IDE.



The main loop is running continuously

The while loop in main is typically important in projects running without an operating system. Check that you get there by setting a breakpoint and maybe add some code to the loop.

It is recommended to browse the project in your IDE to get familiar with the structure. Try also to step from SystemInit to main.

User Code sections

At this step it is important to understand the concept of "User Code sections" used by CubeMX before you start editing your project. All of the source files in the Core/Src folder in your project are generated by CubeMX. When you later change the project configuration in CubeMX, e.g. to enable a UART, some of these files will be regenerated. You have probably also inserted code in some of these files. Your code will be lost when CubeMX regenerates the project unless you follow one single rule:

- **Only write code in User Code sections**

Any code that you write outside of a User Code section will be deleted by CubeMX.

As an example, let us look at the first few lines in Core/Src/main.c:

main.c

```
int main(void)
{
  /* USER CODE BEGIN 1 */
```

```
  /* USER CODE END 1 */


  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  ...
}
```

If you want to insert code here in the beginning of the main function, you must put it between USER CODE BEGIN 1 and USER CODE END 1. If you put code outside this block it will be deleted by CubeMX.

⚠ CAUTION

Do not write code outside User Code sections. Such code will be removed when CubeMX generates code.

Further reading

The documents linked here contains more information about CubeMX:

ⓘ FURTHER READING

- **STM32CubeMX User Manual**
- **STM32CubeIDE resources**
- **Massive Open Online Course on STM32CubeMX and STM32Cube**

Version: 4.16

2. CPU Running

Motivation

In this section we will make sure that the MCU core, internal RAM and flash are running at the desired clock speed.

TouchGFX can run on any MCU speeds, but a wrong clock configuration can lead to lower than necessary performance. Later in your board bring up you need to configure specific timing parameters, e.g. an I2C clock for the Touch Controller. This is impossible without ensuring that the MCU runs with the correct speed.

For STM32 microcontrollers you setup up a system clock. This clock is then divided down to generate the FCLK core clock and various peripheral clocks like APB1 peripheral clock.

Goal

The goal for this section is to modify your project to get the correct clock configuration. You should also verify that your internal RAM and flash are running at the expected speed.

Verification

Here are the verification points for this section:

Verification Point	Rationale
SystemCoreClock variable's value is correct	The microcontroller is configured to run at the desired frequency.
Internal RAM is readable	The microcontroller has the expected amount of internal RAM, it is readable, and the speed is measured
Internal Flash is readable	The microcontroller has the expected amount of internal flash, it is readable, and the speed is measured
Caching is disabled	Running with caches disabled makes the system less complex and easier to understand.

Prerequisites

The following are the prerequisites for this step:

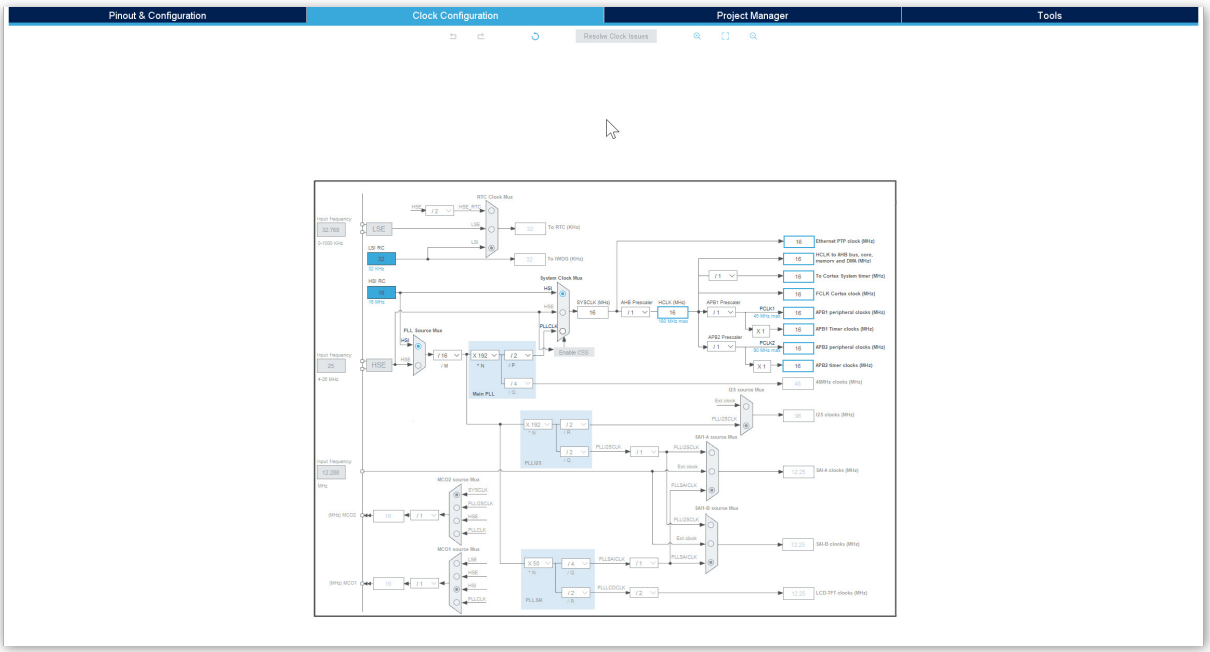
- Information about the clock source on your hardware. It is common to use a crystal, but other solutions are also possible.

Do

We will now go through the steps of adjusting the clock configuration of our project to get the required MCU frequency. Afterwards we will discuss how to measure read speed of the internal flash.

System Clock

In CubeMX click on the "Clock Configuration" tab. This gives you an overview of the clock tree for your specific MCU:



Clock Configuration

In this example the clock source is selected to be HSI. Many projects use an external crystal and must use HSE with a suitable divider (/M) and multiplier (/N). It is out the scope of this guide to advice on the clock configuration. After you have changed the clock configuration you must regenerate the project in CubeMX (click Generate Code in upper right corner).

The core clock (HCLK) can be calculated at runtime by the generated code and saved in a variable. This variable can be used by application code to correctly convert between clock cycles and seconds, and e.g start timers. To get the variable recalculated you must call the *SystemCoreClockUpdate()* function. Insert a call in main.c (in a user code section):

```
78   /* USER CODE END Init */
79
80   /* Configure the system clock */
81   SystemClock_Config();
82
83   /* USER CODE BEGIN SysInit */
84   SystemCoreClockUpdate();
85   /* USER CODE END SysInit */
86
87   /* Initialize all configured peripherals */
88   /* USER CODE BEGIN 2 */
```

SystemCoreClockUpdate

If we set a breakpoint at the end of that function we can see the core clock (according to the configuration):

```
280  /* Compute HCLK frequency --------------------------------------------------*/
281  /* Get HCLK prescaler */
282  tmp = AHBPrescTable[((RCC->CFGR & RCC_CFGR_HPRE) >> 4)];
283  /* HCLK frequency */
284  SystemCore
285  }
286
287  #if defined (DATA_IN_ExtSRAM) && defined (DATA_IN_ExtSDRAM)
```

uint32_t SystemCoreClock = 16000000 (0x00F42400)

SystemCoreClock

Another important point to test is the System Timer. This timer is running on HCLK divided down to give an interrupt every 1 ms. This timer is used by the Cube Firmware to implement millisecond delays.

We can test this by inserting a delay of e.g 5 seconds in main. Verify this with a stop watch or similar means:

```
92   /* Infinite loop */
93   /* USER CODE BEGIN WHILE */
94   HAL_Delay(5000);
95   while (1)
96   {
97     /* USER CODE END WHILE */
```

Measure delay

Flash and RAM size and speed

It is easy to check the reading speed of memory by using the System Timer. The System Timer interrupt increments a variable each millisecond. By reading this variable before and after a piece of code, we can measure the running time of the code (with 1 ms resolution). This scheme can be used to measure a time period in many different places in your application. It is not very precise, but can be done without external devices like oscilloscopes.

To do that we first need two volatile variable to save the result. If we don't save the result here, the optimizing compiler will in some case remove the measuring code:

```
45  /* USER CODE BEGIN PV */
46  volatile uint32_t result;
47  volatile uint32_t time;
48  /* USER CODE END PV */
```

Global volatile variables to hold measurement results

Here is an example where we read the flash from 0x08000000 to 0x08020000 (128 Kb) and time the code:

```
97  uint32_t* code = (uint32_t*)0x08000000;
98  uint32_t* stop = (uint32_t*)0x08020000;
99  uint32_t sum = 0;
100
101 uint32_t start = HAL_GetTick();
102
103 while (code != stop)
104 {
105   sum += *code++;
106 }
107 result = sum;
108 uint32_t end = HAL_GetTick();
109 time = end - start;
110
111 while (1)
112 {
```

Timing a read loop

You can use code like this to verify the speed of your different memories. Once you have created a setup in CubeMX you can measure the read speed and make a note of the result. The measurements can then be repeated later and verified. If you want to measure the bandwidth of your memory (the read speed in kb/s), you can compare the amount of data with the time measured.

On a 16 MHz STM32F429 the code runs in 12 ms giving us a read speed of the internal flash (using this method) of 128kb/0.012s = 10,666 kb/s.

The same loop above can easily be changed to verify that all the internal flash is enabled and readable. Just change the start and end addresses.

The code can also check the internal RAM. On the F429 the RAM starts at address 0x20000000. The core coupled memory is at 0x10000000. Check the datasheet for your specific MCU for the relevant memory addresses.

You should make a few measurements on your different memories and make a note of the result. For RAM test both the read and write speed.

Linker script

Another thing to look at is the linker script. This configuration file tells your linker what are the addresses of the RAM and flashes in your system. The linker script is generated by CubeMX together with the project, but it can be good to study it. Later you will in most cases have to modify it to suit your project's needs.

Cache on F7 and H7

The ARM Cortex-M7 based STM32F7 and STM32H7 microcontrollers include data and instruction caches. It is recommended to disable at least the data cache until you have a stable platform. The data cache improves the performance significantly in many cases, but it also introduces complexity during testing.

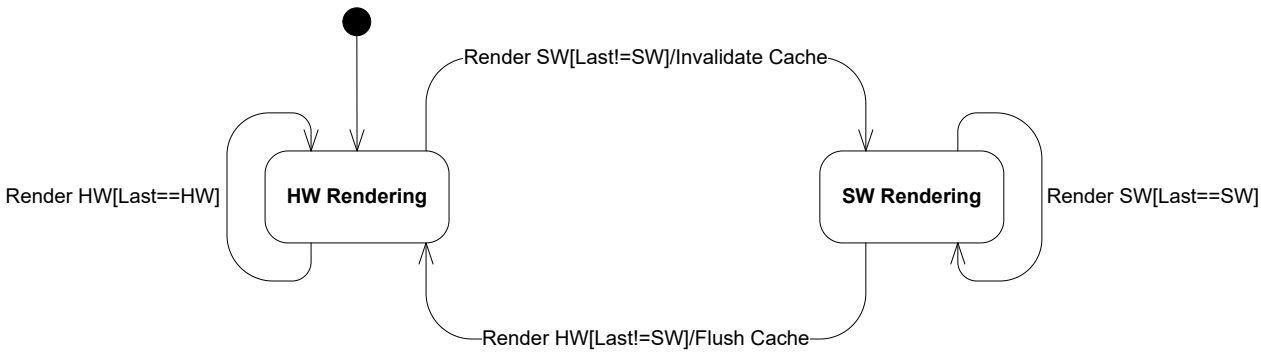
When you have a stable platform, you can enable the data cache. It is easier at that point to identify that a given problem originates from data cache management, since the platform is otherwise functional.

The complexity of the data cache comes from the fact that the MCU core reads and writes to the cache, whereas peripherals like DMA2 and LTDC read directly from memory (and not in the cache). For this reason, you can be in a situation where you write data to e.g. your framebuffer, but some of the data is not seen on the display. This is because the LTDC did not find the new data in the RAM because it is only written to the cache so far. The solution is to flush the cache at certain points in your project, but we recommend to deal with this at a later point.

Caching can be disabled/enabled in CubeMX in the System Core section.

TouchGFX internal DCache State Machine

TouchGFX engine keeps track of the current and last rendering operation, there are two states `HARDWARE` and `SOFTWARE`. The initial state is set to `HARDWARE` as the mijority of draw operations are done by hardware. When a state switch occurs the state machine will call the appropriate virtual function to handle cache invalidation. When the state transit from `HARDWARE` to `SOFTWARE` it will call the virtual method `void touchgfx::HAL::InvalidateCache()` and when the state transitions from `SOFTWARE` to `HARDWARE` it will call the virtual method `void touchgfx::HAL::FlushCache()`. The functionality of these two functions is left for the user to implement in the derived HAL class.



TouchGFX engine internal DCache State Machine

If using TouchGFX Generator the implementation of these derived methods will be created in the TouchGFXGeneratedHAL class with function calls to DCache invalidation and no further action is needed.

Further Readings

The documents linked here contains more information about CubeMX and the STM32 caches:

FURTHER READING

- Section on Clock configuration in the **STM32CubeMX User Manual**
- **Level 1 cache on STM32F7 and STM32H7**

Version: 4.16

3. Framebuffer in internal RAM

Motivation

In this step we will see the display come to life by transferring pixel data from the internal RAM to the display. This step ensures that we can transmit data to the display and that we can continuously update the contents of the display.

In addition to transferring image data to the display we must also make sure that we can continuously send new data to the display without seeing errors on the display. We are also going to measure the speed of the transfer as this has influence on the frame rate we can obtain with the display.

We will place a framebuffer in internal RAM as we know from last section that this RAM is both readable and writable. We will update and transfer this framebuffer to the display repeatedly.

Recall that the size of the framebuffer is calculated by this formula:

width x height x bpp

So, for example, a common 16 bit display with resolution 480 x 272 will take up 480x272x16/8 bytes = 261120 bytes.

If the display size implies a framebuffer too large to be stored in internal RAM, you should not skip this step. Instead configure the display controller to only update a part of the display. This way we can tune the amount of RAM needed for the framebuffer and make it fit internal RAM.

The type of display interface has a large impact on the setup and code needed to transfer the framebuffer. In this section we will first target a display connected to the LTDC. If you are using e.g. a SPI display, the code will be very different, but the tasks and goals are the same.

Goal

The goal in this section is to transfer a framebuffer to the display. You should also verify that you can modify the framebuffer content and resend the framebuffer continuously.

Verification

Here are the verification points for this section:

Verification Point	Rationale
Framebuffer is shown	Display controller or SPI is configured and running
Updated framebuffer is shown	We know how to repeatedly transmit the framebuffer
Colors are correct	The GPIOs are correct (LTDC) or the data format of the display matches our framebuffer
Framerate is correct	The pixel clock and porches are configured to get the required framerate

Prerequisites

The following are the prerequisites for this step:

- Information about the display, typically a datasheet
- Information about the connections between the MCU and the display.

Do

Depending on the display type, the needed setup differs. But for all display types we need a framebuffer in internal RAM. An easy way to allocate that memory is to just declare a global array with the correct size:

main.c

```
uint16_t framebuffer[480*272];  //16 bpp framebuffer
```

If your internal RAM is not big enough to hold the array, declare a array corresponding to a smaller resolution, say 480x200.

The method to transfer the framebuffer to the display depends on the display type. We will look at this now.

Parallel RGB Displays

We will first discuss a parallel RGB display connected to the LTDC controller on the MCU.

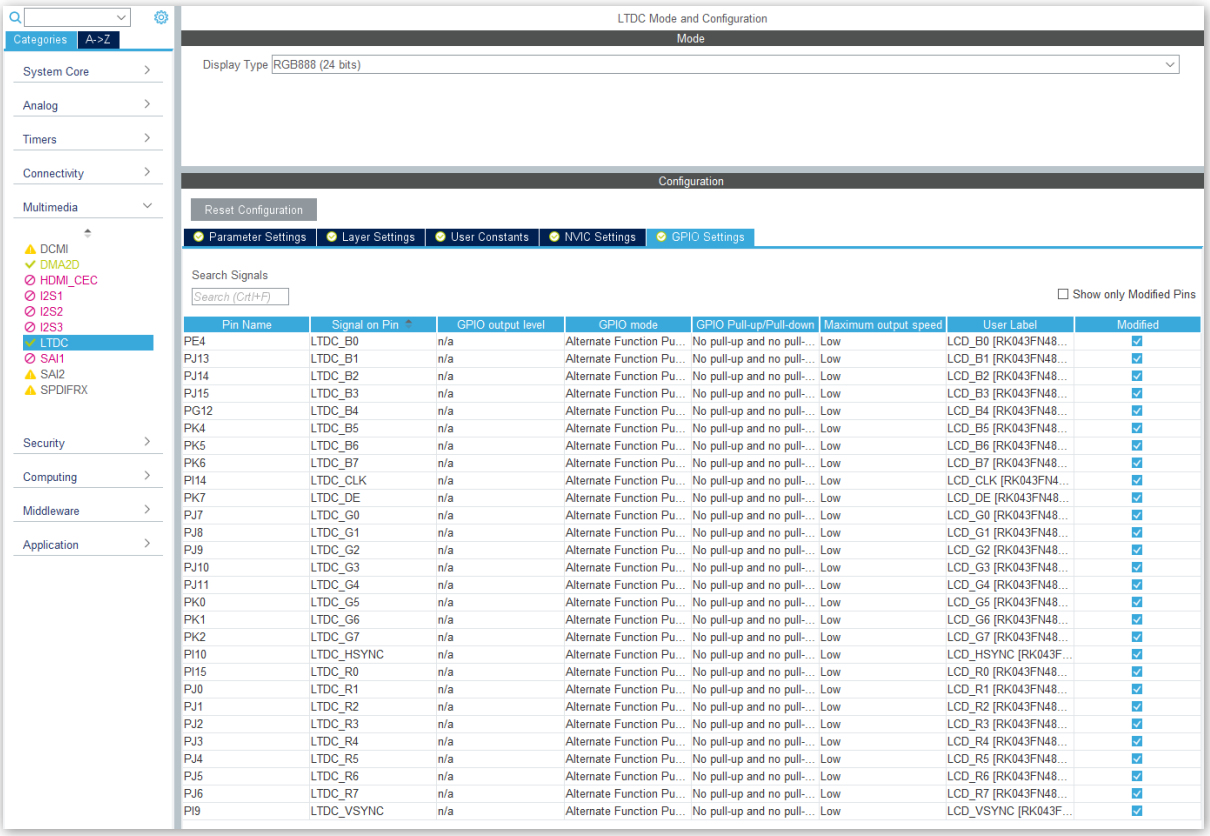
The configuration tasks for a display like this are:

- Configure the GPIO connections to the display
- Configure the LTDC controller
- Configure the LTDC pixel clock
- Setting the framebuffer address
- Check the framerate

As an illustrative example we will use a STM32F746Discovery evaluation kit. This board features a 480*272 display.

Display GPIO

This display is running in 24 BPP mode, so we configure the 24 GPIOs for the connection between the LTDC and the display. This is most easily done in CubeMX under Multimedia -> LTDC -> GPIO Settings:



Configuring display GPIOs

Besides the 24 GPIOs for the pixel transfer (e.g. LTDC_B0) we also configure the 4 display control signals:

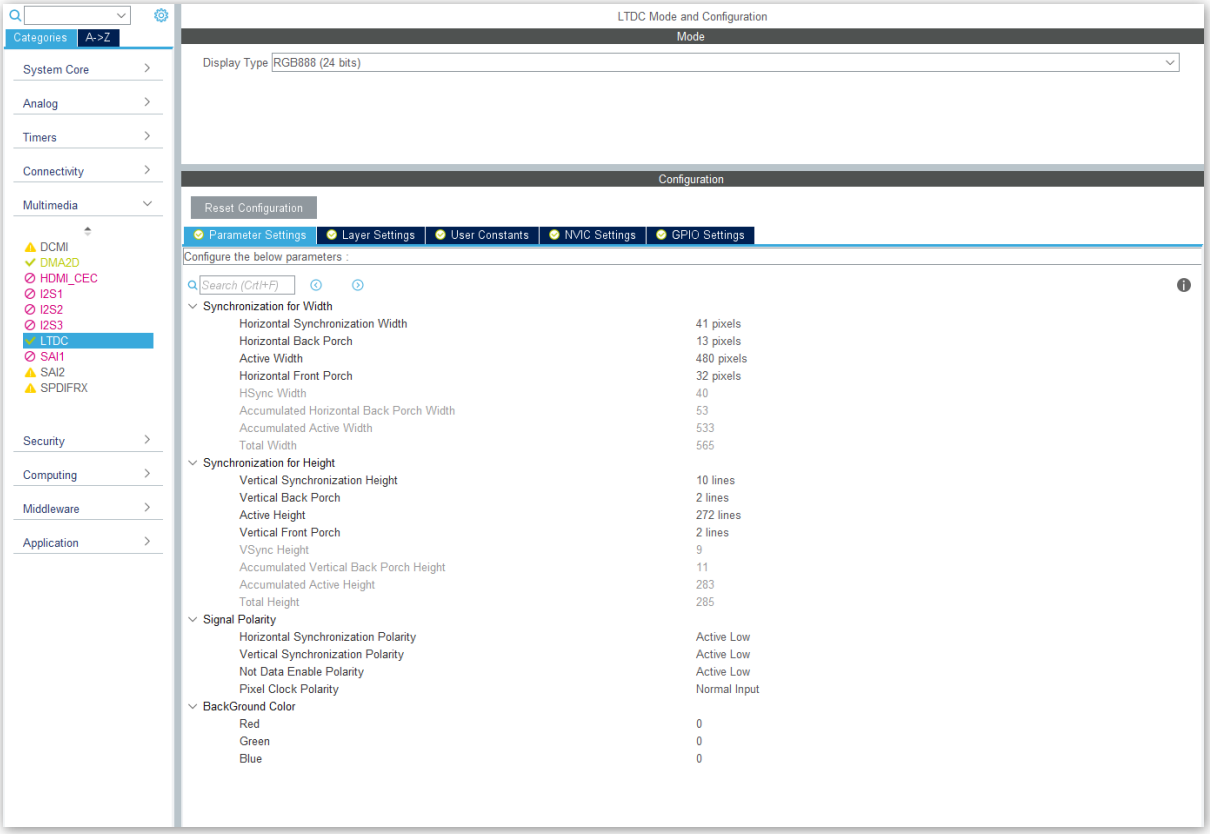
Signal	Function
LTDC_CLK	Pixel clock. Signals to the display when to sample the pixels from the 24 lines
LTDC_DE	Data enable. Pixels are transferred when active

Signal	Function
LTDC_HSYNC	Horizontal synchronisation. Allows the display to find the pixel line start
LTDC_VSYNC	Vertical synchronisation. Allows the display to find the frame start

Check your hardware design and make the corresponding configurations.

LTDC Configuration

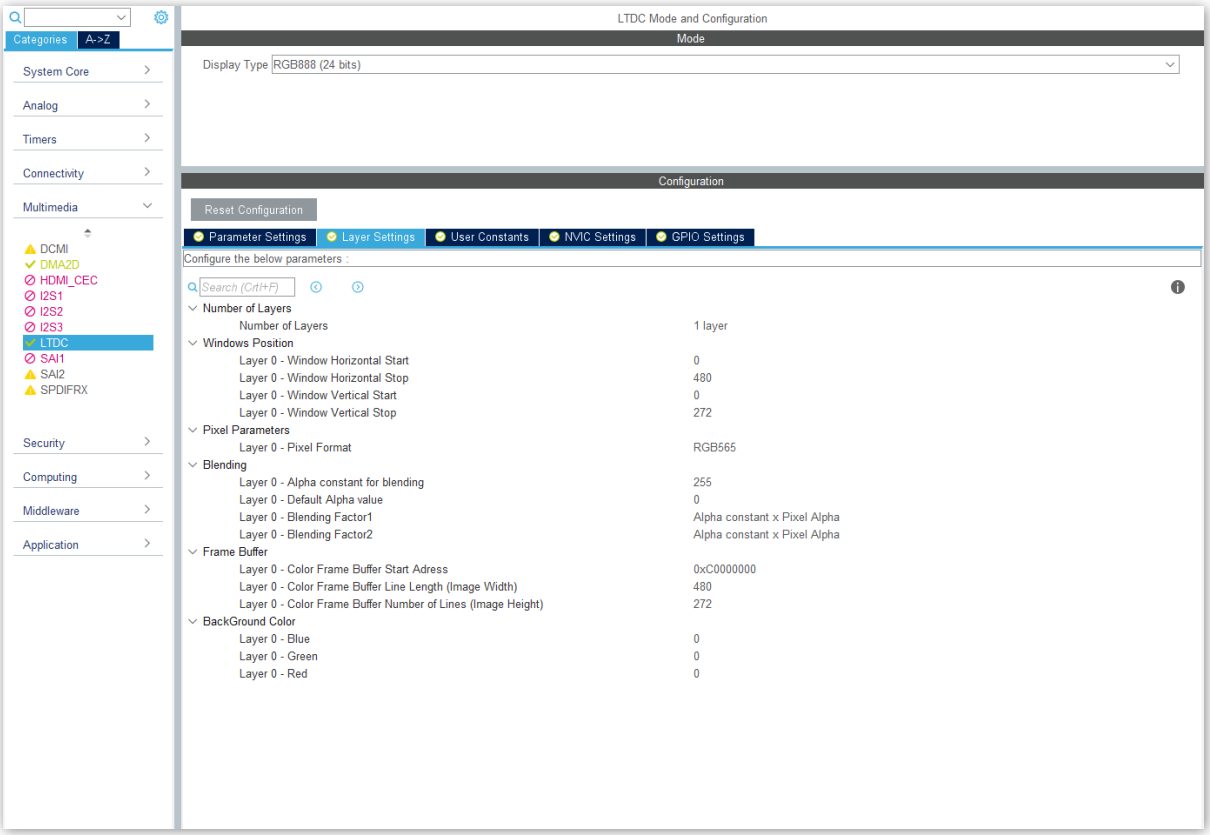
The LTDC configuration is found in CubeMX under Multimedia -> LTDC -> Parameter Settings:



Configuring LTDC Parameters

The active width and height corresponds to the resolution of your display. Check your display datasheet for the synchronization pulse widths and the porch widths. Also pay attention to the signal polarities. The values shown in grey are computed from the other values. These values are written to the LTDC registers (and can be found in the code).

Now go to the LTDC Layer configuration under Multimedia -> LTDC -> Layer Settings:



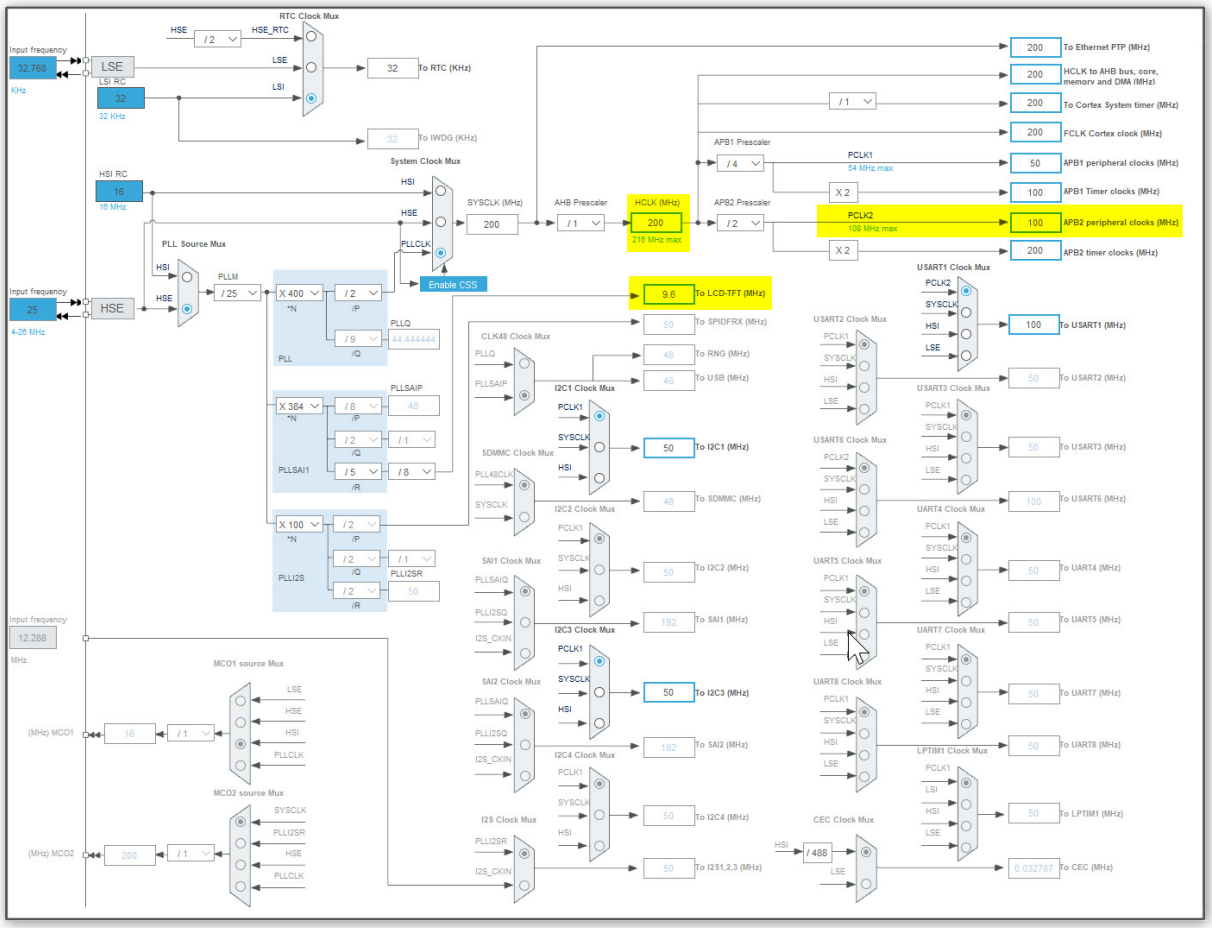
Configuring LTDC Layer Parameters

For this test and in TouchGFX in general we will only use one layer. The resolution of Layer 0 should match the framebuffer dimension. The framebuffer address needs to be set later, so just leave the address unchanged here.

If you declared a framebuffer array smaller than the display resolution, then adjust the layer size to match the framebuffer dimension. The LTDC will transmit the background color for the display pixels not available in the framebuffer. It is recommended to set the background color to something recognisable like red (Blue: 0x00, Green: 0x00, Red: 0xFF).

Clock Configuration

The clock configuration is also important. The clock must be enabled for all the GPIOs and the LTDC. The pixel clock must be in the range acceptable by the display.



Clock configuration

The LTDC depends on 3 clocks: HCLK, PCLK2, and LCD_CLK.

Setting the Framebuffer Address

In CubeMX we configured the framebuffer address of layer 0 to 0xC0000000. We need to change that to the address of our array in internal RAM. This is easily done by using one of the Cube Firmware HAL functions:

main.c

```
/* USER CODE BEGIN 2 */
HAL_LTDC_SetAddress(&hltdc, framebuffer, LTDC_LAYER_1);
/* USER CODE END 2 */
```

Layers are numbered 1, 2, in the HAL functions, but 0, 1 in CubeMX. The LTDC is otherwise fully configured by the code generated by CubeMX in the function *MX_LTDC_Init(void)*.

The LTDC controller transmits the framebuffer to the display repeatedly. The image displayed depends on the values in the framebuffer. Try different values or patterns in the framebuffer. Use e.g. memset to clear the framebuffer to 0xFF to get a white display.

NOTE

On some displays backlight must be turned on to make the frame visible.

Check the Framerate

The LTDC controller raises an interrupt for each frame. This interrupt will be used to drive the application forward.

You should use a debugger to check that this interrupt is raised.

The time between these interrupts is the sum of clocking all the pixels and the porches. You can adjust the porches to adjust the framerate. The porches were part of the LTDC configuration. It is custom to lower the framerate by enhancing the vertical front porch.

An easy way to measure the framerate is to use the HAL_GetTick() in the interrupt handler:

stm32f7xx_it.c

```
volatile int last = 0;
volatile int diff = 0;
void LTDC_IRQHandler(void)
{
  /* USER CODE BEGIN LTDC_IRQn 0 */
  int now = HAL_GetTick();
  diff = last - now;
  last = now;
  /* USER CODE END LTDC_IRQn 0 */
  HAL_LTDC_IRQHandler(&hltdc);
  ...
```

Remember 60 frames per second should have 1000 ms / 60 = 16 ms between each frame.

SPI Display

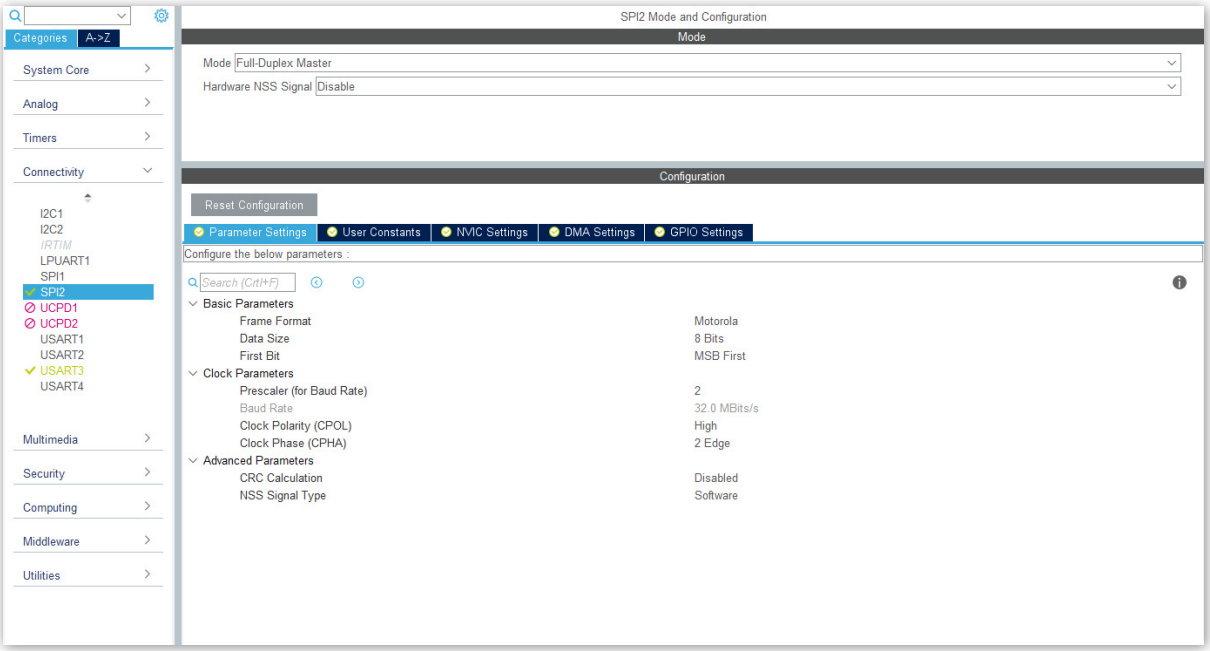
We will now discuss a display connected with an SPI bus.

The configuration tasks for a display like this are:

- Configure the SPI peripheral and GPIOs
- Check the clocks
- Write or find the necessary driver code

SPI Configuration

Start in CubeMX and enable the SPI. The images here are from an STM32G081 project:

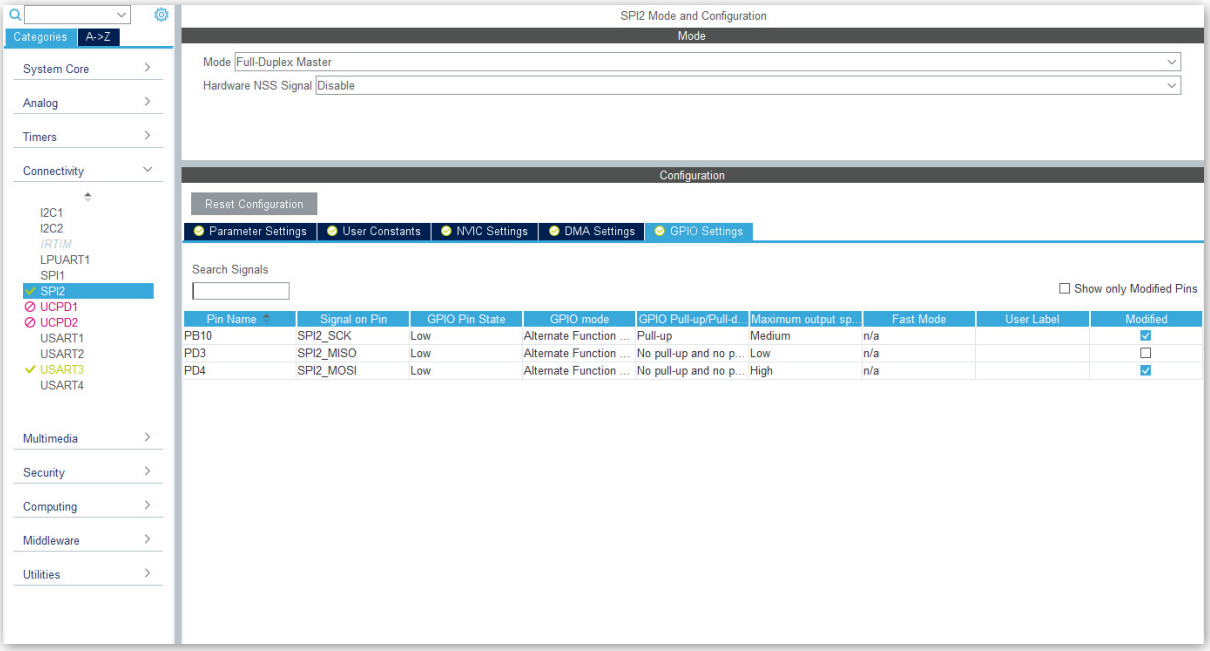


SPI configuration

Check the display datasheet for SPI format used (data size and bit order). Remember the 16 bit words are stored in little endian byte order in the framebuffer. Check if you can configure the display to accept this format. If not, then you have to convert data during transmission.
Also pay attention to the clock polarity and clock phase. These are also specified in the display datasheet.

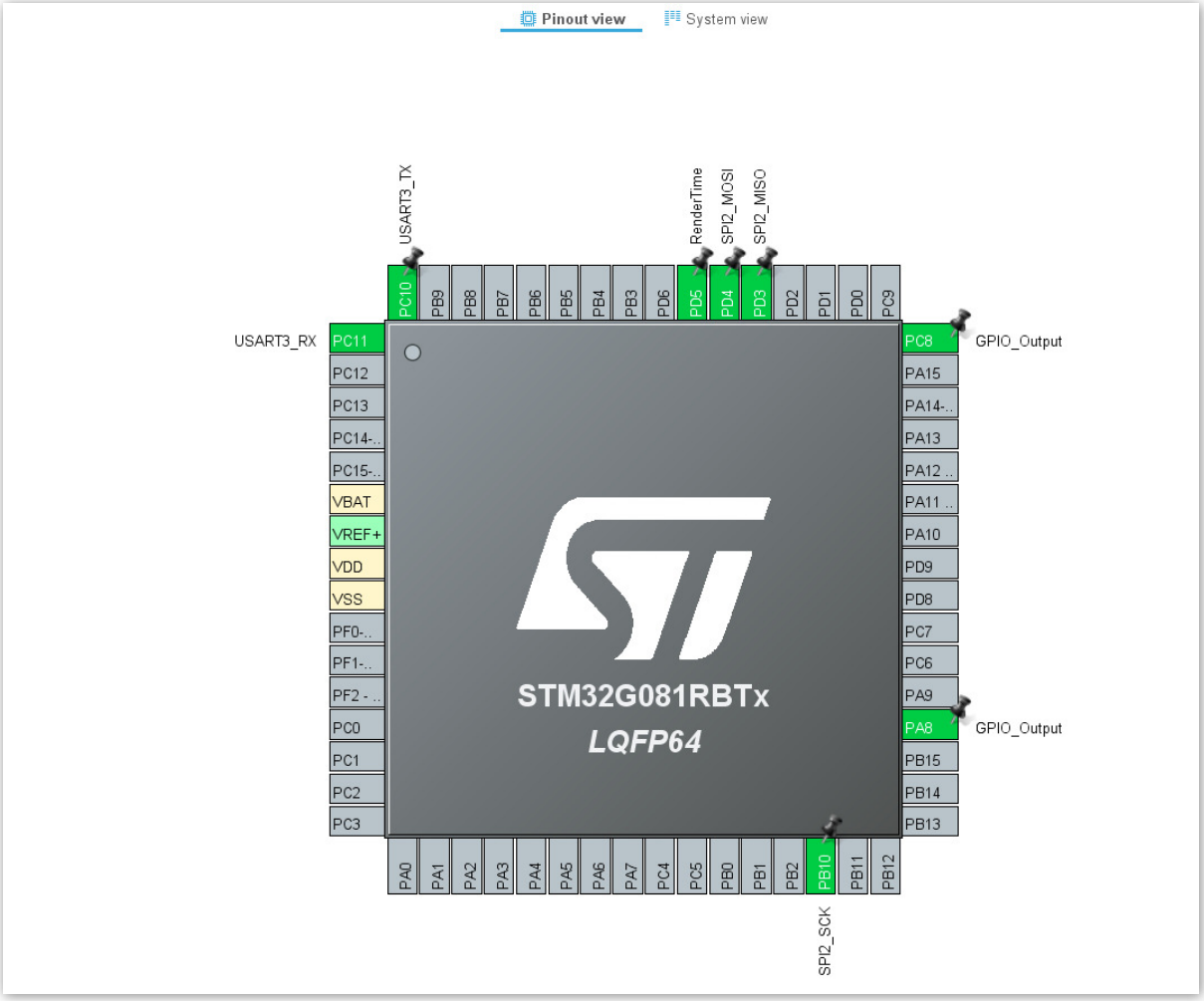
The SPI clock (the bit rate) is controlled by a divider to the FCLK. The minimum divider is 2. If the MCU is running e.g. 64 MHz, the maximum SPI bit rate will be 32 MBit/s.

On the GPIO tab you can check the GPIO selection for the SPI peripheral:



SPI GPIO configuration

Select the GPIOs on Pinout view on the right:



SPI GPIO selection

What is left now is to configure the display and transfer the framebuffer on the SPI channel. CubeMX cannot generate this code for you, as it depends heavily on the display.
For many displays it is necessary to send a sequence of commands and follow a specific power up sequence. After that you typically set the color mode and turn the display to ON. All this should be specified in the datasheet or examples provided by the vendor.

The Cube Firmware contains examples using SPI communication. The Cube HAL contains various helper functions. The basic function to send data is:

stm32g0xx_hal_spi.h

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size,
```

We recommend using these functions until communication is running stable. Afterwards performance can sometimes be improved by writing dedicated functions.

An SPI aware oscilloscope or SPI to usb logger can be very helpful in the process of writing a SPI display driver.

NOTE

Start with a low frequency on SPI to avoid noise problems.

Checking the Display Colors

At this point where you can transmit a framebuffer to the display, it is adviceable to thoroughly check the display colors.

The idea is to write a color to the framebuffer and check the display by visual inspection. Here are a few examples:

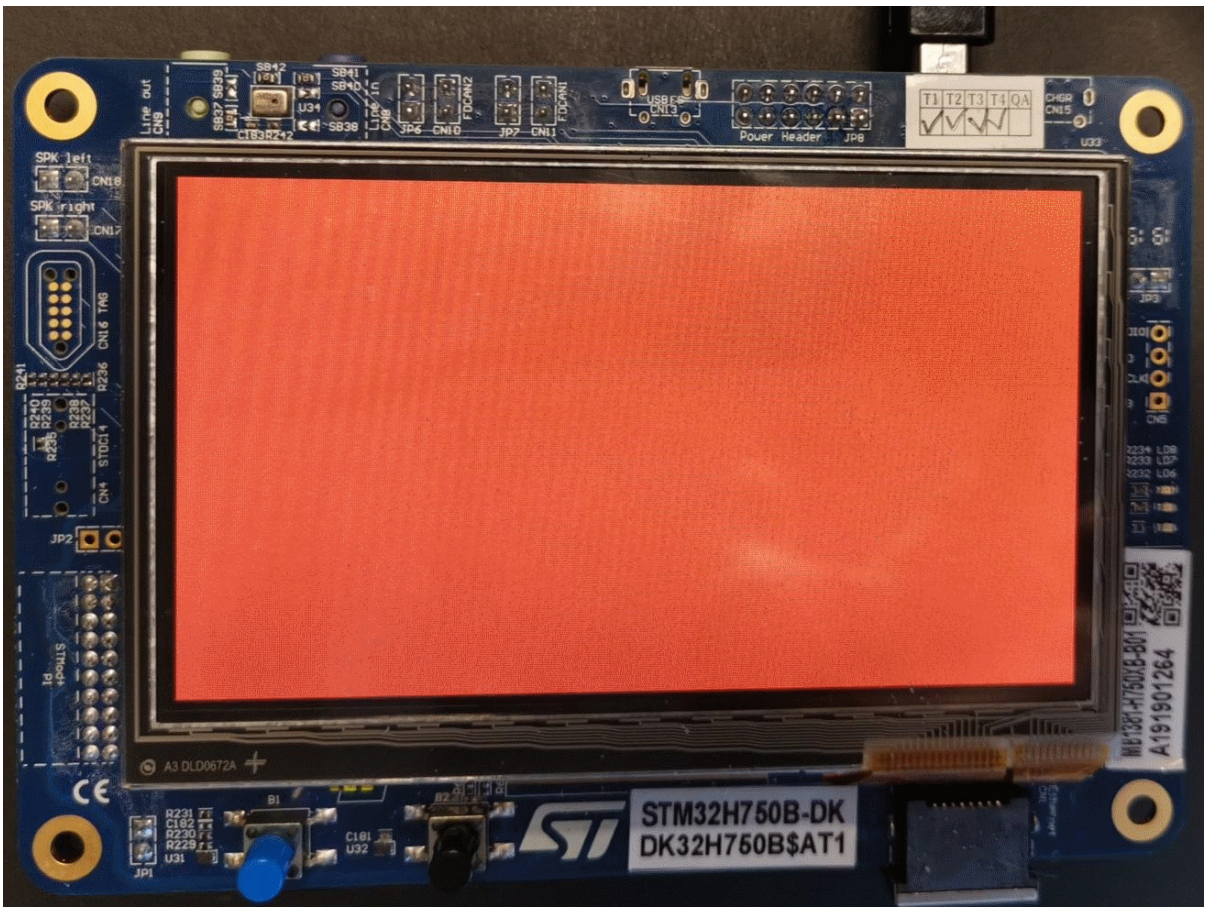
Test	Description
Red	Set red color in the framebuffer. The display must be red also.
Green	Set green color in the framebuffer. The display must be green also.
Blue	Set blue color in the framebuffer. The display must be blue also.
Dark color	A dark color (e.g. 0x8000) for 50% red, must be dark on the display.
Changing color	Change the framebuffer every second and see that the display also updates.

To put a color in the RGB565 framebuffer, the following scheme can be used:

```
uint8_t r    = 0xff, g = 0x00, b = 0x00;             // Solid red
uint16_t col = ((r>>3)<<11) | ((g>>2)<<5) | (b>>3);  // Convert colors to RGB565
// put colors into the framebuffer
for(int i = 0; i < W*H; i++) {
  framebuffer[i] = col;
}
```

For a 24BPP display the code is better formulated using byte pointers (colors are stored in BGR order):

```
uint8_t* framebuffer[480*272*3];  //24 bit framebuffer
...
uint8_t *fb = framebuffer;
while(fb < (uint8_t*)(framebuffer + (480*272*3))) {
  *fb++ = 0x00; // Write blue color and increment pointer by one byte
  *fb++ = 0x00; // Write green color
  *fb++ = 0xFF; // Write red color
}
```



Showing a colored framebuffer

Version: 4.16

4. External RAM

Motivation

In this step we will enable the external SDRAM. External RAM is often required in graphical applications as the framebuffer is too big to fit into the internal RAM in many resolutions. Some applications use two or three framebuffers, making external RAM even more a necessity.

> **NOTE**
>
> **Skip this step if external RAM is not relevant for your board bring up.**

When the framebuffer is to be placed in external RAM it is important that we ensure that the external RAM

- Can be read and written.
- Runs at desired (typically maximum) speed.

Goal

The goal in this section is to enable the external RAM and read and write data from it.

Verification

Here are the verification points for this section:

Verification Point	Rationale
External RAM is readable	External RAM can be used for framebuffer location
External RAM writable	External RAM can be used for framebuffer location
External RAM Performance	Graphics performance is acceptable with framebuffer in external RAM

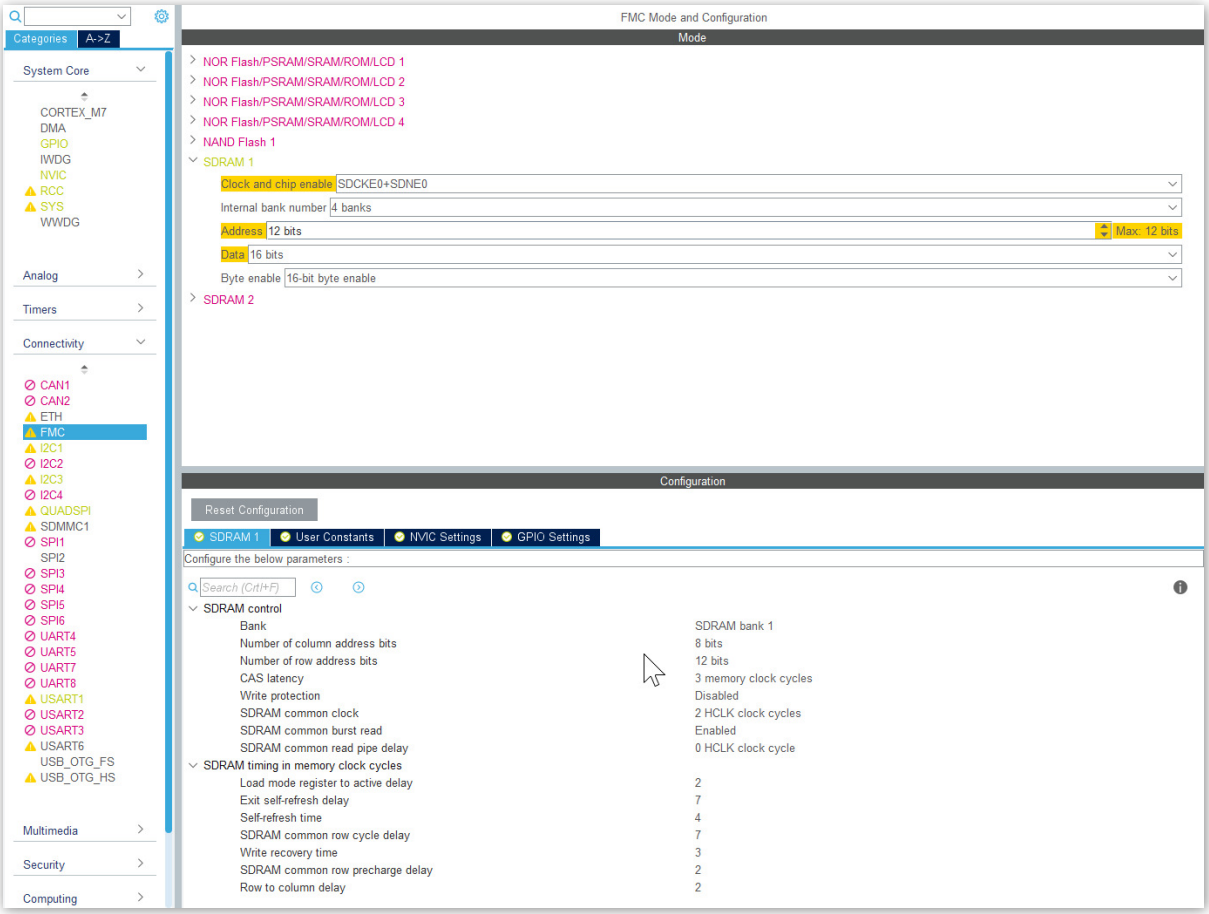
Prerequisites

The following are the prerequisites for this step:

- Information about the RAM, typically a datasheet
- Information about the connections between the MCU and the external RAM

Do

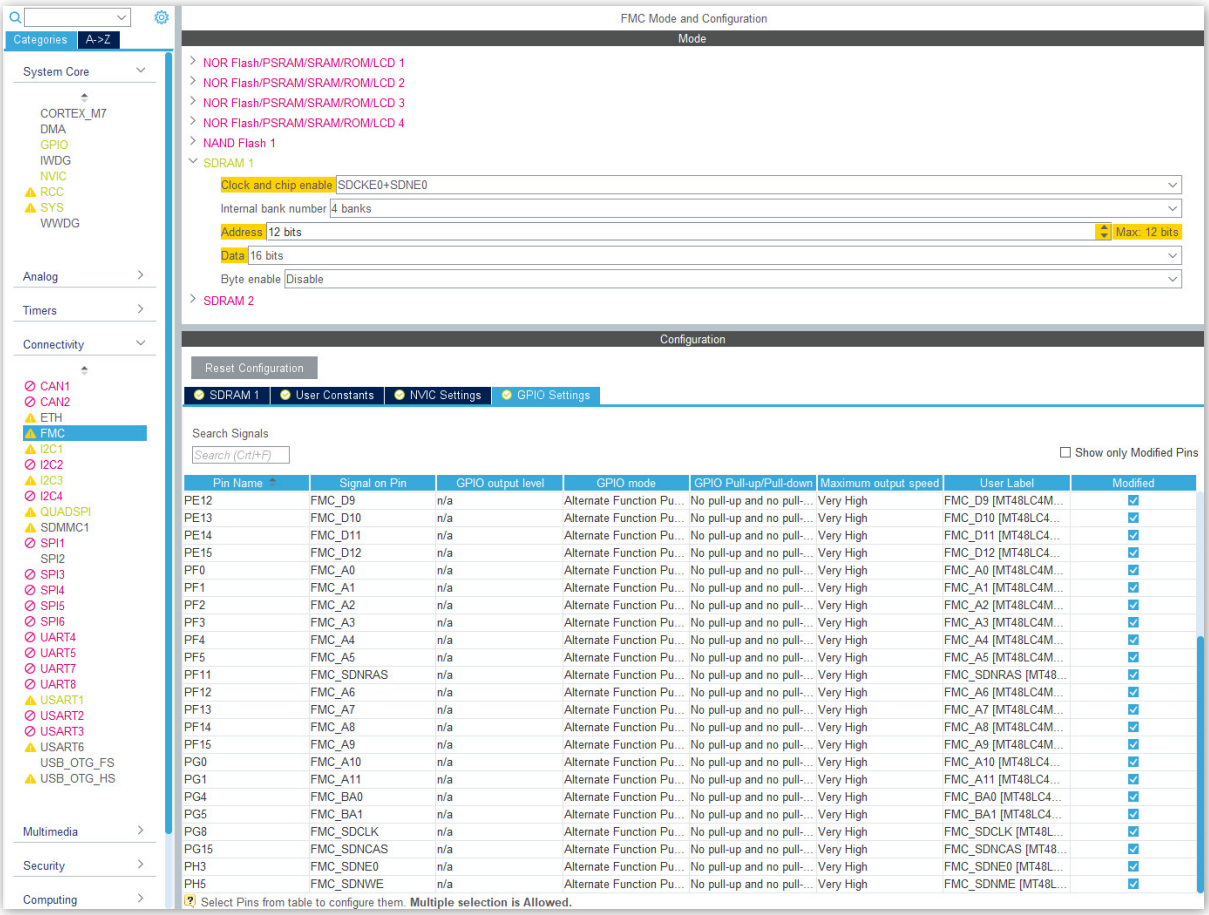
The external SDRAM controller is configured in CubeMX under Connectivity -> FMC -> SDRAM1:



Configuring SDRAM

The AHB clock (HCLK) is reference clock for the FMC memory controller. Check the clock frequency under "Clock Configuration" and use that to calculate the various SDRAM clock cycles.

Remember to configure all the GPIOs used for the SDRAM:



Configuring SDRAM GPIO

Further configuration

For some RAM chips it is necessary to do additional device specific configuration. This cannot be configured in CubeMX, but must be done in the C code. The Cube HAL contains functions to send commands to the device. Here is an example:

main.c

```
FMC_SDRAM_CommandTypeDef Command;

/* Step 1: Configure a clock configuration enable command */
Command.CommandMode            = FMC_SDRAM_CMD_CLK_ENABLE;
Command.CommandTarget          = FMC_SDRAM_CMD_TARGET_BANK1;
Command.AutoRefreshNumber      = 1;
Command.ModeRegisterDefinition = 0;

/* Send the command */
HAL_SDRAM_SendCommand(&hsdram1, &Command, SDRAM_TIMEOUT);
```

Testing the RAM

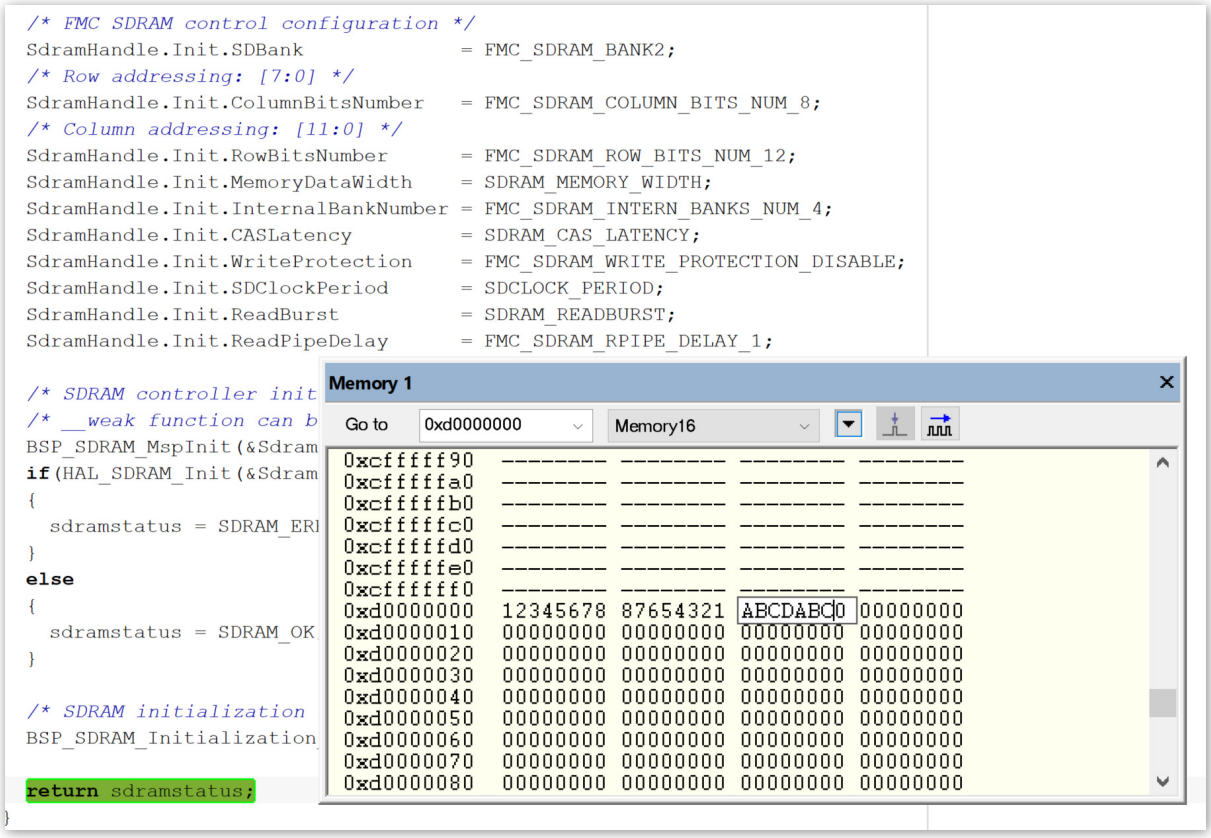
After configuring the external RAM it is important to test it. We should test at least the following:

- RAM is visible in the debugger
- RAM is readable and writeable in the whole range
- Performance is as expected

The memory controller uses a fixed address mapping of external memories based on their type. Check the datasheet for your microcontroller for the addresses. SDRAM is typically mapped to 0xC0000000 (bank1) or 0xD0000000 (bank2).

Test RAM is visible in the debugger.

The first test when the RAM is enabled is to access it from the debugger. This will easily show if you can read and write to the memory. Just open a memory viewer on the address:



Testing memory in Bank2 at 0xD0000000 in the debugger

RAM is readable and writeable in the whole range

The next test is to write small programs to write more data to the external memory. Preferably test the whole memory. Here is a starting point:

```
uint32_t *externalRAM = 0xC000000;
const uint32_t size = 1000;

//write external RAM
for(int i = 0; i < size; i++)
{
    externalRAM[i] = i;
```

```
}
```

Now check the memory again in the debugger. This can reveal some types of error, for example if some of the address pins are not connected or exchanged. You should also try with different value patterns. Writing low numbers like 0, 1, 2, 3, will not reveal if some of the data pins are bad.

We can also read the memory with a little program:

```
uint32_t *externalRAM = 0xC000000;
const uint32_t size = 1000;

//read external RAM
for(int i = 0; i < size; i++)
{
    ASSERT(externalRAM[i] == i, "external RAM not as expected");
}
```

Remember that a test like this will not tell if the addresses are incorrect.

Test all memory cells. Either by running a longer loop, or by changing the starting address.

Performance is as expected

We need now to test the performance of the external RAM. The performance is important when the framebuffer is in external memory. A slow memory will degrade the graphics performance of your system.

Test the speed of reading, writing, and modifying the RAM. Typically, a graphics application copies a lot of data from one memory to another. There will be a lot of writing to the framebuffer during draw operations, and a lot of reading when transmitting to the display. We can mimic that in our test programs:

```
volatile uint32_t *externalRAM = 0xC000000;
uint32_t sourcedata[10000];
const uint32_t size = 10000;

int begin = HAL_GetTick();
//write external RAM
for(int i = 0; i < size; i++)
{
    externalRAM[i] = sourcedata[i];
}
int end = HAL_GetTick();
```

```
int begin = HAL_GetTick();
//Read external RAM
for(int i = 0; i < size; i++)
{
    sourcedata[i] = externalRAM[i];
}
int end = HAL_GetTick();
```

Graphics software reads and writes data in the framebuffer when e.g. blending an image on a background.

```
//Time modifying external RAM
int begin = HAL_GetTick();
for(int i = 0; i < size; i++)
{
    externalRAM[i] += 2;
}
int end = HAL_GetTick();
```

Depending on your memory speed and the accuracy you would like, you may like to loop the tests, say 100 times, to make the results more reliable.

If the external RAM is clocked too fast it can result in incorrect values during read or write operations. This can be difficult to see with these simpler tests, but will be visible on the display.

Version: 4.16

5. Framebuffer in external RAM

Motivation

In this step we will move the framebuffer from internal to external RAM, and make sure that the framebuffer can still be transferred to the display.

> **NOTE**
>
> **Skip this step if external RAM is not relevant for your board bring up.**

This step will stress test the external RAM since the display controller has certain expectations on the transfer speed. This might result in errors. Common errors are LTDC underrun, because the bandwidth of the external RAM is not high enough, or pixel errors because the RAM is configured incorrectly and is running "out of spec".

Goal

The goal in this step is to remove the framebuffer array from internal RAM and use a framebuffer in external RAM.

Verification

Here are the verification points for this section. These are similar to the verification points when the framebuffer is in internal RAM, but should be checked again, as the speed on the external memory may influence the transmission of the framebuffer to the display.

Verification Point	Rationale
Framebuffer is shown	Display controller or SPI is configured and running
Updated framebuffer is shown	We know how to repeatedly transmit the framebuffer
Framerate is correct	The pixel clock and porches are configured to get the required framerate

Prerequisites

The following are the prerequisites for this step:

- Address of the framebuffer in the external RAM

Do

We have these two tasks:

- Place the framebuffer in external RAM
- Setup the display controller to read from the external RAM

When the frambuffer is in external RAM, it is common practice to not allocate an array for it. You just declare a pointer to the correct address. The address in external RAM is then manually selected. It can be anywhere in the external RAM, but the start of the RAM is commonly used:

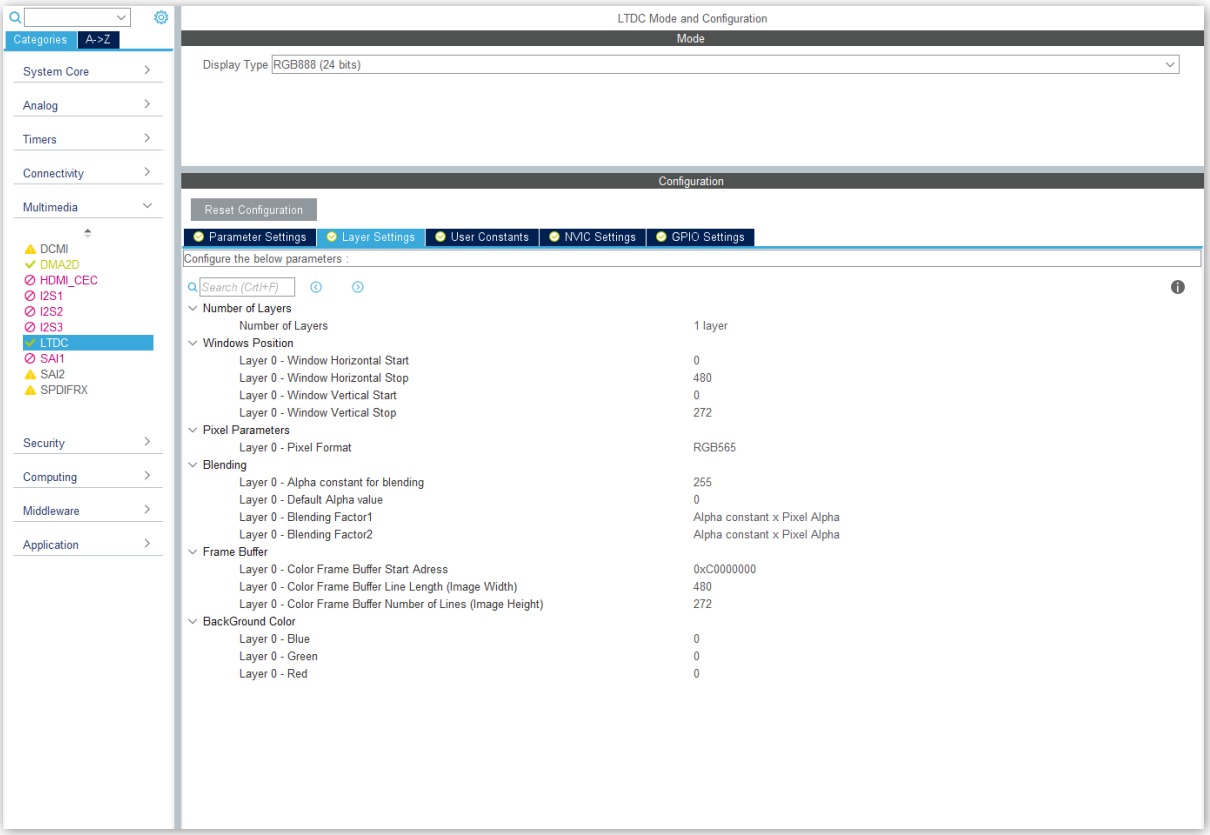
main.c

```
uint16_t* framebuffer = (uint16_t*)0xC0000000;  //16 bpp framebuffer
```

You can reuse the small test programs you created in the steps in Display Internal.

LTDC Layer configuration

Remember to change the configuration of the LTDC Layer. Since we now have a specific address for the framebuffer, we can insert that address in CubeMX (Color Frame Buffer Start Address):



Configuring LTDC Layer Parameters

Remember to remove this line from your program and the framebuffer array:

main.c

```
/* USER CODE BEGIN 2 */
HAL_LTDC_SetAddress(&hltdc, framebuffer, LTDC_LAYER_1);
/* USER CODE END 2 */
```

If the LTDC Layer size was setup to only update a part of the display in step 03 (due to the amount of internal RAM), now is the time to redo that. Reconfigure the LTDC Layer such that the entire display is covered.

Version: 4.16

6. External addressable flash

Motivation

In this step we will enable an external quad or octo SPI flash in memory mapped mode. An external flash is recommended for most project as it allows the application to use many and large images. The internal flash will quickly be full even for modest applications.

> **NOTE**
>
> **Skip this step if external flash is not relevant for your board bring up.**

When data is to be placed in external flash it is important that the external flash can be read by the MCU. The external flash should run at desired (typically maximum) speed to get the best performance.

Goal

The goal for this section is to enable the external flash, change it to memory mapped mode, and read data from it. As the read speed of the external flash is very important to graphics, you should also test the reading speed.

Verification

Here are the verification points for this section:

Verification Point	Rationale
External flash is readable	External flash can be used for image storage
External flash performance	Graphics performance depends a lot of the performance of the image memory

Prerequisites

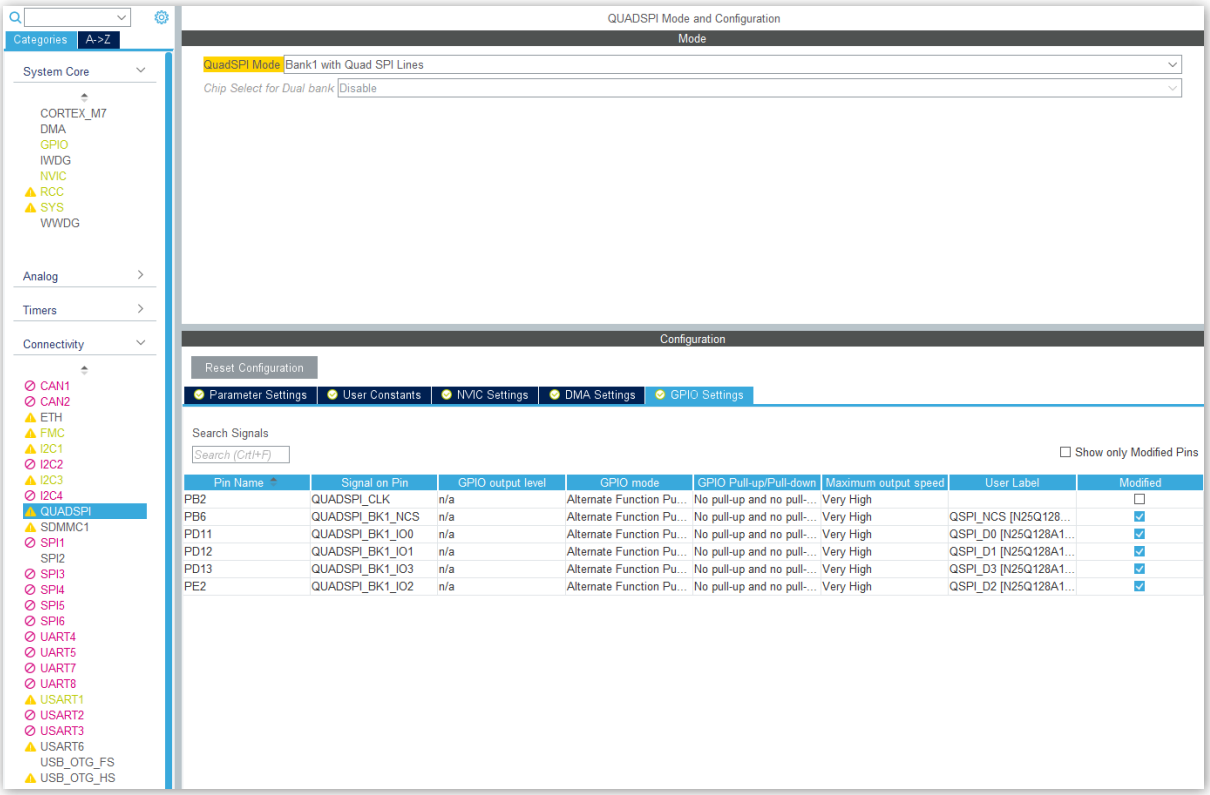
The following are the prerequisites for this step:

- Information about the flash, typically a datasheet

- Information about the connections between the MCU and the external flash

Do

The QSPI controller is configured in CubeMX under Connectivity -> QUADSPI:



Configuring QSPI flash

In the mode section you can configure the flash to single/dual/quad data lines. Quad lines are the fastest. Similar to the external RAM, you also here need to select and configure the GPIOs used for the data lines, chip select and clock signal.

Block Mode

After enabling the flash, we can test it by reading data from it. The Cube Firmware package contains examples for that.

Memory Mapped mode

After enabling the flash and testing it in block mode, it is necessary to change it to memory mapped mode. This will allow the CPU to fetch data directly from the flash.

The STM32 Cube HAL contains functions to change to memory mapped mode. An example is given here. It is necessary to consult the datasheet for the configuration data. The Cube Firmware package for your MCU contains more examples.

main.c

```
QSPI_CommandTypeDef      s_command;
QSPI_MemoryMappedTypeDef s_mem_mapped_cfg;

/* Configure the command for the read instruction */
s_command.InstructionMode   = QSPI_INSTRUCTION_1_LINE;
s_command.Instruction       = QUAD_INOUT_FAST_READ_CMD;
s_command.AddressMode       = QSPI_ADDRESS_4_LINES;
s_command.AddressSize       = QSPI_ADDRESS_24_BITS;
s_command.AlternateByteMode = QSPI_ALTERNATE_BYTES_NONE;
s_command.DataMode          = QSPI_DATA_4_LINES;
s_command.DummyCycles       = N25Q128A_DUMMY_CYCLES_READ_QUAD;
s_command.DdrMode           = QSPI_DDR_MODE_DISABLE;
s_command.DdrHoldHalfCycle  = QSPI_DDR_HHC_ANALOG_DELAY;
s_command.SIOOMode          = QSPI_SIOO_INST_EVERY_CMD;

/* Configure the memory mapped mode */
s_mem_mapped_cfg.TimeOutActivation = QSPI_TIMEOUT_COUNTER_DISABLE;

if (HAL_QSPI_MemoryMapped(&QSPIHandle, &s_command, &s_mem_mapped_cfg) != HAL_OK)
{
  return QSPI_ERROR;
}
```

If you are using the same flash as one of the STM32 evaluation kits, then the BSP packages for these boards (also in the Cube Firmware) contains valuable examples that can be modified for your hardware.

When the flash is in memory mapped mode, you can test it with code similar to what we used for external RAM (find the address in your MCU datasheet):

```
volatile uint32_t *externalFlash = 0x90000000;
const uint32_t size = 1000;
volatile uint32_t result = 0;

//read external Flash
for(int i = 0; i < size; i++)
{
    result += externalFlash[i];
}
```

Reuse the memory performance tests you did in earlier steps to also test the performance of the external flash.

Version: 4.16

7. External flash in block mode

Motivation

When working with Non-Memory-Mapped Flash memory, such as NAND flash, a driver must be developed in order for TouchGFX to use the assets stored within.

Read more about this topic in the Using Non-Memory Mapped flash for storing images section.

> **NOTE**
>
> **Skip this step if nonaddressable external flash is not relevant for your board bring up.**

Goal

The goal of this step is to create a driver that can read a number of bytes from a location in the non-mapped flash memory and store it in an array.

Verification

The verification points for this section are:

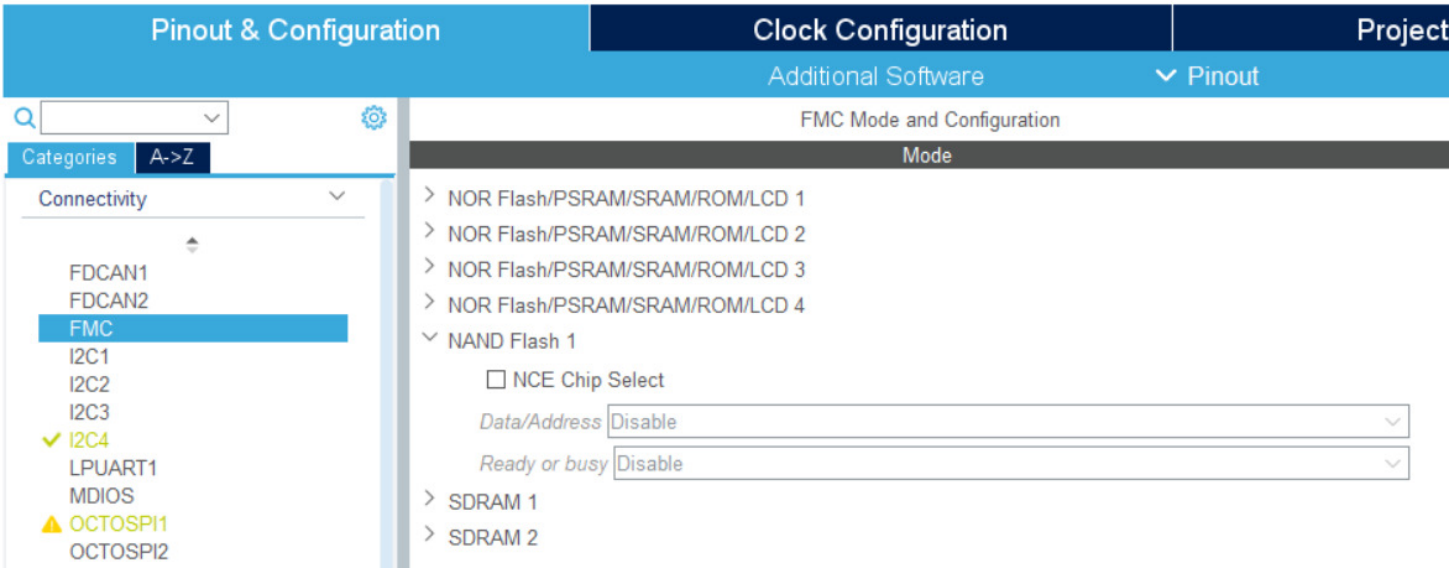
Verification Point	Rationale
Ensure contents of flash	Ensure that the contents read from the flash are correct.
Verify performance	Ensure that read performance is in accordance with MCU configuration.

Prerequisites

- Information about the flash, typically from a datasheet.
- Information about the connections between the MCU and the external flash.
- The flash speed.

Do

Typically, the NAND flash is configured via the FMC on your MCU.



Remember to configure the GPIOs that are connected to the flash.

A non-memory-mapped QSPI flash is configured in CubeMX like a memory-mapped QSPI flash.

Code

Write code that can read a number of bytes from a specific address of the flash. An example of how this might look is provided below. The implementation of the driver depends on your flash chip.

```
void readNonaddressableFlash(uint32_t from, uint8_t *into, uint32_t n)
{
  ...
}

uint8_t bytes[1000];

//read external Flash
readNonaddressableFlash(0xab001212, bytes, 1000);
```

This code will be used later to develop the TouchGFX abstraction layer.

Version: 4.16

8. Hardware acceleration

Motivation

The Chrom-ART (DMA2D) graphics accelerator is capable of transferring parts of image data from memory and drawing it into or composing it onto the framebuffer. Chrom-ART can read data from internal or external memory. Similarly it writes to internal or external memory. This can be utilized when doing graphics, and has the possibility of drastically improving the graphical performance and at the same time significantly lowering the MCU usage of your application.

Many STM32 controllers contain the Chrom-ART accelerator, but not all. Check your datasheet. DMA2D is the code name for Chrom-ART and is used in the code and documentation.

> **NOTE**
>
> **Skip this step if hardware acceleration using Chrom-ART is not relevant for your board bring up.**

Goal

The goal of this step is to enable Chrom-ART and read and write data using it. The goal is **not** to examine the functionality of the Chrom-ART chip, but to verify that memory interfaces are functional from a Chrom-ART perspective.

Verification

Here are the verification points for this section:

Verification Point	Rationale
Chrom-ART is configured	Chrom-ART can be used for drawing the desired graphics
Chrom-ART can read memory	Chrom-ART can be used for drawing graphics (M2M)
Chrom-ART can write memory	Chrom-ART can be used for drawing graphics (M2M and R2M)
Chrom-ART performance	Chrom-ART yields the desired performance for graphics

Prerequisites

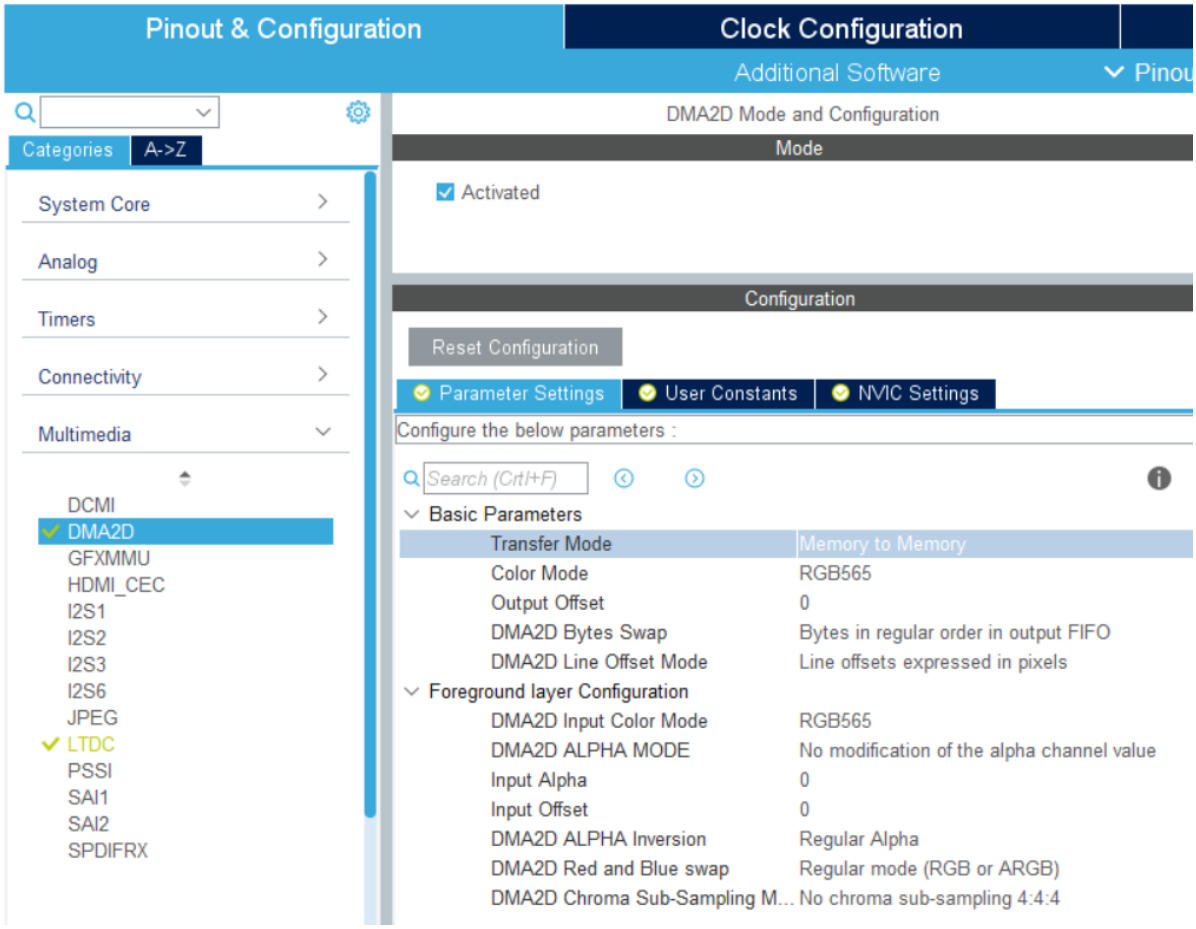
The following are the prerequisites for this step:

- MCU with Chrom-ART.

Do

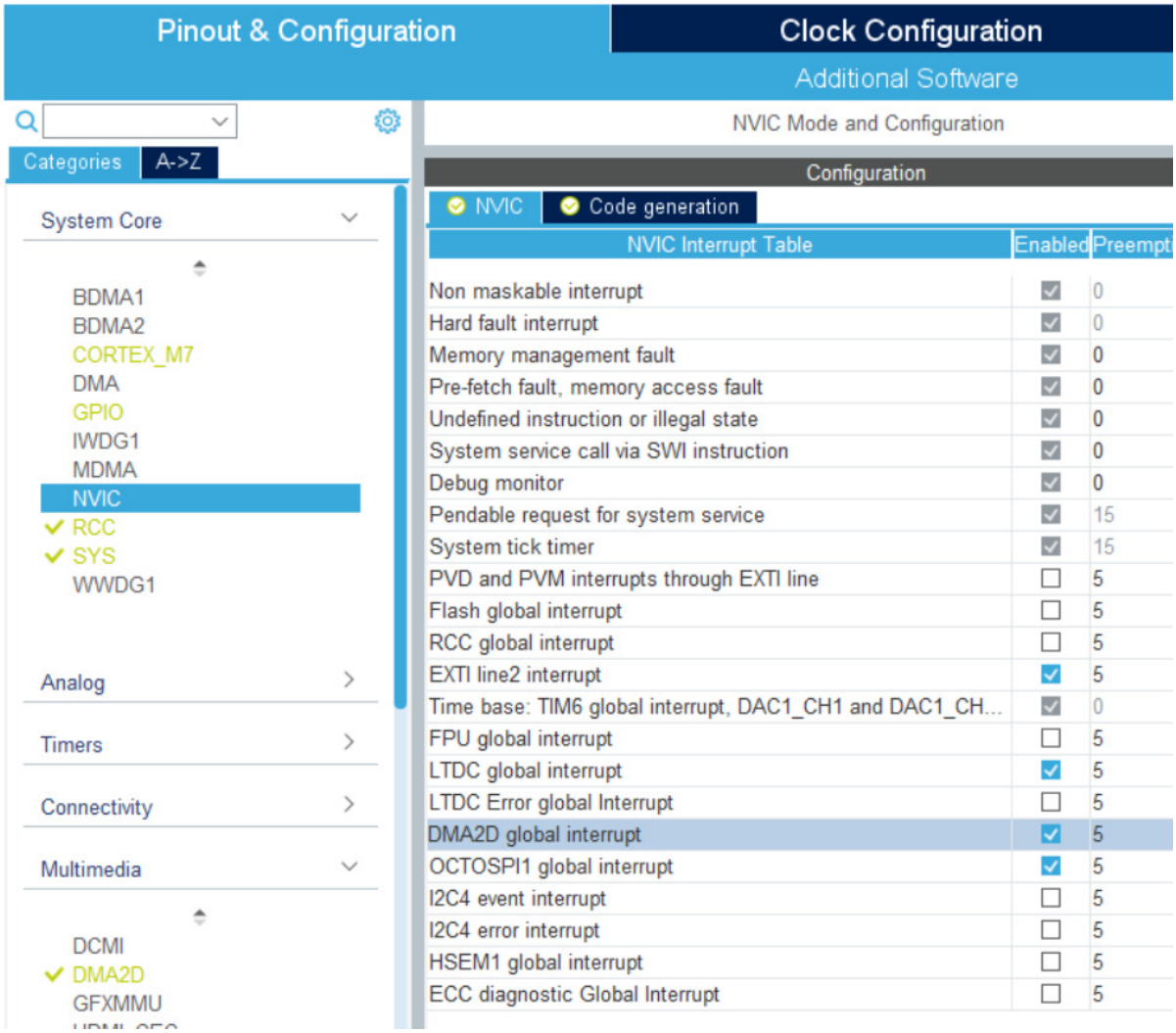
The Chrom-ART is configured in CubeMX under the *Multimedia -> DMA2D* category. Activate DMA2D and configure *Transfer mode* and *Color mode* according to your display.

In the figure below DMA2D is activated and configured for Memory to Memory transfer mode and RGB565 Color Mode. Select the color mode that matches your display.

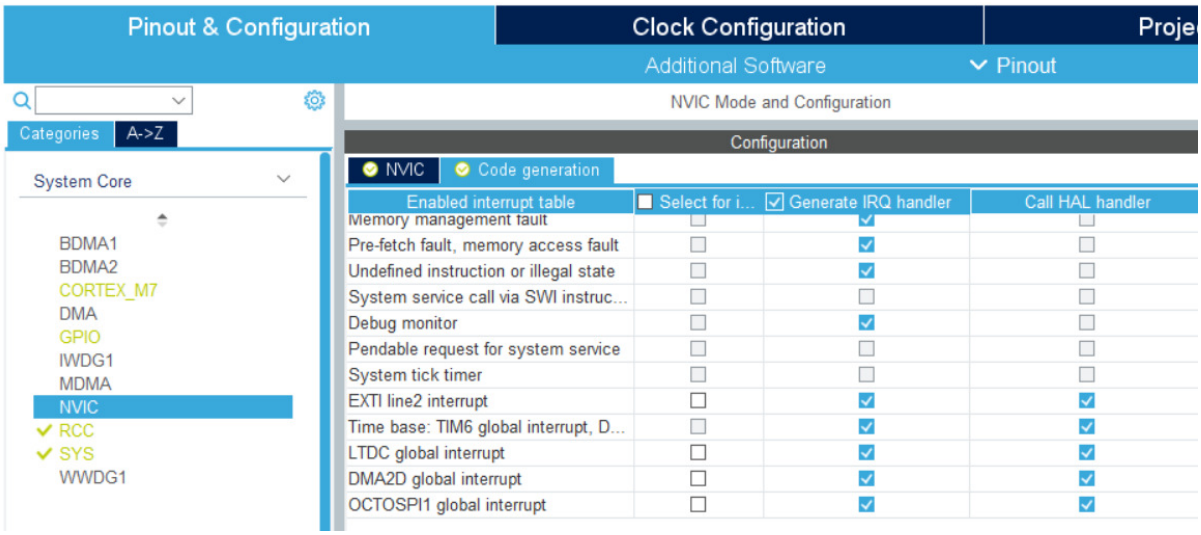


Configuring Chrom-ART

The DMA2D global interrupt is important for the syncronization of framebuffer access in a TouchGFX application. Ensure that the global interupt is enabled (NVIC tab) and that code generation for interrupt handlers is enabled (Code Generation tab) in the CubeMX NVIC settings as shown below. The priority is not important at this stage.



Enabling the Chrom-ART interrupt



Enabling the Chrom-ART interrupt handler code generation

Write to Framebuffer Memory

Here is an overview of code that fills a specific color in a rectangle in target memory (register to memory). Check Cube Firmware packs for a concrete project for your MCU.

main.c

```
#include "stm32f7xx_hal.h"
#include "stm32f7xx_hal_dma2d.h"
...

uint32_t color = 0xF800; //Red in RGB565

hdma2d.Init.Mode = DMA2D_R2M;
hdma2d.Init.ColorMode = DMA2D_RGB565;

MODIFY_REG(hdma2d.Instance->CR, DMA2D_CR_MODE, DMA2D_R2M);
MODIFY_REG(hdma2d.Instance->OPFCCR, DMA2D_OPFCCR_CM, DMA2D_RGB565);
MODIFY_REG(hdma2d.Instance->OOR, DMA2D_OOR_LO, displayWidth - rectangleWidth);

hdma2d.LayerCfg[1].InputColorMode = CM_RGB565;
hdma2d.LayerCfg[1].InputOffset = 0;

HAL_DMA2D_ConfigLayer(&hdma2d, 1);

HAL_DMA2D_Start_IT(&hdma2d, color, (unsigned int)dstPtr, rectangleWidth, rectangleHeight);
```

If the Transfer Completed setup is configured correctly in CubeMX, a custom handler can be assigned to handle this event:

```
    hdma2d.XferCpltCallback = DMA2D_XferCpltCallback;
```

And the handler can be defined as follows to verify the Transfer Completed interrupt configuration:

```
extern "C" {
    static void DMA2D_XferCpltCallback(DMA2D_HandleTypeDef* handle)
    {
        //Ensure that you this callback is called
    }
}
```

Memory-to-Memory can be tested by supplying a pointer to memory with pixel data.

```
HAL_DMA2D_Start_IT(&hdma2d,
    (unsigned int)srcPtr,
    (unsigned int)dstPtr,
    displayWidth - nrOfPixels);
```

Performance is as expected

Compare the performance of the Chrom-ART with the performance results from previous steps of reading and writing memory. It is expected that the code utilizing Chrom-ART will be more performant than the previous CPU read/write operations.

TIP

Use the value of the `CCSTEP` clock cycle register to get a more precise measurement of clock cycles spent between breakpoints than the millisecond counting sysTick.

Version: 4.16

9. Touch Controller

Motivation

Touch coordinates must be readable from a touch controller for the user to be able to interact with the application. The code developed in this step will be used later to develop the TouchGFX abstraction layer at a later stage.

> **NOTE**
>
> **Skip this step if a touch controller is not relevant for your board bring up.**

Goal

The goal of this step is to ensure that touch coordinates can be read from the touch controller on your display.

Verification

Here are the verification points for this section:

Verification Point	Rationale
Touch controller and MCU are configured	MCU must be configured to read from the touch controller over e.g. I2C.
Touch controller registers can be read	The TouchGFX abstraction layer can use this code to get the touch coordinates from the controller.
Reading performs as expected	Polling is a part of application render time. If polling takes too long touch polling should be moved to a different thread or made interrupt based.

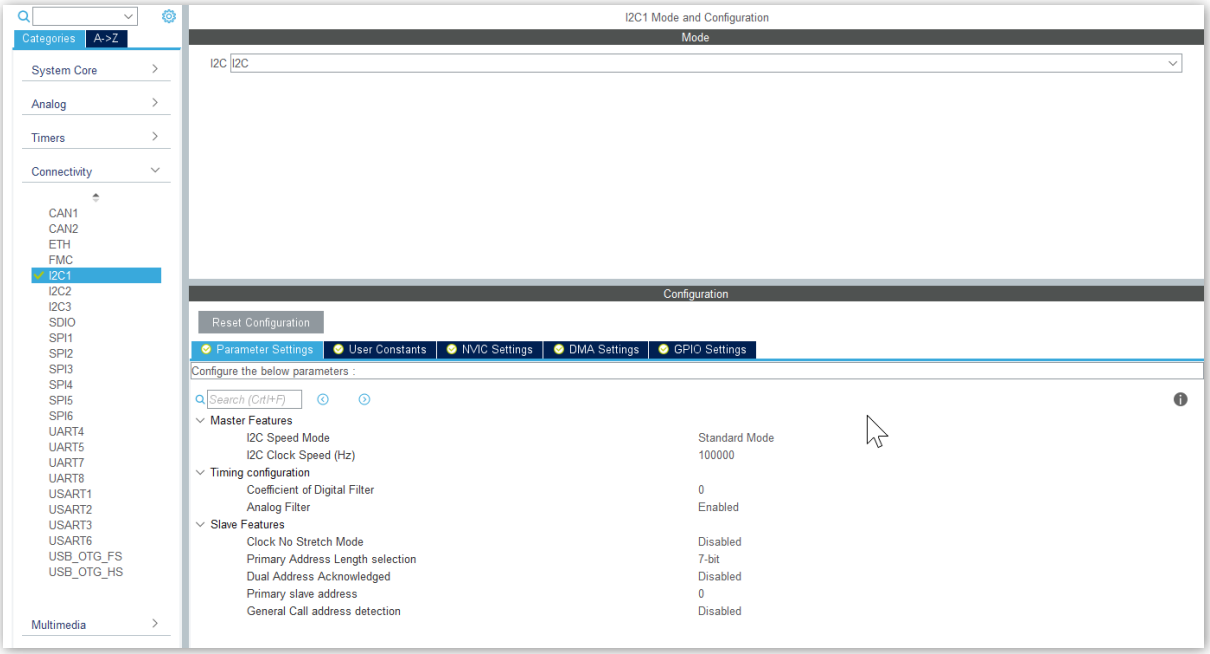
Prerequisites

- Display with touch controller
- Drivers to read from touch controller

Do

This step consists of two elements: Configure the MCU to communicate with the touch controller, and write driver code to talk to the touch controller.

Most touch controllers are connected to a I2C bus. The I2C communication is configured in CubeMX under Connectivity -> I2C1:



Configuring I2C

Many STM32 MCUs have more than one I2C controller, so select the one connected to your touch controller. Remember to configure the relevant GPIOs also.

If you do not have driver code for your touch controller you need to write it from scratch. The Cube Firmware for your MCU contains examples for I2C communication. These can be a start. Check the datasheet for the touch controller what registers to read to get the touch coordinates. The first thing to check is the I2C address of the touch controller and then read a "device id" registers for testing.

When you have the basic I2C running you need to develop a driver function that we will need when integrating with TouchGFX later. The function should return true if there is a touch, false if not, and also provide the coordinates.

The code example below shows how this code might look, driver code being abstracted by the function `myTouchController_GetState`:

main.c

```
    uint16_t x;
    uint16_t y;
```

```
    TouchControllerState state;
    if (myTouchController_GetState(&state))
    {
        x = state.touchY;
        y = state.touchX;
        //break point here
    }
```

Check with your debugger that the correct x and y values are received from the touch controller.

Some touch controllers are able to report multiple touch points. This is not supported by TouchGFX and can be ignored. Most often you just select the first touch point.

In the "TouchGFX AL Development" article Abstraction Layer it is explained how to send these values to TouchGFX.

Performance is as expected

Sampling touch should be possible within 1 ms if the code is executed in the same thread as the TouchGFX Application. If not fast enough, consider moving the code to a separate task, at a later stage.

Version: 4.16

10. Physical Buttons

Motivation

Physical buttons can function as external events usable as triggers from the TouchGFX Designer during application development, or simply used as events in the application backend.

> **NOTE**
>
> **Skip this step if physical buttons are not relevant for your board bring up.**

Goal

The goal of this section is to develop code that can be used in subsequent TouchGFX HAL- and/or application development.

Verification

Here are the verification points for this section:

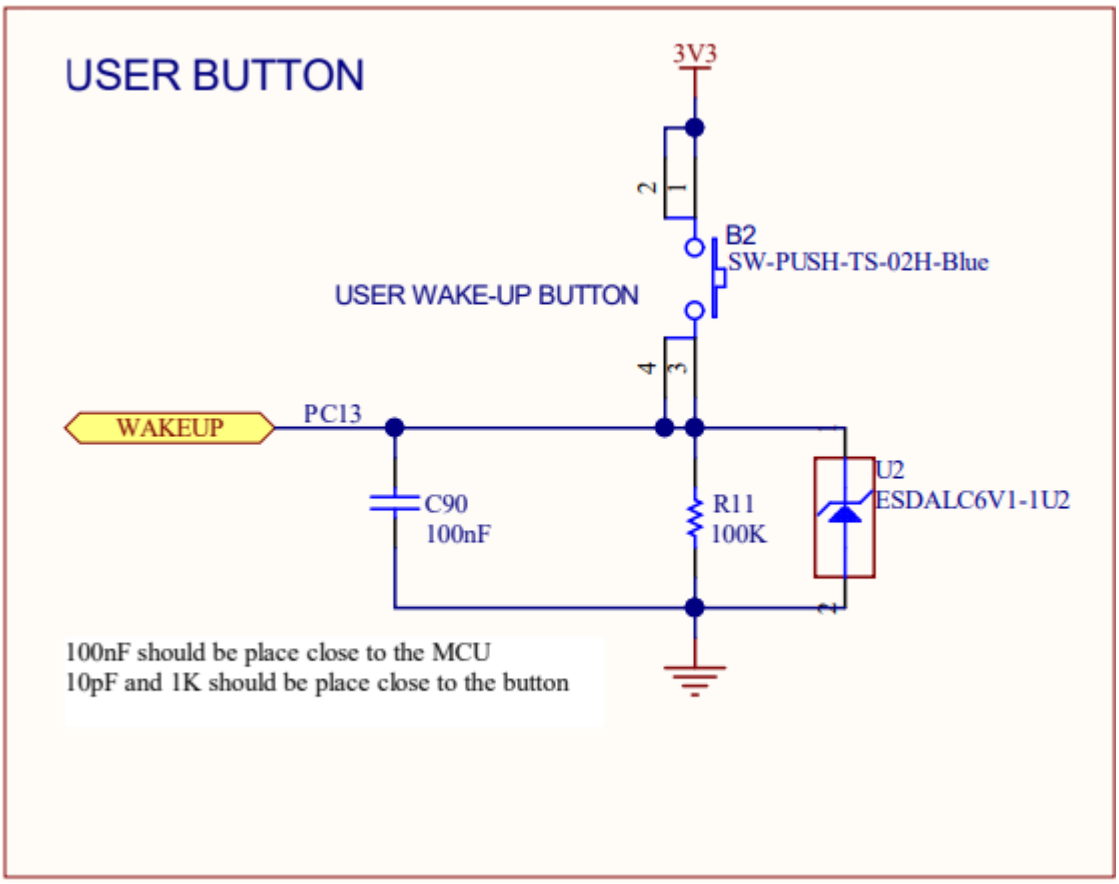
Verification Point	Rationale
MCU is configured	MCU GPIOs must be configured to read the state of connected physical buttons.
Connected GPIO can be read	Once code has been developed to read the physical button state from a GPIO this can be used in sub-sequent TouchGFX HAL development.
Reading performs as expected	Polling is a part of application render time. If polling takes too long this should be moved to a different thread or made interrupt based.

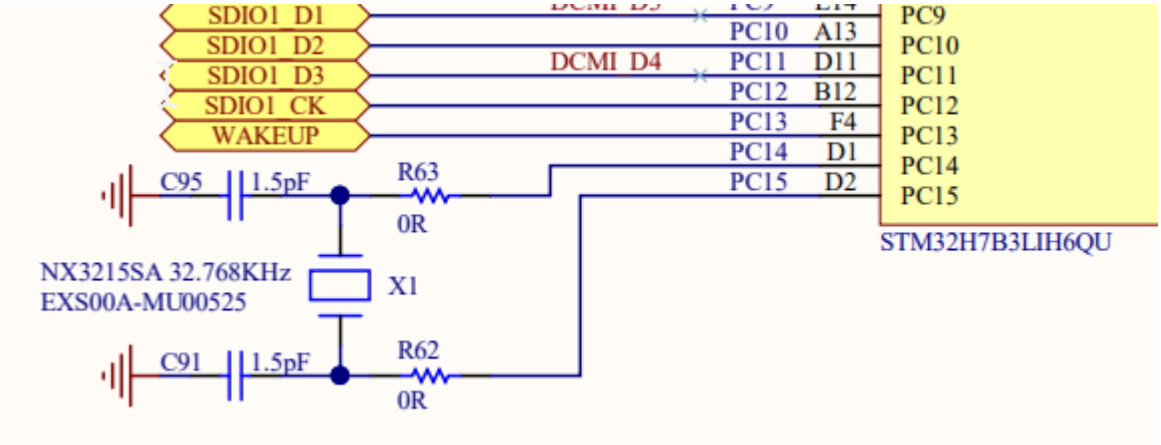
Prerequisites

- Physical buttons must be connected to GPIOs on the MCU

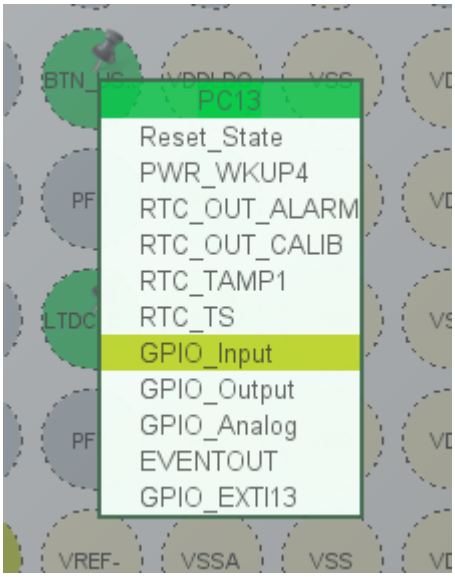
Do

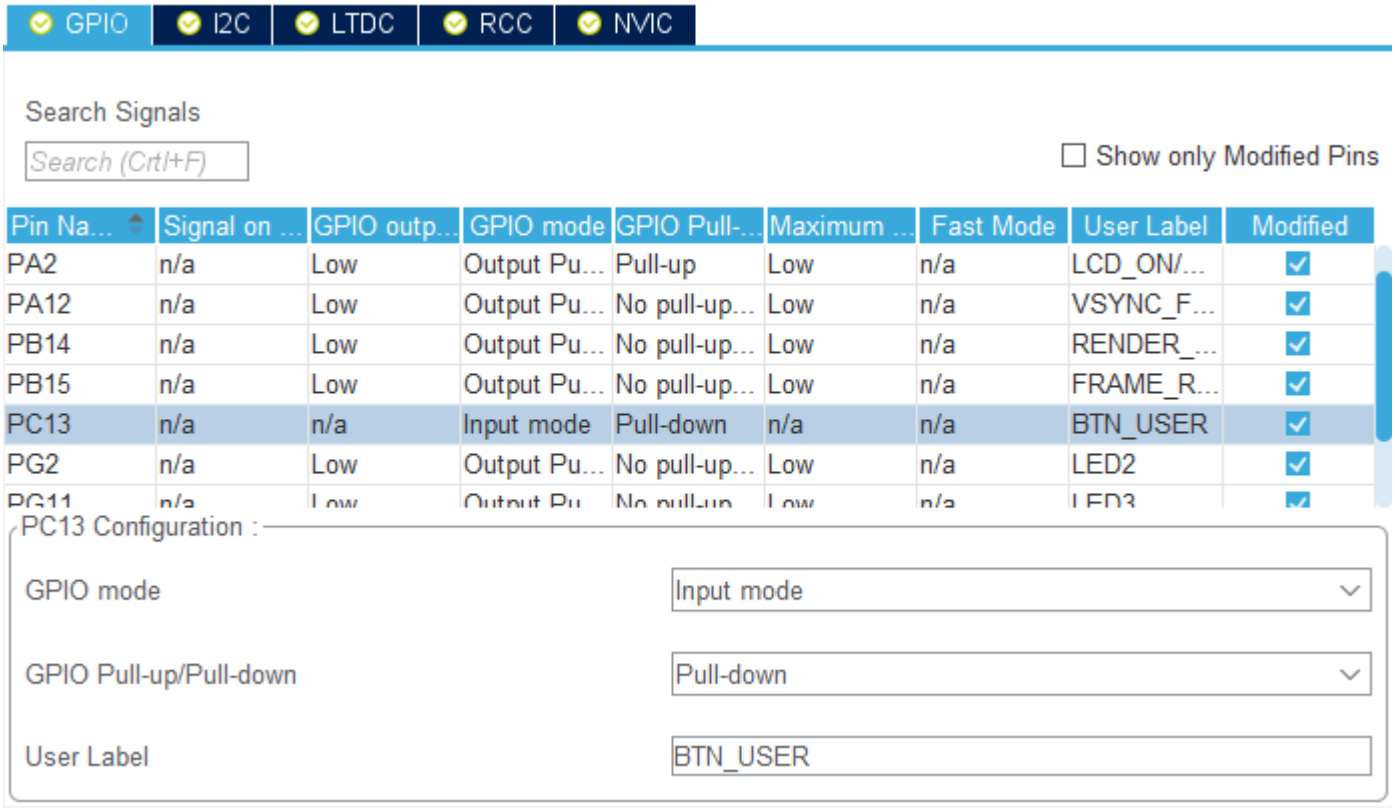
The following images show how the schematics might look for a physical button and how it is connected to the MCU





In CubeMX GPIO Port C Pin 13 (PC13) can be configured as an input and configured as a pull-down in the **GPIO** section of the **System Core** category.





The code generated by CubeMX will setup the appropriate GPIO port(s) which can now be read:

main.c

```
uint8_t key;
if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) != GPIO_PIN_RESET)
{
    key = 1;
}
```

Performance is as expected

Polling the state of physical buttons should be possible within 1 ms if the code is executed in the same thread as the TouchGFX Application. If not fast enough, consider moving the code to a separate task, at a later stage or making it interrupt based.

Version: 4.16

11. Flash Loader

Motivation

In this step we will discuss loading data to the external flash. The compiler will compile the text, fonts, and images in your project and produce a binary or hex file with this data. This data is typically put into the external flash. The internal flash is then reserved for code.

During development we need a way to write data to the external flash, but this is not necessary during runtime where we only read from the flash.

Two ways are common for writing data to the external flash:

- Write a flashloader for STM32CubeProgrammer
- Use a proprietary application-based solution

> **NOTE**
>
> **Skip this step if external flash is not present**

Goal

The goal in this section is to select and develop a mechanism for loading data to the external flash.

Verification

Here are the verification points for this section:

Verification Point	Rationale
Data can be flashed	External flash can be used for image storage

Prerequisites

The following are the prerequisites for this step:

- Information about the flash, typically a datasheet

- Information about the connections between the MCU and the external flash

Do

Flash loader for STM32CubeProgrammer

The STM32CubeProgrammer comes with flash loaders for the various STM32 Evaluation kits. The flash loaders are small programs that are loaded to the RAM of the MCU and facilitate the programming of the flash.

The flash loader consists of two parts: *Configuration of the GPIOs and flash interface that are required to communicate with the flash* The flashing algorithm that knows the sequence of commands required to write in the flash

These parts can often be based on an existing flash loader. If you can find a flash loader for the same flash that you are using, take that as starting point and modify the GPIO part. If you design your hardware by copying the flash circuit from an evaluation kit, then you can use the flash loader for that kit directly. This is the recommended way.

The flash loader projects provided with *STM32CubeProgrammer* are found in the installation folder, typically here: *C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\ExternalLoader*

A few flash loaders are provided as source code projects for IAR Embedded Workbench, Keil, and TrueStudio.

Proprietary application-based solution

Another solution is to include flash loading into the application itself. The idea is that you already have the flash configuration inside your application (to be able to load from it), and maybe you know how to write a block to the flash from your previous testing. You then just need a way of transferring the new flash data to your application. One way is through a UART. The application receives the data stream, and writes the data to the flash, block by block.

The flash cannot be in memory mapped mode while this is running, so the application must typically be put in a special mode.

Open source solutions for the transmission of bytes can be found on the Internet. The Y-modem protocol for example provides 16-bit CRC on the data.

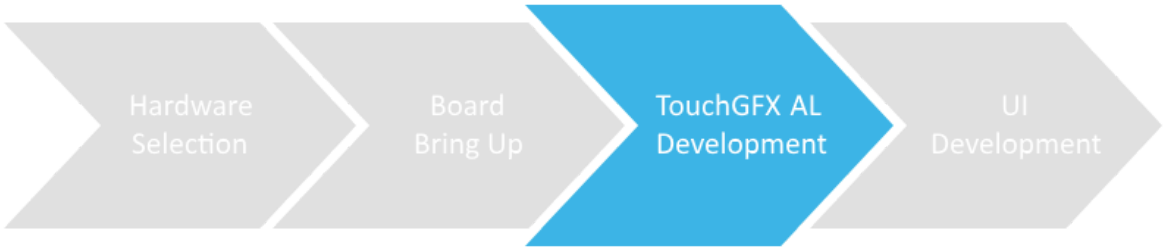
Testing

After the data has been written to the flash, test that it can be read correctly. Use the small test programs developed in the previous sections.

It is advised to test the whole flash thoroughly now, to find any bugs early.

Version: 4.16

TouchGFX AL Development Introduction



Project activity

The TouchGFX Abstraction Layer (AL), in a TouchGFX application, is the software component that sits between the Board Initialization Code, developed during the Board Bring-Up phase, and the TouchGFX Engine. Its main task is to tie together the Engine with the underlying hardware and operating system. This is done by abstracting the specifics of the underlying hardware and OS such that it can be treated in a unified way by the Engine.

The AL consists of two different parts, the Hardware Abstraction Layer (HAL) and the Operating System Abstraction Layer (OSAL).

Project component

In this section you will get a general introduction to the principles and responsibilities of the abstraction layer and how it interacts with the TouchGFX Engine. Details on how this is achieved for particular hardware is described in the following sections.

- Abstraction Layer Architecture details the architecture of the AL and shows you how to implement each of the interaction points, called hooks, between the TouchGFX Engine and the AL.
- Generator User Guide shows you how to use TouchGFX Generator to create the basis for your AL implementation as well as details on more complex issues.
- Scenarios gives you concrete detailed examples on how to create ALs for specific hardware setups.

Responsibilities of the Abstraction Layer

As explaned in the Main Loop section in the Basic Concepts chapter, the TouchGFX Engine has a main loop that performs three basic steps.

1. Collect input (Touch coordinates, Buttons)
2. Update the Scene Model
3. Render the Scene Model to the Framebuffer

These three steps ensure the main responsibility of the TouchGFX engine, which is to update the framebuffer to reflect the current state of the application.

The actual transfer of framebuffer data to the display as well as the collection of external input is not directly handled by the engine, but instead delegated from the engine to the TouchGFX AL.

The main loop will continuously update the framebuffer(s) over and over again. This process must be synchronized with the actual update frequency and readyness of the display to ensure that all frames will be transferred and displayed correctly on the display. If no synchronization takes place the main loop will continuously update and potentionally overwrite the framebuffer(s) before it has been transferred. This synchronization is the responsibility of the AL.

The TouchGFX AL also has the responsibility of controlling the framebuffer memory area and the access to it. This means that all accesses of the framebuffer will go through the AL.

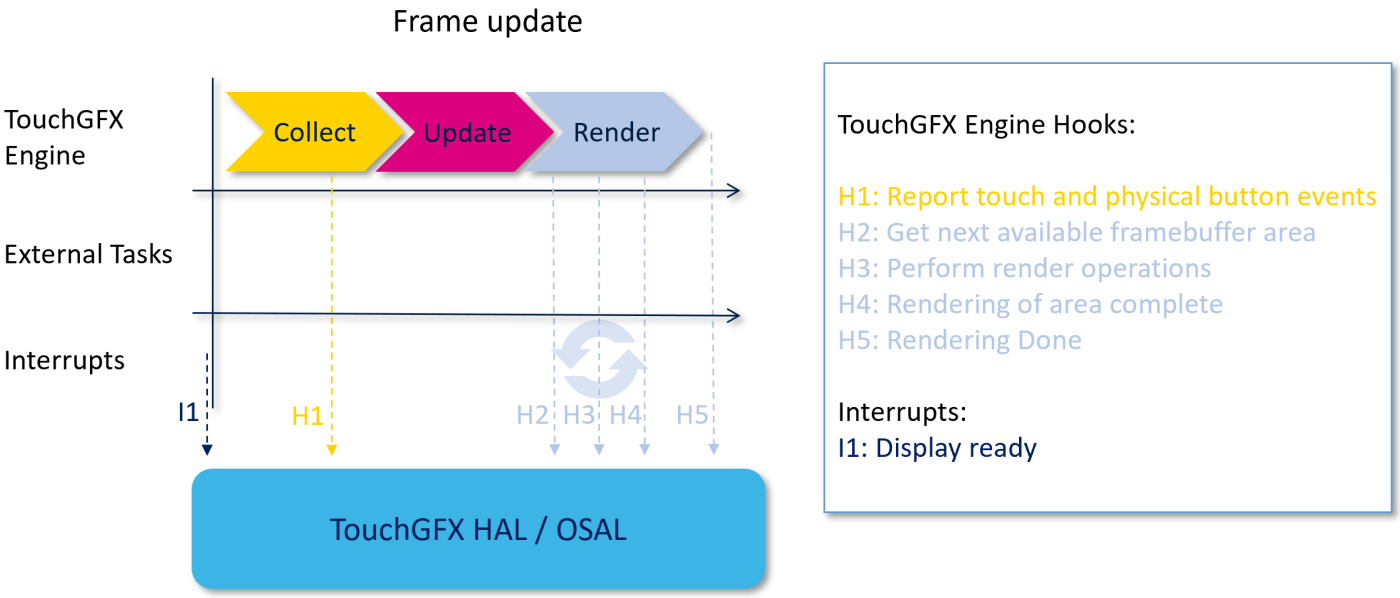
To detail, the responsibilities of the TouchGFX AL are:

Responsibility	Description
Synchronize TouchGFX Engine main loop with display transfer	When the next frame has been calculated and rendered in the available framebuffer, the engine main loop must be halted to make sure that it does not overwrite the newly assembled framebuffer before it has been transferred to the display.
Report touch and physical button events	Sample if a touch event has occurred and the corresponding coordinates hereof. Sample whether or not any physical button or similar has been activated. Report these events to the engine. Note that other external events are to be propagated to the TouchGFX application through a different mechanism. Read more on this in the section on backend communication.
Synchronize framebuffer access	The framebuffer memory is the responsibility of the TouchGFX AL and since it can be accessed by different actors, like the main loop thread and the DMA, TouchGFX AL must offer a way to protect this memory.

Responsibility	Description
Report the next available framebuffer area	The AL must be able to answer which part of the current framebuffer can be updated next. In a standard two framebuffer setup, this will always be the complete framebuffer, since in that case you always have one entire framebuffer dedicated for rendering and one for transferring to the display. In a one or partial framebuffer setup this is more complex.
Perform render operations	While rendering the scene model, the engine main loop will ask the AL to render parts hereof. A specific TouchGFX AL implementation will utilize the underlying hardware to render graphics primitives. One example is rendering bitmaps on MCUs with the Chrom-ART graphics accelarator. TouchGFX comes with optimized rendering methods built-in for all available platforms, so no need to customize this.
Handle framebuffer transfer to display	The engine informs the AL which part of which framebuffer must be transferred. The AL should initiate this transfer making sure that the pixels eventually end up on the physical display.

Since TouchGFX AL is a passive software module, not having its own thread or similar, it must perform its actions through certain hooks (functions) called from the TouchGFX Engine main loop or through interrupts.

The available set of hooks and interrupts are depicted below.



Available hooks and interrupts

It is up to the AL developer to implement these hooks so that the responsibilities of the AL are covered given the underlying hardware and operating system. If the AL developer needs other means to support the responsibilities, the developer can setup interrupts to activate at certain points.

Examples of this is LTDC vertical synchronization interrupt and a hardware timer. The *I1: Display ready* interrupt is an example of a vertical synchronization interrupt. Note that the setup of these interrupts is considered a part of the AL development.

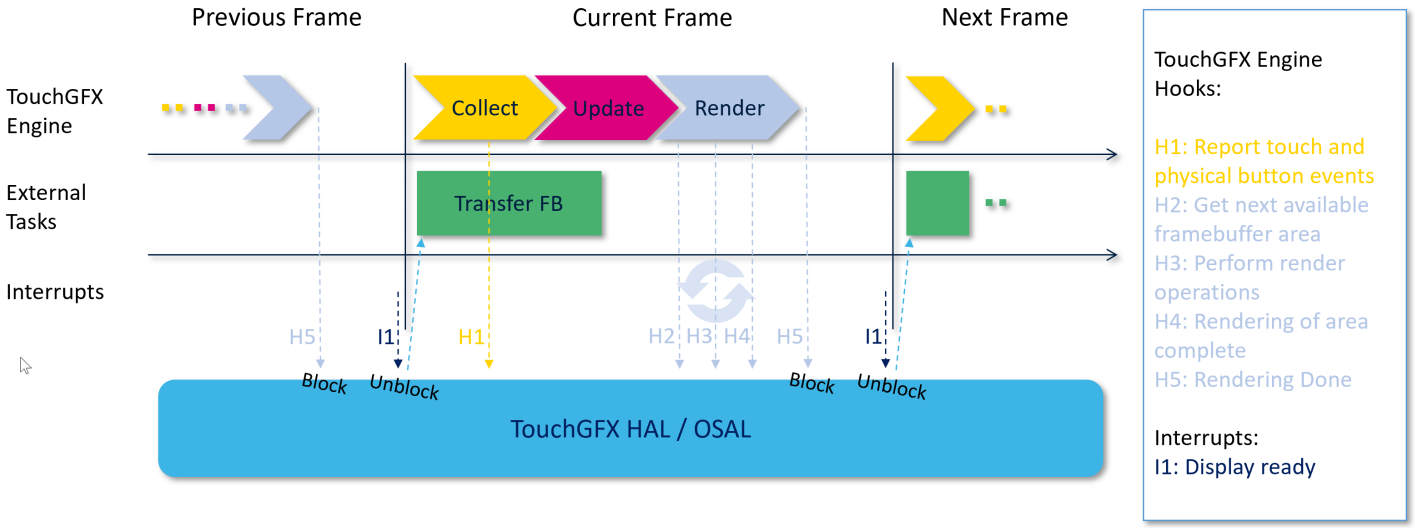
Example setup: Two framebuffers - MCU with LTDC

One common setup is having two framebuffers with an MCU with LTDC. The AL actions for each hook will in this setup most often be as follows.

Setup the AL to react to the LTDC VSYNC interrupt such that I1 is executed each time the display is ready to receive a new frame. This is used to synchronize the main loop with the display.

Hooks and Interrupts	Actions
I1: Display ready	Setup the LTDC VSYNC interrupt to trigger this. Unblock the main loop and initiate framebuffer transfer of the framebuffer prepared in previous frame
H1: Report touch and physical button events	Return any information on touch events or physical button clicks
H2: Get next available framebuffer area	Using the double buffer setup simply return the entire framebuffer area of the framebuffer not currently being transferred to the display
H3: Perform render operations	Depends on the capabilities of the MCU. Perform the hardware assisted render operations and software fallback for the rest
H4: Rendering of area complete	No action
H5: Rendering done	Block the main loop

This setup gives the following execution flow:



Execution flow in setup with two framebuffers and an MCU with LTDC

This describes the overall design of the AL for this setup. The following sections goes into depth on how to implement Abstraction Layers.

Version: 4.16

Abstraction Layer Architecture

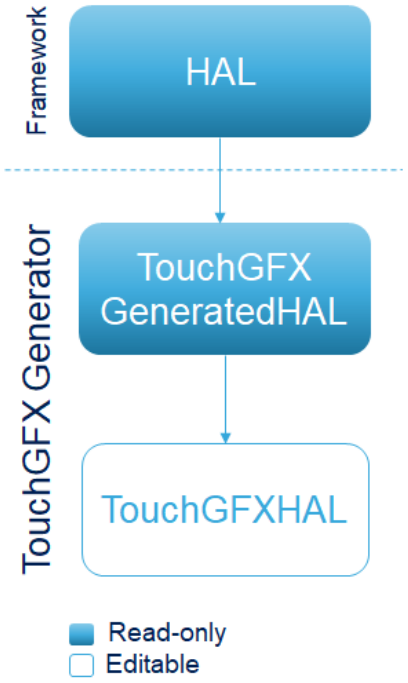
As described in the previous section, the TouchGFX AL has a particular set of responsibilities. Responsibilities are either implemented in the hardware part of the AL (HAL) or the part of the AL that synchronizes with TouchGFX Engine, typically through an RTOS (OSAL). The following table summarizes these responsibilities which were outlined in the previous section:

Responsibility	Operating system or Hardware
Synchronize TouchGFX Engine main loop with display transfer	Operating system and hardware
Report touch and physical button events	Hardware
Synchronize framebuffer access	Operating system
Report the next available framebuffer area	Hardware
Perform render operations	Hardware
Handle framebuffer transfer to display	Hardware

Each of the following subsections highlight what should be done to fulfill the above responsibilities. For custom hardware platforms the TouchGFX Generator, inside STM32CubeMX, can generate most of the AL and accompanying TouchGFX project. The remaining parts, that the AL developer must implement manually, are pointed out through code comments and notifications through the TouchGFX Generator. Read more about the TouchGFX Generator in the next section.

Abstraction Layer Classes

The HAL is accessed by the TouchGFX Engine through concrete sub-classes of `HAL`. These sub-classes are generated by the TouchGFX Generator. The generator, which is *the* primary tool for creation of the Abstraction Layer, can generate both the part of the HAL that reflects configurations from CubeMX, as well as the OSAL for CMSIS V1 and V2. Please read the section on TouchGFX Generator for further details. Generally, the architecture of the HAL is in the following figure.



Hierarchy of generated code

Synchronize TouchGFX Engine main loop with display transfer

The main idea behind this step is to block the TouchGFX Engine main loop when rendering is done, ensuring that no further frames are produced. Once the display is ready the OSAL signals the blocked Engine main loop to continue producing frames.

In order to fulfil this responsibility the typical way of a TouchGFX AL is to utilize the engine hook *Rendering done* and the interrupt *Display Ready*, as outlined in the previous section. The OSAL defines a function `OSWrappers::signalVSync` in which developers can signal the semaphore that the engine waits upon when it calls `OSWrappers::signalVSync`

TIP

The TouchGFX Generator can create a complete OSAL for CMSIS V1 and V2.

Rendering Done

The *Rendering done* hook, `OSWrappers::waitForVSync`, is called by the TouchGFX Engine after rendering is complete.

When implementing this AL method, the AL must block the graphics engine until it is time to render the next frame. The standard method to implement this block is to peform a blocking read from a

message queue. The HAL developer is free to use any method to implement the block if this is not feasible.

> **TIP**
>
> The TouchGFX Generator can also generate an empty OSAL that uses spinlocks to wait, rather than RTOS primitives if such software is not available.

When `OSWrappers::signalVSync` is signaled (or the semaphore/queue used in `OSWrappers::waitForVSync` is signaled) TouchGFX will start rendering the next application frame. The following code based on CMSIS V1 causes the TouchGFX engine to block until an element is added to the queue by another part of the system, typically an interrupt syncronized with the display.

RTOS_OSWrappers.cpp

```
static osMessageQId vsync_queue = 0; //Queue identifier is assigned elsewhere

void OSWrappers::waitForVSync()
{
    //Wait for next VSYNC to occur, by reading from the queue
    osMessageGet(vsync_queue, osWaitForever);
}
```

If not using an RTOS, the TouchGFX Generator provides the following implementation for `waitForVSync` using a volatile variable.

NO_OS_OSWrappers.cpp

```
static volatile uint8_t vsync_sem = 0;

void OSWrappers::waitForVSync()
{
    while(!vsync_sem)
    {
        // Perform other work while waiting
        ...
    }
}
```

> **TIP**
>
> - ***While*** TouchGFX Engine is waiting to produce the next frame other tasks can do important work.

Display ready

The *Display ready* signal to unblock the main loop should come from an interrupt from a display controller, from the display itself or even from a hardware timer. The source of the signal is dependant on the type of display.

The `OSWrappers` class defines a function for this signal: `OSWrappers::signalVsync`. The implementation of the function must unblock the main loop by satisfying the wait condition used in `OSWrappers::waitForVSync`.

Continuing from the above CMSIS RTOS example, the following code puts a message into the message queue `vsync_queue` which unblocks the TouchGFX Engine.

RTOS_OSWrappers.cpp

```
void OSWrappers::signalVSync()
{
    if (vsync_queue)
    {
        osMessagePut(vsync_queue, dummy, 0);
    }
}
```

This `OSWrappers::signalVSync` method must be called at hardware level from an interrupt for e.g. an LTDC, an external signal from the display, or a hardware timer.

If not using an RTOS use a variable and assign a non-zero value to break the while-loop.

NO_OS_OSWrappers.cpp

```
void OSWrappers::signalVSync()
{
    vsync_sem = 1;
}
```

Report touch and physical button events

Before rendering a new frame, the TouchGFX Engine collects external input from the `TouchController` and `ButtonController` interfaces.

Touch Coordinates

Coordinates from the touch controller are translated into click-, drag- and gesture events by the engine and passed to the application. The following code is generated by the TouchGFX Generator:

TouchGFXConfiguration.cpp

```
static STM32TouchController tc;
static STM32L4DMA dma;
static LCD24bpp display;
static ApplicationFontProvider fontProvider;
static Texts texts;
static TouchGFXHAL hal(dma, display, tc, 390, 390);
```

During the TouchGFX Engine render cycle, when collecting input, the engine calls the `sampleTouch()` function on the *tc* object:

```
bool STM32TouchController::sampleTouch(int32_t& x, int32_t& y)
```

The implementation, provided by the AL developer, should assign the read touch coordinate values to x and y and return whether or not a touch was detected (true or false).

TIP

The TouchGFX Generator will generate a class that defines the TouchController interface functions as empty. The HAL developer must fill in the implementation.

There are multiple ways of implementing this function:

1. ***Polling in sampleTouch()***: Read touch status from the hardware touch controller (typically I2C) by sending a request and polling for the result. This impacts the overall render time of the application as the i2C round-trip is often up to 1ms during which the graphics enging is blocked.
2. ***Interrupt based***: Another possibility is to use interrupts. The I2C read command is started regularly by a timer or as a response to an external interrupt from the touch hardware. When the I2C data is available (another interrupt) the data is made available to the `STM32TouchController` through a message queue or global variables. The code below from `STM32TouchController.cpp` (created by TouchGFX Generator) shows how `sampleTouch` could look for a system with an RTOS:

STM32TouchController.cpp

```
bool STM32TouchController::sampleTouch(int32_t& x, int32_t& y)
{
    if (osMessageQueueGet(mid_MsgQueue, &msg, NULL, 0U) == osOK)
    {
        x = msg.x;
        y = msg.y;
        return true;
    }
    return false;
}
```

The location of this file will be outlined in the next chapter on TouchGFX Generator

Other External Events

The Button Controller interface, `touchgfx::ButtonController`, can be used to map hardware signals (buttons or other) to events to the the application. The reaction to these events can be configured within TouchGFX Designer.

The use of this interface is similar to the Touch Controller above, except that it is not mandatory to have a ButtonController. To use it, create an instance of a class implementing the `ButtonController` interface, and pass a reference to the instance to the HAL:

MyButtonController.cpp

```
class MyButtonController : public touchgfx::ButtonController
{
  bool sample(uint8_t& key)
  {
    ... //Sample IO, set key, return true/false
  }
};
```

TouchGFXConfiguration.cpp

```
static MyButtonController bc;
void touchgfx_init()
{
  ...
  hal.initialize();
  hal.setButtonController(&bc);
}
```

The *sample* method in your ButtonController class is called before each frame. If you return true, the key value will be passed to the *handleKeyEvent* eventhandler of the current screen.

FURTHER READING

See the **Interactions** article for further information on how to use values sampled through the ButtonController as triggers for interactions in the designer.

Synchronize framebuffer access

Multiple actors may be interested in accessing the framebuffer memory.

1	CPU	Reads and writes pixels during rendering
2	DMA2D*	Reads and writes pixels during hardware assisted rendering
3	LTDC	Reads pixels during transfer to parallel RGB display
4	DMA	Read pixels during transfer to SPI display

The TouchGFX Engine synchronizes framebuffer access through the `OSWrappers` interface and peripherals (e.g. DMA2D) that also wish to access the framebuffer must do the same. The normal design is to use a semaphore to guard the access to the framebuffer, but other synchronization mechanisms can be used.

The following table shows a list of functions in the `OSWrappers` class (OSWrappers.cpp) that can be generated by the TouchGFX Generator or manually by the user.

Method	Description
`takeFrameBufferSemaphore`	Called by graphics engine to get exclusive access to the framebuffer. This will block the engine until the DMA2D is done (if running)
`tryTakeFrameBufferSemaphore`	Ensure that the lock is taken. This method does not block, but ensures that the next call to takeFrameBufferSemaphore will block its caller
`giveFrameBufferSemaphore`	Releases the framebuffer lock
`giveFrameBufferSemaphoreFromISR`	Releases the framebuffer lock from an interrupt context

TIP

The TouchGFX Generator can generate a ChromART driver that synchronizes using the OSWrappers interface as well as implementations for functions that perform this synchronization depending on choice of RTOS.

Report the next available framebuffer area

Regardless of rendering strategy TouchGFX Engine must know, in each tick, which memory area it should render pixels to. Using single- or double framebuffer strategies the TouchGFX Engine will write

pixel data to a memory area according to the full width, height, and bit depth, of the framebuffer. The graphics engine takes care of swapping between the two buffers in a double buffer setup.

It is possible to limit the access to the framebuffer to part of the framebuffer. The method `HAL::getTFTCurrentLine()` can be reimplemented in your HAL subclass. Return the line number above which it is save for the graphics engine to draw.

Using a Partial Framebuffer strategy the developer defines one or more blocks of memory that TouchGFX Engine will use when rendering. Read more about that here.

> **TIP**
>
> TouchGFX Generator can provide configurations for all supported framebuffer strategies.

Perform Render Operations

Rendering and displaying graphics are rarely the sole purposes of an application. Other tasks also need to use the CPU. One goal of TouchGFX is to draw the user interface using as few CPU cycles as possible. The HAL class abstracts the DMA2D found on many STM32 microcontrollers (or other hardware capabilities) and makes this available to the graphics engine.

When rendering assets such as bitmaps to the framebuffer, the TouchGFX Engine checks if the HAL has the capability to 'blit' a portion of- or all of the bitmap into to the framebuffer. If so, the drawing operation is delegated to the HAL rather than being handled by the CPU.

The engine calls the method `HAL::getBlitCaps()` to get a description of the capabilities of the hardware. Your HAL subclass can reimplement this to add the capabilities.

When the engine is drawing the user interface it will call operations on the HAL class, e.g. `HAL::blitCopy`, that queue the operations for the DMA. If the HAL does not report the required capability, the graphics engine will use a software rendering fallback.

> **TIP**
>
> Many STM32 MCUs have a ChromART chip which can move data from e.g. external Flash memory into the framebuffer while alpha blending pixels.
> For many MCUs, TouchGFX Generator can generate a ChromART driver which adds the capability of several 'blit' operations using the ChromART chip.

Handle framebuffer transfer to display

In order to transfer the framebuffer to the display the hook "Rendering of area complete" is often utilized in a TouchGFX AL. The engine signals the AL once rendering of a part of the framebuffer has been completed. The AL can choose how to transfer this part of the framebuffer to the display.

Rendering of area complete

In code this hook is the virtual function `HAL::flushFrameBuffer(Rect& rect)`.

On STM32 microcontrollers with LTDC controllers we don't need to do anything to transmit the framebuffer after every rendering. This happens continuously with a given frequency after the LTDC has been initialized and therefore we can leave the implementation of this method empty.

For other display types like SPI or 8080 you need to transfer the framebuffer manually.

The implementation of this function allows developers to initiate a manual transfer of that area of the framebuffer to a display with GRAM:

```
void TouchGFXHAL::flushFrameBuffer(const touchgfx::Rect& r)
{
    HAL::flushFrameBuffer(rect); //call superclass

    //start transfer if not running already!
    if (!IsTransmittingData())
    {
        const uint8_t* pixels = ...; // Calculate pixel address
        SendFrameBufferRect((uint8_t*)pixels, r.x, r.y, r.width, r.height);
    }
    else
    {
       ... // Queue rect for later or wait here
    }
}
```

FURTHER READING

Read through the scenarios for concrete examples of how to support various display interfaces.

Version: 4.16

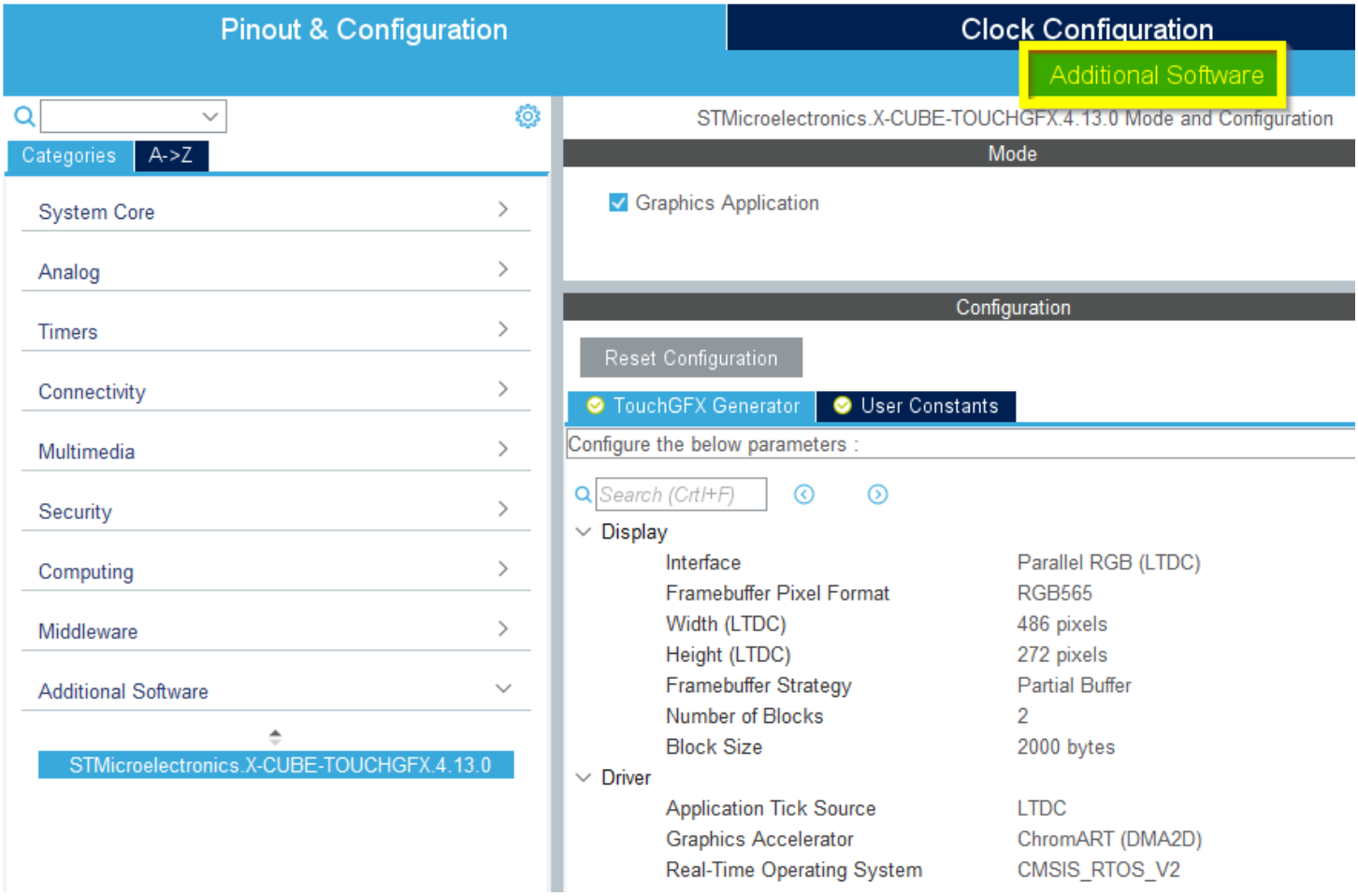
Generator User Guide

TouchGFX Generator, a part of X-CUBE-TOUCHGFX, is a CubeMX Additional-Software component that helps developers configure TouchGFX to run on their hardware platform. Based on existing CubeMX settings and user input TouchGFX Generator will generate the files required to configure a working TouchGFX application. They include files for TouchGFX HAL, TouchGFX OSAL and TouchGFX Configuration.

Once code is generated through CubeMX, the TouchGFX project can be opened through TouchGFX Designer where the UI is developed. TouchGFX Designer automatically adds any additional generated code files to the target IDE project that was configured for the project in CubeMX.

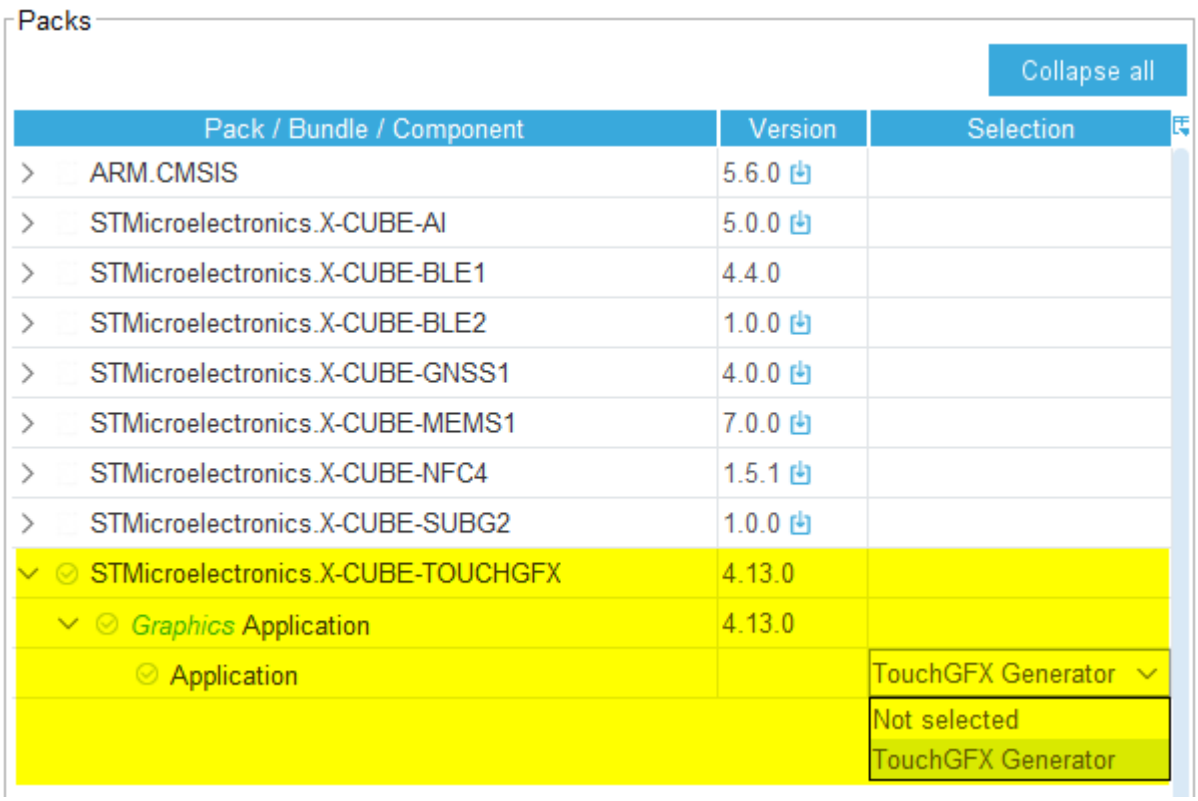
Enabling TouchGFX Generator

The following figure shows a project with TouchGFX Generator already enabled under the *Additional Software* category. Users gain access to adding functionality from X-CUBEs by pressing the "Additional Software" button.



Selecting Additional Software in CubeMx

The following figure shows how TouchGFX Generator can be enabled for a project:



Enabling TouchGFX Generator

Generated Code Architecture

Before describing the features of TouchGFX Generator it is important to understand the architecture of the generated code and how developers can use it to customize configuration and behavior.

Handwritten user code in files generated by CubeMX is protected through the use of **User Code** sections placed strategically throughout the code generated by CubeMX (C code). In the code generated by TouchGFX Generator (C++ code) this flexibility is accomplished through inheritance.

When TouchGFX Generator is added through *Additional Software* and enabled, CubeMX will always create a *TouchGFX* folder for the project. The folder always contains the same files, regardless of configuration, while the content of the files changes according to CubeMX and User configuration.

The listing below shows an overview of the content of a CubeMX project with TouchGFX Generator *enabled*, with emphasis on TouchGFX related files. The table following the list outlines the responsibility of the most important entries.

TouchGFX Folder

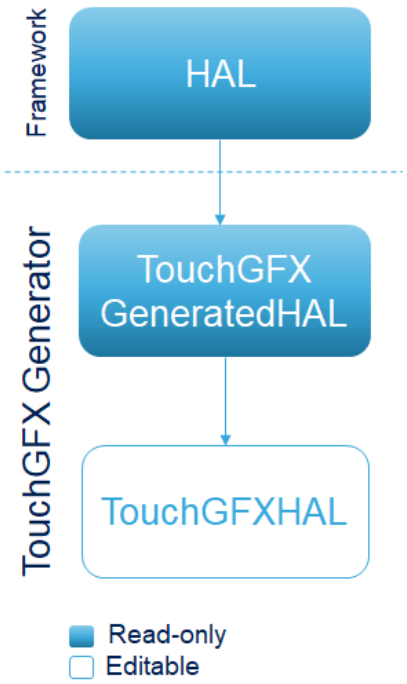
```
|   .mxproject
|   myproject.ioc
├──Core
```

```
├──Drivers
├──EWARM
├──Middlewares
└──TouchGFX
    │   ApplicationTemplate.touchgfx.part
    ├──App
    │       app_touchgfx.c
    │       app_touchgfx.h
    └──target
        │   STM32TouchController.cpp
        │   STM32TouchController.hpp
        │   TouchGFXGPIO.cpp
        │   TouchGFXHAL.cpp
        │   TouchGFXHAL.hpp
        │
        └──generated
                OSWrappers.cpp
                TouchGFXConfiguration.cpp
                TouchGFXGeneratedHAL.cpp
                TouchGFXGeneratedHAL.hpp
```

Folder	Responsibility
myproject.ioc	CubeMX Project file
Core	`main.c` and startup code
Drivers	CMSIS and MCU family drivers
EWARM	IDE project folder. Can be EWARM, MDK-ARM or STM32CubeIDE
Middlewares	Contains TouchGFX library/headerfiles and third party software like FreeRTOS.
ApplicationTemplate.touchgfx.part	The .part file is updated by CubeMX with information that is relevant to TouchGFX Designer project, e.g. screen dimensions and bit depth
App	X-CUBE interface to CubeMX. `app_touchgfx.c` contains definitions for the functions `MX_TouchGFX_Process(void)` and `MX_TouchGFX_Init(void)` which are used to initialize TouchGFX and start its main loop.

Folder	Responsibility
target/generated	This sub-folder contains the read-only files that get overwritten by CubeMX when configurations change. `TouchGFXGeneratedHAL.cpp` is a sub-class of the TouchGFX class `HAL` and contains the code that CubeMX can generate based on its current configuration. `OSWrappers.cpp` (The OSAL) contains the functions required to synchronize with TouchGFX Engine, and finally `TouchGFXConfiguration.cpp` which contains the code to construct and configure TouchGFX, including the HAL.
target	Contains the bulk of files that can be modified by the user to extend the behavior of the HAL or to override configurations generated by CubeMX. `STM32TouchController.cpp` contains an empty touch controller interface. `TouchGFXHAL.cpp` defines a sub-class, `TouchGFXHAL`, of `TouchGFXGeneratedHAL`.

It is important to know that `TouchGFXConfiguration.cpp` contains a function that constructs the HAL and a function that starts the main loop of TouchGFX. Additional configuration can be done in the editable user-class `TouchGFXHAL`. The general architecture of the HAL is seen below:

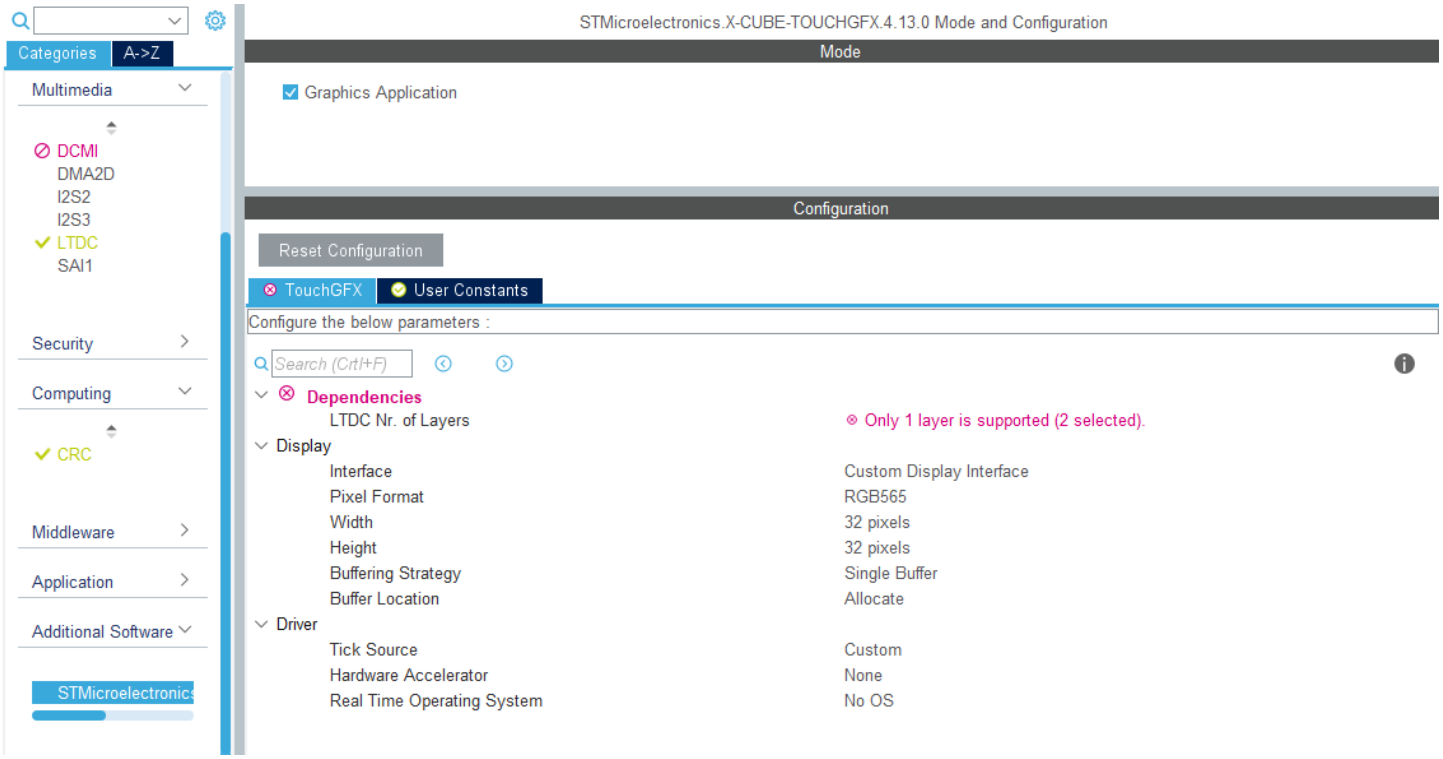


Hierarchy of generated code

Feature Overview

Having enabled TouchGFX Generator, three groups exist in the user interface:

- ***Dependencies*** - This group contains notifications to the developer about dependencies, warnings or concrete errors in the configuration. The group is hidden if no entries exist.
- ***Display*** - This group contains settings related to display such as interface, framebuffer bitdepth, width and height. These settings directly impact the size of the canvas of the TouchGFX project as well as the code generated for assets.
- ***Driver*** - This group allows the user to opt-in for a number of ready-made drivers related to the tick source of the application, graphics acceleration and RTOS. Since CubeMX supports FreeRTOS (CMSIS RTOS v1 and v2) configurations, TouchGFX Generator provides drivers for each of these options.



TouchGFX Generator has three groups: Dependencies, Display, Driver

Display

The *Display* group contains configurations related to display, such as interface, dimensions and buffering strategies.

Interface and dimensions

Multiple display interfaces are usable today with STM32 microcontrollers, e.g.:

- Parallel RGB
- MIPI DSI
- FMC
- SPI

In the case of MCUs with an LTDC *TouchGFX Generator* can generate a driver to transfer the framebuffer to the connected display. For DSI, FMC and SPI interfaces drivers must be implemented by developers themselves.

> **FURTHER READING**
>
> See section **Scenarios** for concrete examples of drivers for different display interfaces.

Buffering Strategies

The following frame buffer strategies can be configured through TouchGFX generator:

- **Single Buffer** - Use only one application frame buffer. Possibly limits performance but uses less memory. Can be used with the "Buffer Location" configuration to place it in internal RAM. For further optimization the user can define a function that returns the current line being processed by the display controller. This method is used by the framework to allow updates to memory that has already been transferred to the display during this frame.
- **Double Buffer** - Use two frame buffers. Usually allows for better performance at the cost of memory.
- **Partial Buffer** - Use one or more user defined chunks of memory as the frame buffer. This strategy is targeted at low cost solutions that do not rely on external RAM, but have displays for which a full frame buffer would exceed available memory.

In the case of Single Buffer and Double Buffer users are allowed to configure their location through the "Buffer Location" configuration which offers the following options:

- **By Allocation** - Lets the linker place frame buffer memory according to linker script. Default is in internal RAM.
- **By Address** - Allows the user to define one (Single) or two (Double) frame buffer addresses.

The **Partial Buffer** strategy allows the user to define the following parameters:

- Number of blocks (always placed in internal RAM)
- Block size (bytes)

To understand some core concepts regarding the Partial Buffer strategy please read the dedicated article on Lowering Memory requirements using partial Frame Buffers. The article shows, conceptually, how to achieve partial frame buffers and the code shown in this article will differ slightly from what is generated by TouchGFX Generator. Please see Frame Buffer Strategies for concrete examples of the generated code for these strategies.

Driver

The driver section allows developers to select drivers for various responsibilities of a TouchGFX AL.

Application Tick Source

The application tick source for an application defines how to drive an application forward. The developer has the following options:

- **LTDC** - If LTDC is selected as the Interface in the "Display" group the Application Tick Source can be "LTDC". This means that TouchGFX Generator will install a driver function (LTDC interrupt handler) in `TouchGFXGeneratedHAL` class that drives the application forward by calling `OSWrappers::signalVSync()`.
- **Custom** - In this case, the developer is required to implement a handler that drives the application forward by calling `OSWrappers::signalVSync()` repeatedly.

Graphics Accelerator

The developer has three options when it comes to graphics acceleration:

- **None** - The application uses only the CPU to render frames.
- **Chrom-ART (DMA2D)** - The application uses the Chrom-ART chip when possible to move and blend pixels, freeing up CPU cycles. The driver is installed by TouchGFX Generator and does not require any action from the developer.

The Chrom-ART (DMA2D) driver supplied by TouchGFX Generator supports two ways of receiving a TransferCompleteInterrupt.

1. Uses the STM32Cube HAL driver where it registers a callback funtion to the dma2d handle `hdma2d.XferCpltCallback`.
2. Uses the `DMA2D_IRQHandler()` interrupt handler directly.

Switching between these two is done by enabling or disabling the DMA2D global interrupt in the NVIC Settings in CubeMX for DMA2D IP. Enabling the global interrupt generated code for option 1), disabling the global interrupt generates code for option 2).

NOTE

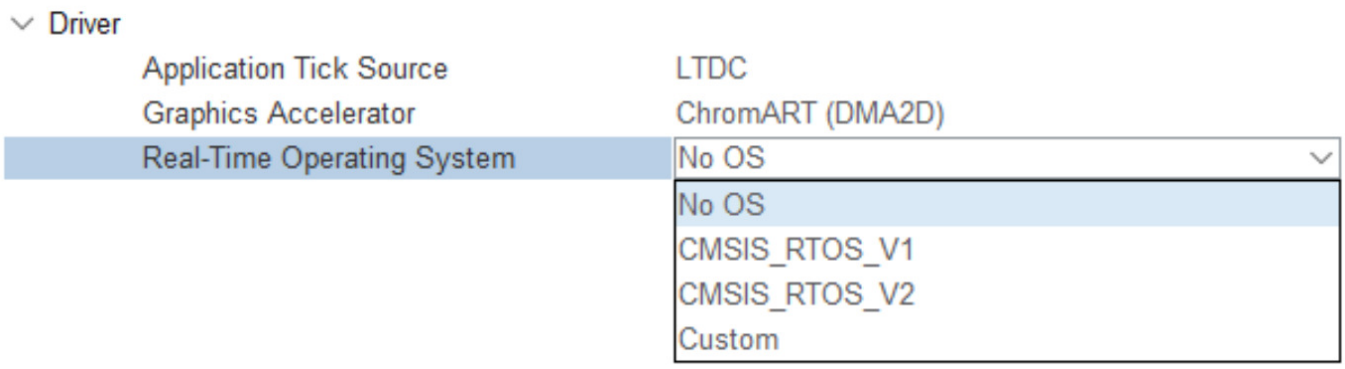
- **When using global interrupt for DMA2D, ensure that the "IRQ handler" calls the "DMA2D HAL handler", this is default behaviour.**
- **If disabling "IRQ handler" and "Call HAL handler" for DMA2D while global interrupt is enabled will cause the registered callback to never be called.**

Real-Time Operating System

Developers can use any RTOS with TouchGFX (even No OS). As described in the Abstraction Layer Architecture the TouchGFX Engine uses the `OSWrappers` interface to synchronize its main event loop as well as framebuffer access with the users choice of RTOS.

FreeRTOS can be configured directly from within CubeMX and the TouchGFX Generator granting the user generated code for both task definitions and TouchGFX RTOS driver. TouchGFX Generator can generate CMSIS V1 and CMSIS V2 compliant RTOS drivers which work with any CMSIS compliant RTOS. In this case, developers cannot rely on CubeMX for code generation of task definitions and this must be done in *user code*.

The following figure shows the options available through the TouchGFX Generator.



RTOS driver options

The TouchGFX main loop is entered when calling the following function.

```
void MX_TouchGFX_Process(void);
```

Developers are required to call this function in the task handler for the task they intend to run the TouchGFX application in. If the user configured a FeeRTOS task from CubeMX called `DefaultTask` then the following example shows how `MX_TouchGFX_Process()` is called to start TouchGFX in the user code section of its task handler.

```
void StartDefaultTask(void *argument)
{
  /* USER CODE BEGIN 5 */
  MX_TouchGFX_Process();
  /* USER CODE END 5 */
}
```

When FreeRTOS is enabled, CubeMX will also generate a call to `osKernelStart();` which starts the scheduler.

Other CMSIS compliant OS

When developers require a different CMSIS compliant OS than what CubeMX can offer (FreeRTOS) he must perform RTOS configuration and task definition manually. Generally, the following manual steps are required:

1. Configure the RTOS
2. Define a task to run TouchGFX (`MX_TouchGFX_Process`)
3. Start the scheduler

Here's an example of how to perform steps 2 and 3 for *Azure RTOS*. Since CubeMX cannot help with any of this configuration everything must be done in the provided *user code sections* to avoid code being overwritten. The following code shows pseudo code for the GUI task definition. Generally, any code that is not generated by CubeMX should be placed in *user code sections* that are scattered throughout the file `main.c` .

```
/* BEGIN USER CODE SECTION */
#include "tx_api.h"

#define GUI_THREAD_STACK_SIZE         1024
TX_THREAD gui_thread;
void    gui_thread_entry(ULONG thread_input); //Thread prototype
/* END USER CODE SECTION */

int main()
{
  /* BEGIN USER CODE SECTION - Choose an appropriate one from main.c */
  /* Allocate the stack for gui thread  */
  tx_byte_allocate(...);

  /* Create the gui thread.  */
  tx_thread_create(&gui_thread, "GUI Thread", gui_thread_entry, 0,
              pointer, GUI_TASK_STACK_SIZE,
              1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

  /* END USER CODE SECTION*/
```

Call `MX_TouchGFX_Process` to start the TouchGFX Engine Main Loop inside the task handler.

```
/* BEGIN USER CODE SECTION */
void gui_thread_entry(ULONG thread_input)
{
  MX_TouchGFX_Process();
```

```
}
/* END USER CODE SECTION*/
```

Start the scheduler to start the GIU task and your TouchGFX Application.

```
/* BEGIN USER CODE SECTION */
tx_kernel_enter();
/* END USER CODE SECTION*/
```

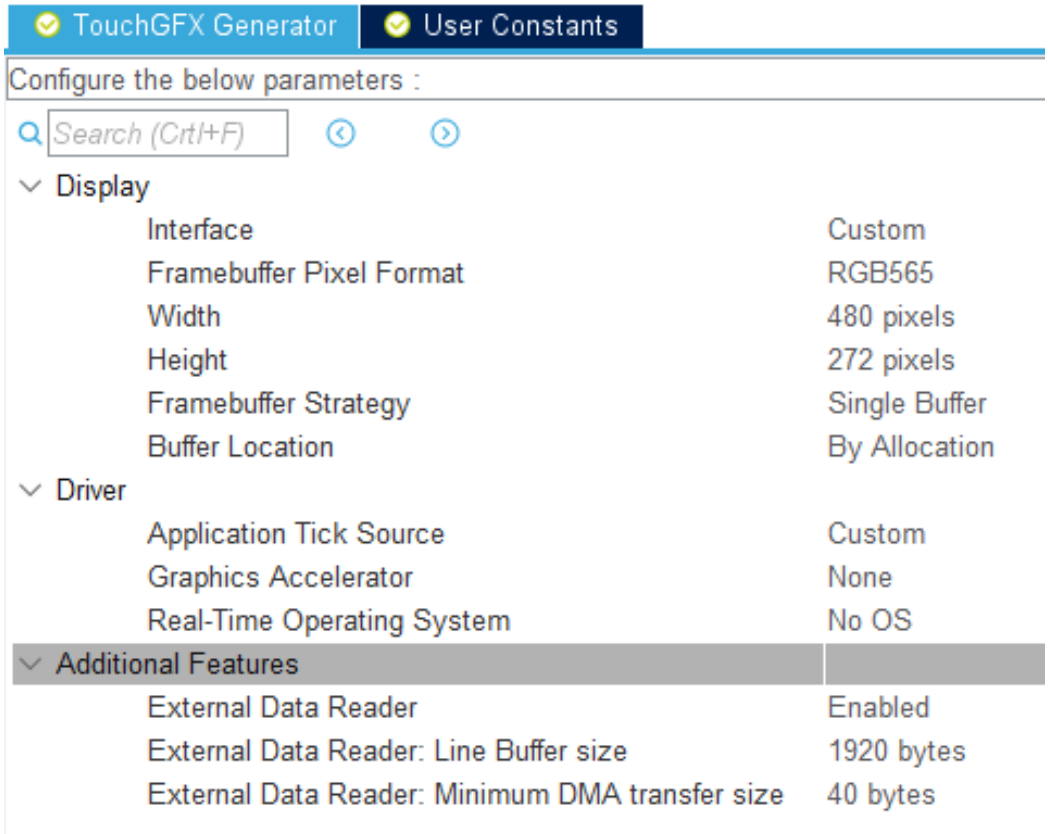
Additional features

External Data Reader

For the RGB565 Framebuffer Pixel Format touchgfx supports a so called *Data Reader* interface that allows developers to read data directly from a non-memory-mapped serial flash instead of caching which comes at the cost of an additional buffer in memory. Please see the SerialFlash article for a example on how to implement a DataReader to retrieve application assets from a non-memory mapped flash chip.

The *Data Reader* option is typically used for low cost solutions (e.g. STM32G0) that do not have enough memory for additional buffers. It cannot be enabled if DMA2D is also enabled.

Once *RGB565* is selected as the Framebuffer Pixel Format, *Additional Features* group becomes available.



The following configurations can be made by the developer:

- **External Data Reader**: Enable or Disable the feature. Enabling will cause TouchGFX to retrieve data for assets directly through the generated interface. If disabled, developers are then required to *cache* images to a buffer in memory instead.
- **External Data Reader: Line Buffer Size**: Creates two buffers for blending images or text into the framebuffer. Default value is one screen width*4 bytes to support full size images in ARGB8888 pixel format.
- **External Data Reader: Minimum DMA transfer size**: Set minimum required bytes to start a DMA transfer. If fewer bytes are requested, DMA will not be used.

After generating code with *External Data Reader* enabled, the following, additional files are created to support the retrieval of assets directly from a non-memory mapped flash.

- `TouchGFX/target/generated/TouchGFXGeneratedDataReader.cpp`
- `TouchGFX/target/generated/TouchGFXGeneratedDataReader.hpp`
- `TouchGFX/target/TouchGFXDataReader.cpp`
- `TouchGFX/target/TouchGFXDataReader.hpp`

As usual, for code generated by TouchGFX Generator, `TouchGFXGeneratedDataReader` is read-only and user modifications should be made inside the `TouchGFXDataReader` class. `TouchGFXGeneratedDataReader` is of type `touchgfx::FlashDataReader`.

Modifications will be made to the following files to configure TouchGFX HAL to use the `DataReader`.

- `TouchGFX/target/generated/TouchGFXConfiguration.cpp`
- `TouchGFX/target/generated/TouchGFXGeneratedHAL.cpp`
- `TouchGFX/target/generated/TouchGFXGeneratedHAL.hpp`

> **NOTE**
>
> **The DataReader Additional Feature is only available if DMA2D and LTDC are disabled.**

8bit LTDC Color Look-up Table

When the LTDC is configured to read the framebuffer in L8 format and TouchGFX renders in either ABRG2222, ARGB222, BGRA2222, or RGBA2222, TouchGFX Generator will provide a CLUT which is loaded into the LTDC during `TouchGFXHAL::initialize()`. Please refer to the STM32 MCU reference manual for more details on usage of LTDC and CLUT.

Generated project

TouchGFX works with (at least) the following IDEs when generating code using the **Generate Code** button in CubeMX:

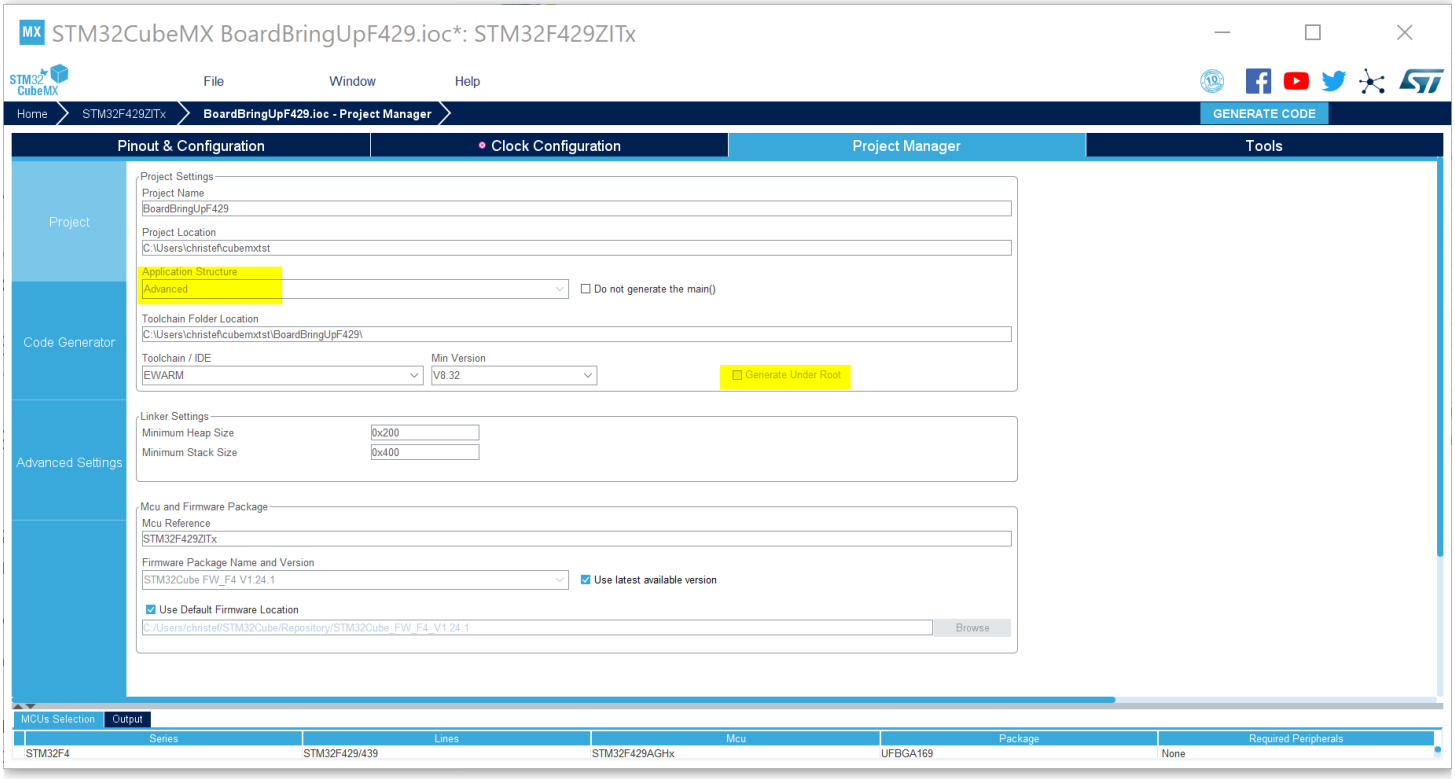
1. EWARM
2. MDK-ARM
3. STM32CubeIDE

For optimal project structure select the following options for project generation:

- Application structure: **Advanced**
- Disable **Generate under root** (STM32CubeIDE only)

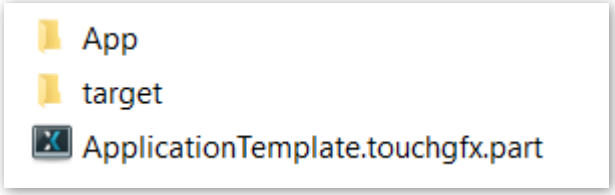
For optimal project structure select the following options for project generation:

- Application structure: **Advanced**
- Disable **Generate under root** (STM32CubeIDE only)



Select Advanced application structure and deselect Generate under root

CubeMX will also generate a *TouchGFX* folder with the following structure:



TouchGFX folder structure

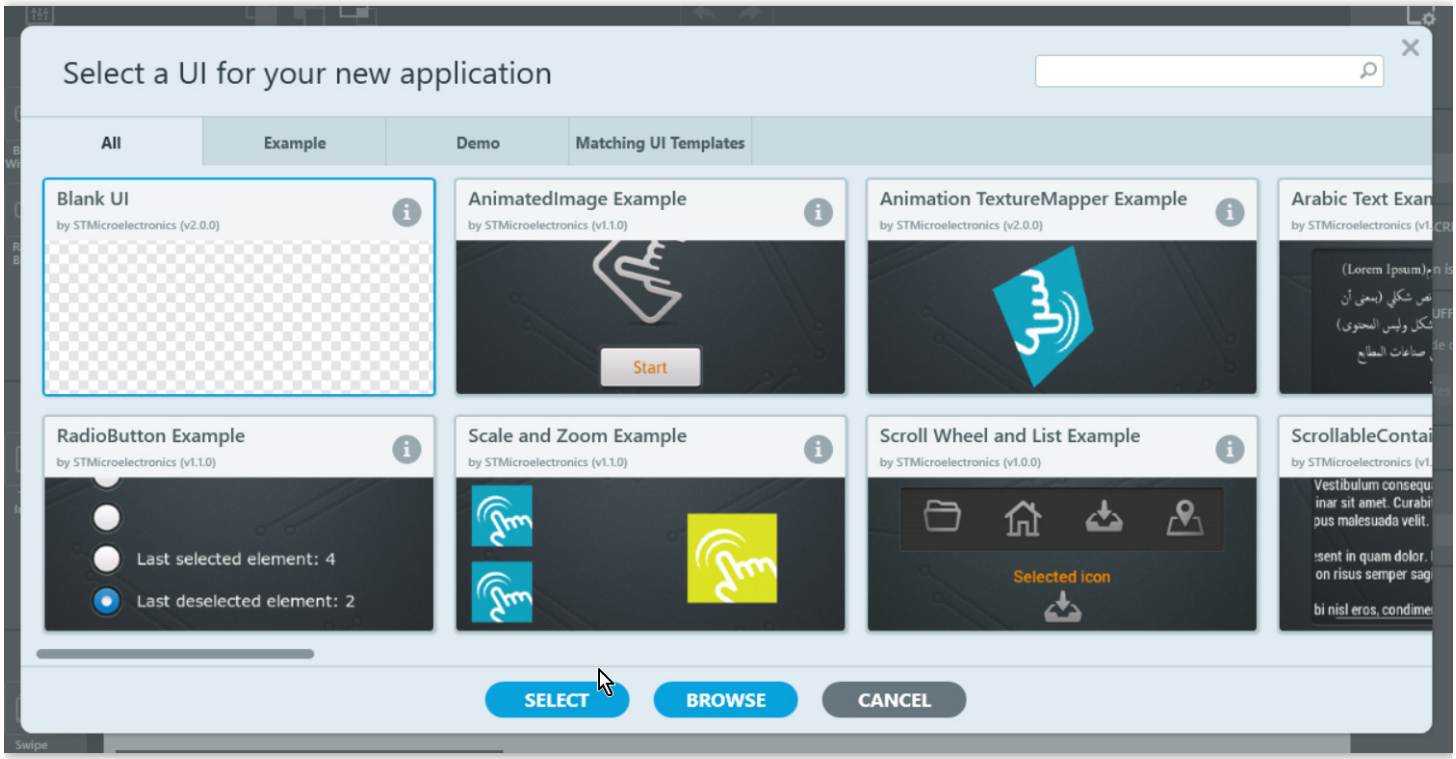
- The *App* folder which contains code to initialize and start TouchGFX.
- The *target* folder which contains read-only, generated code (inside generated/) and modifiable user classes (`STM32TouchController.cpp` , `TouchGFXGPIO.cpp` and `TouchGFXHAL.cpp`)
- The *.part* file which is opened using the TouchGFX Designer in order to create a full TouchGFX project complete with TouchGFX header files and libraries The part file contains relevant application information such as pixel format, and canvas dimensions that the designer uses when generating TouchGFX application code.

TouchGFX Designer Project

The following code is an example of the contents of the `.part` file mentioned in the Generated Code Architecture section. The post-generate command, seen below, will update the project selected in CubeMX (e.g. EWARM) when new files are created by the TouchGFX designer (e.g. new screens and assets).

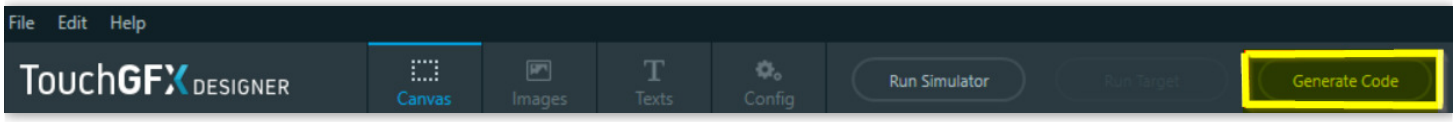
```
{
 "Application": {
 "Name": "my_project",
 "TouchGfxPath": "../Middlewares/ST/touchgfx",
 "AvailableColorDepths": [ 16 ],
 "AvailableLCDs":
  {
    "16": "LCD16bpp"
 },
 "AvailableResolutions" :
 [
  {
    "Width": 320,
    "Height": 240
  }
 ],
 "PostGenerateTargetCommand" : "touchgfx update_project --project-file=../my_project.ioc -
 },
 "Version": "4.13.0"
}
```

When opening the *.part* file with TouchGFX Designer developers are presented with the option to load a concrete UI or start from a blank template.

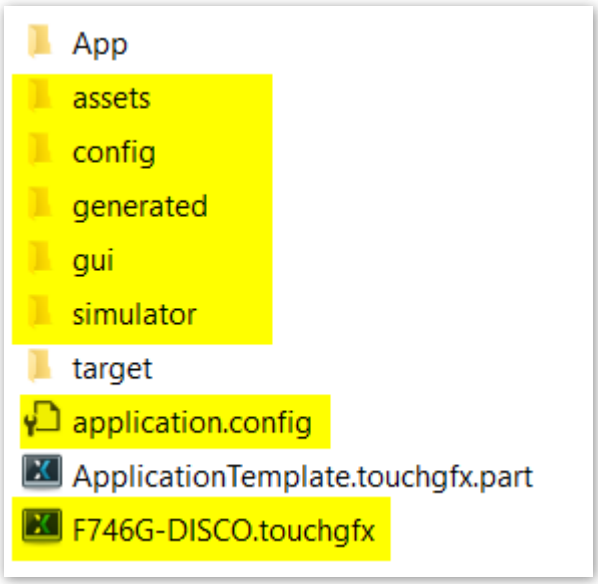


Choose UI

After pressing *Generate Code* in TouchGFX Designer the structure of the TouchGFX folder now looks like the following. The following image shows a concrete example of a TouchGFX folder structure and highlights the files and folders that are new after generation.



Generate Code



TouchGFX Folder Structure

TouchGFX will detect the selected IDE from the .ioc CubeMX file (For STM32CubeIDE, EWARM, MDK-ARM) and update the project file with new, generated files like files for screen definitions, image- and font assets.

At this point, developers can work interchangeably in CubeMX, TouchGFX Designer and toolchain/IDE where:

- CubeMX updates the IDE project with drivers
- CubeMX updates the TouchGFX *.part* file with UI related changes that are instantly picked up by the designer
- CubeMX generates HAL code (TouchGFX/target/generated/) based on TouchGFX Generator Configuration necessary for TouchGFX to work on a specific platform.
- The TouchGFX designer updates the project with generated code.

Modifying Generated Behavior

It important to know that, due to the class hierarchy of the HAL, users can override HAL configuration or behavior that was generated by CubeMX. In the example below, developers can modify the `initialize` function to configure TouchGFX additionally or to modify an existing configuration set in `TouchGFXGeneratedHAL`.

TouchGFXHAL.cpp

```
void TouchGFXHAL::initialize()
{
    // Calling parent implementation of initialize().
    //
    // To overwrite the generated implementation, omit call to parent function
    // and implemented needed functionality here.
    // Please note, HAL::initialize() must be called to initialize the framework.

    TouchGFXGeneratedHAL::initialize();

    //Overriding configurations
    hal.lockDMAToFrontPorch(true);
    hal.setFingerSize(4);
    hal....
}
```

Upgrading Projects

TouchGFX Generator parameters are stored in .ioc files (CubeMX project). When a new version of TouchGFX Generator is released the parameters of the old version may be incompatible with the new version and may require migration.

Since cubeMX does not support upgrading between X-CUBE versions the upgrade is automatically performed by TouchGFX Designer when *Generate Code* is pressed due to the following command in the `PostGenerateTargetCommand` section of the `.touchgfx` file.

.touchgfx

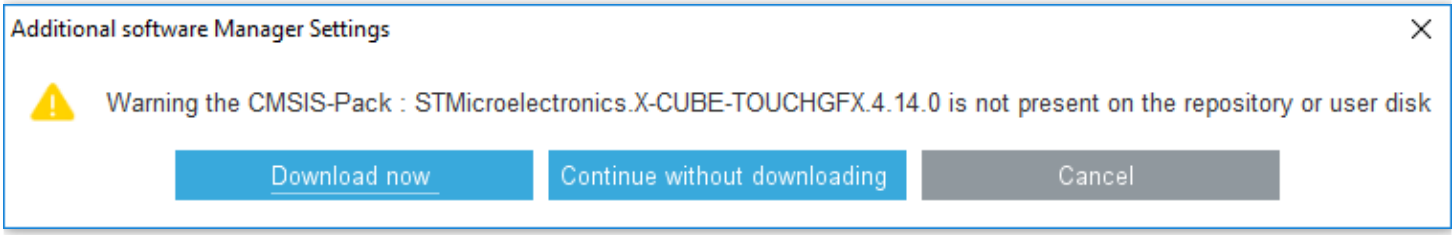
```
"PostGenerateTargetCommand" : "touchgfx update_project --project-file=../upgrade.ioc --pla
```

The command will read the `.ioc` file and update the parameters to fit the current version of X-CUBE-TOUCHGFX. Below is an example of running the script (X-CUBE-TOUCHGFX 4.14.0) by hand on an .ioc file created with X-CUBE-TOUCHGFX 4.13.0.

Upgrade example using STM32F746 DISCO Application Template from 4.13.0 to 4.14.0

```
$ touchgfx update_project --project-file=../STM32F746G_DISCO.ioc
  TouchGFX Generator 4.13.0 found
  Creating backup of ../STM32F746G_DISCO.ioc as ../backup_STM32F746G_DISCO.ioc
  Performing upgrade 4.13.0 -> 4.14.0 ... OK
```

Opening the updated project with CubeMX prompts the user to install the version of X-CUBE-TOUCHGFX that is represented by the .ioc file (if not already installed). Clicking *Download now* will download and install X-Cube-TouchGFX-4.14.0.



Additional Software Component Missing: TouchGFX Generator 4.14.0

All configurations in TouchGFX Generator will be kept during the upgrade procedure and a backup of the .ioc file will be placed beside the original on prepended with `backup_`.

NOTE

To use the new features provided by TouchGFX Generator, *Generate Code* must be performed in CubeMX.

CAUTION

If upgrading X-CUBE-TOUCHGFX through CubeMX for an existing TouchGFX Project and the upgrade process is not run by TouchGFX Designer, TouchGFX Generator parameters will be reset to default since they are applicable to a different version.

Version: 4.16

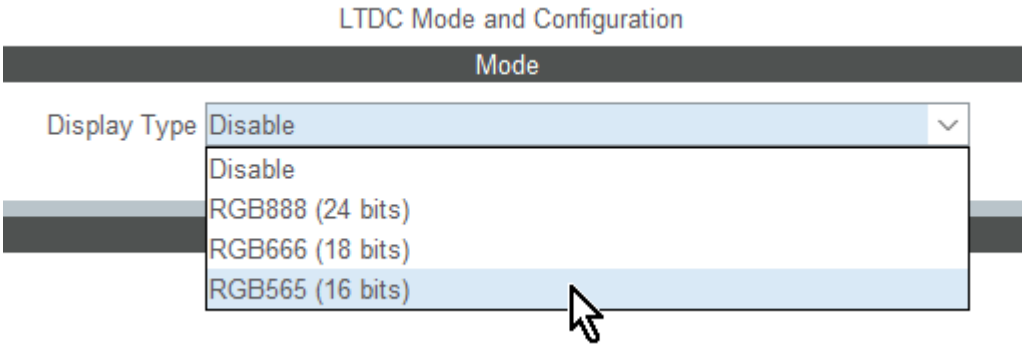
LTDC/Parallel RGB

For MCUs with a TFT controller (e.g. STM32F429, STM32F746, STM32H7), the *TouchGFX Generator* can generate the part of the HAL that configures the LTDC to transfer pixels from the framebuffer memory to the display. The generated code both starts the correct framebuffer transfer and unblocks the TouchGFX Engine main loop by calling `OSWrappers::signalVSync()` once a VSYNC interrupt is raised by the LTDC.

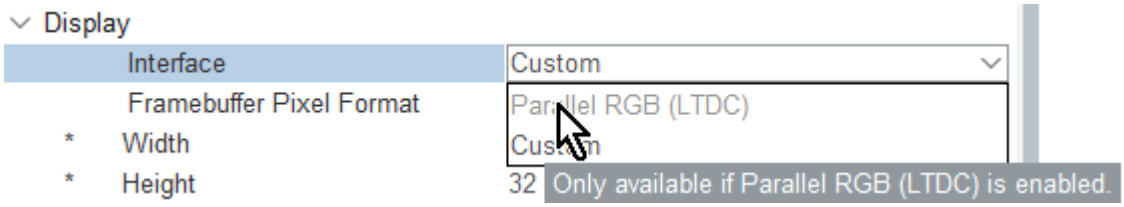
Display Interface

As opposed to a "Custom" display interface, where the developer must implement the whole driver by hand, for LTDC the TouchGFX Generator can generate all the code necessary for the TouchGFX HAL to support an LTDC configuration.

For "Parallel RGB (LTDC)" to be a selectable option through the TouchGFX Generator the *LTDC* must be enabled from the *Multimedia* group in the CubeMX category list.



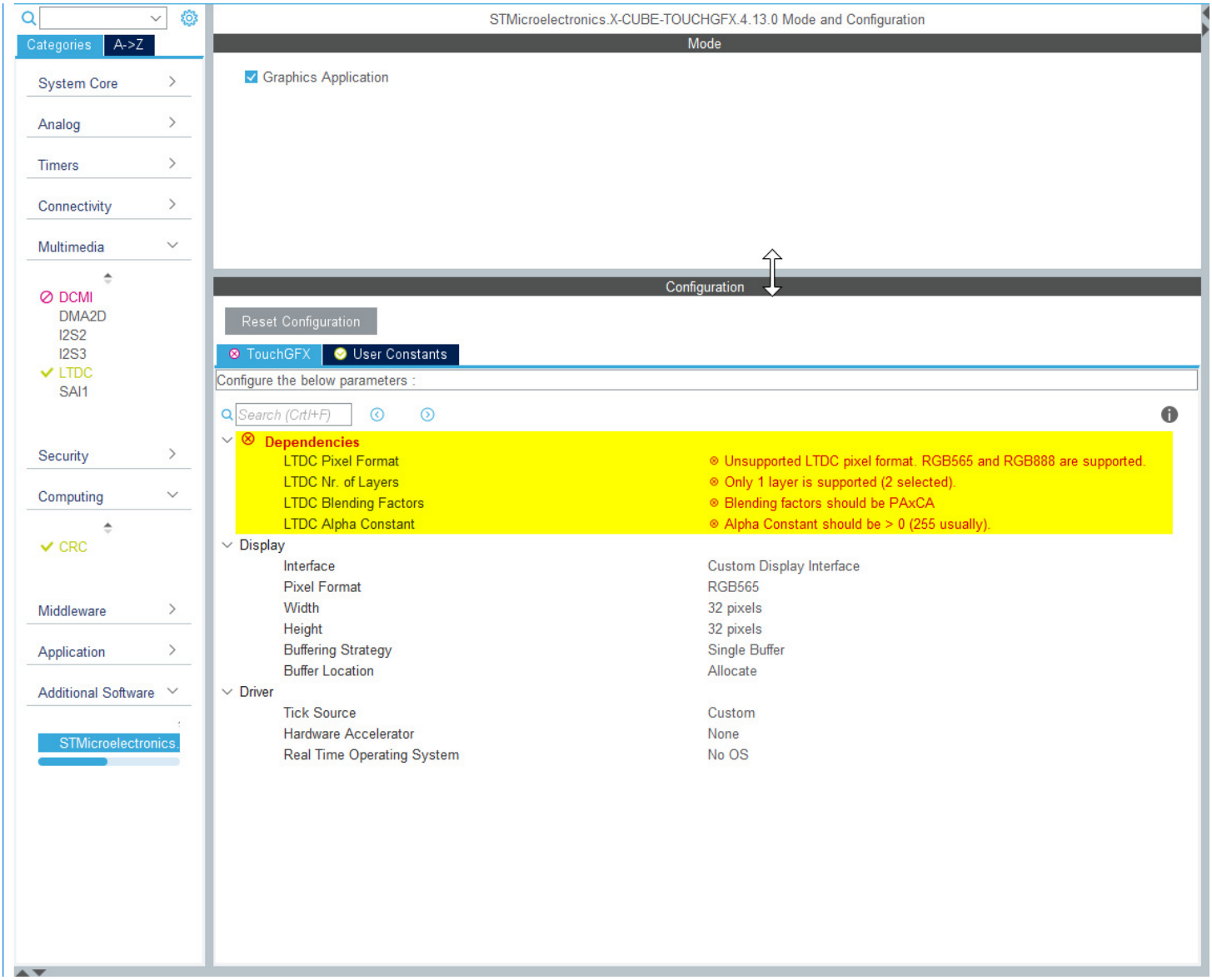
Once LTDC is enabled, the *Parallel RGB (LTDC)* option becomes available through the *Display* section of the TouchGFX Generator.



Even after LTDC is enabled through CubeMX, some work is required in order to:

1. Configure LTDC (GPIO and timings) to match connected display specifications
2. Configure LTDC to match desired TouchGFX application specifications.

The TouchGFX Generator can read various configurations from CubeMX and provide a list of warnings, recommendations or errors that are called *Dependencies*. The image below shows the list of dependencies present when initially enabling LTDC in CubeMX for any MCU (in this example we used an F429):



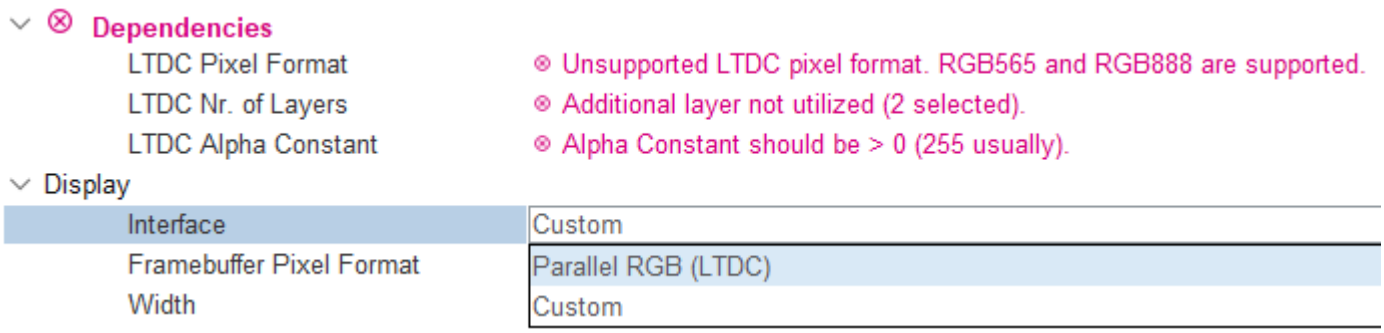
NOTE

LTDC recommendations, warnings and errors will be visible in the TouchGFX Generator interface as soon as LTDC is enabled through CubeMX.

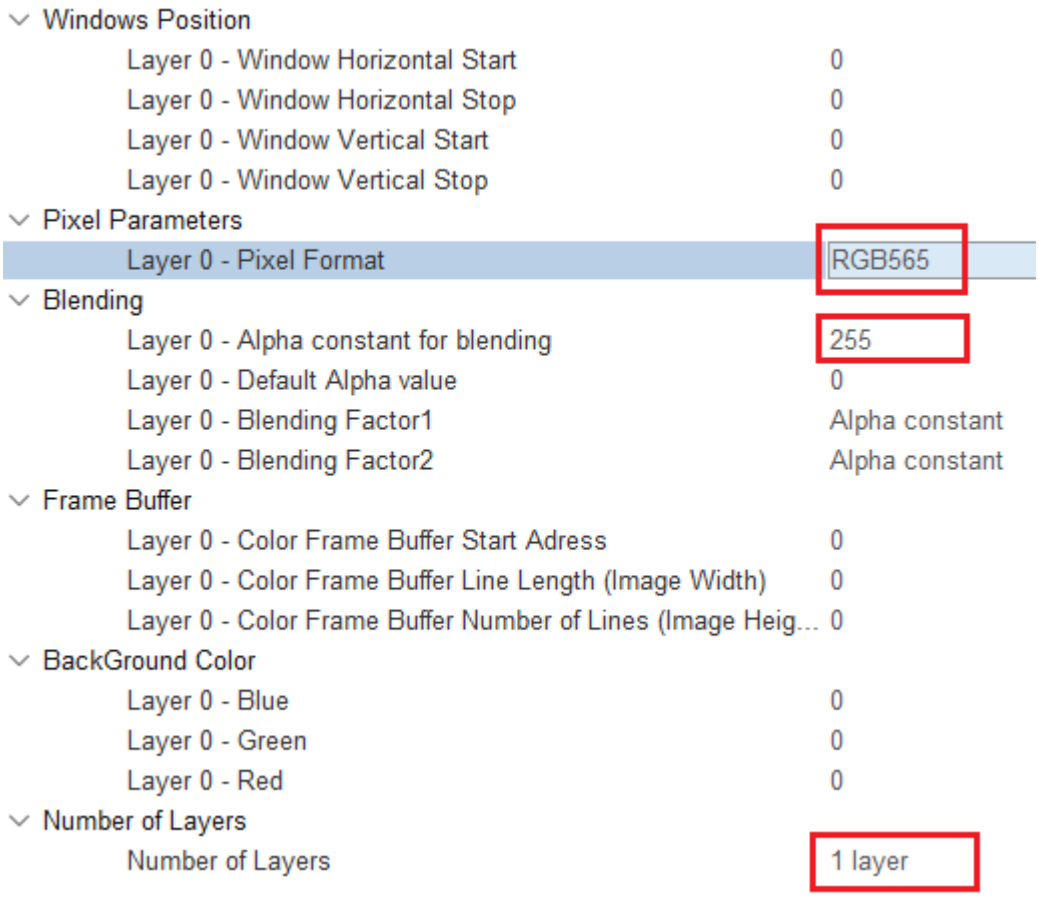
Dependency	Description
Unsupported pixel format	The framebuffer drivers of TouchGFX are only available in RGB565 (16-bit) and RGB888 (24-bit). The pixel format of the LTDC must match the chosen driver for the TouchGFX HAL.

Dependency	Description
Additional layers configured	TouchGFX is only capable of utilizing a single layer. While TouchGFX applications can work with two layers enabled, this is a warning to the user that the LTDC is potentially misconfigured. Change the number of layers in the LTDC block.
Blending factors should be PAxCA	By default, the blending factor is Alpha Constant. This should be Alpha Constant x Pixel Alpha
Alpha Constant is 0	By default, the alpha constant of LTDC layers is 0. This should be > 0 and preferably 255 unless there is an intent to have a global alpha at all times in an application.

Remember to actually select the Parallel RGB (LTDC) display interface after enabling the LTDC peripheral in the Multimedia section.

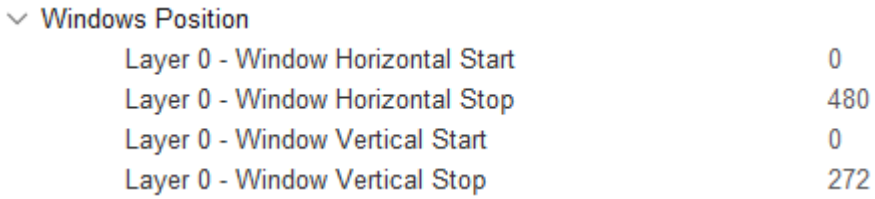


The following image shows the LTDC configuration that satisfies the conditions of the warnings, causing the Dependencies group to disappear from the TouchGFX Generator interface.

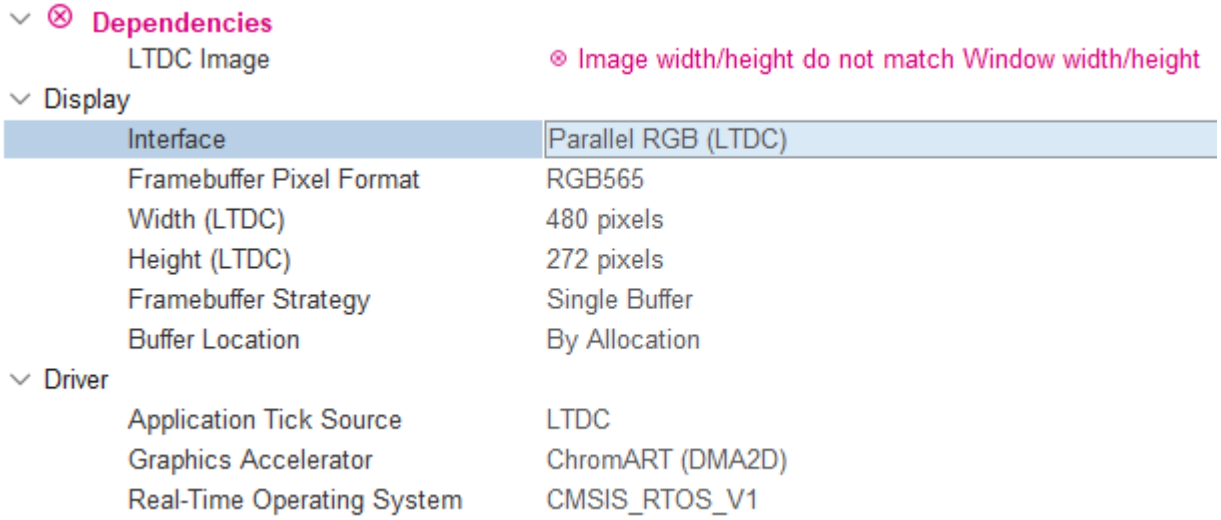


Reading settings from CubeMX

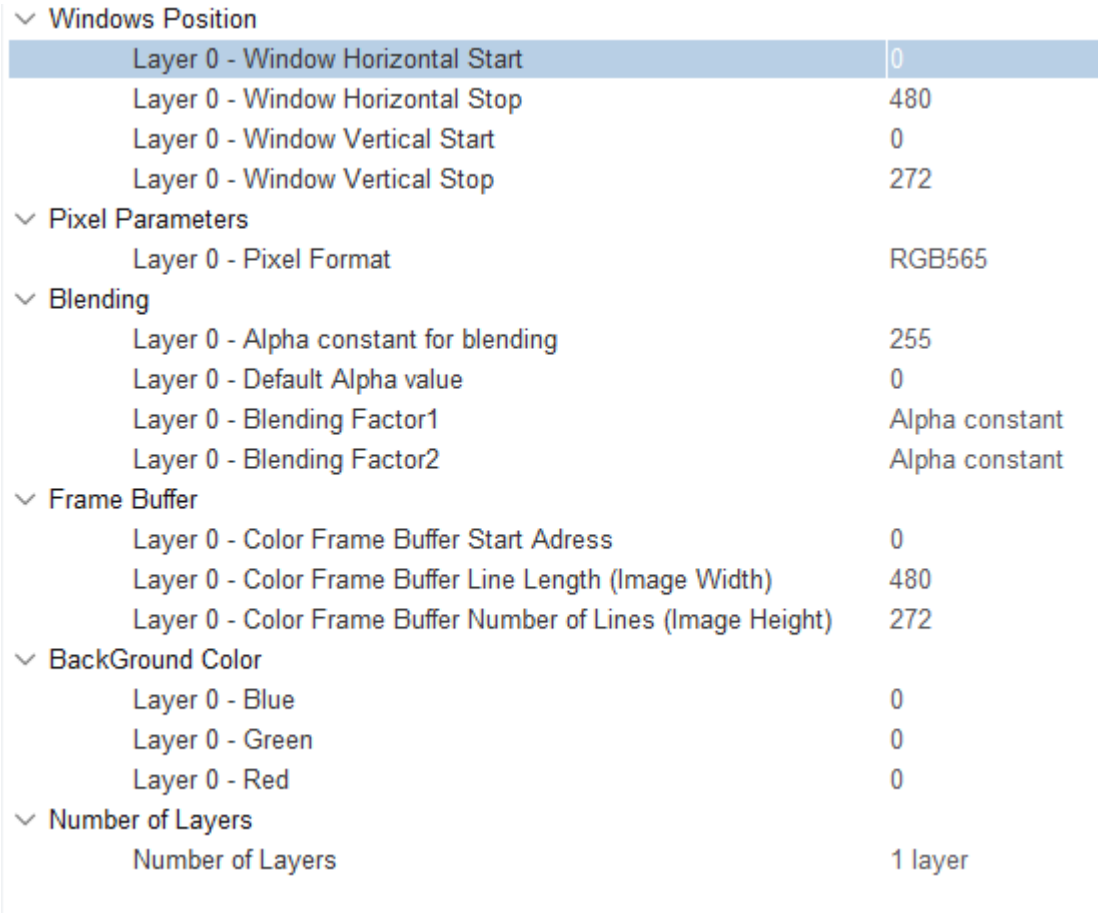
By selecting *Parallel RGB (LTDC)* as the display interface through TouchGFX Generator, the *width* and *height* of the framebuffer is inherited from the LTDC configuration *horizontal start/stop* and *vertical start/stop*.



Defining the dimensions of Layer *0* according to the display and application dimensions a new entry in the *Dependency* window appears.



Ensuring that Framebuffer *Image Width* and *Image Height* match the size of the window, which is usually desired, will satisfy the warning.

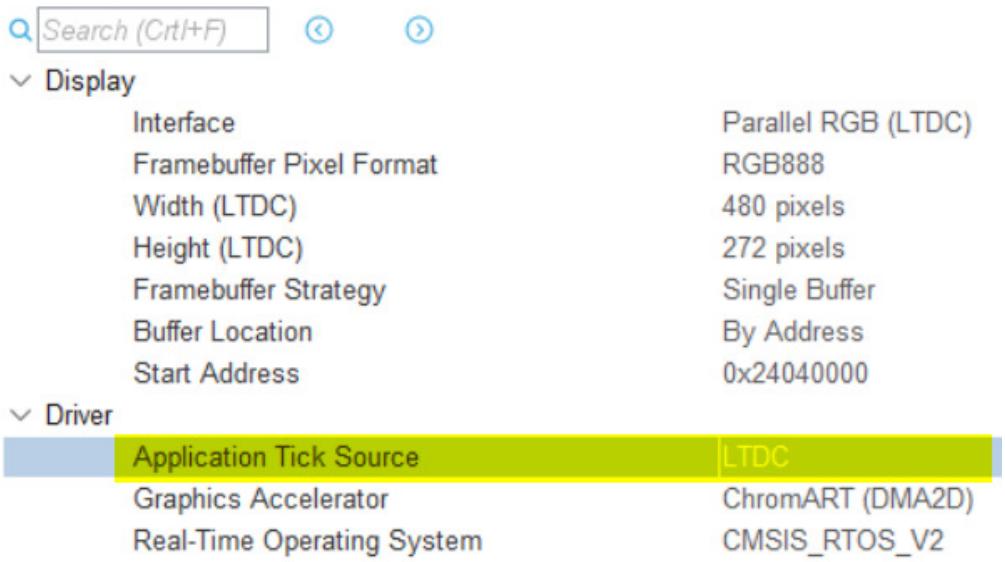


CAUTION

TouchGFX Generator inherits the **Width** and **Height** values from the LTDC configuration, if LTDC is enabled. However, **Width** and **Height** can still be modified from the TouchGFX Generator interface. Changing these values can lead to a configuration mismatch if they do mot respect the Window LTDC Layer configuration.

TouchGFX Driver / VSYNC Signal

Once *Parallel RGB (LTDC)* is selected as Display Interface, developers gain access to the *LTDC* Application Tick Driver or TouchGFX Driver.

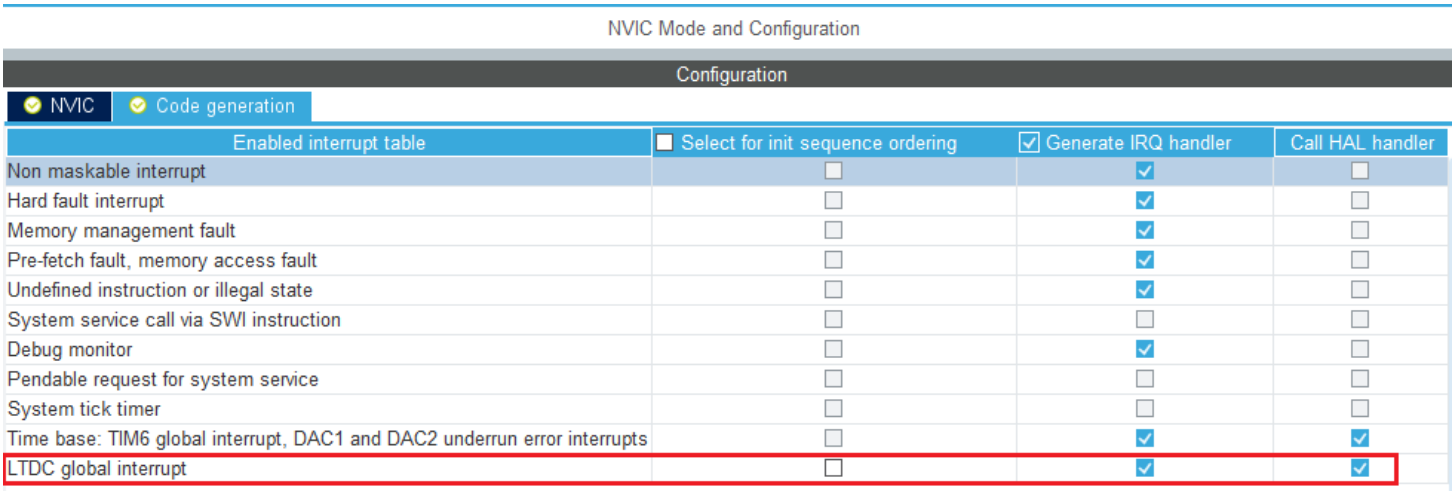


The following code is the interrupt handler (STM32F7) for the LTDC interrupt generated according to LTDC configuration. The generated handler automatically unblocks the TouchGFX Engine main loop.

```
extern "C"
irq void LTDC_IRQHandler(void)
{
    if (LTDC->ISR & 1)
    {
        LTDC->ICR = 1;
        if (LTDC->LIPCR == (LTDC->AWCR & 0x7FF) - 1)
        {
            //entering active area
            OSWrappers::signalVSync();
        }
    }
}
```

NOTE

For the LTDC driver to work, users must enable the LTDC global interrupt through the LTDC NVIC settings or through Global NVIC settings, and also enable generation of handler code.


NVIC Mode and Configuration

Configuration

NVIC | Code generation

Enabled interrupt table	Select for init sequence ordering	Generate IRQ handler	Call HAL handler
Non maskable interrupt	☐	☑	☐
Hard fault interrupt	☐	☑	☐
Memory management fault	☐	☑	☐
Pre-fetch fault, memory access fault	☐	☑	☐
Undefined instruction or illegal state	☐	☑	☐
System service call via SWI instruction	☐	☐	☐
Debug monitor	☐	☑	☐
Pendable request for system service	☐	☐	☐
System tick timer	☐	☐	☐
Time base: TIM6 global interrupt, DAC1 and DAC2 underrun error interrupts	☐	☑	☑
LTDC global interrupt	☐	☑	☑

Conclusion

Enabling the *Parallel RGB (LTDC)* display interface option through TouchGFX Generator allows all required HAL code to be generated.

- Sets the width, height and pixel format of the TouchGFX application in accordance with the CubeMX LTDC configuration. The LTDC layer *width* and *height* impact the size of the canvas in TouchGFX Designer and the LTDC Pixel Format impacts the required TouchGFX framebuffer driver and also the format for generated assets.
- Allow the LTDC application tick source to be selected which generates a handler to drive the TouchGFX Engine Main loop. Usually, with LTDC Configurations developers would always use the provided Application Tick Driver.

Version: 4.16

FMC and SPI Display Interface

The following scenario shows, generally, the steps involved in writing a TouchGFX driver when selecting *Custom* display interface in the TouchGFX generator using an LCD connected to either an FMC or through SPI.

FURTHER READING

the STM32L496-DISCO Application template available from the designer uses FMC and can be inspected for inspiration on how to implement a TouchGFX display driver.

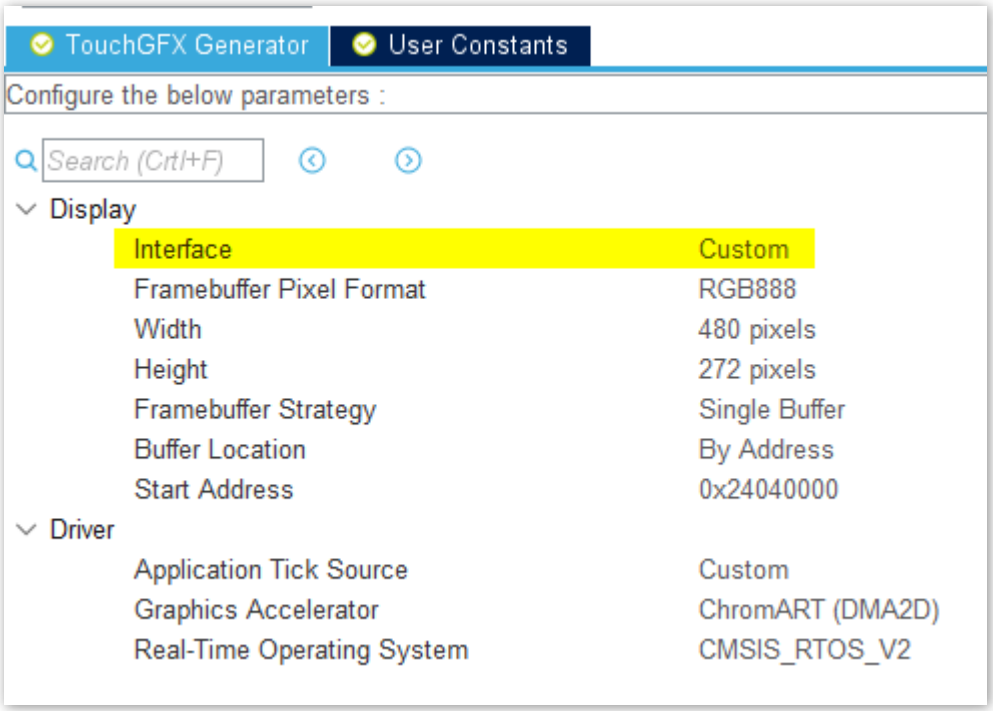
The process of writing a TouchGFX display driver for MCUs without embedded display controllers over FMC or SPI is identical. The scenario described in this section uses an ST7789H2 LCD Controller to exemplify.

Once FMC or SPI is configured according to board specifications in CubeMX the TouchGFX Generator can be used to generate a HAL, selecting the *Custom* display interface, which allows developers to write custom code to transfer the updated parts of the application framebuffer to a connected display.

The figure below shows a TouchGFX Generator configuration with *Custom* Display Interface selected. This instructs the TouchGFX Generator that the developer would like to configure and transfer pixels from the framebuffer memory to the display manually and generates the handles to accomplish this.

NOTE

The driver code shown in this section for the ST7789H2 would have been developed during the Board Bringup phase and, once working, can more or less be copied to the HAL class generated by the TouchGFX Generator.
The driver must be able to transfer pixels to the display, and to control the memory writing position of the display. Check its datasheet to find appropriate commands, outlined below, and further details.



TouchGFX Generator Configuration

Generally, for displays with embedded GRAM such as 8080 or SPI displays, the driver works as follows:

1. Based on the area of the framebuffer to be redrawn, move the "display cursor" and "active window" to a place in GRAM that matches this area.
2. Prepare to write incoming pixel data to GRAM.
3. Send pixel data.

Transferring the framebuffer

When an area of the framebuffer has been updated, the TouchGFX Engine calls `HAL::flushFrameBuffer(Rect r)`. This function can be overridden when developers must implement a driver for a "Custom" display interface.

```
void  TouchGFXHAL::flushFrameBuffer(const Rect& rect)
{
    /* Set Cursor */
    __ST7789H2_SetDisplayWindow(rect.x, rect.y, rect.width, rect.height);

    /* Prepare to write to LCD RAM */
    ST7789H2_WriteReg(ST7789H2_WRITE_RAM, (uint8_t*)NULL, 0);

    /* Send Pixels */
    this->copyFrameBufferBlockToLCD(rect);
}
```

The following function `__ST7789H2_SetDisplayWindow` sets the `x` and `y` coordinates for the virtual "cursor" in GRAM by writing to specific registers, which is usual for displays using GRAM.

```
extern "C"
void __ST7789H2_SetDisplayWindow(uint16_t Xpos, uint16_t Ypos, uint16_t Width, uint16_t He
{
    uint8_t   parameter[4];

    /* CASET: Column Address Set */
    parameter[0] = 0x00;
    parameter[1] = Xpos;
    parameter[2] = 0x00;
    parameter[3] = Xpos + Width - 1;
    ST7789H2_WriteReg(ST7789H2_CASET, parameter, 4);

    /* RASET: Row Address Set */
    parameter[0] = 0x00;
    parameter[1] = Ypos;
    parameter[2] = 0x00;
    parameter[3] = Ypos + Height - 1;
    ST7789H2_WriteReg(ST7789H2_RASET, parameter, 4);
}
```

The following function `TouchGFXHAL::copyFrameBufferBlockToLCD` is a private function that sends one line of the updated area (`Rect`) at a time, ensuring to progress the framebuffer pointer accordingly.

```
void TouchGFXHAL::copyFrameBufferBlockToLCD(const Rect& rect)
{
    __IO uint16_t* ptr;
    uint32_t height;

    // This can be accelerated using regular DMA hardware
    for (height = 0; height < rect.height ; height++)
    {
        ptr = getClientFrameBuffer() + rect.x + (height + rect.y)  * BSP_LCD_GetXSize();
        LCD_IO_WriteMultipleData((uint16_t*)ptr, rect.width);
    }
}
```

Instead of advancing `ptr` manually, the TouchGFX Generator will generate a function `advanceFrameBufferToRect` that advances `ptr` according to the position of `Rect` in the framebuffer.

```
inline uint8_t* TouchGFXGeneratedHAL::advanceFrameBufferToRect(uint8_t* fbPtr, const touch
{
    //       Advance vertically                  Advance horizontally
```

```
    fbPtr += rect.y * lcd().framebufferStride() + rect.x * 2;
    return fbPtr;
}
```

Returning from HAL::flushFrameBuffer()

Once the function returns TouchGFX Engine continues to draw the rest of the frame. If developers wish to use DMA to transfer pixels to the display, they must ensure that `HAL::flushFrameBuffer(Rect& rect)` does not return immediately by e.g. waiting on a semaphore signaled by a *DMA Completed* interrupt.

The following pseudo-code example shows an example of how `HAL::flushFrameBuffer()` could be structured in case DMA is used. The code uses a FreeRTOS semaphore `screen_frame_buffer_sem`.

```
void TouchGFXHAL::flushFrameBuffer(const touchgfx::Rect& rect)
{
    uint16_t* fb = HAL::lockFrameBuffer();

    //Prepare display
    prepare();

    //Try to take a display semaphore - Always free at this point
    xSemaphoreTake(screen_frame_buffer_sem, portMAX_DELAY);

    //Set up DMA
    screenDMAEnable();

    // Wait for the DMA transfer to complete
    xSemaphoreTake(screen_frame_buffer_sem, portMAX_DELAY);

    //Unlock framebuffer and give semaphore back
    HAL::unlockFrameBuffer();
    xSemaphoreGive(screen_frame_buffer_sem);
}
```

TouchGFX Driver / Tearing Effect Signal

As can be seen in TouchGFX Generator configuration above, the "Application Tick Source" is also set to "Custom", which is general for MCUs without embedded TFT Controllers.

As described in the Abstraction Layer Architecture section, the TouchGFX Engine main loop is unblocked by calling `OSWrappers::signalVSync()`, usually at the time when a display signals.

For displays with a serial or 8080 display interface, the embedded display controller typically raises a periodic Tearing Effect (TE) signal that can be connected to a GPIO on the MCU. In this case, the MCU is usually configured to raise an interrupt when the GPIO is signalled. This "Tearing Effect" interrupt will then unblock the TouchGFX Engine Main loop to render the next frame. Remember to configure the GPIO to input and enable the external interrupt for the pin in CubeMX.

```
extern "C"
void TE_Handler(void)
{
    ...
    /* Unblock TouchGFX Engine Main Loop to render next frame */
    OSWrappers::signalVSync();
    ...
}
```

Conclusion

Selecting *Custom* Display Interface through the TouchGFX Generator is an expression of a developer's intent to write code to transfer pixels from an application frame buffer to a display, manually.

The TouchGFX Generator will generate a function `TouchGFXHAL::flushFrameBuffer(Rect& rect)` that is called automatically by TouchGFX after rendering an area of the framebuffer that developers can use to transfer affected pixels to a display, SPI, FMC or otherwise.

Selecting a *custom* display interface also requires developers to implement a custom TouchGFX Application Tick driver that signals `OSWrappers::signalVSync()` to unblock the TouchGFX Engine Main loop. Usually, displays used along with MCUs that have no TFT Controllers can provide a *Tearing Effect* signal that is connected to the MCU.

Version: 4.16

Framebuffer Strategies

This section shows how to configure the TouchGFX Generator to generate a TouchGFX HAL that uses one of the following Frame Buffer strategies:

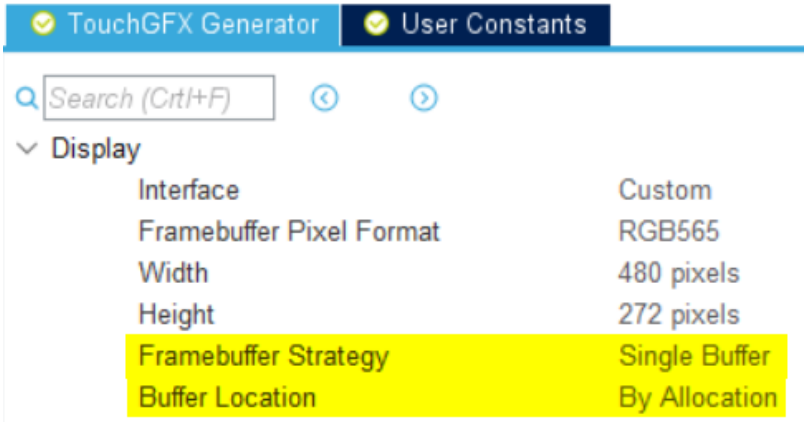
- Single
- Double
- Partial

Single Frame Buffer

Choosing *Single Buffer* as the buffering strategy developers are able to let the compiler allocate memory for the framebuffer in internal RAM but can also choose a specific location for the buffer.

By Allocation

When choosing *By Allocation* TouchGFX Generator will allocate an array based on the dimensions and bitdepth of the application.



Single framebuffer, by allocation

Code is generated to configure the HAL to use this array as the framebuffer.

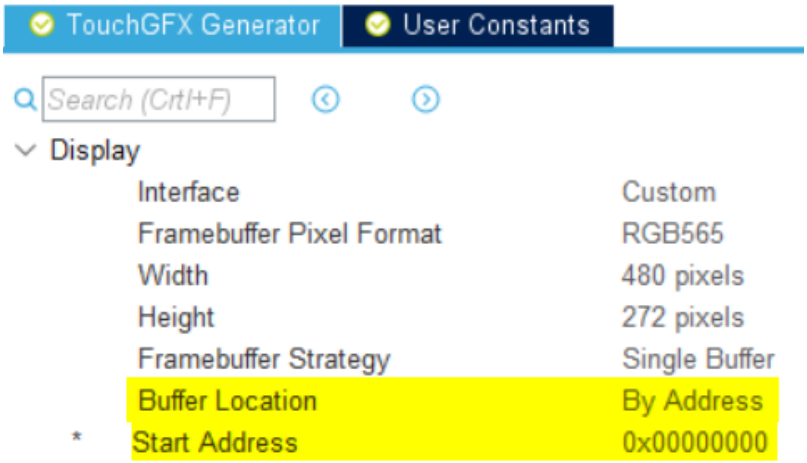
TouchGFXGeneratedHAL.cpp

```
namespace {
    // Use the section "TouchGFX_Framebuffer" in the linker script
    // to specify the placement of the buffer
    LOCATION_PRAGMA("TouchGFX_Framebuffer")
    uint32_t frameBuf[(480 * 272 * 2 + 3) / 4] LOCATION_ATTRIBUTE("TouchGFX_Framebuffer");
}
```

```
void TouchGFXGeneratedHAL::initialize()
{
    HAL::initialize();

    setFrameBufferStartAddresses((void*)frameBuf, (void*)0, (void*)0);
}
```

By Address

When choosing *By Address* for the location of the framebuffer TouchGFX Generator will use the specified Start Addresses during HAL initialization.



Single framebuffer, by address

TouchGFXGeneratedHAL.cpp

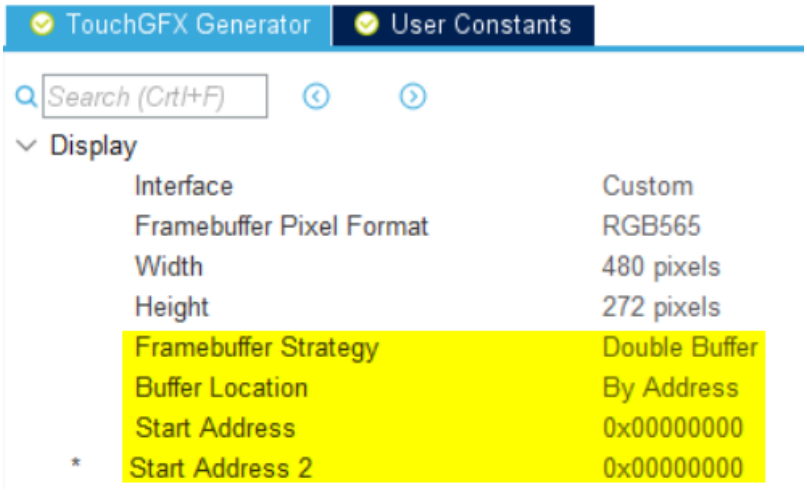
```
void TouchGFXGeneratedHAL::initialize()
{
    HAL::initialize();

    setFrameBufferStartAddresses((void*)0xC0000000, (void*)0, (void*)0);
}
```

Double Frame Buffer

In a double frame buffer configuration, code to swap farmebuffers will be generated in the HAL by TouchGFX Generator depending on the selected Framebuffer strategy and display interface. This memory interface to frame buffer location is used by the TouchGFX Engine during the main event loop.

By Address

When choosing *By Address* TouchGFX Generator will use the two specified Start Addresses during HAL initialization.



Double framebuffer, by address

TouchGFXGeneratedHAL.cpp

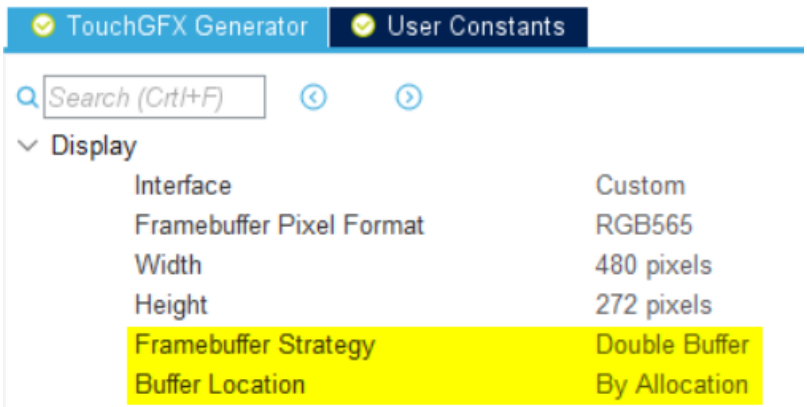
```
void TouchGFXGeneratedHAL::initialize()
{
    HAL::initialize();

    setFrameBufferStartAddresses((void*)0xC0000000, (void*)0xC003FC00, (void*)0);
}
```

TIP

When using Parallel RGB (LTDC) as display interface, the start address will be inherited from the LTDC Layer settings.

By Allocation

When choosing *By Allocation* TouchGFX Generator will allocate an array based on the dimensions and bitdepth of the application, exactly as with a Single Frame Buffer, only twice the size.



Single framebuffer, by allocation

TouchGFXGeneratedHAL.cpp

```
namespace {
    // Use the section "TouchGFX_Framebuffer" in the linker to specify the placement of th
    LOCATION_PRAGMA("TouchGFX_Framebuffer")
    uint32_t frameBuf[(480 * 272 * 2 + 3) / 4 * 2] LOCATION_ATTRIBUTE("TouchGFX_Framebuffe
}

void TouchGFXGeneratedHAL::initialize()
{
    HAL::initialize();

    setFrameBufferStartAddresses((void*)frameBuf, (void*)(frameBuf + sizeof(frameBuf)/(siz
}
```

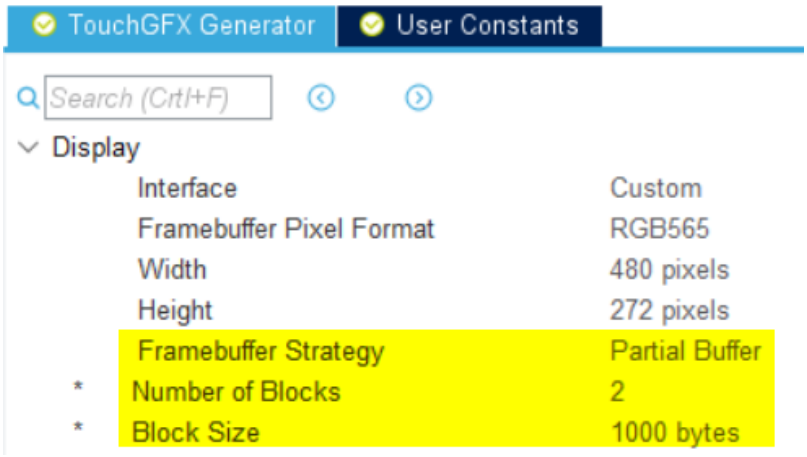
Partial Frame Buffer

Selecting the *Partial Buffer* strategy allows developers to choose a number of blocks and a size for each of these to be used as frame buffers. This strategy uses what TouchGFX calls a *Frame Buffer Allocator* and is different from supplying either a pointer to external memory where the frame buffer is located, or allocating a fixed sized array in internal memory.

See the article on Framebuffer for a general overview of the concept of frame buffers.

TIP

Usually, STM32G0 does not have enough internal RAM to fit framebuffer. "Partial Buffer" would be a perfect match for a low cost solution using this MCU.



Partial framebuffer

Since a partial buffering strategy is typically only used with low cost MCU with no TFT controller and little internal RAM the Partial Buffer Strategy expects the developer to implement the transfer of the contents of the framebuffer to the display. See FMC/SPI Scenario for how to transmit pixels to e.g. a serial display on MCUs with no TFT Controller.

In order to synchronize with TouchGFX when using the Partial Framebuffer strategy developers are required to provide implementations for the following two functions. The code displayed below is generated by CubeMX inside `TouchGFX/target/generated/TouchGFXGeneratedHAL.cpp` and defines the interface from developer to the TouchGFX Engine.

TouchGFXGeneratedHAL.cpp

```
/* ****************************************************
* Functions required by Partial Frame Buffer Strategy
* ****************************************************
*
*  *  uint8_t isTransmittingData() must return whether or not data is currently being tra
*  *  void transmitFrameBufferBlock(uint8_t* pixels, uint16_t x, uint16_t y, uint16_t w, u
*       when the framework wants to send a block. The user must then transfer
*       the data represented by the arguments.
*
*  A user must call touchgfx::startNewTransfer(); once transmitFrameBufferBlock() has succ
*  E.g. if using DMA to transfer the block, this could be called in the "Transfer Complete
*
*/
extern "C" void transmitFrameBufferBlock(uint8_t* pixels, uint16_t x, uint16_t y, uint16_t
extern "C" uint8_t isTransmittingData();
```

The following function is also generated by CubeMX inside the read-only `TouchGFXGeneratedHAL` class inside `TouchGFX/target/generated/TouchGFXGeneratedHAL.cpp`.

> **NOTE**
>
> This flushFrameBuffer() function is generally used for MCUs with no TFT Controller. In the case of Partial Frame Buffers the TouchGFX Generator can generate a definition for this method specifically for that frame buffer strategy.

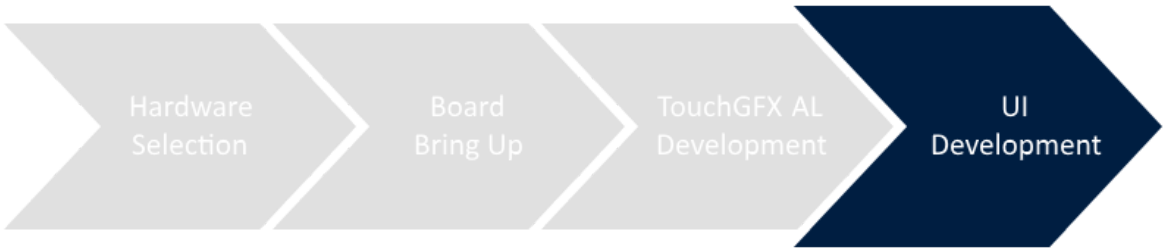
TouchGFXGeneratedHAL.cpp

```
void TouchGFXGeneratedHAL::flushFrameBuffer(const touchgfx::Rect& rect)
{
    HAL::flushFrameBuffer(rect);

    // Once flushFrameBuffer() is called by the framework a block is ready for transfer
    // Mark it ready for transfer and transmit it if user defined method
    // isTransmittingData() does not return false
```

```
    // If data is not being transmitted, transfer the data with user defined method
    // transmitFrameBufferBlock().
    frameBufferAllocator->markBlockReadyForTransfer();
    if (!isTransmittingData())
    {
        touchgfx::Rect r;
        const uint8_t* pixels = frameBufferAllocator->getBlockForTransfer(r);
        transmitFrameBufferBlock((uint8_t*)pixels, r.x, r.y, r.width, r.height);
    }
}
```

Version: 4.16

UI Development Introduction



Developing a functional UI is an integral part to having a successful embedded graphics product and as such, TouchGFX aims to not only provide fast performance but also a smooth development experience.

The UI Development chapter focuses on the ins and outs of how a TouchGFX application is developed:

- **Software Architecture** - describes the overall architecture and design of a TouchGFX application and the relationship between generated code from TouchGFX Designer and user code.
- **Working with TouchGFX** - contains information on the workflow of developing a TouchGFX application and the different tools used in the process - from the PC simulator to the numerous supported IDEs.
- **Designer User Guide** - contains an extensive guide and tips and tricks on how to use the different components of TouchGFX Designer.
- **TouchGFX Engine Features** - contains information on the different TouchGFX Engine features such as bitmap caching, partial framebuffer, multi language support etc.
- **UI Components** - contains information on every UI component found in TouchGFX - from widgets to containers.
- **Scenarios** - contains different scenarios that developers might run into and how to solve them.

Version: 4.16

Model-View-Presenter Design Pattern

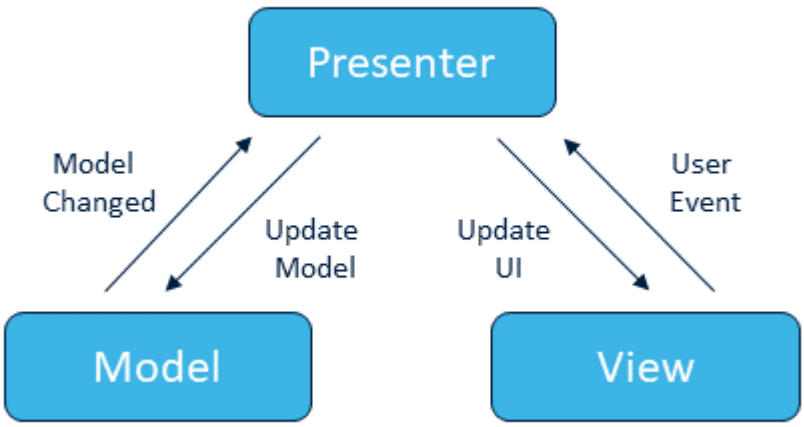
TouchGFX user interfaces follow an architectural pattern known as Model-View-Presenter (MVP) which is a derivation of the Model-View-Controller (MVC) pattern. Both of them are widely used for building user interface applications.

The main benifits of the MVP pattern are:

- **Separation of Concerns**: Dividing your code into separate parts, each having their own responsibility. This makes the code simpler, more reusable and easier to maintain.
- **Unit Testing**: Since the logic (the presenter) of the UI is separated from the visual layer (the view) it is much easier to test these parts in isolation.

In MVP the three classes are defined as follows:

- The *model* is an interface defining the data to be displayed or otherwise acted upon in the user interface.
- The *view* is a passive interface that displays data (from the model) and routes user commands (events) to the presenter to act upon that data.
- The *presenter* acts upon the model and the view. It retrieves data from repositories (the model), and formats it for display in the view.

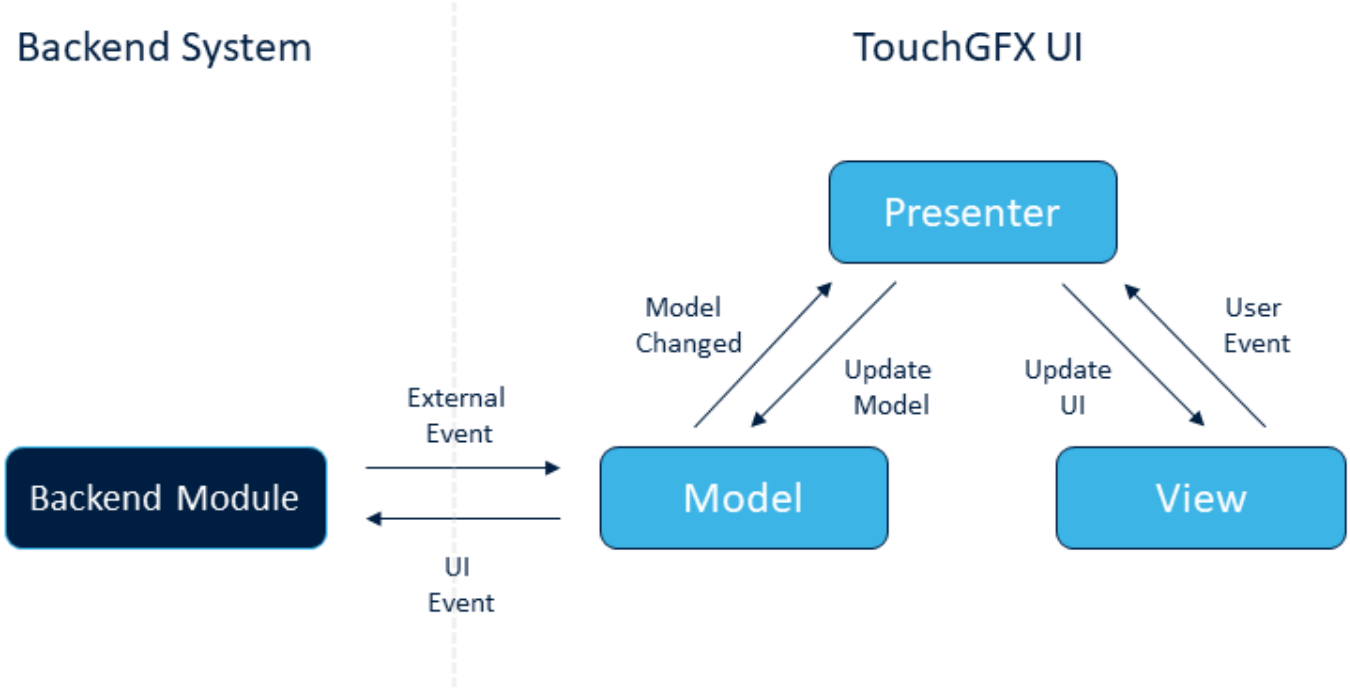


Model-View-Presenter Design Pattern

In TouchGFX the communication with the non-UI part of the application, here called the backend system, is done from the Model class. The backend system is a software component that both receives events from the UI and feeds events into the UI, such as new measurements from sensors. The

backend system can run as a separate task on the same MCU, on a separate processor, a cloud module or something else. From the perspective of TouchGFX, this does not really matter, as long as it is a component that it is able to communicate with.

The specific communication protocol used is not managed by TouchGFX. It simply supplies a function that is called once each tick of TouchGFX, in which the needed communication can be handled. Read more on this subject in Backend Communication.



Model-View-Presenter and external communication

For more concrete details on how MVP is implemented and used in TouchGFX UI development see the Code Structure section.

Version: 4.16

The Screen Concept

In TouchGFX applications, you can have any number of "Screens". A screen in TouchGFX is a logical grouping of UI elements (widgets) and their associated business logic. A screen consists of two classes: a View class containing all the widgets that are shown on this screen, and a Presenter containing business logic for this screen.

You can choose to implement your entire application within the context of a single screen (meaning you only have one View and one Presenter), but we recommend splitting unrelated parts of your UI into different screens, for two reasons:

1. TouchGFX includes a memory allocation scheme that automatically allocates the necessary RAM needed for the most RAM-consuming screen. Only this amount will be allocated, and this RAM block is reused across all screens in your application.
2. Having several screens makes your UI code much easier to maintain.

Defining Screens

There are no exact rules as to how your application should be divided into screens, but there are certain guidelines that might assist you in deciding what screens should make up your specific application. Areas of the UI that are visually and functionally unrelated should be kept in different screens.

If you consider a very simple thermostat application which has a main temperature readout display and a configuration menu, it would be a good idea to create a "Main Screen" for the temperature readout and a "Settings Screen" for showing the configuration menu.

The View for the Main Screen would contain widgets for a background image, a few text areas for showing temperature and a button for switching to the configuration menu. The View for the configuration on the other hand would probably contain widgets for showing a list of configuration options and a different background image. If the configuration menu is capable of editing many different types of settings (dates, names with keyboard, temperatures, units etc.), this screen will grow large in complexity.

In that case it might be beneficial to further divide the configuration menu into one screen showing the overall tree of menu options, and a different screen for editing a specific value. But this is something you will learn as your project progresses.

Currently Active Screen

Because of the way TouchGFX allocates memory for screens (only allocating for biggest View and biggest Presenter), only one View and one Presenter can be active at a time. So if your thermostat application is displaying the temperature readout, then the configuration menu screen is not running anywhere, and in fact is not even allocated.

If events are received from the "backend" (all your non-UI code that does the actual work of the thermostat) or from hardware peripherals, then these events can be delegated to the currently active screen.

This provides a useful separation of concerns because some events will be of interest only to certain screens in your application. For instance, a received event notifying of a change in current temperature could be handled only by the main screen (which would update the text area showing current temperature), whereas the configuration screen could simply discard this event as it is of no interest since current temperature is not being displayed in this screen.

Model-View-Presenter in TouchGFX

TouchGFX follows the Model-View-Presenter (MVP) design pattern as described in Model-View-Presenter Design Pattern. The TouchGFX screen concept ties into the overall Model-View-Presenter architecture by classes that inherit from the View and Presenter classes in TouchGFX. So when adding a new screen to your application in TouchGFX Designer it creates both a new specific View class and a Presenter class to represent that particular screen.

The content and responsibility of the MVP classes in a TouchGFX application are as follows.

Model

The Model class is a singleton class which is always alive and has two purposes:

1. Store state information for the UI. The Views and Presenters are deallocated when switching screen, so they cannot be used to store information which should be kept across screen transitions. Use the Model for this instead.
2. Act as an interface towards the backend system, relaying events to and from the currently active screen.

The Model class is automatically setup to have a pointer to the currently active presenter. When changes occur in the Model the current active Presenter is notified of the change. This is done via methods in the ModelListener interface in the application.

New applications generated by the Designer will automatically have a Model class ready to be used by the UI.

View

The View class (or more specifically, a class that derives from the TouchGFX View class) contains the widgets that are shown in this view as member objects. It also contains a `setupScreen` and a `tearDownScreen` function, which gets automatically called when this screen is entered/exited. Typically you would configure your widgets in the `setupScreen` function.

Your View will also contain a pointer to the associated Presenter. This pointer is set up automatically by the framework. Using this pointer you can communicate UI events like button clicks to the Presenter.

Presenter

Your Presenter class (again, a class that derives from the TouchGFX Presenter class) is responsible for the business logic of the currently active screen. It will receive "backend" events from the Model, and UI events from the View and decide which action to take. For instance, if an alarm event is received from the Model, the Presenter might decide to tell the View that an alarm popup dialog should be displayed.

Version: 4.16

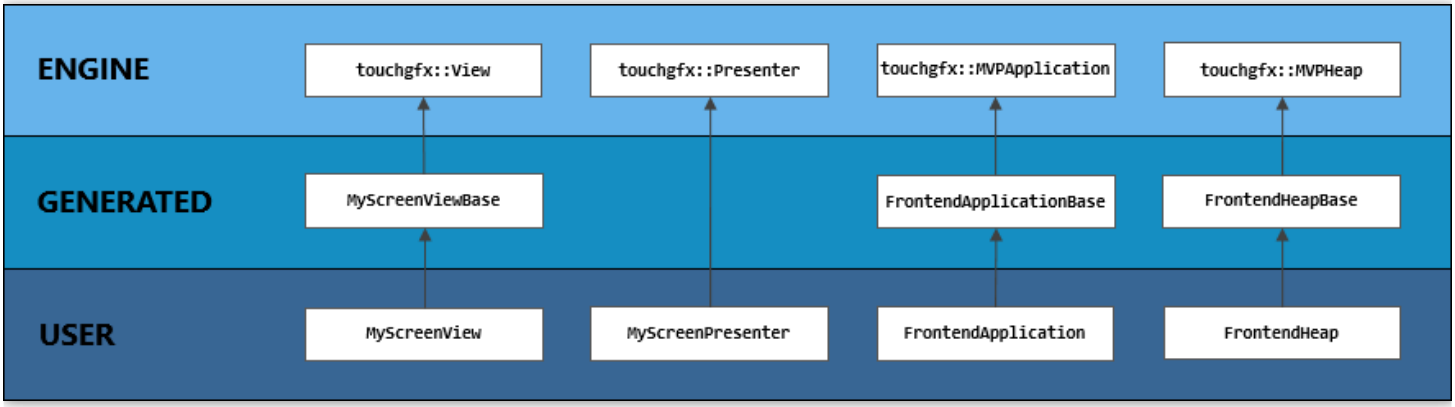
Code Structure

This section explains the structure of a TouchGFX project - from the code generated by TouchGFX Designer to the extending code written by the user.

Generated Code vs. User Code

The code generated by TouchGFX Designer will be completely separate from the code written by the user. In fact, the generated code is placed in the folder `generated/gui_generated`, whereas the handwritten code is placed in the `gui` folder.

The `gui_generated` code serves as base classes for user code classes. The base classes contain all the setup code configured in TouchGFX Designer. The following class diagram shows the relationship of the classes:



Class hierarchy of engine, generated and user classes

As shown above, TouchGFX Designer applications consist of 3 different layers of code:

- **Engine**: these classes are the standard classes provided by TouchGFX. These act as base classes for the generated classes.
- **Generated**: these classes and corresponding files will be regenerated whenever TouchGFX Designer generates code. Therefore, these classes and files should not be edited manually, as any manual changes will be overwritten on the next run of the code generator. These classes are base classes for the user classes.
- **User**: these classes are intended for handwritten code. The user is free to put any code in this layer. The user classes will be generated if not present, but will never be altered by TouchGFX Designer. They *belong to the user*.

NOTE

The architecture of the applications generated by TouchGFX Designer is open in the sense that there should be no limits to what you can create. If something (e.g. a widget, animation, or effect) is not supported by TouchGFX Designer, you can add them in user classes. The code generated by TouchGFX Designer is by design not allowed to restrict you in your way of doing TouchGFX applications.

Files Generated by TouchGFX Designer

As a rule, TouchGFX Designer will *only* regenerate files within the `generated` folder in a TouchGFX project and it is therefore important that you do not manually edit these, as they will be overwritten. However, TouchGFX Designer can also generate missing files needed for compilation, e.g. `application.config`, `simulator/main.cpp` and skin images located in `assets/images/__designer`. In actuality, TouchGFX Designer only needs the following to be able to generate, compile and run a project:

- The `.touchgfx` file describing the project
- User code (if any) located in the `gui` folder
- User images (if any) located in the `assets/images` folder
- Texts (if any) located in the `assets/texts/texts.xlsx` file
- Fonts (if any) located in the `assets/fonts` folder

TIP

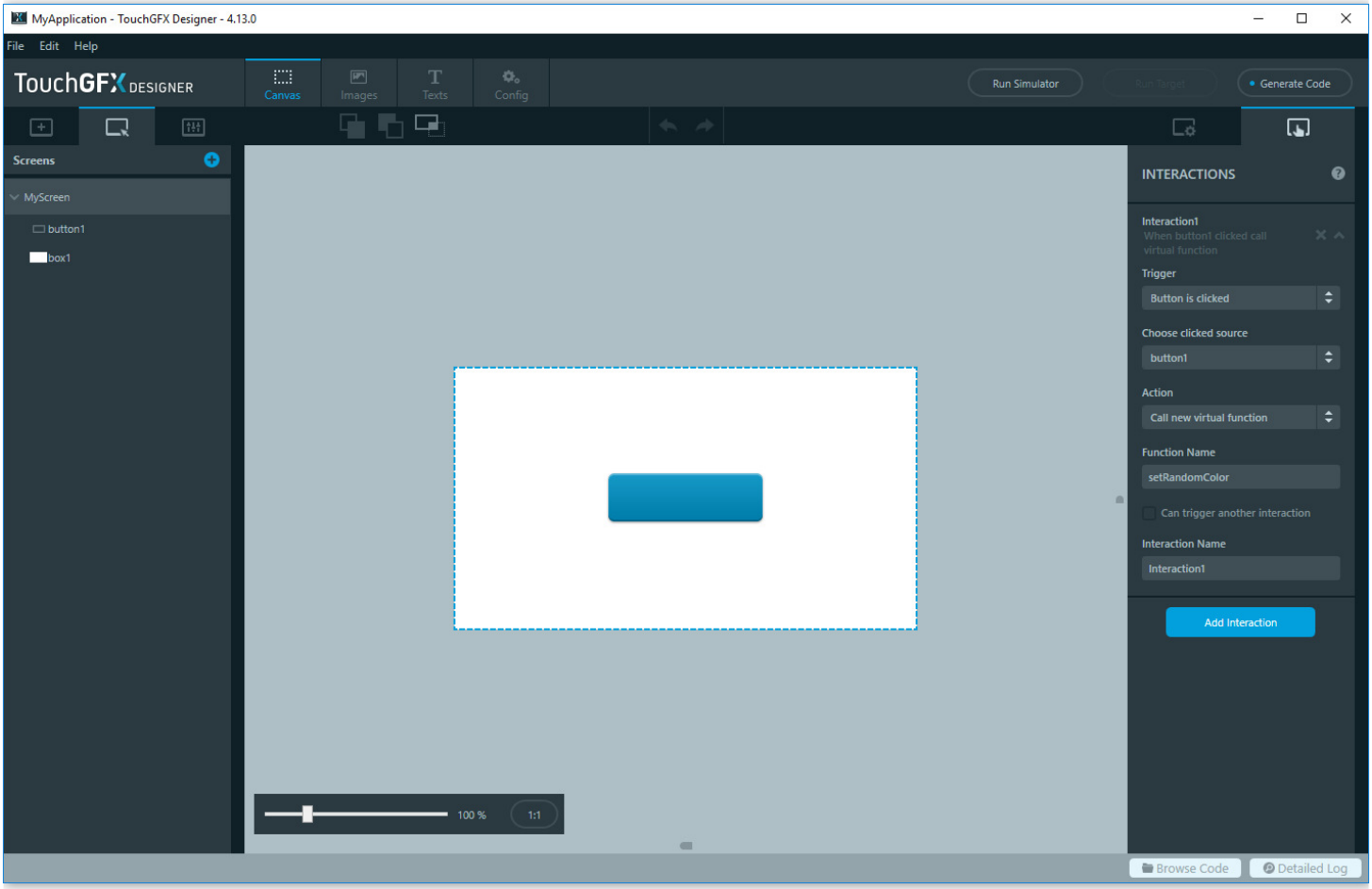
When using version control such as Git, this means that only the files listed above actually need to be commited to a repository. The rest can be generated by TouchGFX Designer itself.

Example

A simple example of an application having both generated and user code will be given below. This should illustrate the aspects above in more detail.

TouchGFX Designer View

The following example has just one screen. The screen `MyScreen` consists of a Box `box1` and a Button `button1`. We have set up an interaction to call the virtual function `setRandomColor()` when `button1` is clicked.

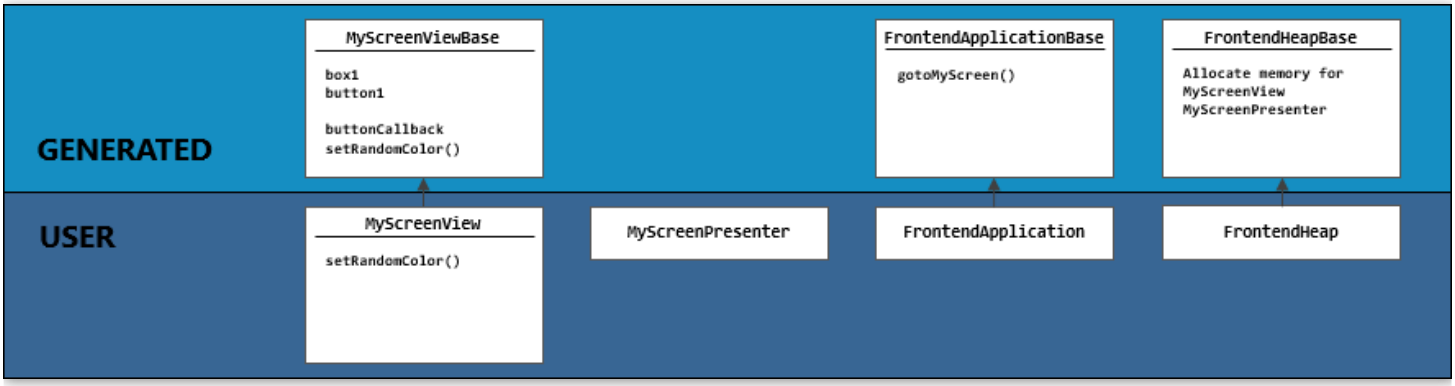


TouchGFX Designer application

Whenever we press the button we would like to change the color of the background box to a new random color. To demonstrate how to write your own custom code, we will implement this behavior in user code.

Layers

The different classes involved in this example can be seen below:



Example classes

We see that:

- `MyScreenViewBase`, `FrontendApplicationBase` and `FrontendHeapBase` are in the generated space, implying that:

    - They will be regenerated whenever a change is made in TouchGFX Designer
    - The user should not manually edit these classes
- `MyScreenView`, `MyScreenPresenter`, `FrontendApplication` and `FrontendHeap` are created in the user code space, meaning that:
    - These will not be regenerated when changes are made in TouchGFX Designer
    - The user is free to add custom code here
- All the setup of `box1` and `button1` is done in the generated view base class `MyScreenViewBase`.
- All the functions for transitioning between screens are in the generated application base class `FrontendApplicationBase`.
- All the book keeping, making sure that the right amount of memory is allocated, is located in the generated heap base class `FrontendHeapBase`.

The user is free to edit the user code classes. For instance you could add more widgets. For now we will just implement the `setRandomColor` function to actually change the color of `box1`.

Code

Looking at the view base code, we see the setup of the box and button generated by TouchGFX Designer. We also see the setup of and the call to the virtual function `setRandomColor` in the `buttonCallbackHandler`, but at the moment this function does not do anything:

MyApplication/generated/gui_generated/src/my_screen/MyScreenViewBase.cpp

```
/*********************************************************************************/
/********** THIS FILE IS GENERATED BY TOUCHGFX DESIGNER, DO NOT MODIFY ***********/
/*********************************************************************************/
#include <gui_generated/myscreen_screen/MyScreenViewBase.hpp>
#include <touchgfx/Color.hpp>
#include "BitmapDatabase.hpp"

MyScreenViewBase::MyScreenViewBase() :
    buttonCallback(this, &MyScreenViewBase::buttonCallbackHandler)
{
    box1.setPosition(0, 0, 800, 480);
    box1.setColor(touchgfx::Color::getColorFrom24BitRGB(255, 255, 255));

    button1.setXY(155, 106);
    button1.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID), touchgfx
    button1.setAction(buttonCallback);

    add(box1);
    add(button1);
}

void MyScreenViewBase::setupScreen()
{
```

```
}

void MyScreenViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
{
    if (&src == &button1)
    {
        //Interaction1
        //When button1 clicked call virtual function
        //Call setRandomColor
        setRandomColor();
    }
}
```

Looking at the header file for the view base class, we see the declaration of `setRandomColor` and an instruction to override and implement the function in user code:

MyApplication/generated/gui_generated/src/my_screen/MyScreenViewBase.hpp

```
/*********************************************************************************/
/********** THIS FILE IS GENERATED BY TOUCHGFX DESIGNER, DO NOT MODIFY ***********/
/*********************************************************************************/
#ifndef MYSCREENVIEWBASE_HPP
#define MYSCREENVIEWBASE_HPP

#include <gui/common/FrontendApplication.hpp>
#include <mvp/View.hpp>
#include <gui/myscreen_screen/MyScreenPresenter.hpp>
#include <touchgfx/widgets/Box.hpp>
#include <touchgfx/widgets/Button.hpp>

class MyScreenViewBase : public touchgfx::View<MyScreenPresenter>
{
public:
    MyScreenViewBase();
    virtual ~MyScreenViewBase() {}
    virtual void setupScreen();

    /*
     * Virtual Action Handlers
     */
    virtual void setRandomColor()
    {
        // Override and implement this function in MyScreen
    }

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }
```

```
    /*
     * Member Declarations
     */
    touchgfx::Box box1;
    touchgfx::Button button1;

private:

    /*
     * Callback Declarations
     */
    touchgfx::Callback<MyScreenViewBase, const touchgfx::AbstractButton&> buttonCallback;

    /*
     * Callback Handler Declarations
     */
    void buttonCallbackHandler(const touchgfx::AbstractButton& src);

};

#endif // MYSCREENVIEWBASE_HPP
```

So let's do just that. Navigate to the user code header file `MyScreenView.hpp` and override the function:

MyApplication/generated/gui_generated/src/my_screen/MyScreenView.hpp

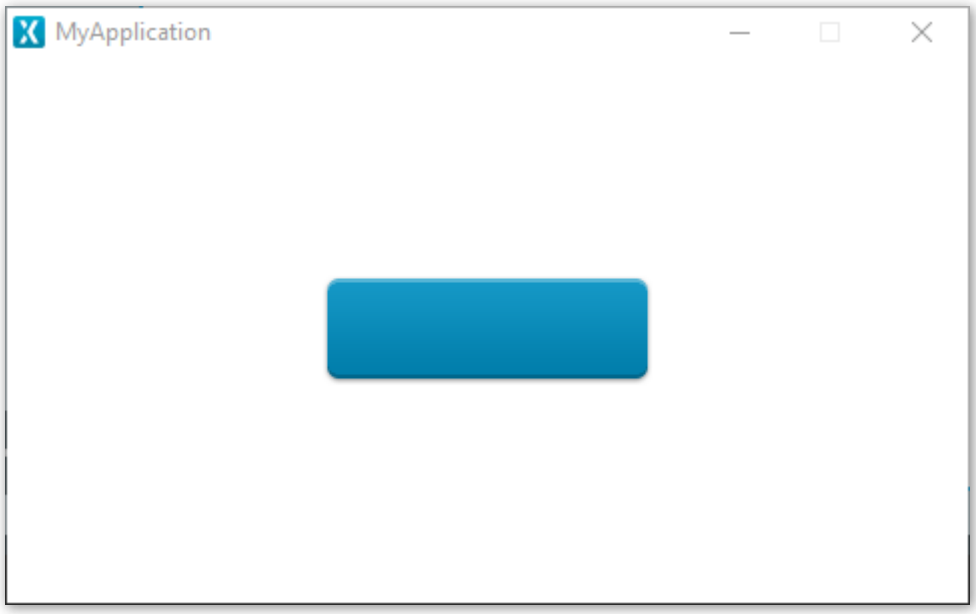
```
#ifndef MYSCREENVIEW_HPP
#define MYSCREENVIEW_HPP

#include <gui_generated/myscreen_screen/MyScreenViewBase.hpp>
#include <gui/myscreen_screen/MyScreenPresenter.hpp>

class MyScreenView : public MyScreenViewBase
{
public:
    MyScreenView();
    virtual ~MyScreenView() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void setRandomColor();
protected:
};

#endif // MYSCREENVIEW_HPP
```

Then we need to implement the actual behavior of `setRandomColor` in the cpp file for `MyScreenView`:

MyApplication/gui/src/my_screen/MyScreenView.cpp

```
#include <gui/myscreen_screen/MyScreenView.hpp>
#include <touchgfx/Color.hpp>

MyScreenView::MyScreenView()
{

}

void MyScreenView::setupScreen()
{
    MyScreenViewBase::setupScreen();
}

void MyScreenView::tearDownScreen()
{
    MyScreenViewBase::tearDownScreen();
}

void MyScreenView::setRandomColor()
{
    box1.setColor(touchgfx::Color::getColorFrom24BitRGB(rand()&0xff, rand()&0xff, rand()&0
    box1.invalidate();
}
```

Running the simulator now shows that we have succeeded in creating a simple application utilizing both generated and user code - the background box now changes to a random color everytime you click the button.



Simulator showcasing the application

TIP

Of course you do not need to use any features of the code generated by TouchGFX Designer but it will potentially save a lot of time. It is possible to do TouchGFX applications the old-school way by just adding the necessary files by hand.

It is also possible to mix and match. For example, you can have screens that are not defined within the TouchGFX Designer project. You can add the `gotoXYZ` functions to the `FrontendApplication` class and add the views, presenters and transitions you need for your screen to the `FrontendHeap`.

Version: 4.16

Using IDEs with TouchGFX

When creating a new TouchGFX project, either through the TouchGFX Designer or CubeMX, the following project files and libraries for using proprietary IDEs are available:

- Keil uVision *(Target only)*
- IAR Embedded Workbench *(Target only)*
- CubeIDE *(Target only)*
- Microsoft Visual Studio *(Simulator only)*

Note that not all project files are present in your project at the same time. The tool chain selected in CubeMX is the one that will be generated, by default CubeIDE is selected.

FURTHER READING

How to change the tool chain is described **here**.

In addition makefiles and libraries for shell-based compilation with a GCC cross compiler for ARM targets are also provided. This article will help you configure third-party GCC-based IDEs for TouchGFX application development. Basically any IDE which is able to invoke the GCC cross compiler should be useable.

NOTE

Please note that this article describes only the setup procedure in general terms - support for all the various IDEs cannot be provided, but hopefully the information presented here is sufficient for you to use TouchGFX with your favorite IDE.

This article will describe two different approaches to getting TouchGFX to work with other IDEs. One approach is to invoke the TouchGFX Makefile from within the IDE. This is probably the easiest approach, but is not always desirable if you want to have more control over the compilation process and file locations. Alternatively you can manually add the necessary TouchGFX files and configuration options to your existing project.

Prerequisite: GCC version

This article assumes that you will use either the GCC cross compiler toolchain distributed with the TouchGFX environment shell, or alternatively your own GCC toolchain of a flavor that is able to link with the TouchGFX core library compiled using the environment shell toolchain.

The GCC compiler used:

```
$ arm-none-eabi-gcc.exe -v
Target: arm-none-eabi
Thread model: single
gcc version 6.3.1 20170620 (release) [ARM/embedded-6-branch revision 249437] (GNU Tools fo
```

The compiler can be obtained from https://launchpad.net/gcc-arm-embedded.

Invoke TouchGFX Makefile from IDE

A hopefully quick-and-dirty way of compiling TouchGFX applications from within your IDE is to simply invoke the Makefile included in the projects created by the TouchGFX Designer. To use the TouchGFX environment shell to compile an application for target, you must navigate to the TouchGFX application root folder and execute the following command:

```
$ make -f target/gcc/Makefile
```

Now, instead of invoking the make command from the TouchGFX environment shell, it is also possible to invoke it from within your IDE. The executables used by the shell (make, arm-none-eabi-gcc, ..) are actually normal Windows x86 executables, so the make application can be executed by a normal command prompt, provided that the following environment variables have been configured.

```
C:\<touchfx_installation_directory>\touchgfx\env\MinGW\bin
C:\<touchfx_installation_directory>\touchgfx\env\MinGW\msys\1.0\Ruby193\bin
C:\<touchfx_installation_directory>\touchgfx\env\MinGW\msys\1.0\bin
C:\<touchfx_installation_directory>\touchgfx\env\MinGW\msys\1.0\gnu-arm-gcc\bin
```

After setting up the needed Windows environment variables it is now possible to invoke the `make` command on the appropriate TouchGFX makefile directly from within your IDE. The exact command you need to execute is the same as when compiling from the shell, namely:

```
$ make -f target/gcc/Makefile
```

NOTE

Please note that your current directory must be the root directory of the application you want to compile.

Add TouchGFX code files to your own project

If you instead wish to have more control over the build process and file locations, you can instead integrate the relevant TouchGFX code files into your own existing project, and add the necessary include paths and compiler switches in order to make it compile.

Required files

Basically you will need to add the same TouchGFX files to your IDE project as are compiled when building with make from the TouchGFX environment shell. Exactly which files to include depend on your target board, since the low-level drivers are different for each board. In order to determine what files you need, the recommended approach is to simply try compiling the application using the TouchGFX environment shell for the appropriate board. The compilation process will mention each file being compiled, thereby giving you a list of exactly the files you need to add.

Include paths

You will need to add the following include paths to your compilation (here mentioned relative to the directory where you have unpacked TouchGFX):

```
<touchgfx_application_root_directory>/gui/include
<touchgfx_application_root_directory>/generated/gui_generated/include
<touchgfx_application_root_directory>/platform/os
<touchgfx_application_root_directory>/generated/fonts/include
<touchgfx_application_root_directory>/generated/images/include
<touchgfx_application_root_directory>/generated/texts/include
<touchgfx_application_root_directory>/touchgfx/framework/include
```

TIP

In addition to the above include paths, you also need to add include paths for the board specific code. Take a look in the `target/gcc/Makefile` for this information.

Compiler switches

Like with include paths, there are some generic compiler switches which must be enabled, and also some that are specific for the processor core and concrete board. The compiler switches used to compile the TouchGFX core library are listed below, for each core. Some of these options will be mandatory for the compilation of your code as well in order for the linker to work, and some are optional. Those that are known to be mandatory are marked in bold.

Cortex-M3 cores

-mcpu=cortex-m3 -march=armv7-m -Wno-psabi -DCORE_M3 -D__irq="" -fno-rtti -fno-exceptions -fno-strict-aliasing -fdata-sections -ffunction-sections

Cortex-M4f cores

-fno-rtti -fno-exceptions -mfpu=fpv4-sp-d16 -mfloat-abi=softfp -mcpu=cortex-m4 -D__irq=”“ -mthumb -mno-thumb-interwork -std=c99 -Os -fno-strict-aliasing -fdata-sections -ffunction-sections -Wno-psabi -DCORE_M4 -march=armv7e-m

Cortex-M7 cores

-fno-rtti -fno-exceptions -mfpu=fpv5-sp-d16 -mfloat-abi=softfp -mcpu=cortex-m7 -D__irq=”“ -mthumb -mno-thumb-interwork -std=c99 -Os -fno-strict-aliasing -fdata-sections -ffunction-sections -Wno-psabi -DCORE_M7

Linker

Core library

You must link with the TouchGFX core library. Depending on your MCU, this would be either

```
touchgfx/lib/core/cortex-m4f/gcc/libtouchgfx.a
touchgfx/lib/core/cortex-m7/gcc/libtouchgfx.a
```

Linker options

In addition, you will need a few linker options. The following options are what TouchGFX uses:

```
Cortex-M4f: -Wl,-static -nostartfiles -mthumb -mno-thumb-interwork -fno-exceptions -fno-rt
Cortex-M7: -Wl,-static -nostartfiles -mthumb -mno-thumb-interwork -fno-exceptions -fno-rtt
```

Asset generation

To compile a project, assets must be generated as well. This can be done either by invoking the generated Makefile with the option 'assets':

```
make -f <path_to_Makefile> assets
```

Another way to generate assets, is to use the imageconverter and text/font-converter directly.

Imageconverter The imageconverter can be found in your projects touchgfx folder `touchgfx/framework/tools/imageconvert/build` built for Linux and Windows.

```
usage: imageconvert [-c configfile] [-f inputfile -o outputfile | -r inputdir -w outputdi
```

When calling the imageconvert.out executable, it will look for a configfile (`application.config`) file in the folder it is called from.

Textconverter The textconverter can be found in your projects touchgfx folder `touchgfx\framework\tools\textconvert` as a ruby file: `main.rb` .

```
usage: main.rb file.xlsx path/to/fontconvert.out path/to/fonts_output_dir path/to/localiza
```

Flashing and debugging

Depending on your linker settings, you will most likely get an `.elf` or `.hex` file produced as output. It is possible to deploy and debug TouchGFX applications from within most IDEs, typically using a GDB server, or whichever other approach your IDE provides. Concrete pointers for each available IDE cannot be provided, but you might find inspiration in the Compiling & Flashing article, which explains how to flash a board with a GCC-produced ELF/HEX file.

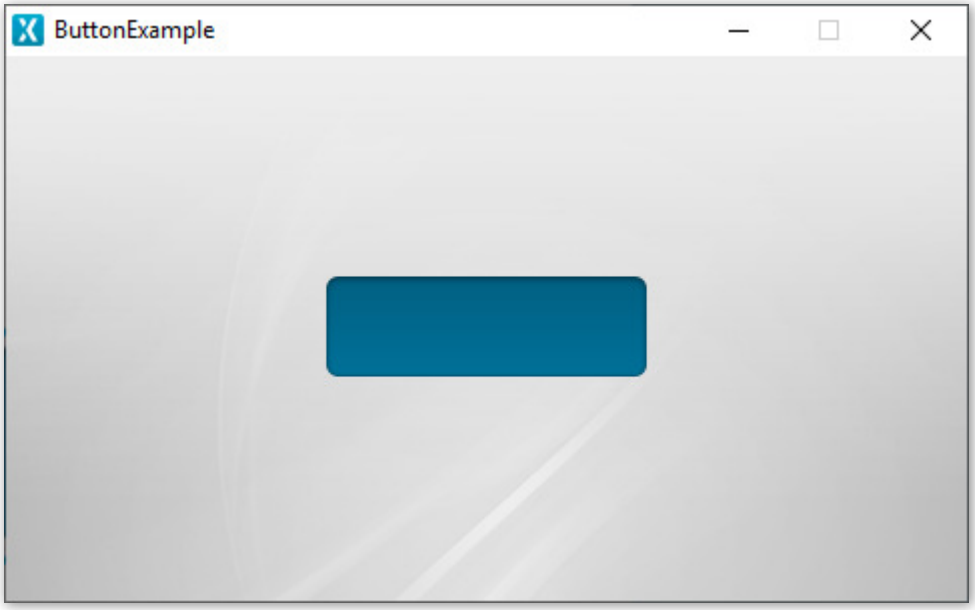
Version: 4.16

Widgets and Containers

This section of the documentation will go over two of the most fundamental concepts of building a TouchGFX application: widgets and containers. These are two of the building blocks you will be using throughout the development of your UI. Both include premade components supplied with TouchGFX, while also being open-ended enough to support the creation of custom implementations.

Widgets

TouchGFX and the TouchGFX Designer tool supplies numerous standard widgets which users can freely use to build their UI, such as TextArea, Button and TextureMapper. But on a fundamental level, a widget in TouchGFX is simply an abstract definition of something that can be drawn on the screen and can be interacted with.



A Button widget with an Image widget as background

Using TouchGFX Designer, users can add any widgets they want to their screens and customize them how they want with the supplied properties specific to each widget. Widgets can also be grouped by using the different types of containers that TouchGFX also supplies.

You can also add widgets in user code if you want by using the `add(widget_instance_name);` function or adding it to a container by using the containers add function e.g. `myContainer.add(widget_instance_name);`. The order in which you add the widgets will determine the z-order. The widget added last will appear front-most on the screen.

The coordinates of a widget are always relative to its parent node which is either the root container (the screen) or a container.

FURTHER READING

You can create your own widgets to meet any specific need you might have. Read more on this in the **Custom Widgets section**.

Containers

A container is a component in TouchGFX that can contain child nodes, such as widgets and other containers.

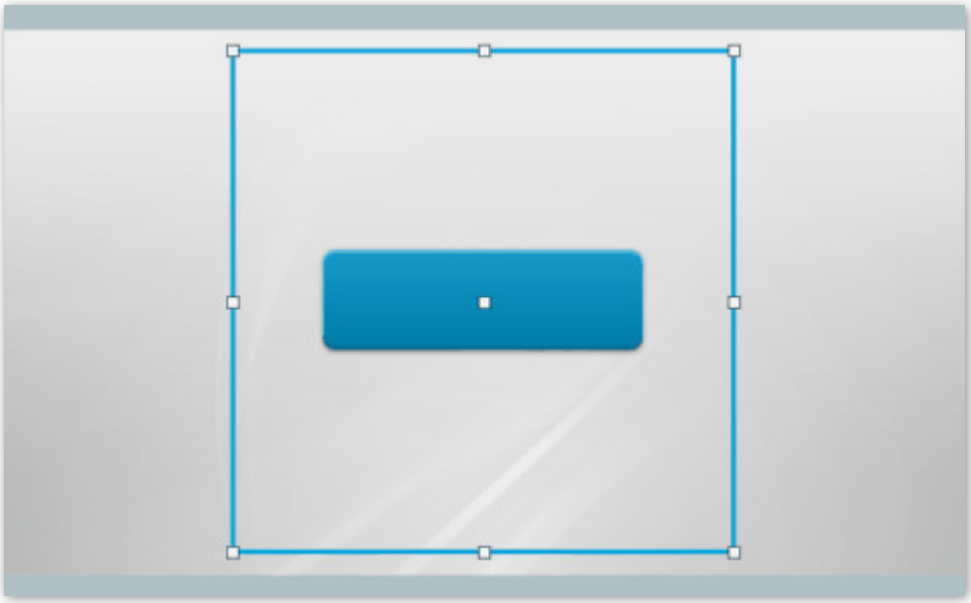
In TouchGFX Designer, containers are found under the Containers category in the Widgets tab and adding widgets to a containers is done by dragging widgets into the container in the tree view.

The z-order of children is determined by the order in which children are added to the container - the child added last will appear front-most on the screen.

Since the position of widgets in TouchGFX is defined relative to their parents, changing the position of a parent container will also move the children accordingly.

Containers act as viewports, meaning that only the parts of the children that intersect with the geometry of the container will be visible.

In the animation below, you can see how the viewport aspect of containers work. First we see the outline of the container of which the button is a child:



A Container acting as a viewport

FURTHER READING

You can create your own container to meet any specific need you might have. Read more on this in the **Custom Containers section**.

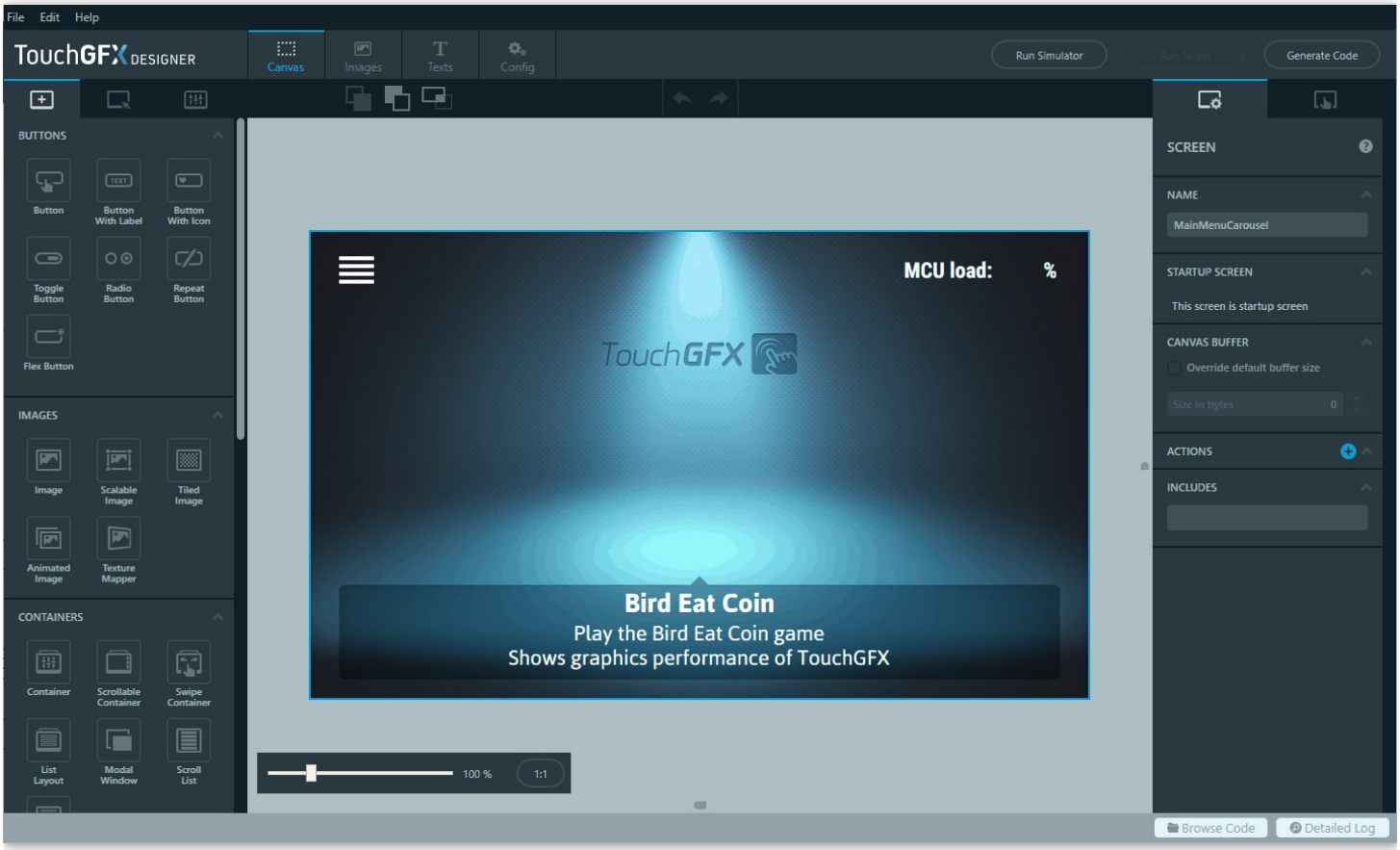
Version: 4.16

Simulator

Building a TouchGFX UI often involves a lot of tweeking of the graphics details to match the specification of the UI.

To speed up the development process it is important to have a fast turnaround time when trying out and debugging your application. Flashing a board can often take quite some time so doing this each time you have made a small change to your application will really slow down the development. To alleviate this, the TouchGFX PC simulator is a great addition to your development tools.

You simply compile your application for your PC and run the application there. The code executed is exactly the same as on target hardware except for the Board Bring Up code and Abstraction Layer which are made for the PC instead of your board. This means that you can test things like placement of widgets, interactions, animations, state machines and so on just as precise as on target hardware. You can even debug your code using IDEs like Visual Studio if you like. Of course things like performance analysis and interactions with real backend systems must be done on your board.

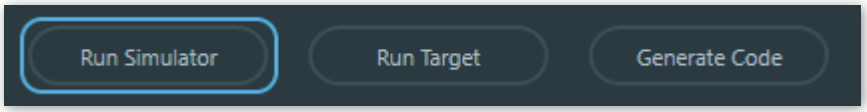


Simulator example

How To Run

Using TouchGFX Designer

To launch the simulator from within TouchGFX Designer, simply press the "Run Simulator" button in the top right corner or press `F5` on your keyboard.



Launching the simulator from TouchGFX Designer

Using TouchGFX Environment

To launch the simulator using the TouchGFX environment, follow these steps:

1. Open the TouchGFX Environment
2. Navigate to the location of your TouchGFX application
3. Run the command `make -f simulator/gcc/Makefile -j6` to compile the simulator
4. Run the command `./build/bin/simulator.exe` to launch the simulator

Run steps 3 and 4 whenever you have made a change to your TouchGFX application.

Simulator Features

Apart from capturing mouse input, the TouchGFX simulator also includes other useful features, listed below:

Shortcut	Feature
`F1`	Enables/disables debug info.
`F2`	Enables/disables highlighting invalidated area.
`F3`	Takes a screenshot and places the image under the `screenshots` folder.
`CTRL + F3`	Takes screenshots of the next 50 frames and places the images under the `screenshots` folder.
`SHIFT + F3`	Takes a screenshot and places it in your clipboard.
`F4`	If a simulator skin is used - enables/disables the simulator skin. If a simulator skin is *not* used - enables/disables window border.

Shortcut	Feature
`F9`	Pauses the simulator by preventing ticks to be sent to the application. Pressing F9 again will resume normal execution.
`F10`	While the simulator is paused (after pressing F9) it is possible to send a single tick to the application by pressing F10 thereby "single step" the application.
`ESC`	Close the simulator.

Simulator Only User Code

If you have some code that should only run when using TouchGFX simulator, you can use `#IFDEF Simulator` in your C++ code:

```
#IFDEF Simulator
  // Your simulator specific user code here
#ENDIF
```

If you want to output a debug text to the console you can use the `touchgfx_printf` function. This is a printf like function that will only be included when building simulator code and thus will not interfere when running on target hardware. Therefore there is no need to use `#IFDEF Simulator` in this case.

```
int i = 0;
touchgfx_printf("Application is running through simulator! \n");
touchgfx_printf("Print our value for integer i = %i \n", i);
```

Version: 4.16

Compiling & Flashing

This section describes how to go from TouchGFX application code to executing program, that is how to compile and flash in a specific setup.

Compiling TouchGFX Applications

When compiling a TouchGFX application, there are two options; compiling for the PC simulator or compiling for the target hardware.

Compiling for PC Simulator

There are two options for compiling projects for the PC Simulator; GCC and Visual Studio.
Both of these options are always available, since they are generated by TouchGFX Designer.

GCC

The makefile is located at `<touchgfx_application_root_folder>/simulator/gcc/Makefile`

TouchGFX includes a MinGW environment, that comes preinstalled with a GCC compiler and GNU Make, making it easy to execute the generated Makefile for the PC simulator.

The TouchGFX Environment can be launched either from `<touchgfx_installation_directory>/env/MinGW/msys/1.0/msys.bat` or from the shortcut added to the Windows start menu, named "TouchGFX x.y.z Environment" where x, y and z describe the version number.

After launching the TouchGFX Environment and navigating to the TouchGFX Application root folder, the simple command below can be executed to produce a simulator.exe file.

```
make -f simulator/gcc/Makefile
```

The simulator executable can then be launched from the TouchGFX Environment with the following command.

```
./build/bin/simulator.exe
```

The PC Simulator can also be compiled and launched from TouchGFX Designer, by using the Run Simulator command.

Visual Studio

To compile the PC Simulator using Visual Studio, simply open the generated solution file located at `<touchgfx_application_root_folder>/simulator/msvs/Application.sln` using Visual Studio.

The PC Simulator can be launched directly from Visual Studio, enabling code debugging.

> **NOTE**
>
> **Before being able to compile with GCC or Visual Studio, Run the Generate command from TouchGFX Designer.**

Compiling for Target Hardware

Compiling projects for STM32 Evaluation Kits is quite simple for Application Template based applications.

Each application template contains project files for GCC, CubeIDE, IAR and Keil:

- GCC: `<project_root_folder>/gcc/MakeFile`
- CubeIDE: `<project_root_folder>/STM32CubeIDE/.cproject`
- IAR: `<project_root_folder>/EWARM/Project.eww`
- Keil: `<project_root_folder>/MDK-ARM/<STM32_evaluation_kit_name>.uvprojx`

The active tool chain is set from CubeMX and is set to CubeIDE by default. *Please note that all project files are not present at the same time. The generated project file depends on the selected tool chain in CubeMX*

TouchGFX includes a MinGW environment, that comes preinstalled with the GNU Embedded Toolchain for Arm and GNU Make, making it easy to execute the included Makefile for the target hardware.

The TouchGFX Environment can be launched either from `<touchgfx_installation_directory>/env/MinGW/msys/1.0/msys.bat` or from the shortcut added to the Windows start menu "TouchGFX x.y.z Environment"

After launching the TouchGFX Environment and navigating to the project root folder, the simple command below can be executed to compile the project for the target hardware.

```
make -f gcc/Makefile
```

NOTE

Before being able to compile with GCC, CubeIDE, IAR or Keil, run the Generate command from TouchGFX Designer.

Flashing STM32 Evaluation Kits

Flashing projects to STM32 Evaluation Kits is quite simple with projects based on a premade Application Template.

Each project, when built, produces a binary that can be flashed by either ST Link Utility or Cube Programmer

Therefore these tools must be installed to proceed with flashing.

It is suggested to install these tools to their default location.

- ST Link Utility default install location:
  `C:/Program Files (x86)/STMicroelectronics/STM32 ST-LINK Utility/ST-LINK Utility`
- Cube Programmer default install location:
  `C:/Program Files/STMicroelectronics/STM32Cube/STM32CubeProgrammer`

NOTE

The Application Templates do not provide any flash loaders for flashing directly from within IAR, Keil, CubeIDE or other IDEs.

GCC & TouchGFX Designer

The Makefile included with an Application Template located at `<project_root_folder>/gcc/MakeFile` has a built-in flash command, as shown below, that uses either ST Link Utility or Cube Programmer (depending on the AT) to flash the STM32 Evaluation Kit. You can of course also use the desktop version of the flash tools to flash the boards with the generated .hex files.

```
make -f gcc/Makefile flash
```

The .hex file is located at `<project_root_folder>/TouchGFX/build/bin/target.hex`

It is also possible to only write to the internal flash and thus skipping the external part. This can reduce the flash time considerably if you have a large set of images. However, you need to be sure that the content for the external flash has not changed since you wrote the external flash last time. If it has, and you do not reflash it, you will see strange behaviour. In this case reflash both the internal and external flash.

```
make -f gcc/Makefile intflash
```

The .hex file is located at `<project_root_folder>/TouchGFX/build/bin/inttarget.hex`

The Application Template also provides the configuration for TouchGFX Designer to be able to flash projects via the Run Target Command. The command used by TouchGFX Designer to flash can be seen and overridden in the Build Section of the Config View in TouchGFX Designer.

CubeIDE

Application Templates provide support for flashing project compiled with CubeIDE, by using the .elf file output by CubeIDE, with the Cube Programmer.

The .elf file is located at
`<project_root_folder>/STM32CubeIDE/Debug/<STM32_evaluation_kit_name>.elf`

e.g. C:/TouchGFXProjects/MyApplication/STM32CubeIDE/Debug/STM32F746G_DISCO.elf

IAR

The Application Templates provide support for flashing project compiled with IAR, by using the .hex file output by IAR, with the Cube Programmer.

The .hex file is located at
`<project_root_folder>/EWARM/<STM32_evaluation_kit_name>/Exe/<STM32_evaluation_kit_name>.hex`

e.g. C:/TouchGFXProjects/MyApplication/MDK-ARM/STM32F469I-DISCO/STM32F469I-DISCO.hex

Keil

The Application Templates provide support for flashing project compiled with Keil, by using the .hex file output by Keil, with the Cube Programmer.

The .hex file is located at `<project_root_folder>/MDK-ARM/<STM32_evaluation_kit_name>/<STM32_evaluation_kit_name>.hex`

e.g. C:/TouchGFXProjects/MyApplication/MDK-ARM/STM32F469I-DISCO/STM32F469I-DISCO.hex

Flashing Custom Hardware

If instead what needs to be flashed is custom hardware, and not a predefined hardware setup like an STM32 Evaluation Kit, you can still use STM32CubeProgrammer. STM32CubeProgrammer does not necessarily come with a flash loading mechanism for your specific external memory. It is however possible to create a custom flash loader. Read the user manual on developing customized loaders for your external memory to find more info.

Debugging

As a TouchGFX application is a set of C++ files generated by TouchGFX Designer, TouchGFX Generator and written by the developer, it can be debugged as any other C++ application.

Target Debugging

If you are using an IDE like IAR Workbench, Keil uVision or CubeIDE, debugging on target is straight forward using the available mechanisms of that IDE. TouchGFX projects generated by TouchGFX Generator or directly from an Application Template is ready for debugging using the selected tool chain.

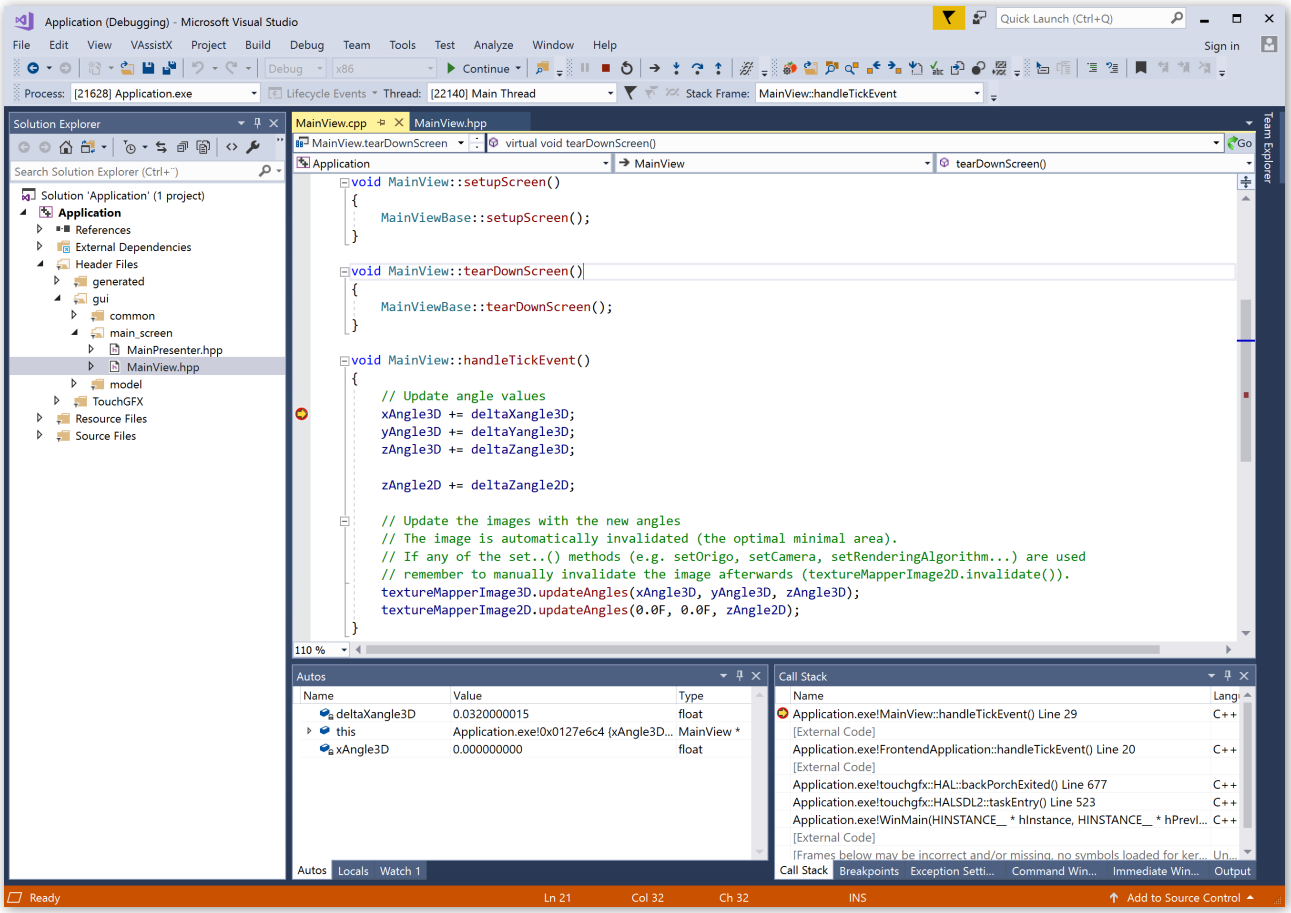
Board bring up code and TouchGFX AL is executed only on your target board and needs to be debugged there. The UI part of your application can be debugged either on target or using the simulator. On target you will typically debug things like performance issues, such as animation speed, update frequency and responsiveness. Other UI specific issues can be debugged on target but is often faster to debug using the simulator.

Simulator Debugging

Debugging UI related issues such as animation movements, transitions, element updates, program logic and so on, is in most cases much more efficient to test and debug using the simulator instead of flashing your board each time.

TouchGFX applications comes with support for Visual Studio and supplies and maintains a project file for it. Using Visual Studio you are able to run the simulator in debug mode, utilizing all the debug features of the IDE.

You are not restricted to Visual Studio if you want to debug using the simulator. The application can be compiled with GCC and if you are using a different IDE this can most likely be set up to debug a GCC compiled project. However, you need to configure your IDE to do this on your own.



Debugging in Visual Studio

Using the DebugPrinter

The DebugPrinter class is an easy way to print debug messages on the display without adding a TextArea or other widgets to the screens. For example, this can be used to show events from a backend or measurements like FPS or rendering time.

Before you can use the DebugPrinter you need to allocate an instance and pass it to the Application class. This can be done e.g. in the constructor of FrontendApplication:

The debug printer needs to be compatible with the framebuffer format. Here we add a 16bpp debug printer in `gui/src/common/FrontendApplication.cpp`:

```
#include <gui/common/FrontendApplication.hpp>

#include <platform/driver/lcd/LCD16bpp.hpp>
LCD16DebugPrinter lcd16bppDebugPrinter;

FrontendApplication::FrontendApplication(Model& m, FrontendHeap& heap)
  : FrontendApplicationBase(m, heap)
{
  lcd16bppDebugPrinter.setPosition(0, 0, 240, 40);
  lcd16bppDebugPrinter.setScale(2);
  lcd16bppDebugPrinter.setColor(0x00); //black
```

```
  Application::setDebugPrinter(&lcd16bppDebugPrinter);
}
```

Here we have configured the DebugPrinter to write in the upper 240 x 40 pixels.
In your application you can now print a string using:

```
char debugStringBuffer[30];
void updateDebugString()
{
  static int count = 0;
  count++;
  snprintf(debugStringBuffer, sizeof(debugStringBuffer), "tick: %d", count);
  Application::getDebugPrinter()->setString(debugStringBuffer);
  Application::invalidateDebugRegion();
}
```

NOTE

Characters from ascii 32 (space) to ascii 126 ('~') are available.

DebugPrinter Classes

The DebugPrinter instance must be compatible to the LCD class used. This table lists the DebugPrinter class names:

LCD class	DebugPrinter class
LCD1bpp	LCD1DebugPrinter
LCD2bpp	LCD2DebugPrinter
LCD4bpp	LCD4DebugPrinter
LCD8bpp_ARGB2222	LCD8ARGB2222DebugPrinter
LCD8bpp_ABGR2222	LCD8ABGR2222DebugPrinter
LCD8bpp_RGBA2222	LCD8RGBA2222DebugPrinter
LCD8bpp_BGRA2222	LCD8BGRA2222DebugPrinter
LCD16bpp	LCD16DebugPrinter
LCD16bppSerialFlash	LCD16DebugPrinter

LCD class	DebugPrinter class
LCD24bpp	LCD24DebugPrinter
LCD32bpp	LCD32DebugPrinter

Use the DebugPrinter class that matches the LCD class you are using.

Version: 4.16

Examples

To help further explore the possibilities and features of TouchGFX, multiple premade examples are made available to the user. These examples can be accessed through the Startup Window of TouchGFX Designer and all include free-to-use images, code, etc., which means that they can even be used as a base to build your own unique application from. Examples are combined with Application Templates to create TouchGFX applications. If you are new to TouchGFX, this can be a great starting source of inspiration and knowledge about how TouchGFX applications function.

Some examples focus on single features while others implement several different functionalities found in TouchGFX. Examples are divided into two different types.

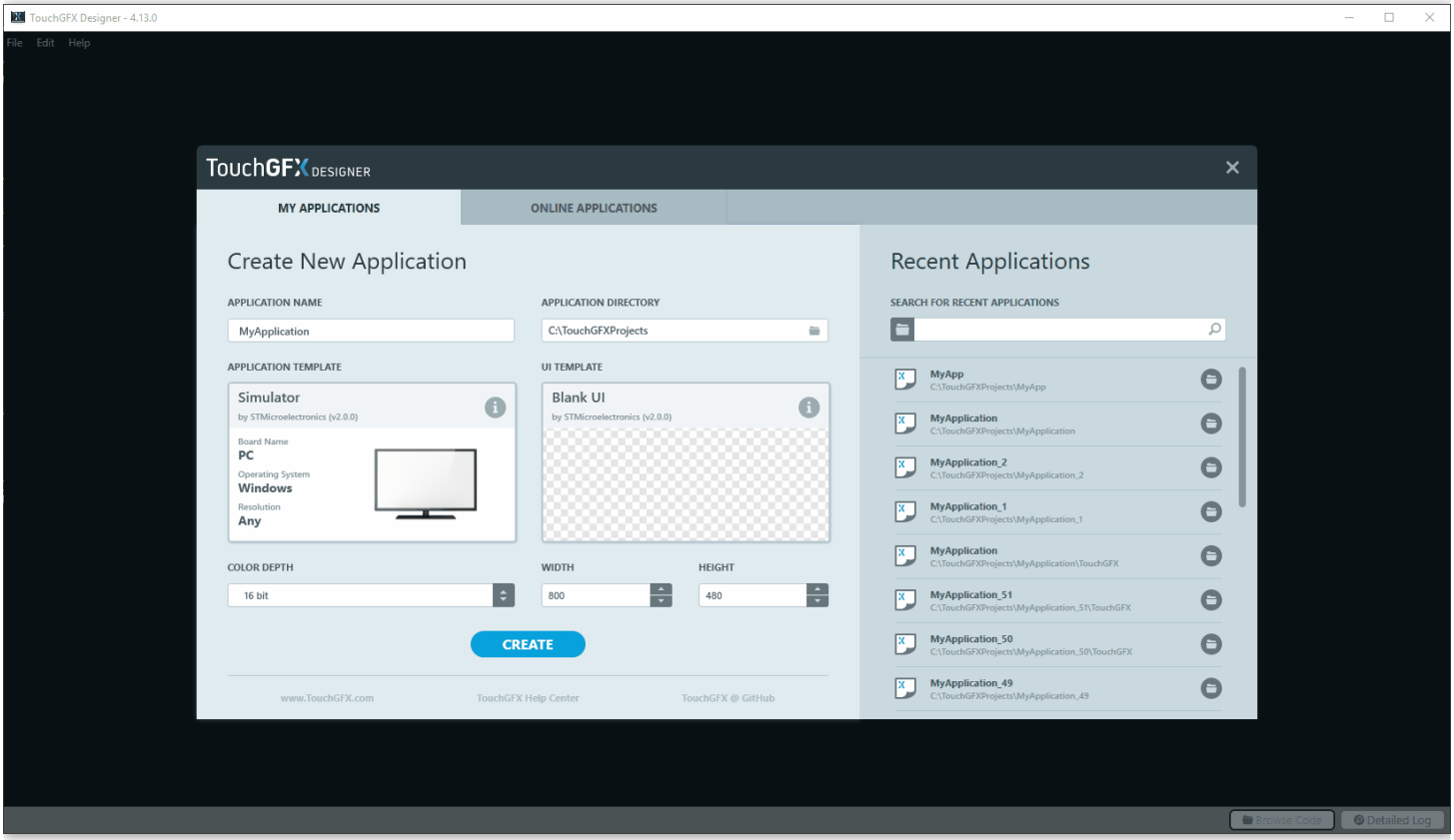
UI Templates

UI Templates are generally smaller, more self-contained examples that mostly focus on specific features such as different widgets. UI Templates can run on any STM32 evaluation kit and the PC simulator, but for the best experience it is encouraged to create projects where the resolution of the UI Template fits the resolution of your board. Some UI Templates are also built with specific color depths in mind, which means they might not display as well on lower color depth displays.

The UI Templates also include several demos made by the TouchGFX team which showcase more features with higher quality UI design. These can be a great place to start to get a feeling for what TouchGFX is capable of.

To create an application using a UI Template, start by pressing the card under the 'Application Template' label to see the available Application Templates. Click whichever Application Template you want and then press 'Select'. Next, press the card under the 'UI Template' label to see the available UI Templates. Click whichever UI Template you want and then press 'Select'. Optionally, select another resolution and color depth in the drop downs. Finally, press 'Create' to create an application from the selected Application Template and UI Template. Press either 'Run Simulator' or 'Run Target' to see the application running.

An animation of these steps can be seen below:



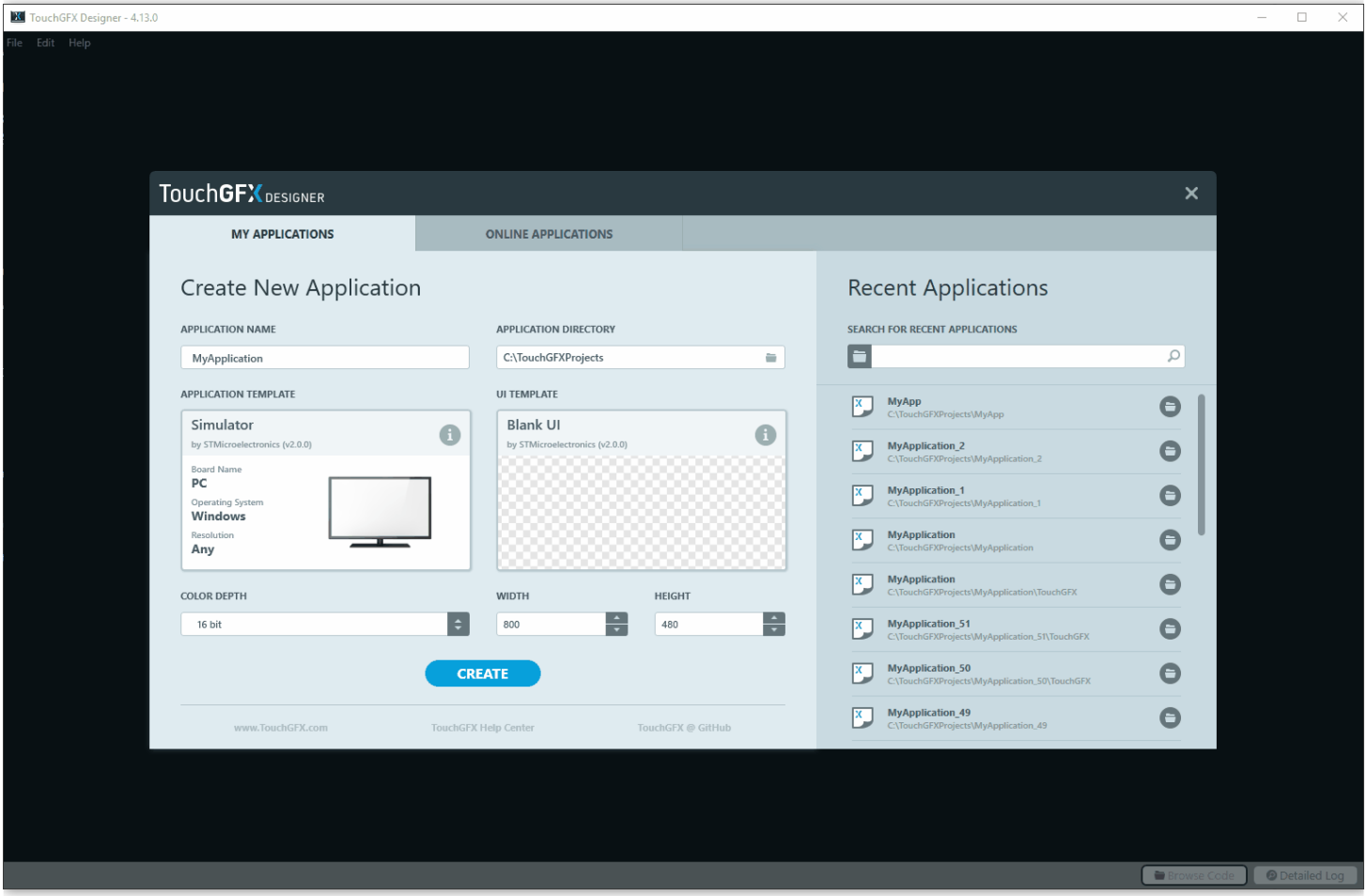
Creating a project using a UI template

Online Applications

Online Applications are out-of-the-box applications for specific hardware solutions and it is therefore not possible to run these on any other STM32 evaluation kit than the one it was created for (aside from the PC simulator). These examples are generally a lot larger and explore multiple different features of the TouchGFX framework and can also include sample integration with the hardware.

To create an application using an Online Application, first access these by clicking the 'Online Applications' tab in the top of the startup window. Click the card under the 'Online Application' label to show what applications are available. Click the application you want and press 'Select'. Finish by pressing 'Create' and either press 'Run Simulator' or 'Run Target' to see the application running.

An animation of these steps can be seen below:



Creating a project using an online application

FURTHER READING

To read more about how to create applications using examples, see the **Startup Window section.**

Version: 4.16

Keyboard Shortcuts

Listed below are a full list of the keyboard shortcuts supported by TouchGFX to increase productivity.

TouchGFX Designer Features

File Management	
`CTRL + N`	Show startup window
`CTRL + O`	Open project from File Explorer
`CTRL + S`	Save current project

Startup Window	
`ESC`	Close current window

Add Widget Menu	
`A`	Show add widget menu (focus search textbox if already open)
`ENTER`	Insert currently highlighted widget and close add widget menu
`ESC`	Close add widget menu

Canvas	
`DEL`	Delete selected widget(s)
`CTRL + C`	Copy
`CTRL + V`	Paste
`CTRL + Z`	Undo
`CTRL + Y`	Redo
`CTRL + F`	Bring selected widget(s) forward
`CTRL + B`	

	Send selected widget(s) backward
`CTRL + A`	Select all widgets
`← / ↑ / → / ↓`	Move selected widget(s) 1 pixel
`CTRL + ← / ↑ / → / ↓`	Move selected widget(s) 10 pixel
`MOUSE WHEEL`	Scroll up / scroll down
`SHIFT + MOUSE WHEEL`	Scroll left / scroll right
`CTRL + MOUSE WHEEL UP / '+'`	Zoom in
`CTRL + MOUSE WHEEL DOWN / '-'`	Zoom out
`CTRL + 0`	Reset zoom
`MOUSE DRAG`	Select widgets
`CTRL + MOUSE DRAG`	Pan canvas

Debugging	
`F5`	Run Simulator
`F6`	Run Target
`F7`	Generate Code
`ALT + L`	Show/hide detailed log

TouchGFX Simulator Features

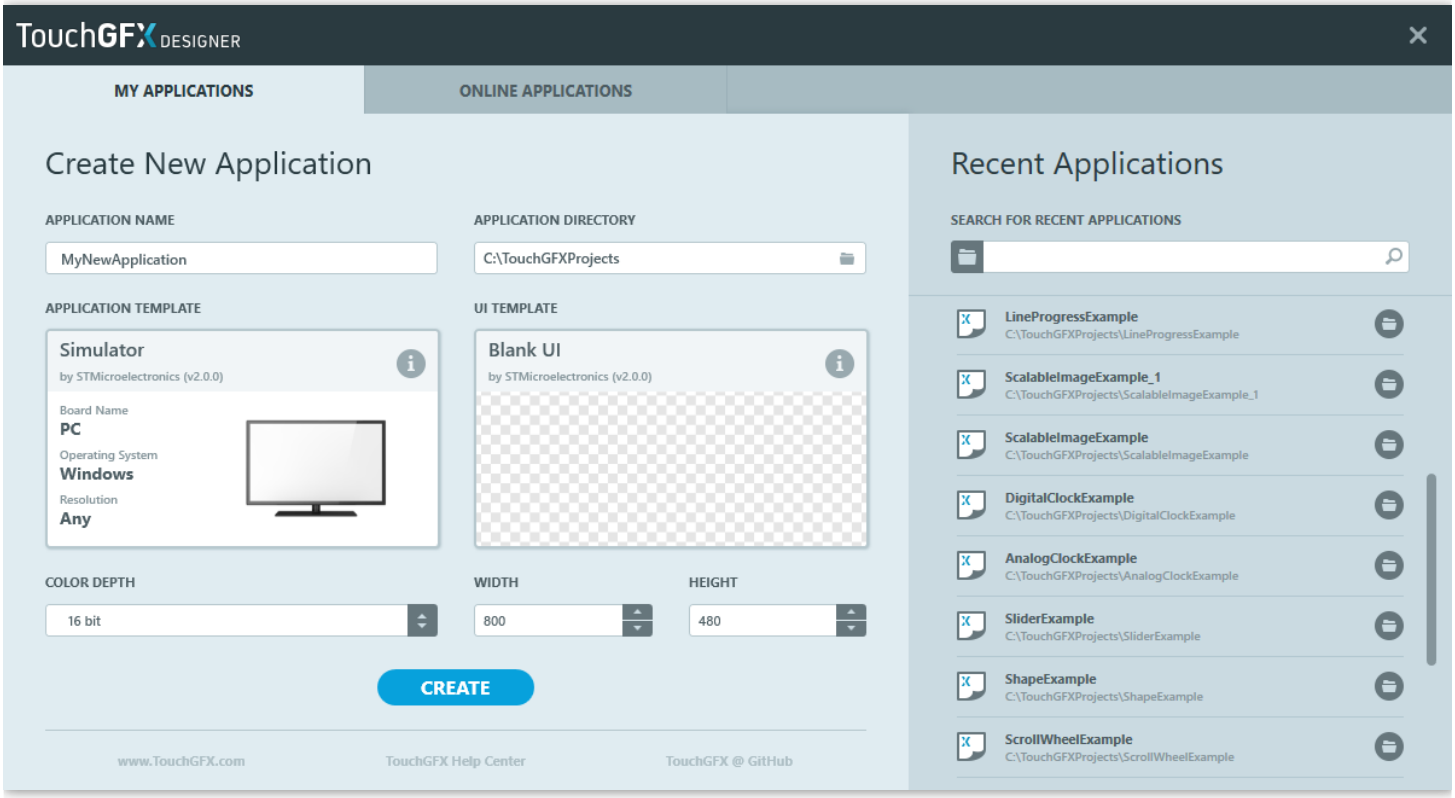
Simulator	
`F1`	Enables/disables debug info
`F2`	Enables/disables highlighting invalidated area
`F3`	Takes a screenshot and places the image under the `screenshots` folder
`CTRL + F3`	Takes screenshots of the next 50 frames and places the images

	under the `screenshots` folder
`SHIFT + F3`	Takes a screenshot and places it in your clipboard
`F4`	If a simulator skin is used - enables/disables the simulator skin If a simulator skin is *not* used - enables/disables window border
`F9`	Pauses the simulator by preventing ticks to be sent to the application. Pressing F9 again will resume normal execution.
`F10`	While the simulator is paused (after pressing F9) it is possible to send a single tick to the application by pressing F10 thereby "single step" the application.
`ESC`	Close the simulator

Version: 4.16

Startup Window

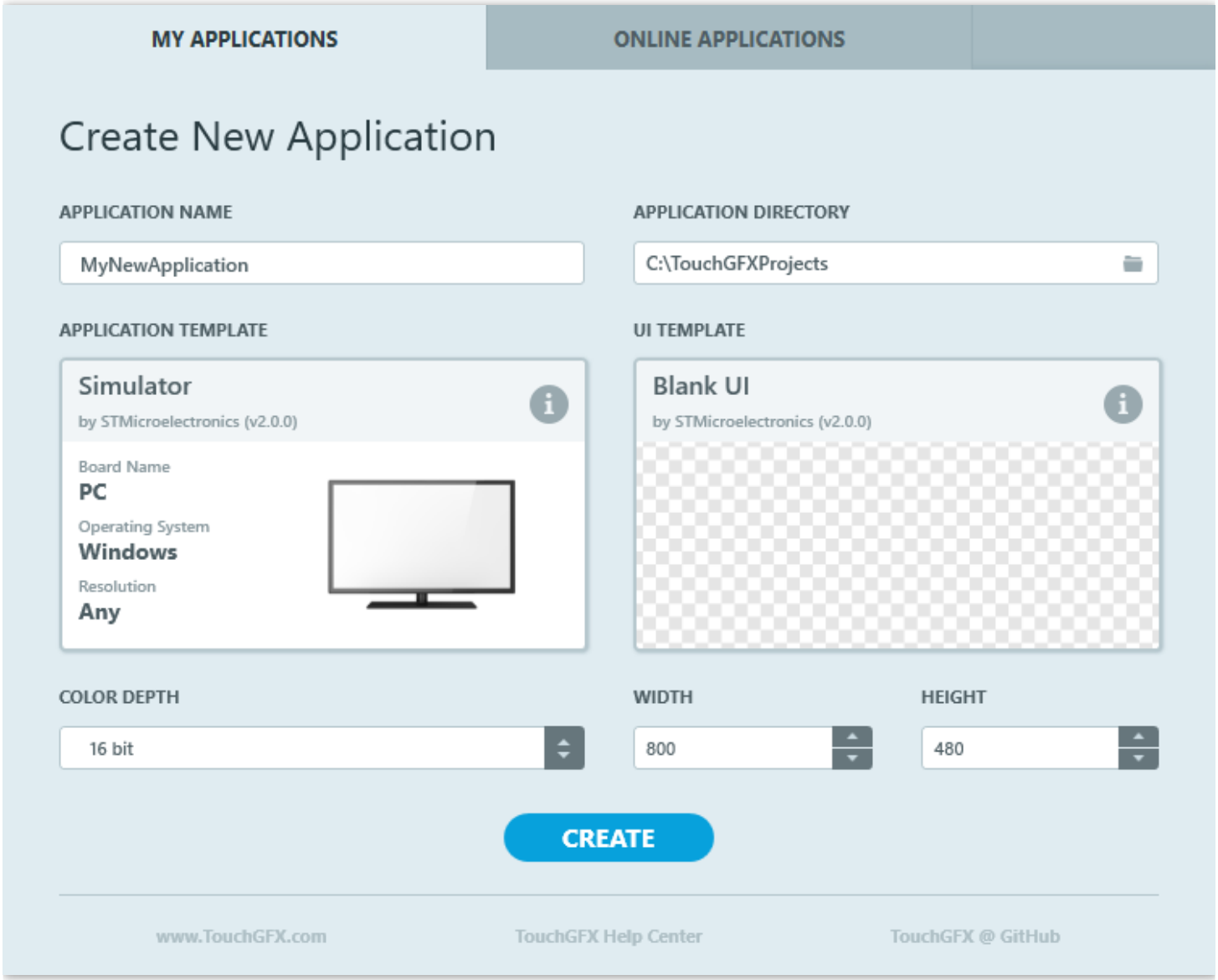
The Startup Window of TouchGFX Designer is where new projects can be created, demo projects can be explored and existing projects can be opened from.
The Startup Window can also be opened by pressing `CTRL + N`.



Startup Window in TouchGFX Designer

Create New Application

In the My Applications tab new projects can be created by combining an Application Template and UI Template.



Create New Application view in the Startup Window

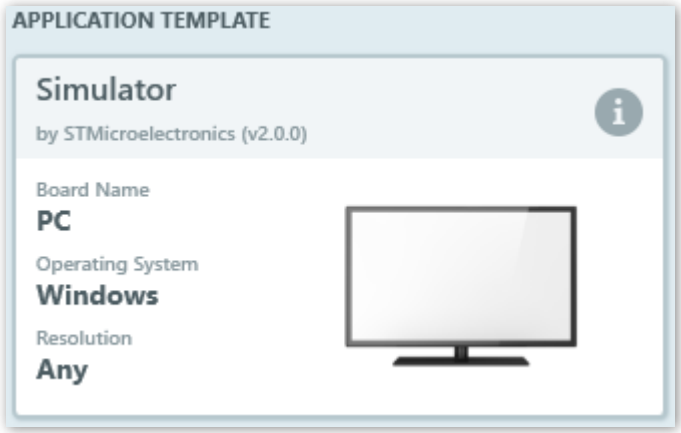
Application Name
This will determine the name of the new project, as well as the name of the folder the new project will be contained in.

Application Directory
This will determine the location of the new project.

Application Template
This selects the Application Template used when creating the new project.

To reveal more information about the selected Application Template, the icon with an 'i' located in the top right corner, as can be seen in the image below, can be clicked.

To change the Application Template, simply click the Application Template to bring up all available Application Templates.



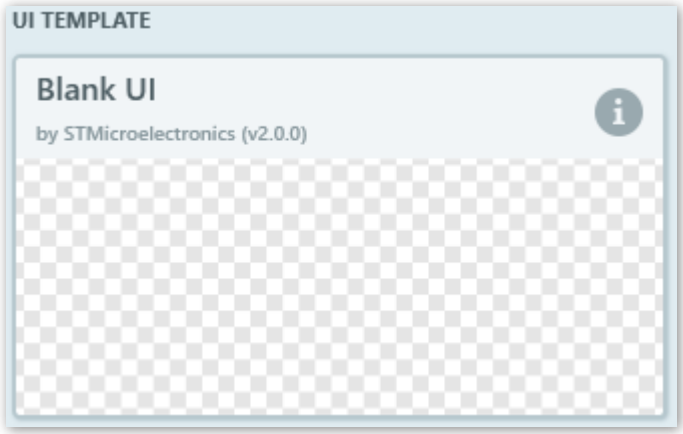
Application Template selector in the Startup Window

UI Template

This selects the UI Template used when creating the new project.

To reveal more information about the selected UI Template, the icon with an 'i' located in the top right corner, as can be seen in the image below, can be clicked.

To change the UI Template, simply click the UI Template to bring up all available UI Templates.



UI Template selector in the Startup Window

Color Depth

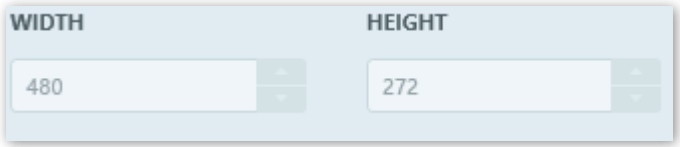
This dropdown will contain the color depths supported by the selected Application Template.

Width & Height

The adjustment of width and height will vary depending on which Application Template has been selected. The Simulator application template will support any resolution size between 0 x 0 and 2000 x 2000

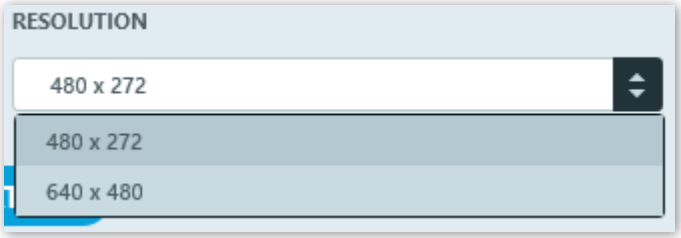
Width and Height free size adjustment in the Startup Window

Some Application Templates only support one resolution size, therefore the width and height adjustment will be locked to whatever the selected Application Template dictates the resolution size should be.



Width and Height locked size adjustment in the Startup Window

Some Application Templates support multiple resolutions, therefore the width and height adjustment will be changed to a dropdown with available resolution sizes from the selected Application Template.

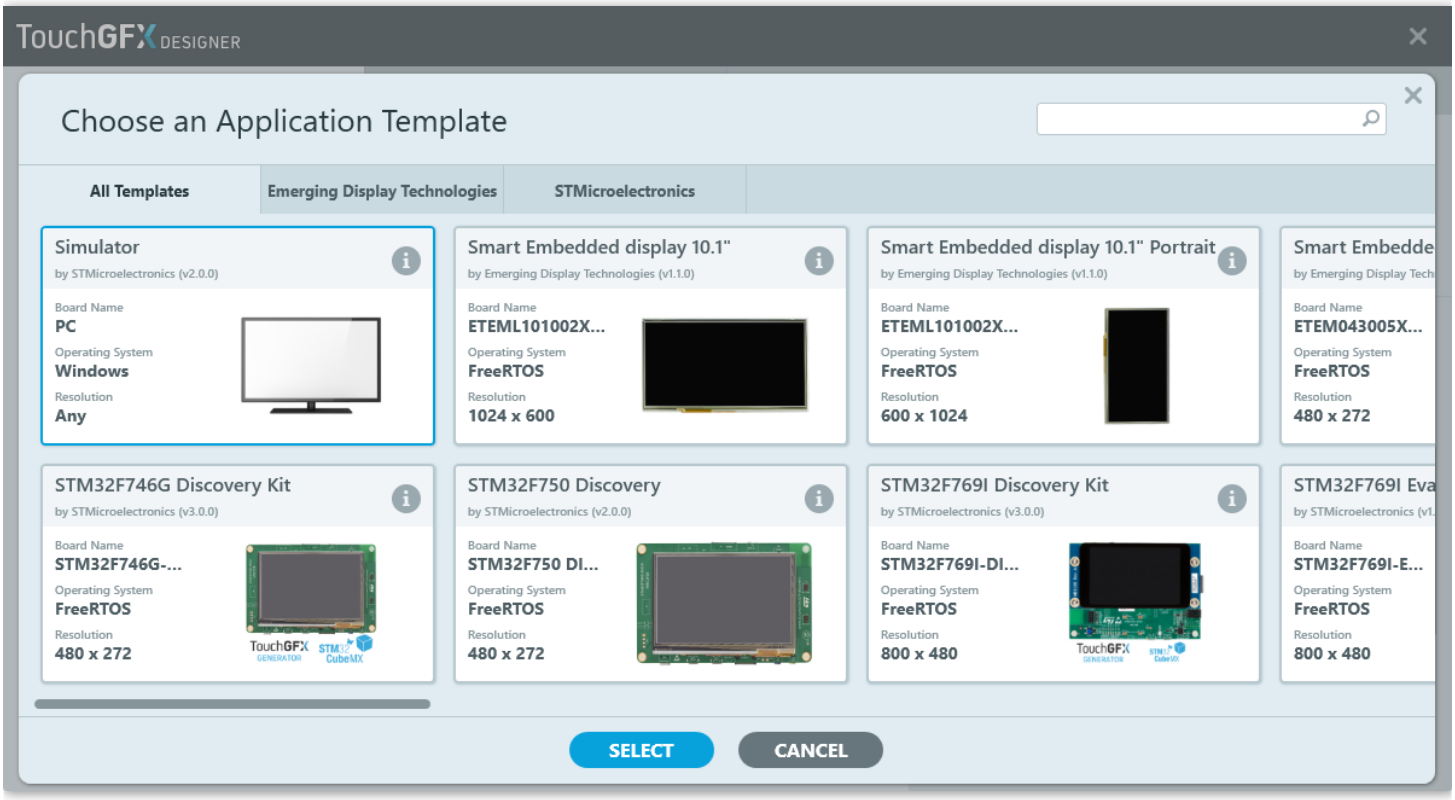


Resolution dropdown size adjustment in the Startup Window

Application Templates

The Application Templates View consists of a search bar in the top right corner, and a tab control that will filter the Application Templates by their provider.

To select an Application Template, simply click the entry, which will mark it with a blue border, as can be seen on the 'Simulator' Application template in the image below, and click the blue button with the label 'SELECT'.



Application Templates in the Startup Window

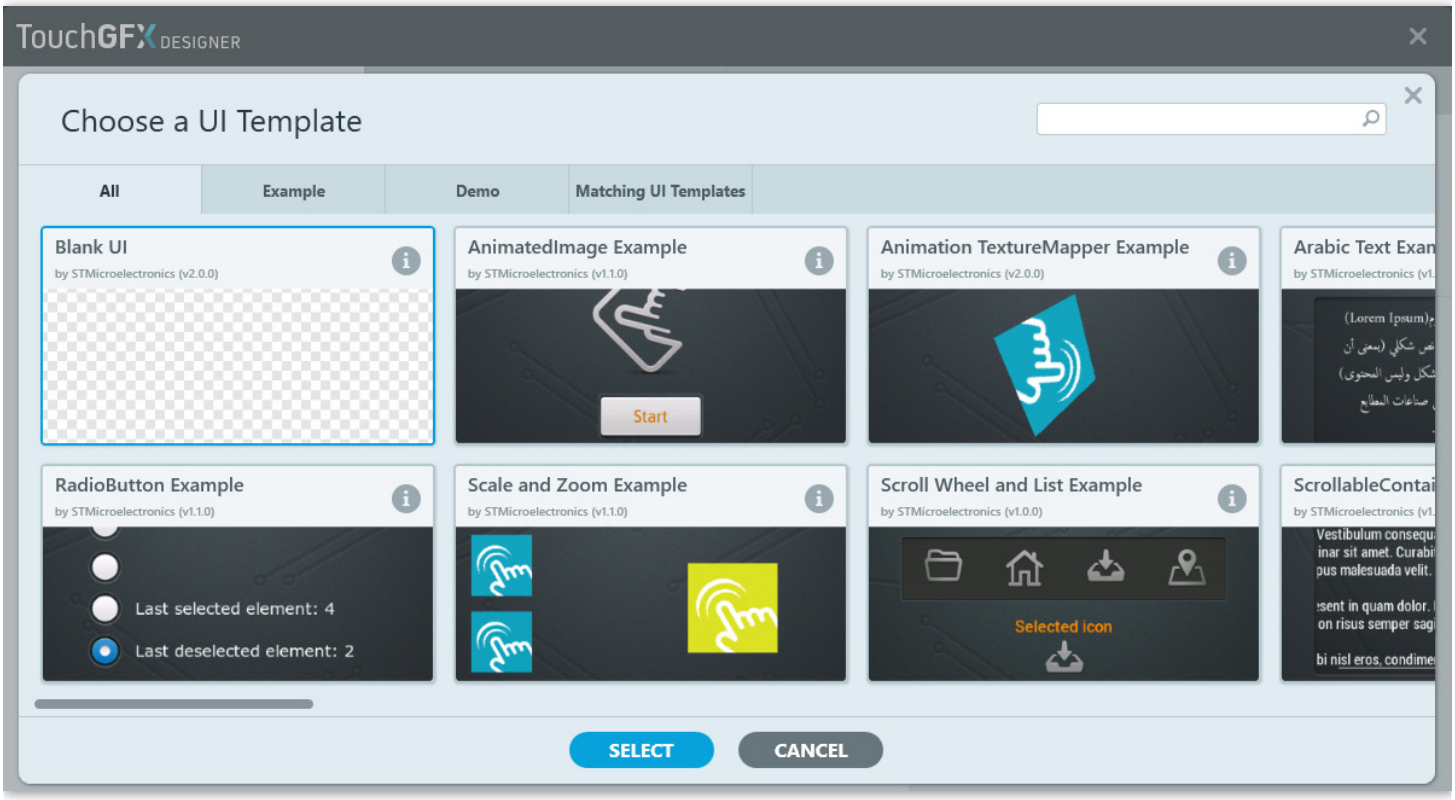
To view more information about an Application Template, each of the Application Templates, as shown in the image above have an icon with an 'i' located in their top right corner, that can be clicked. When clicked the window in the image below will be displayed, here it is also possible to choose the version of the Application Template to use.

Application Template info in the Startup Window

UI Templates

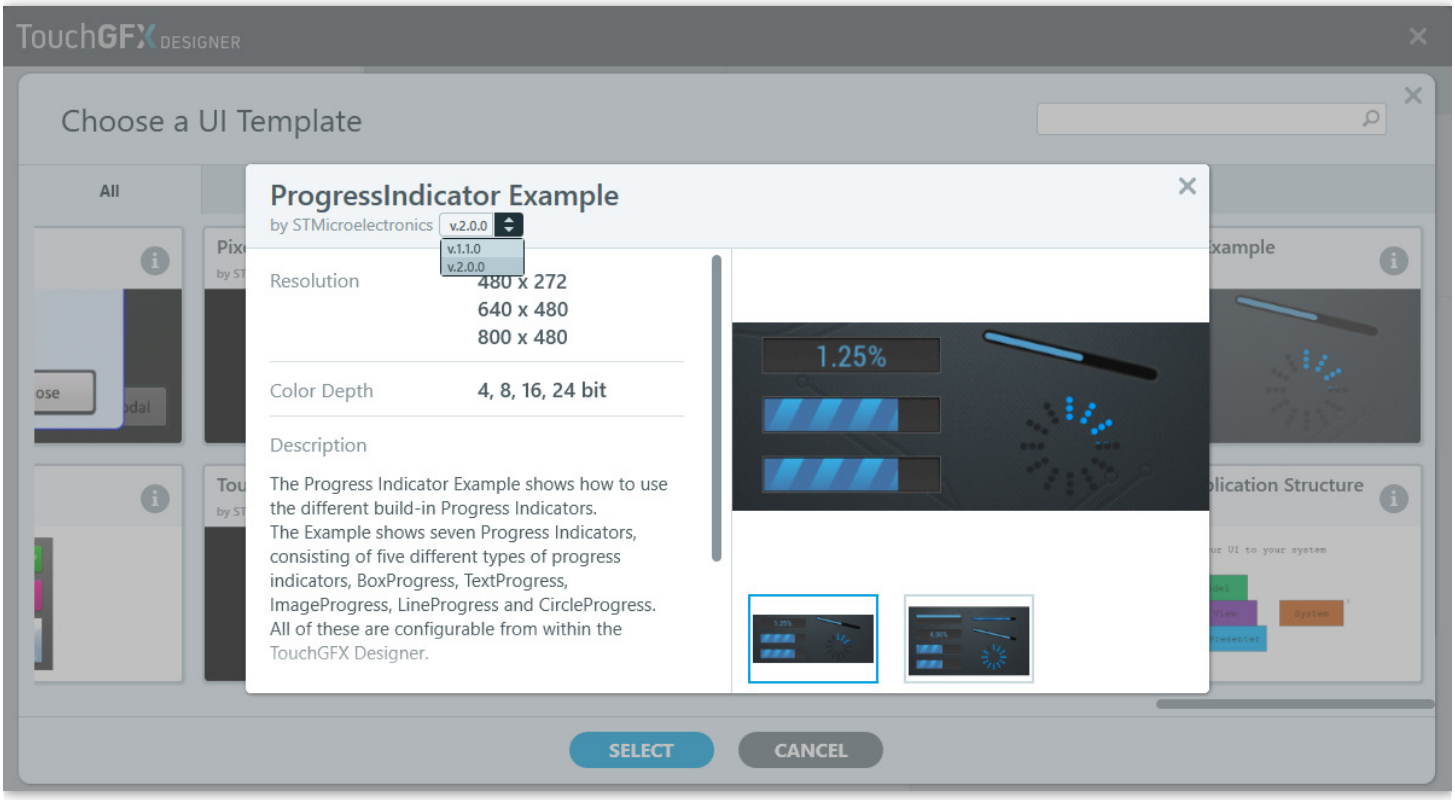
The UI Templates View consists of a search bar in the top right corner, and a tab control that will filter the UI Templates by their category.

To select an UI Template, simply click the entry, which will mark it with a blue border, as can be seen on the 'Blank UI' UI Template in the image below, and click the blue button with the label 'SELECT'.



UI Templates in the Startup Window

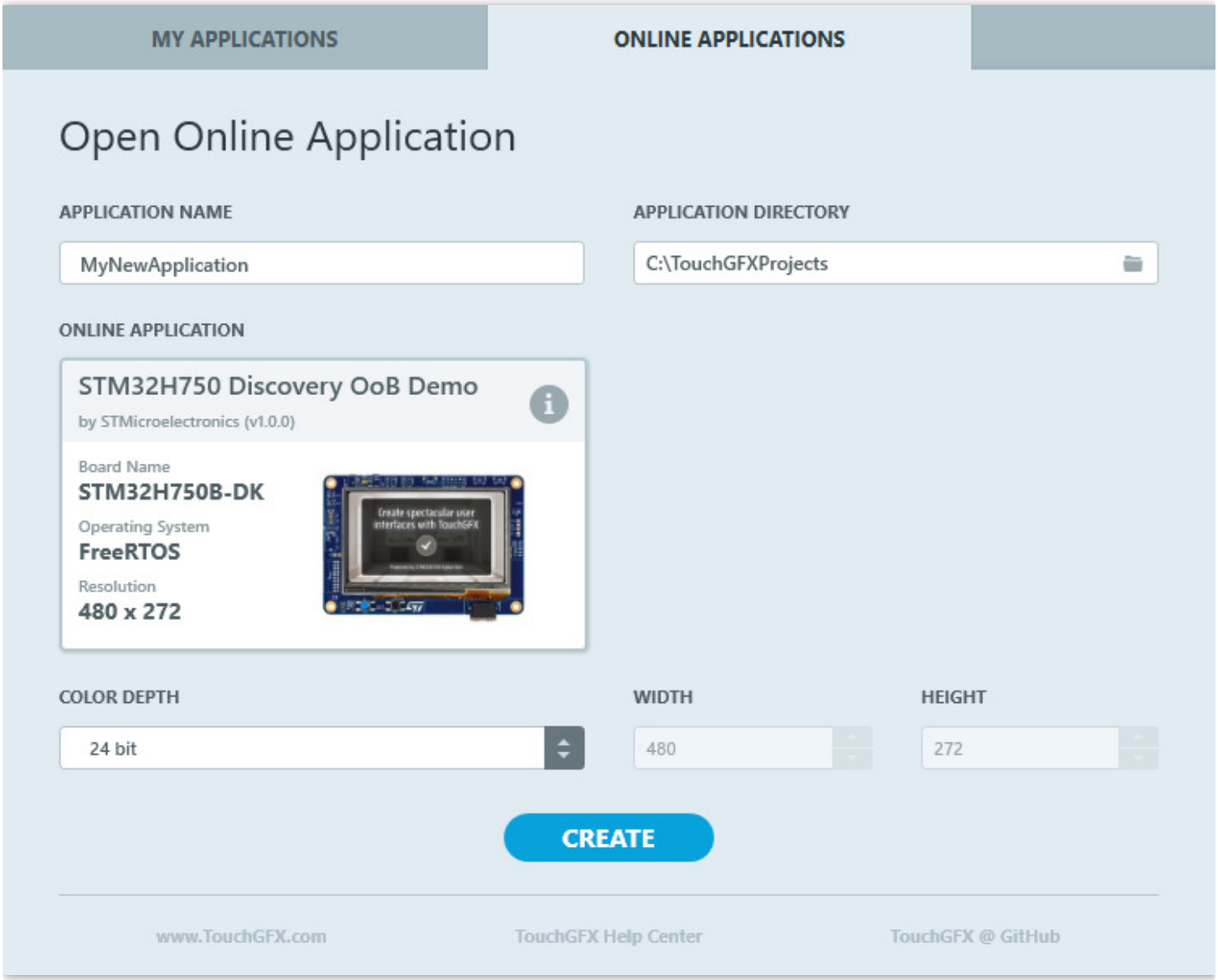
To view more information about an UI Template, each of the UI Templates, as shown in the image above have an icon with an 'i' located in their top right corner, that can be clicked. When clicked the window in the image below will be displayed, here it is also possible to choose the version of the UI Template to use.



UI Template info in the Startup Window

Open Online Applications

In the Online Applications tab new projects can be created by using an Online Application.



Open Online Application view in the Startup Window

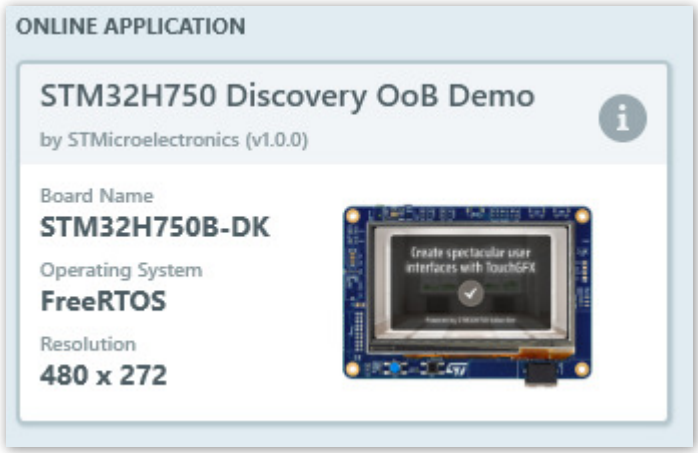
Application Name
This will determine the name of the new project, as well as the name of the folder the new project will be contained in.

Application Directory
This will determine the location of the new project.

Online Application
This selects the Online Application used when creating the new project.

To reveal more information about the selected Online Applications, the icon with an 'i' located in the top right corner, as can be seen in the image below, can be clicked.

To change the Online Applications, simply click the UI Template to bring up all available Online Applications.



Online Application selector in the Startup Window

Color Depth

This dropdown will contain the color depths supported by the selected Online Applications.

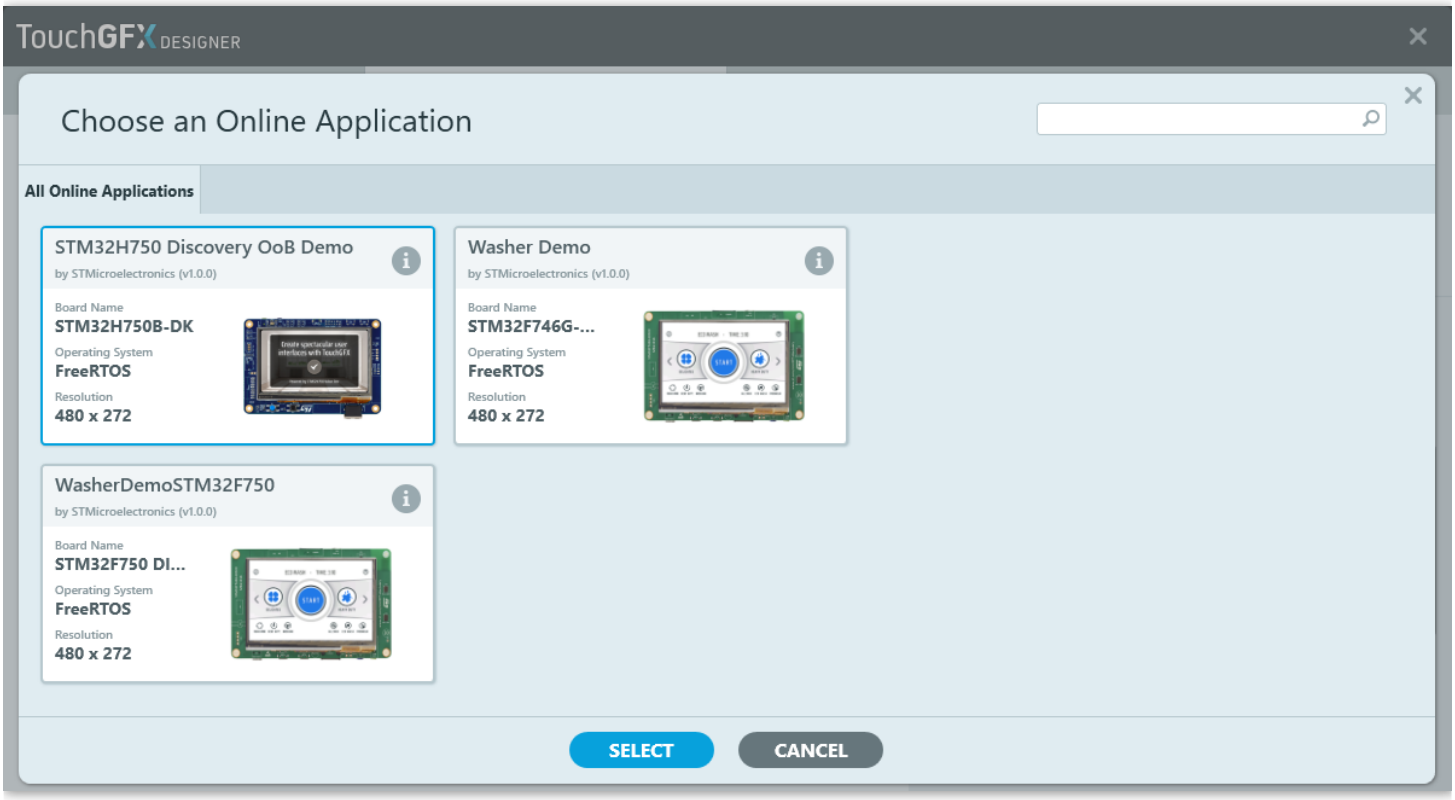
Width & Height

Online Applications only support one resolution size, therefore the width and height adjustment will be locked to whatever the selected Online Applications dictates the resolution size should be.

Online Applications

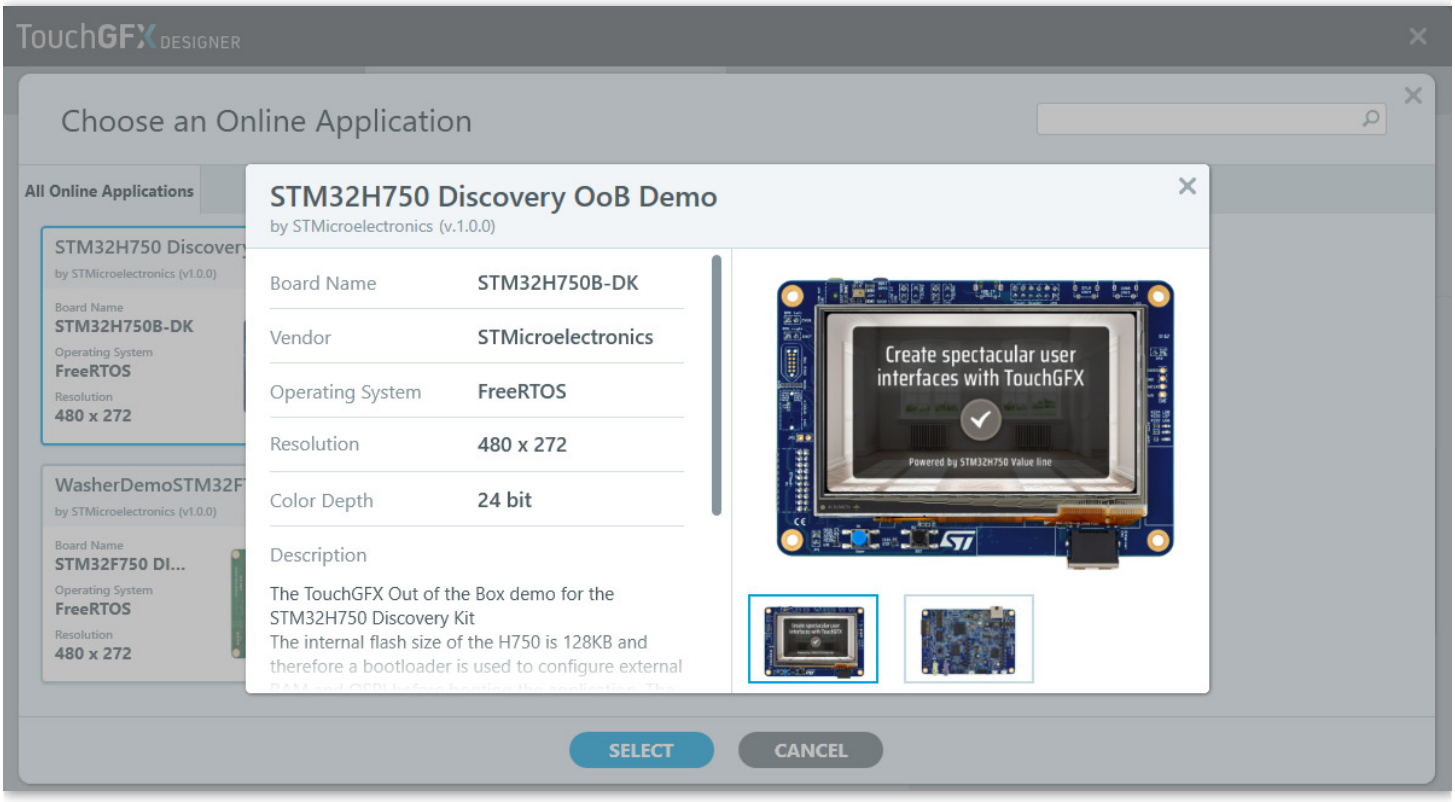
The Online Applications View consists of a search bar in the top right corner, and a list of the available Online Applications.

To select an Online Application, simply click the entry, which will mark it with a blue border, as can be seen on the 'STM32H750 Discovery OoB Demo' Online Application in the image below, and click the blue button with the label 'SELECT'.



Online Applications in the Startup Window

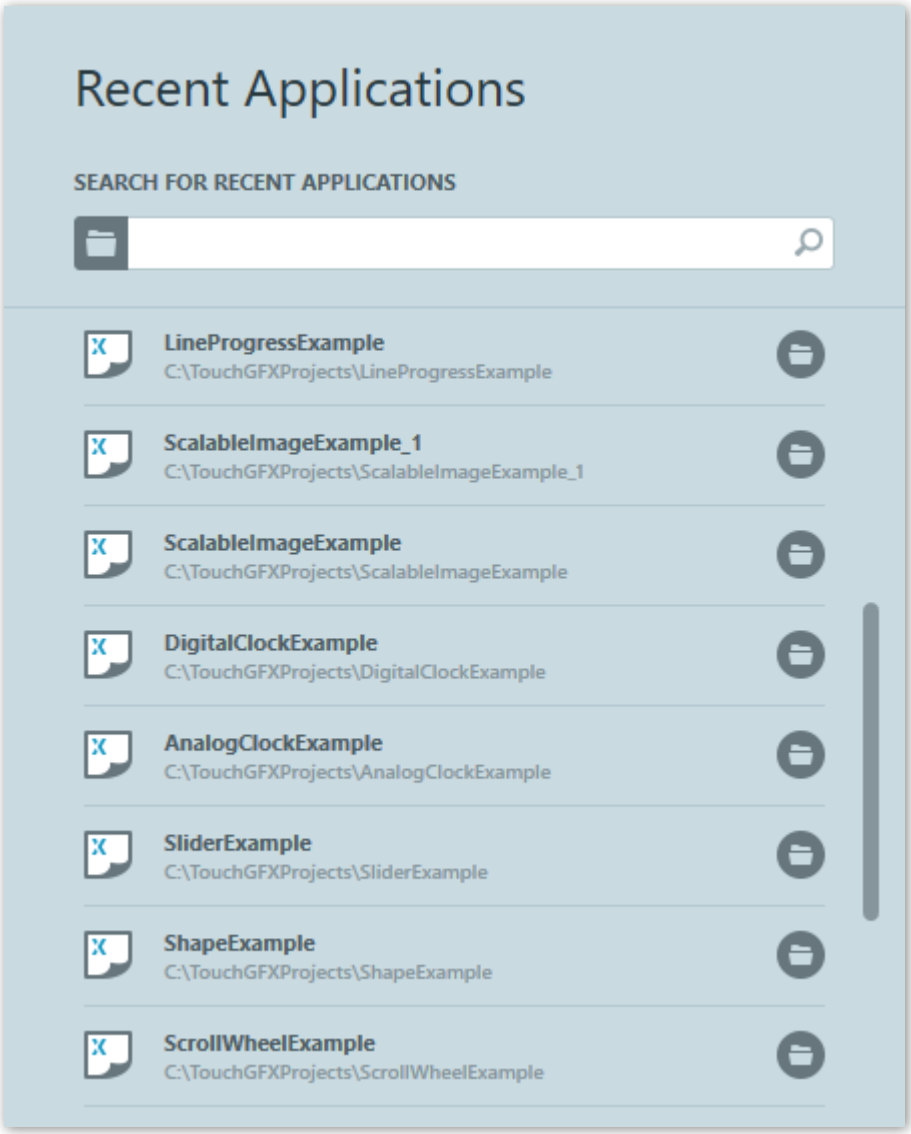
To view more information about an Online Application, each of the Online Applications, as shown in the image above have an icon with an 'i' located in their top right corner, that can be clicked. When clicked the window in the image below will be displayed.



Online Application info in the Startup Window

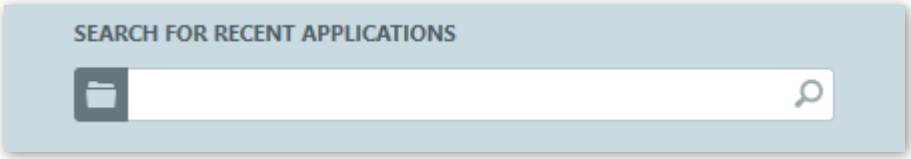
Recent Applications

In the Recent Applications view the projects that have most recently been created, opened or modified can be found.



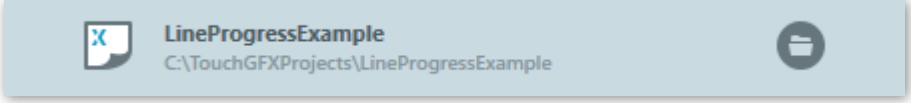
Recent Applications view in the Startup Window

As can be seen in the image above the Recent Applications view consists of a search bar with a list of recent applications below it.



Recent Applications Search Bar in the Startup Window

The search bar will filter the recent applications list as characters are entered into it. At the left side of the search bar a button with a folder icon is located, pressing this button will open a file browser allowing for manual navigation and opening of a .touchgfx project file.



Recent Application Entry in the Startup Window

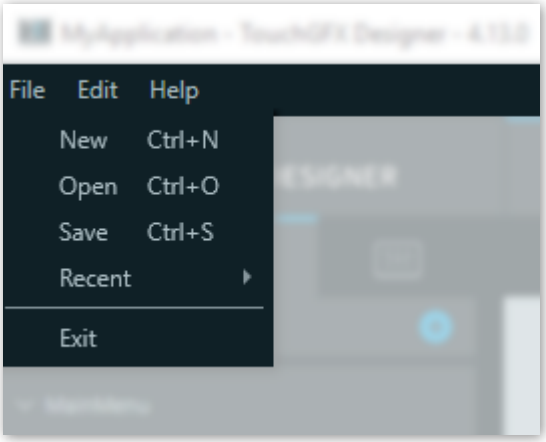
Each entry in the recent applications list consists of the name given to the project, the path to the .touchgfx project file and a button with a folder icon. Clicking an entry will load the project and close the Startup Window.
Hovering the folder icon will reveal the full path to the .touchgfx project file, and clicking it will open the folder containing the file in a file explorer.

Version: 4.16

File Menu

The File Menu of TouchGFX Designer consists of a File-, Edit- and Help menu item.

File



File menu item in File Menu

New
Clicking the New button, will open the Startup Window, where a new project can be created.

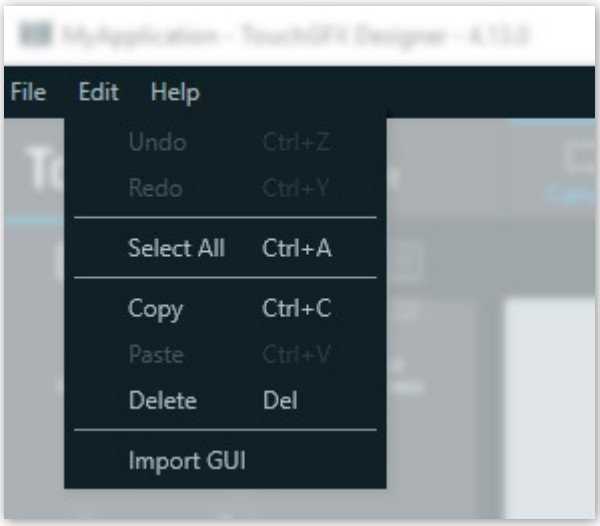
Open
Clicking the Open button, will open a file explorer, allowing for navigation to and loading of a TouchGFX Designer project file (.touchgfx)

Save
Clicking the Save button, will save the project that is currently open, into its TouchGFX Designer project.
The project is also saved when running the simulator, flashing to target and generating code.

Recent
Hovering/clicking the Recent button, will reveal the projects that have most recently been created, opened or modified, clicking any of these will load that project.

Exit
Clicking the Exit button, will shutdown the TouchGFX Designer.

Edit



Edit menu item in File Menu

Undo

Clicking the Undo button, will undo changes made in the Canvas View. This button may be grayed out if there are no changes to undo, or the Canvas View is not currently selected.

Redo

Clicking the Redo button, will redo changes made in the Canvas View. This button may be grayed out if there are no changes to redo, or the Canvas View is not currently selected.

Select All

Clicking the Select All button, will select all widgets added to the Screen or Custom Container that is currently visible in the Canvas View.

Copy

Clicking the Copy button, will add the Widget, Screen or Custom Container that is currently selected in the Canvas View, to the copy/paste buffer.

Paste

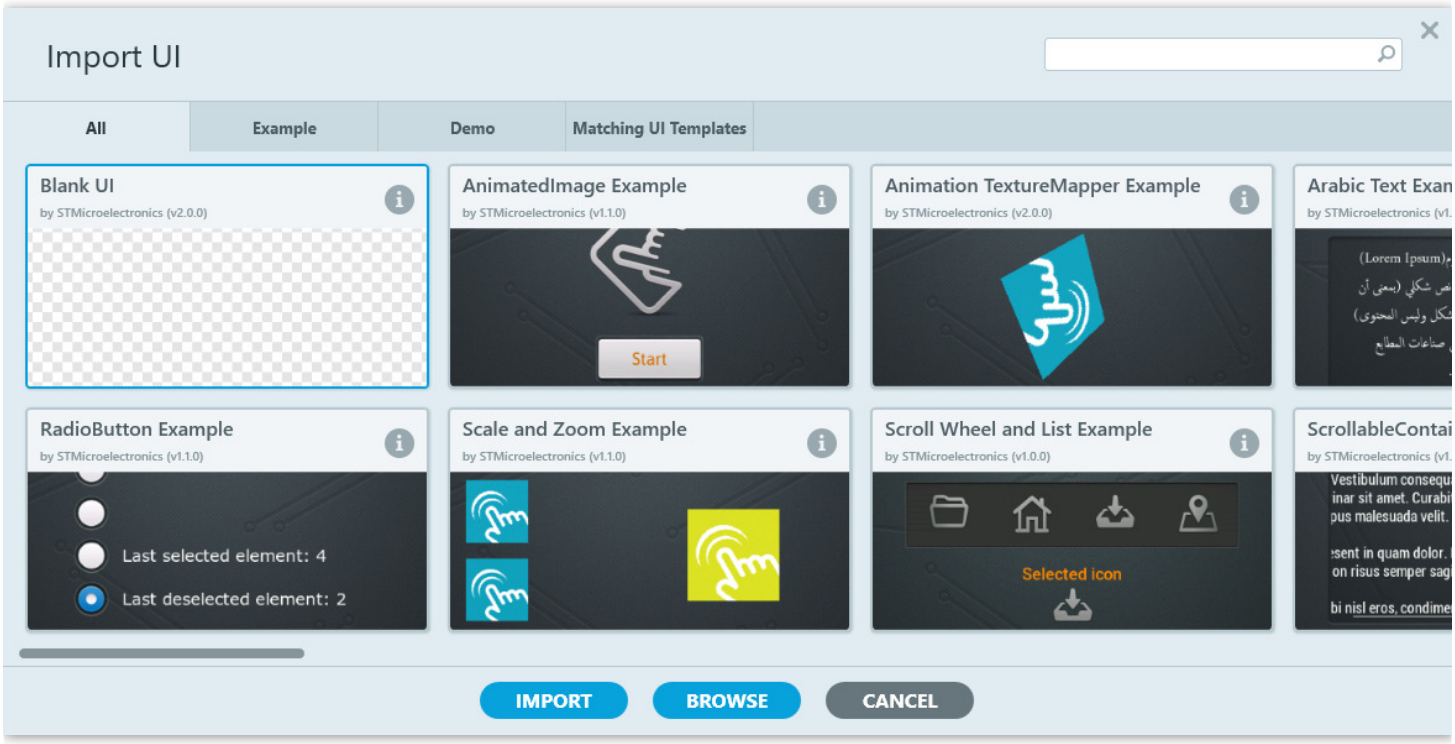
Clicking the Paste button, will paste the Widget, Screen or Custom Container that is currently in the copy/paste buffer. Paste is not available if no object has been copied.

Delete

Clicking the Delete button, will delete the Widget, Screen or Custom Container that is currently selected in the Canvas View.

Import GUI

Clicking the Import Gui button, will open the Import GUI window.



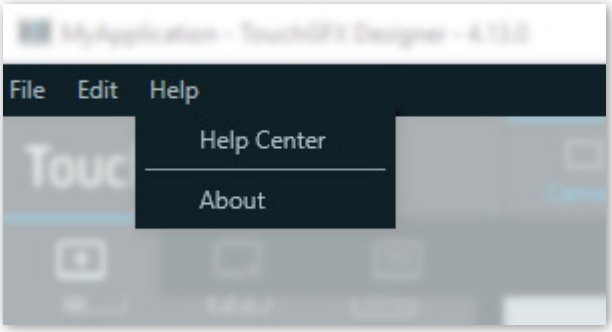
Import UI window in TouchGFX Designer

Here a UI can be imported into the project that is currently open. A UI can be chosen from the Examples and Demo's provided by STMicroelectronics, or by clicking 'Browse' allowing for selection of a TouchGFX Designer project (.touchgfx) to be imported into the current project.

⚠ CAUTION

Importing a UI will overwrite the UI that is already present in a project

Help



Help menu item in File Menu

Help Center

Clicking the Help Center button, will open support.touchgfx.com in your operating systems default browser.

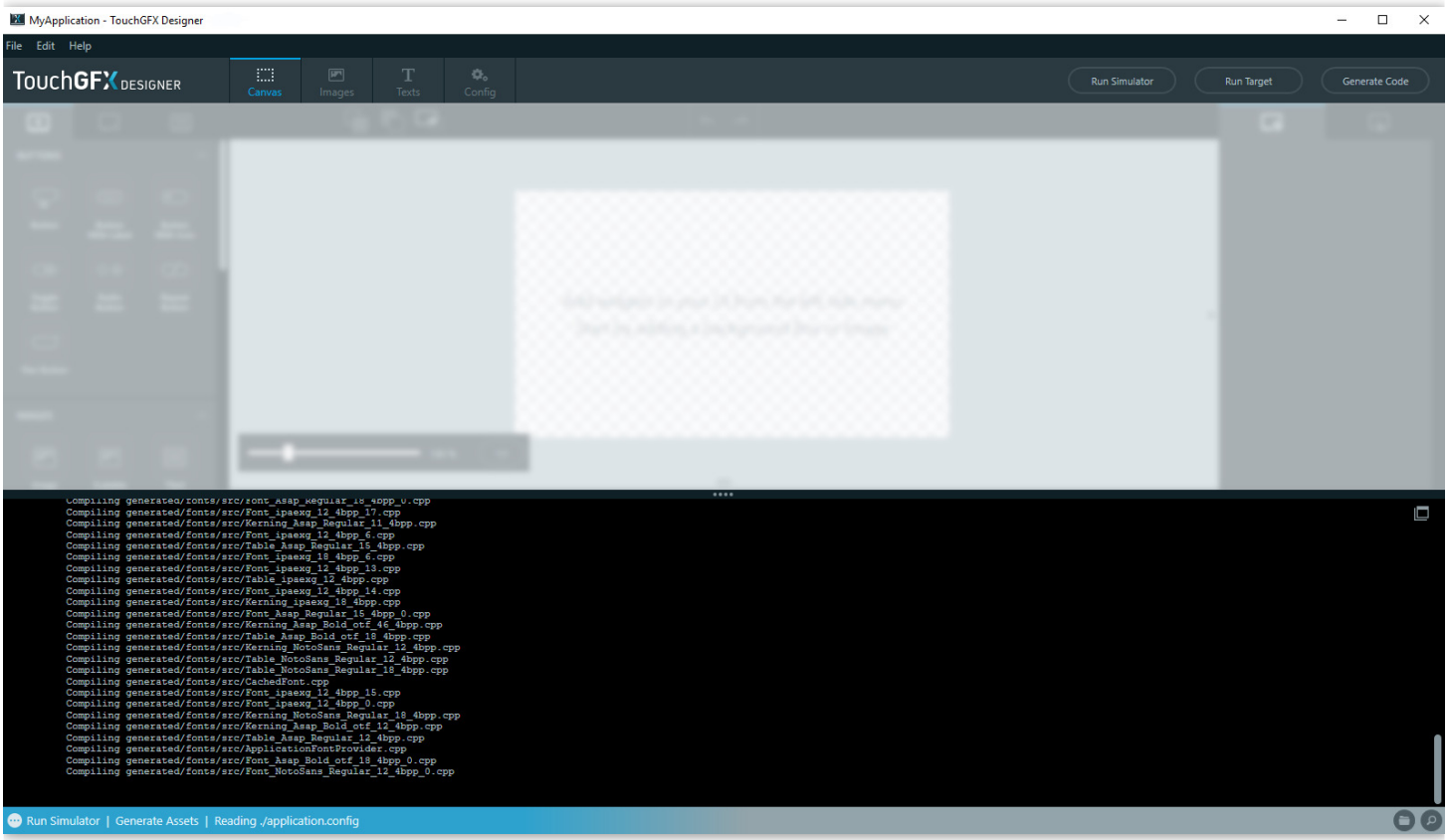
About

Clicking the About button, will open a window containing the Software License Agreement.

Version: 4.16

Main Window

The Main Window of TouchGFX Designer consists of a Navigation Bar, Command Buttons, Notification Bar, and Detailed log. The Main Window forms a frame around the 'View' *(The 'View' area, is the area that has been blurred in the image below)*

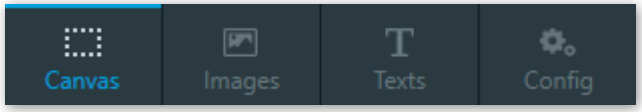


Main window of TouchGFX Designer

Navigation Bar

In TouchGFX Designer, navigation is done through the Navigation Bar (see image below), here the View can be changed to one of the following views:

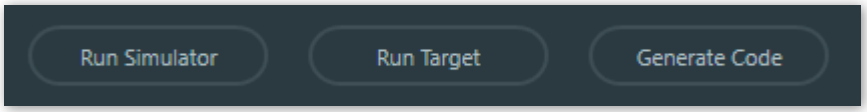
- Canvas used for drag and drop application building.
- Images used for management of the images used in a project.
- Texts used for management of texts and typographies in a project.
- Config used for configuration of various settings for a project.



Navigation bar in TouchGFX Designer

Commands

In the Commands section of TouchGFX Designer three buttons can be found: 'Run Simulator', 'Run Target' and 'Generate Code'. *(See image below)*. These buttons each execute a combination of commands.



Command buttons in TouchGFX Designer

The commands these buttons execute can be overwritten in the Build section in Config.

Run Simulator

The Run Simulator command triggers a complete code generation, then executes the following commands:

- Generate Assets Command
- Post Generate Command
- Compile Simulator Command
- Run Simulator command

The Run Simulator command can also be triggered by pressing `F5`

Run Target

The Run Target command triggers a complete code generation, then executes following commands:

- Generate Assets Command
- Post Generate Command 'Post
- Generate Target Command
- Compile Target Command
- Flash Target command

The Run Target command can also be triggered by pressing `F6`

Generate Code

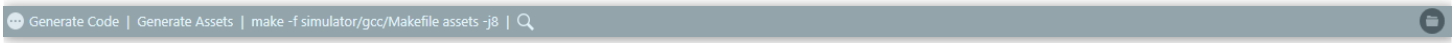
The Generate Code command triggers a complete code generation, then executes following commands:

- Generate Assets Command
- Post Generate Command
- Post Generate Target Command

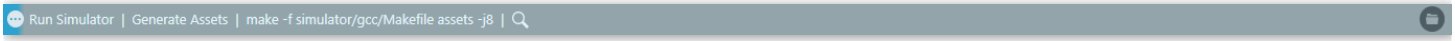
The Generate Code command can also be triggered by pressing `F7`

Notification Bar

The Notification Bar at the bottom of the Main Window, shows the status of the current command being run. If a command fails, the bar will turn red and an error icon will be displayed along with the command that failed. Commands that succeed will first turn green and then will be cleared from the Notification Bar, whereas commands that fail will continue to be displayed until another command is started.



Notification bar success in TouchGFX Designer



Notification bar failed in TouchGFX Designer

Detailed Log

Pressing anywhere on the Notification Bar opens a window showing the full log of the last command that was run by the designer. The output of a command will stream into this window *(See GIF below)*, the window can also be undocked/docked from the Main Window, by pressing the undock/dock icon in the top right corner of the Detailed Log window.

The Detailed Log window can also be toggled with `ALT + L`

Detailed log in TouchGFX Designer

Browse Code

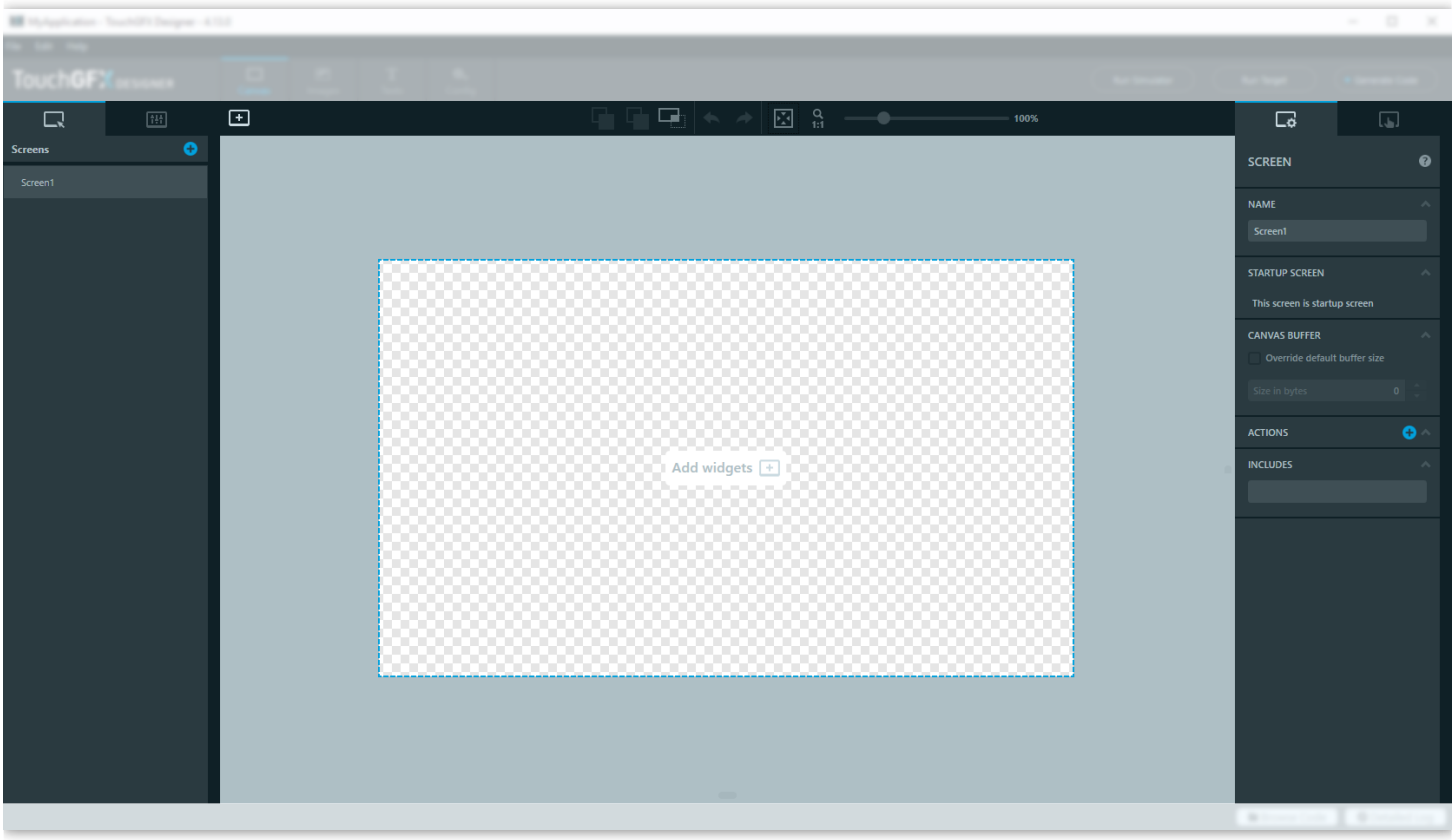
The Browse Code button allows for easy access to the directory of the project by opening the directory in a new File Explorer window.

Browse Code and Detailed Log in TouchGFX Designer

Version: 4.16

Canvas View

The Canvas is the view used for building the graphical parts of an application by providing a visual representation of the interface as it will look while running. The dynamic aspects, like animations and interactions between parts of the system, are described here.



The Canvas View of TouchGFX Designer

Left Side Bar

The side bar to the left contains a tab control, with navigation between and Screens & Custom Containers.

Screens & Custom Containers

Both the Screens tab and Custom Containers tab contain a tree giving an overview of the widgets in each screen/custom container, every widget in the tree, that can contain other widgets can be collapsed by pressing the chevron next to the widgets name.

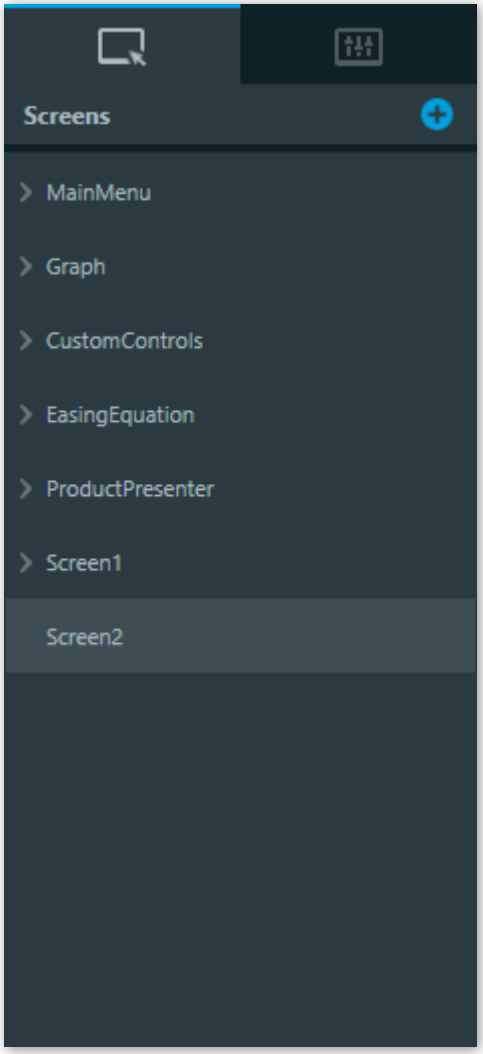
The order of the screens, custom containers, and widgets within can be changed by dragging them below or above other screens, custom containers and widgets, the order can also be changed with the

widget order controls. Changing the order of widgets will determine which widgets is rendered on top of other widgets.

Widgets that are container types, can have children added to them by dragging widget on top of them in the tree view. widgets can also be dragged from one screen to another.

Select multiple widgets by pressing and holding CTRL while clicking widgets.

Screens/Custom Containers can be added to the project by pressing the blue icon with a plus. The Custom Containers can be added to screens and other custom containers from the Widgets tab, they can be found in their own category named 'Custom Containers'.

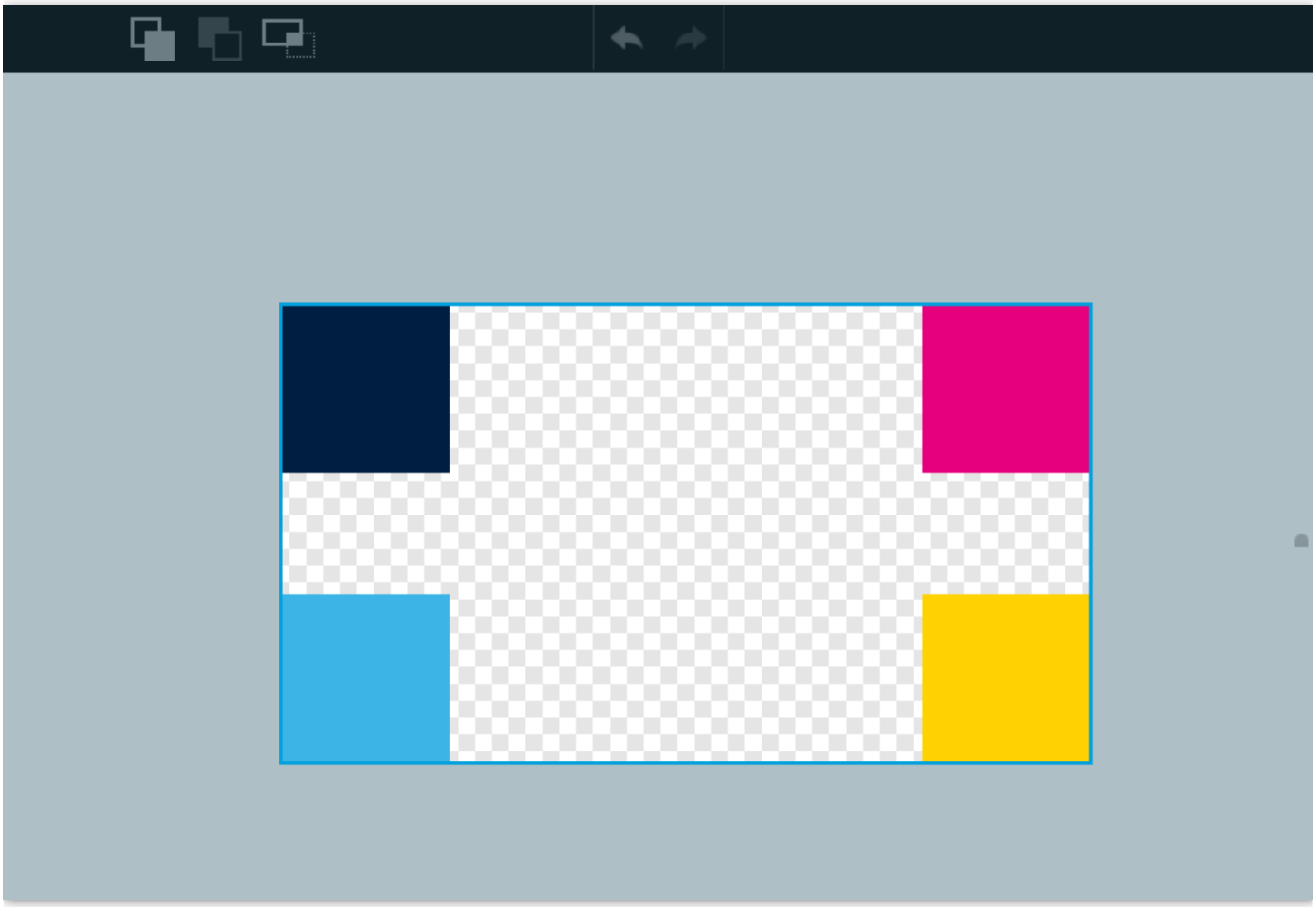


The Screens tree in the left side bar

Canvas

The center of the view contains the canvas which displays the view of the screen or custom container that is currently selected.

The canvas is surrounded by various controls: Widget Order Controls, Content Clipping Control, History Controls and Zoom Controls



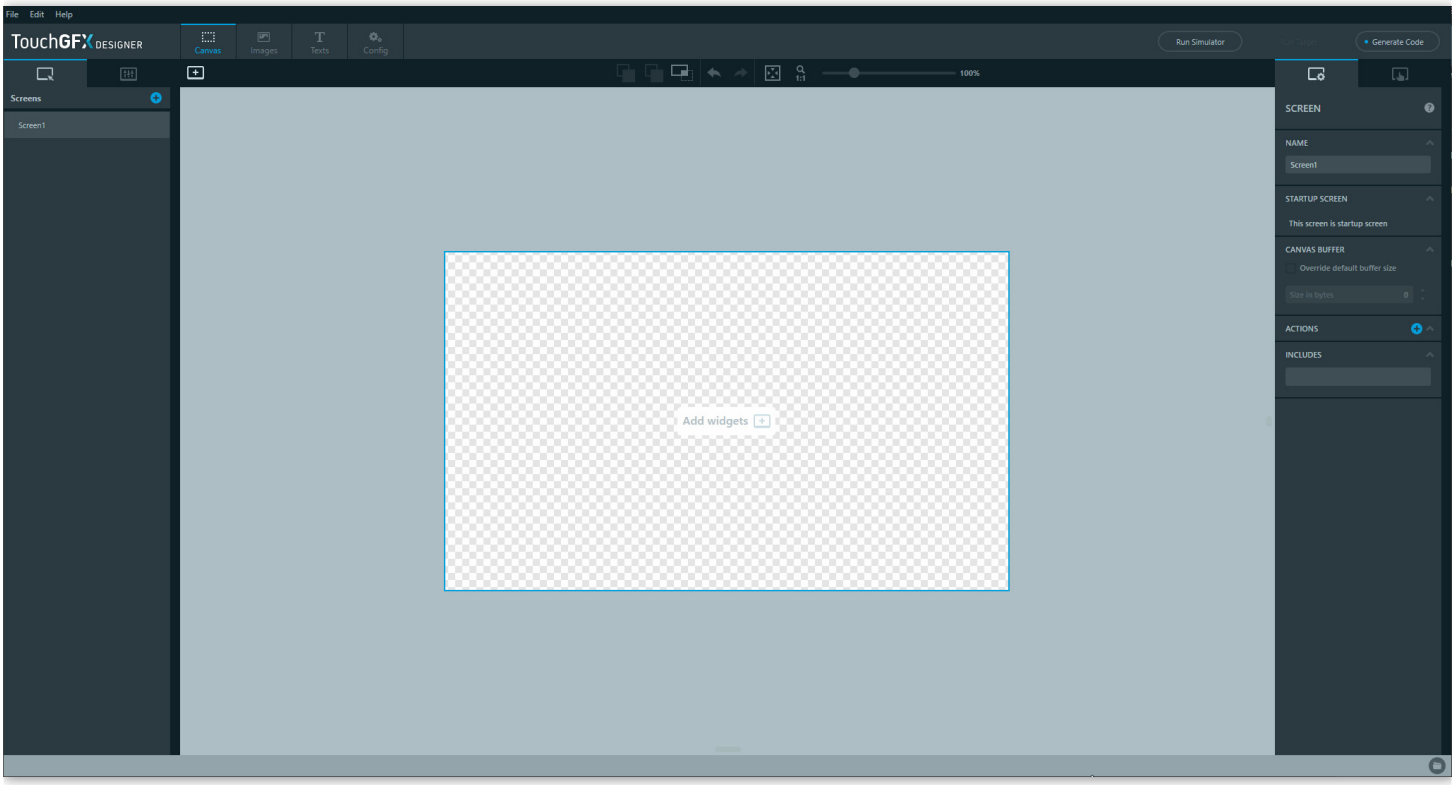
The Canvas in the Canvas View

Add Widget Menu

Clicking the Add Widget button or its shortcut `A` opens up the Add Widget Menu which contains all available widgets grouped into categories. Each category can be expanded and collapsed by pressing the chevron next to the category name.

Clicking a widget, will add it to the canvas of the Screen or Custom Container that is currently selected and visible. A widget can also be added by dragging the widget directly to the canvas.

The Add Widget Menu also contains a search field which, powered by fuzzy search, helps find the widget which suits the search input best. The best result is highlighted. Due to fuzzy search, it is for example possible to input "bwl" and get "Button With Label" due to abbreviations being a factor. After inputting something into the search field, hitting `ENTER` will insert the highlighted widget on canvas and close the Add Widget Menu.



The Add Widget Menu in the Canvas view

Widget Selection

As shown in the animation above, it is possible to select a widget by simply clicking it on the canvas. Multi-selection is also possible by clicking multiple widgets while holding down `CTRL` on the keyboard.

Widget Positioning

As shown in the animation above, it is possible to move and resize widgets by dragging their thumbs.

It is also possible to move selected widgets by 1 pixel using the arrow keys. Holding down `CTRL` while using the arrows keys will move selected widgets by 10 pixels.

Widget Order Controls

The z-order of widgets can be manipulated by the Bring Forward and Send Backwards controls, this also changes their order in the Screens or Custom Container tree.

Bring Forward can also be triggered by pressing `CTRL + F`

Send Backward can also be triggered by pressing `CTRL + B`

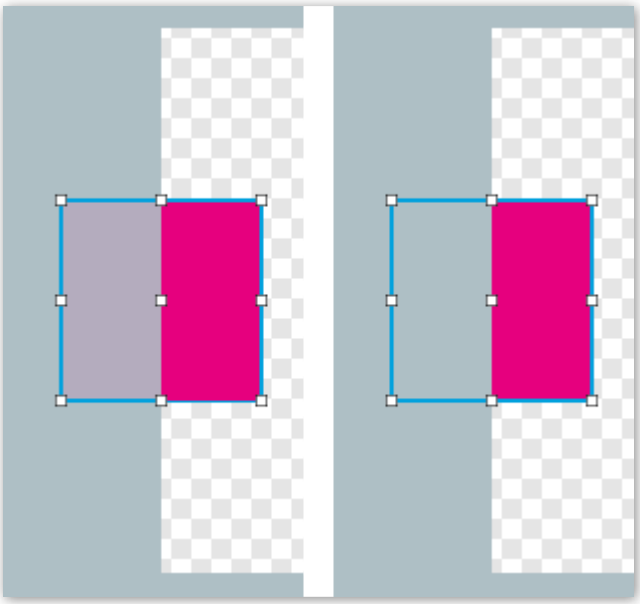
Bring Forward and Send Backwards controls in the Canvas View

Content Clipping Control

In the canvas, widgets outside the borders of the Screen, Custom Container and children of container type widgets are clipped. The content clipping control toggles between clipping the widgets fully or showing them partially anyway.

Show/Hide Clipped Content control in the Canvas View

In the image below a Box is placed halfway outside the borders of a screen, when the content clipping control is set to show clipped widgets, the Box will not be shown fully but instead have the part that is outside desaturated.



Box widget appearance difference when showing and hiding clipped content

History Controls

The history of changes performed on the canvas can be undone and redone through the history controls, located at the top center of the canvas.
Each screen and custom container maintains its own history, therefore to undo or redo a change performed on a certain screen, that specific screen has to be visible on the canvas.

The History controls can also be triggered by pressing `CTRL + Z` and `CTRL + Y`

History controls in the Canvas View

Zoom Controls

The zoom level of the canvas can be controlled with the zoom control in the top right corner of the canvas.

To return to 1:1 zoom scale, press the button next to the zoom slider label '1:1'.

To center the canvas, press the center canvas button.

Zoom controls in the Canvas View

Zooming can also be achieved by using the following shortcuts:

Zoom in	`CTRL + MOUSE WHEEL UP`	`CTRL + '+'`
Zoom out	`CTRL + MOUSE WHEEL DOWN`	`CTRL + '-'`
Zoom reset	`CTRL + 0`	`CTRL + NUMPAD 0`

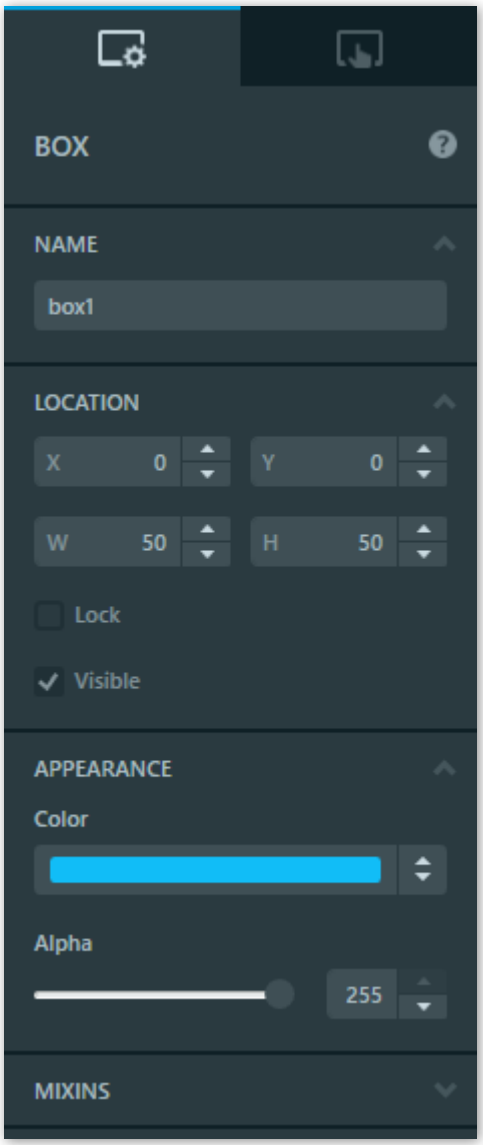
Right Side Bar

The side bar to the right contains a tab control, with navigation to the Properties of the widget currently selected, and the Interactions of the Screen or Custom Container that is currently viewed on the canvas.

Properties

The Properties tab will show the properties of the Widget/Screen/Custom Container that is currently selected. The name of the selected component is displayed at the top of the properties list, next to the name there is a questionmark icon. Clicking this question mark icon will expand the section, displaying a description text and a link to the documentation for the component.

Each of the sections in the properties list can be collapsed and expanded by pressing the chrevron next to the section name.

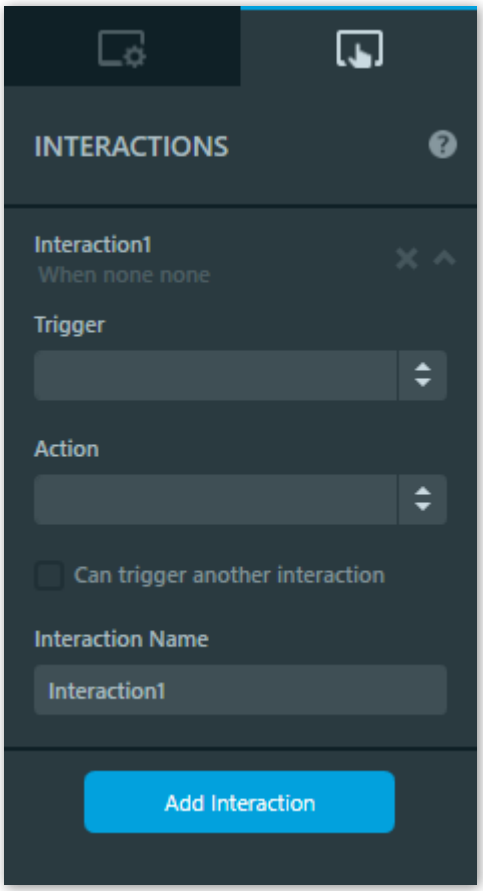


Properties of Box

Interactions

The Interactions tab will show the interactions of the Screen or Custom Container that is currently displayed on the canvas. The questionmark at the top can be pressed to reveal a description of interactions and a link to an article.

Interactions can be added by pressing the blue button labeled 'Add Interaction'. Each interaction can be collapsed and expanded by pressing the chevron next to the name of the interaction. Next to the chevron, a cross is located, this cross will delete the interaction when clicked.



Interactions of a Screen

FURTHER READING

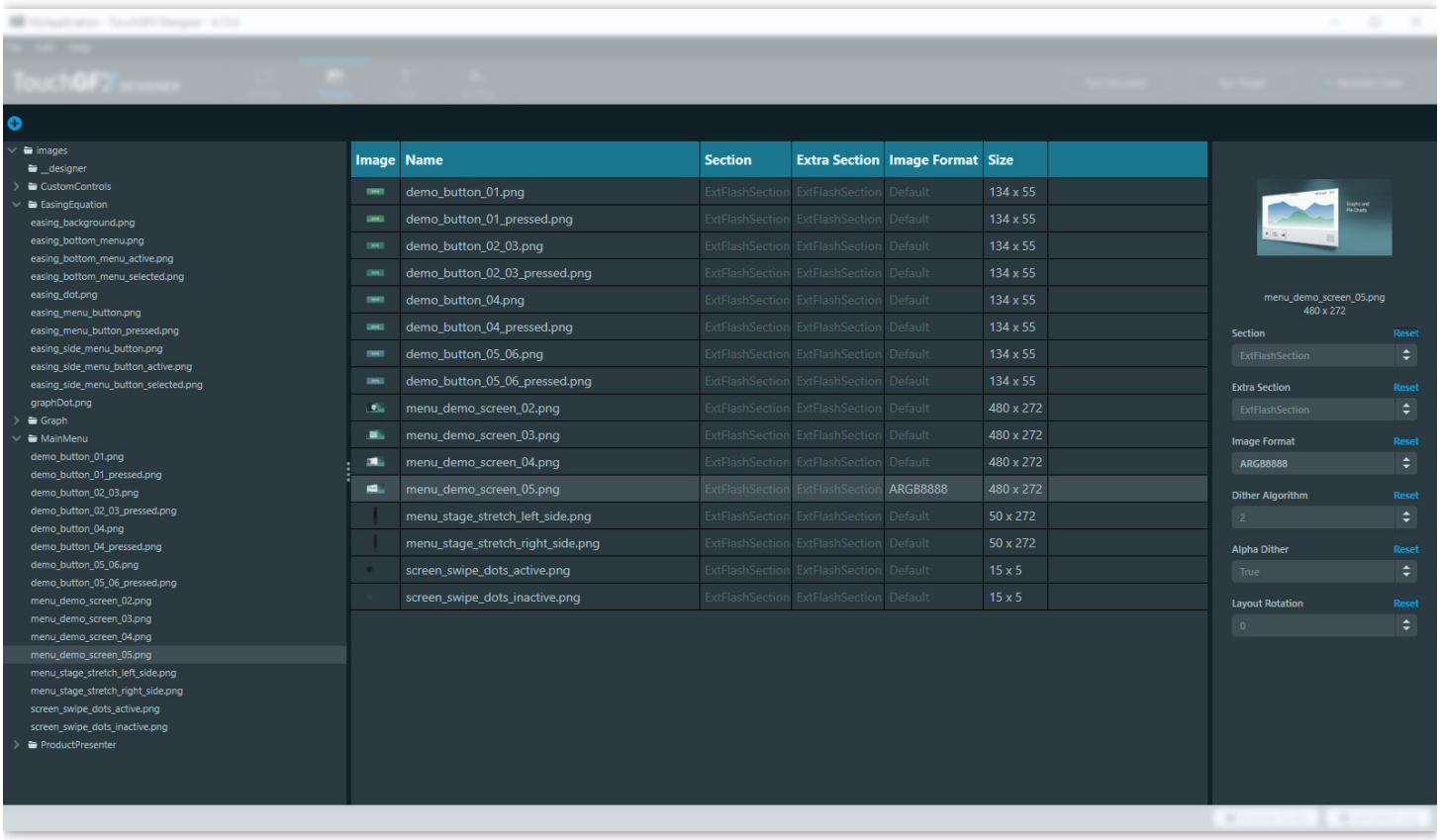
- **Interactions**

Version: 4.16

Images View

The Images View is used to manage the images used in a TouchGFX application (located under the assets\images folder). It includes 3 sections: the tree view (left side), the table view (middle) and the properties view (right side).

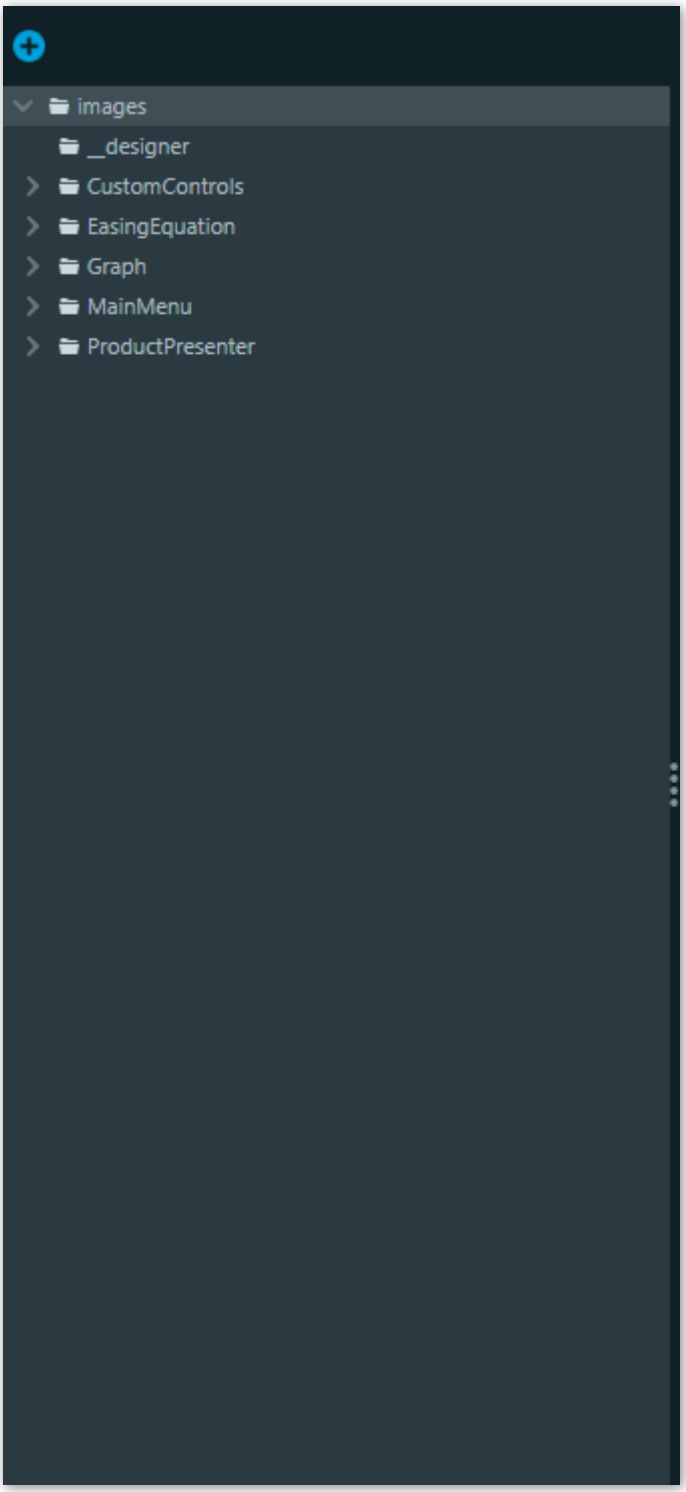
The default configuration values for an image can be changed in the Default Image Configuration in the Config View.



Images view in TouchGFX Designer

Tree View

The tree view provides an overview of the images and folders present in your application. The width of the tree view can be changed by dragging the grid-splitter thumb to suit your needs.



The tree view in Images

You can add images to the assets\images folder by clicking the blue button with a plus icon or by dragging the images directly to the Designer from the File Explorer. Images added to the assets\images folder will automatically show up in the Image Manager.

Clicking on a folder node will show the images in that folder in the table view (clicking on the root folder “images” will show all images in the application, including images located in subfolders).

Clicking on an image node will also show the other images under the same folder in the table view and select it such that its properties can be changed in the right side properties view.

Clicking the chevron next to a folder will collapse/expand the folder.

Clicking the x button while hovering over a node lets you delete that item from disk.

Table View

The table view shows a list of the images located under the currently selected folder and contains different columns corresponding to different properties for an image. Clicking the header of a column sorts the list either ascending or descending.

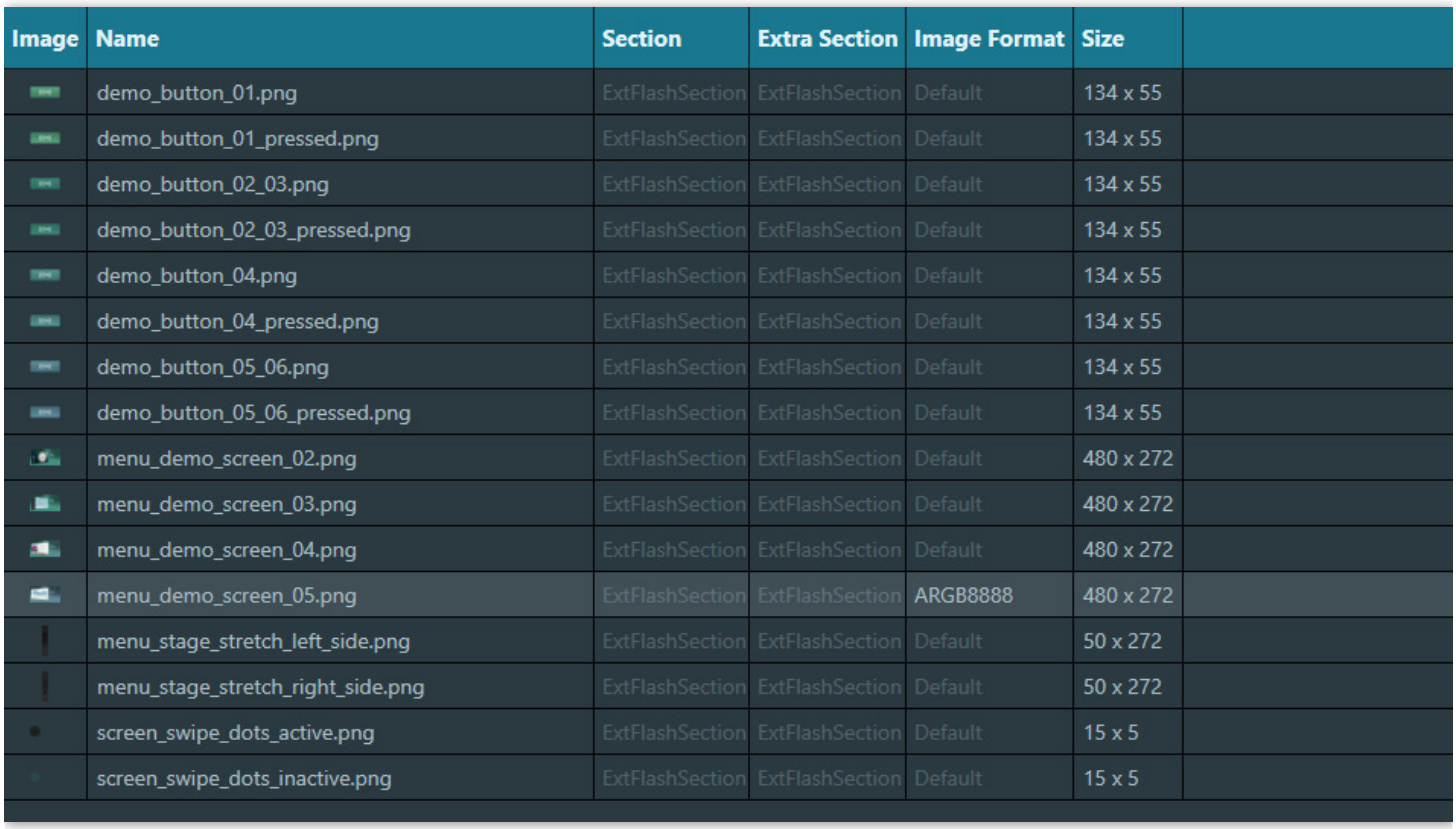


Image	Name	Section	Extra Section	Image Format	Size
	demo_button_01.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_01_pressed.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_02_03.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_02_03_pressed.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_04.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_04_pressed.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_05_06.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	demo_button_05_06_pressed.png	ExtFlashSection	ExtFlashSection	Default	134 x 55
	menu_demo_screen_02.png	ExtFlashSection	ExtFlashSection	Default	480 x 272
	menu_demo_screen_03.png	ExtFlashSection	ExtFlashSection	Default	480 x 272
	menu_demo_screen_04.png	ExtFlashSection	ExtFlashSection	Default	480 x 272
	menu_demo_screen_05.png	ExtFlashSection	ExtFlashSection	ARGB8888	480 x 272
	menu_stage_stretch_left_side.png	ExtFlashSection	ExtFlashSection	Default	50 x 272
	menu_stage_stretch_right_side.png	ExtFlashSection	ExtFlashSection	Default	50 x 272
	screen_swipe_dots_active.png	ExtFlashSection	ExtFlashSection	Default	15 x 5
	screen_swipe_dots_inactive.png	ExtFlashSection	ExtFlashSection	Default	15 x 5

The table view in Images

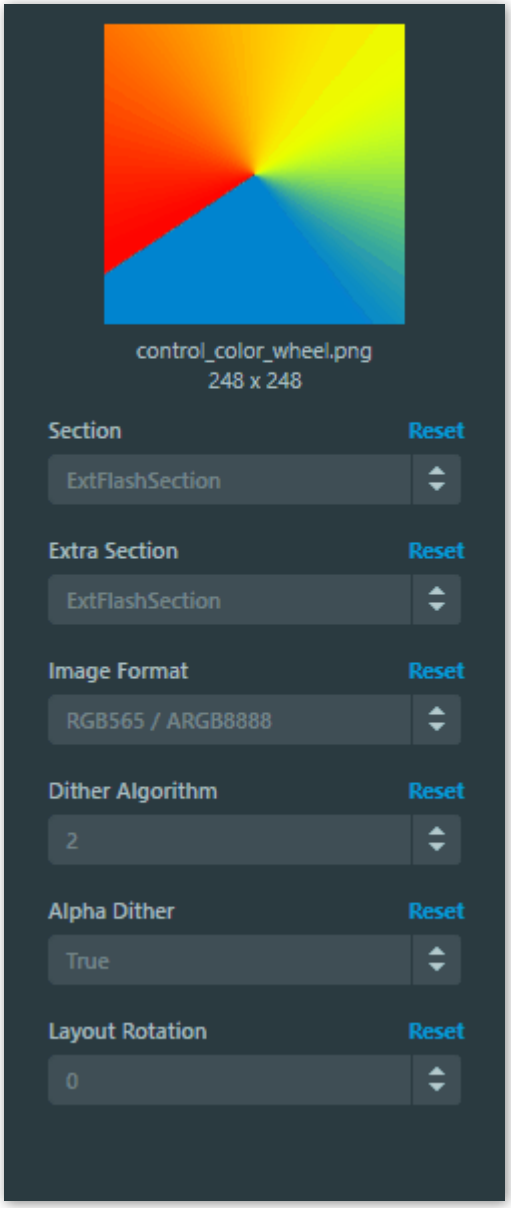
The Image column contains a preview of the image. Hovering over the image preview will show the preview in full size. Clicking the preview will open the image in the default application associated with .png images (for example paint.net).

Clicking a row will select the image such that its properties can be changed in the right side properties view.

When the value of a cell is grayed out, it means that the default value is used. When an explicit value has been set on an image, the cell will light up, as seen in the image above, where the image 'menu_demo_screen_05.png' has had its default Image Format value changed to ARGB8888.

Properties View

The properties view is used to change the properties of an image. It includes an image preview at the top, which, like the preview in the table view, will show a full size version if you hover over it, and will also open up the default application associated with .png files when clicked.



The properties view in Images

The different properties all have a default value. When no explicit value has been set on a property, the default value is shown in a grayed out manner in the selection boxes. When an explicit value is set, the value is shown in a lit up matter as shown below.

Pressing the Reset button will remove the current value and reset it back to its default value. When an explicit value has been set, changing the default value will not have an impact on that specific property.

Version: 4.16

Texts View

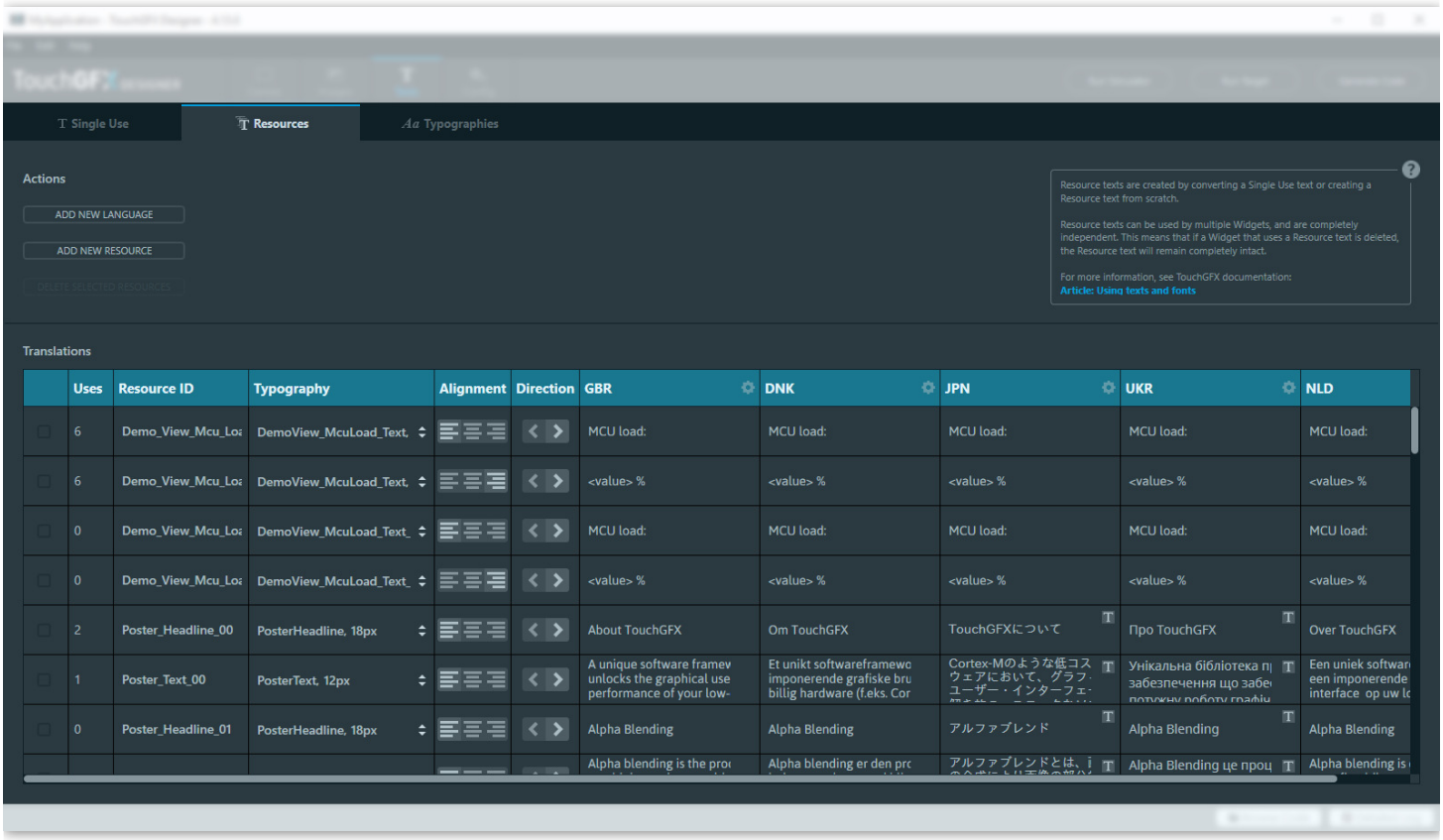
The Texts View in the TouchGFX Designer is used for configuring texts, translations and typographies in a project. The view consists of three tabs: Single Use, Resources and Typographies.

Single Use & Resources

The Single Use and Resouces tabs both contain an overview of texts, they are however different from each other.

Resource Text

Resource texts can be reused on any number of widgets and actions in the TouchGFX Designer. To add a new Resource texts, click the button labeled 'ADD NEW RESOURCE' in the Resources tab. One or more Resource texts can be deleted by setting a checkmark in the first column of the desired text rows and clicking the button labeled 'DELETE SELECTED RESOURCES'.

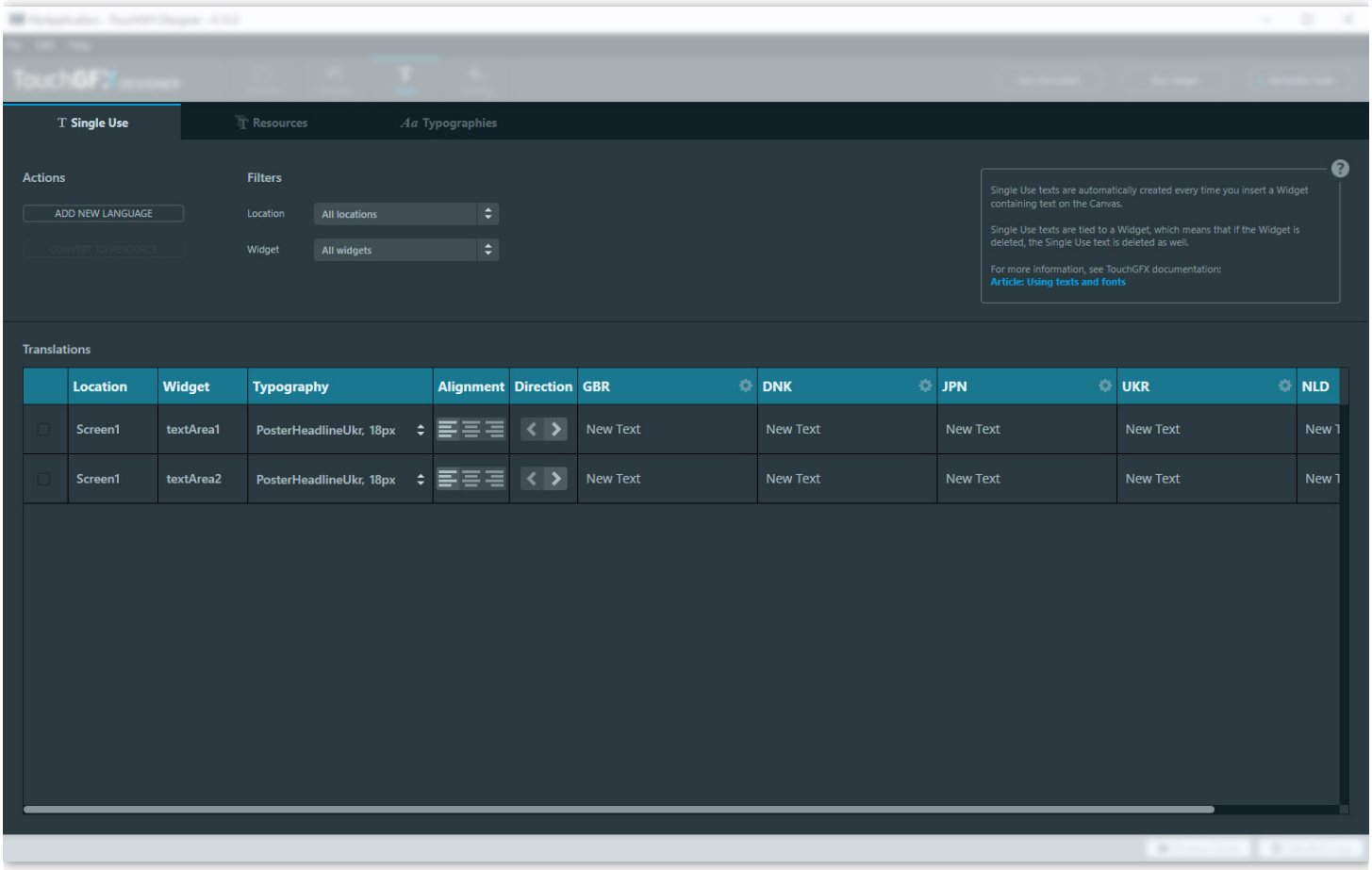


Resource texts in Texts View

Single Use Text

Single Use texts are only used once, and cannot be used by more than one widget or action at a time. They are added automatically when used on a widget or action, are deleted automatically when the widget or action is deleted or changed to use a Resource text instead.
One or more Single Use texts can be converted to a Resource text by setting a checkmark in the first column of the desired text rows and clicking the button labeled 'CONVERT TO RESOURCE'.

The Single Use text overview columns labeled 'Location' and 'Widget' show which Screen/Custom Container and Widget the widget is used on.



Single Use texts in Texts View

Translations

Typography:
Specifies which typography the text and all its translations should use as default. These can be added and configured in the Typographies tab

Alignment:
Specifies the horizontal alignment the text and all its translations should use as default. Possible values are Left, Right, and Center.

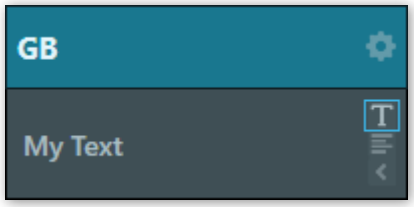
Direction:
Specifies which text direction the text and all its translations should use as default. Possible values are

LTR (Left-to-Right) and RTL (Right-to-Left), the default being LTR. The RTL option is primarily used for Arabic, Hebrew or other languages written from right to left.

Translations Specifics

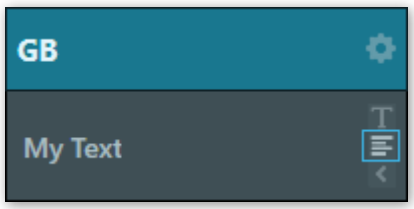
Each translation can overwrite the default Typography, Alignment and Direction, to reveal these controls, simply hover the mouse cursor over a translation.

Translation Specific Typography:
Setting a translation specific typography is easily done through the inline Typography selector, as shown in the figure below.



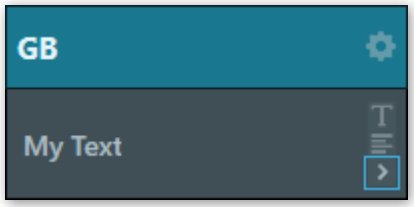
How to set Translation Specific Typography

Translation Specific Alignment:
Setting a translation specific alignment is easily done through the inline Alignment selector, as shown in the figure below.



How to set Translation Specific Alignment

Translation Specific Direction:
Setting a translation specific direction is easily done through the inline Direction selector, as shown in the figure below.

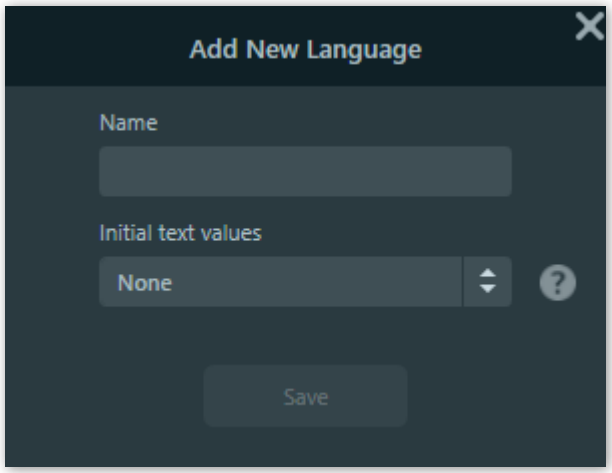


How to set Translation Specific Direction

Adding languages

To add a new language, simply press the button labeled 'ADD NEW LANGUAGE' and the popup in the figure below will appear, where the name of the language can be configured, and whether or not to

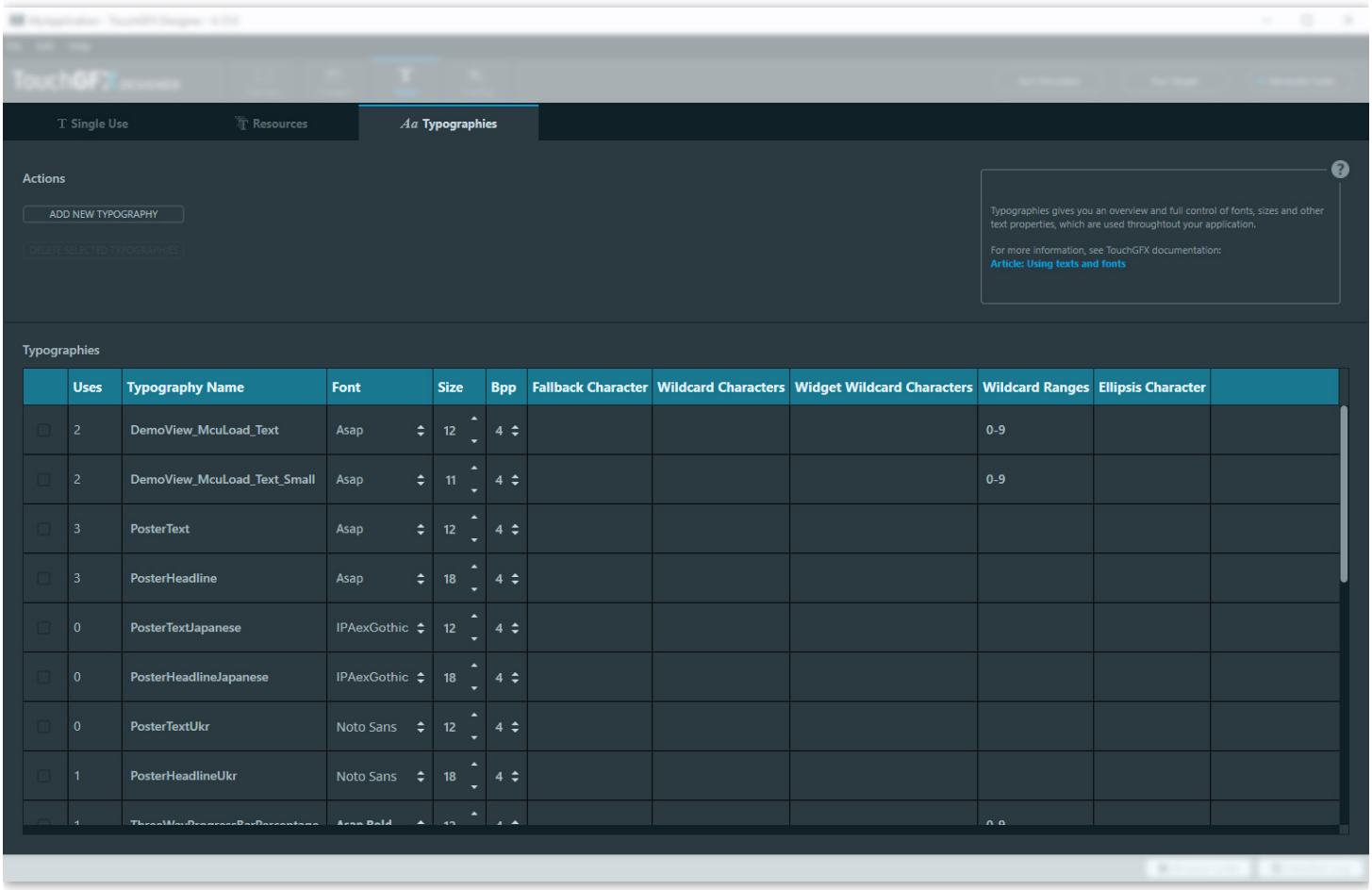
use the translations from another language.



Add New Language popup

Typographies

In the Typographies tab an overview of all typographies in a project can be found, as shown in the figure below.



Texts view in TouchGFX Designer

One or more Typographies can be deleted by setting a checkmark in the first column of the desired typography rows and clicking the button labeled 'DELETE SELECTED TYPOGRAPHIES'.

Uses:
The number of times the typography has been used in texts

Typography Name:
The name of the typography, which can be referenced in code.

Font:
The name of the font to use for the given typography.
You can choose between all installed fonts in Windows, or add your own fonts in the `assets/fonts` folder. *When adding fonts to this folder, the TouchGFX Designer needs to be restarted to load them.*

Size:
The font size of the typography.

Bpp:
Bits per pixel. The number of bits that are used per pixel to represent the font. Legal values are 1, 2, 4, 8.

Fallback Character:
If TouchGFX needs to render a character, but the glyph is unavailable, the character specified in this column is used. Value can be a single character, a unicode value *(in decimal or hexadecimal e.g. 0xABCD)*, the special keyword 'skip' or simply blank.

Wildcard Characters:
Characters that should be available to display in the TouchGFX application. This is recommended over using a dummy text. A dummy text will generate all glyphs, but also the actual string (e.g. "0123456789-"). Putting "0123456789-" in this column will generate the glyphs, but not a string.

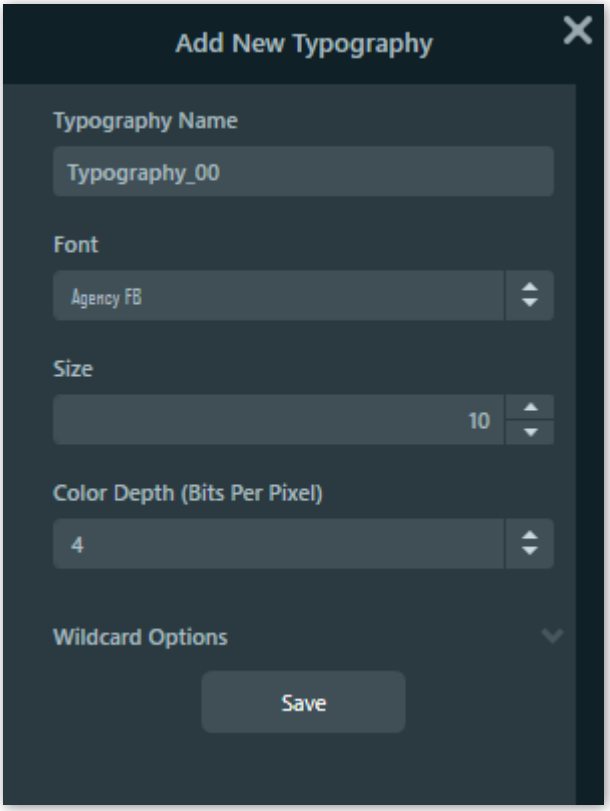
Widget Wildcard Characters:
These are characters that some widgets that require a wildcard will add, the Digital Clock widget will add "0123456789 :APM" to this field.

Wildcard Ranges:
This is similar to Wildcard Characters, but ranges can easily be specified, e.g. "0-9,A-F" will be the same as putting "0123456789ABCDEF" in the Wildcard Characters column. Ranges can also be specified as numbers, so for example "0-9" can also be specified as "48-57" or "0x30-0x39". Please note that the quotes should not be entered.

Ellipsis Character:
This character is used to truncate long text in text areas.

Add New Typography

To add a new typography, simply press the button labeled 'ADD NEW TYPOGRAPHY' and the popup in the figure below will appear, where the name of the typography, font, size and color depth can be configured.

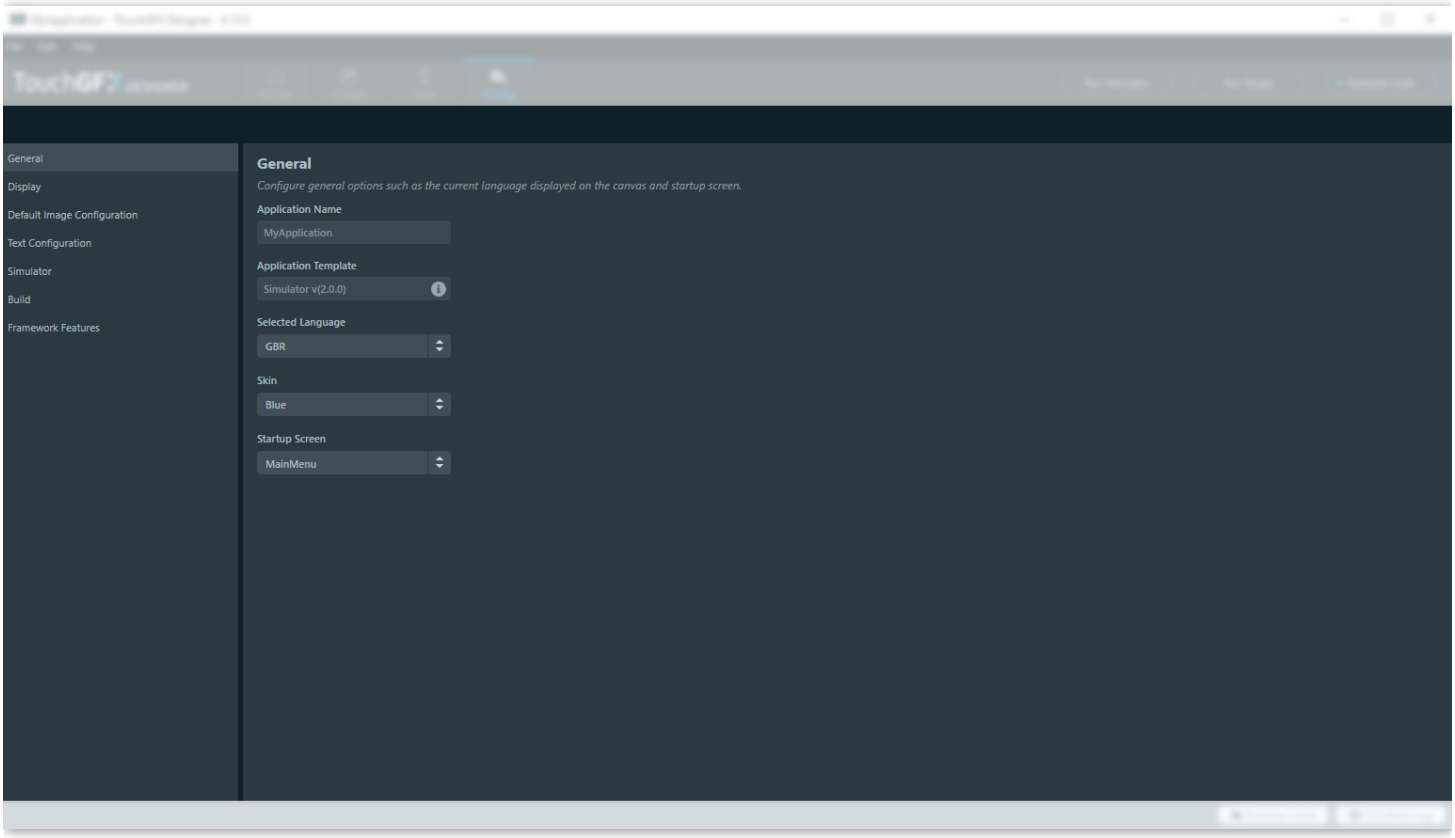


Add New Typography popup

Version: 4.16

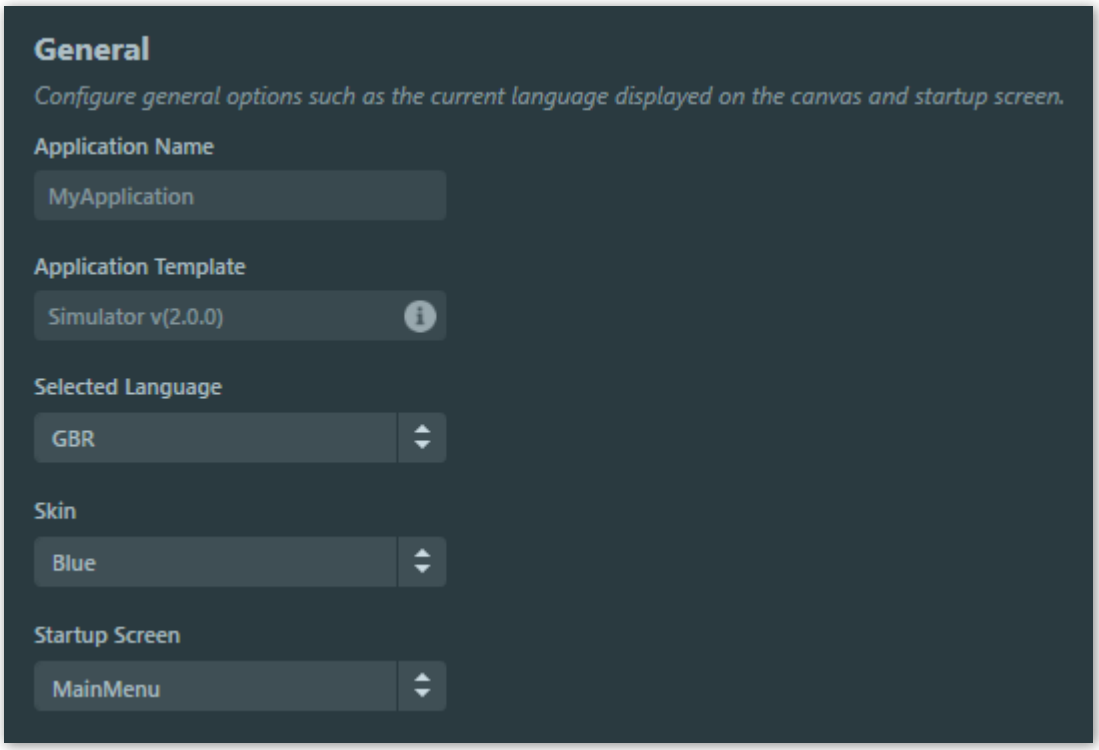
Config View

In the Config View various settings affecting the project can be configured.



General settings in Config View

General



General settings in Config View

Application Name
Application name is a readonly field, displaying the name that was chosen for the application when it was created.

Application Template
This field shows the application template the application was created with, if this information is not available 'N/A' will be displayed.
If this information is available, the name of the application template will be displayed along with the version. There will also be an icon with an 'i' *(see image above)*, clicking this will display more information about the Application Template.

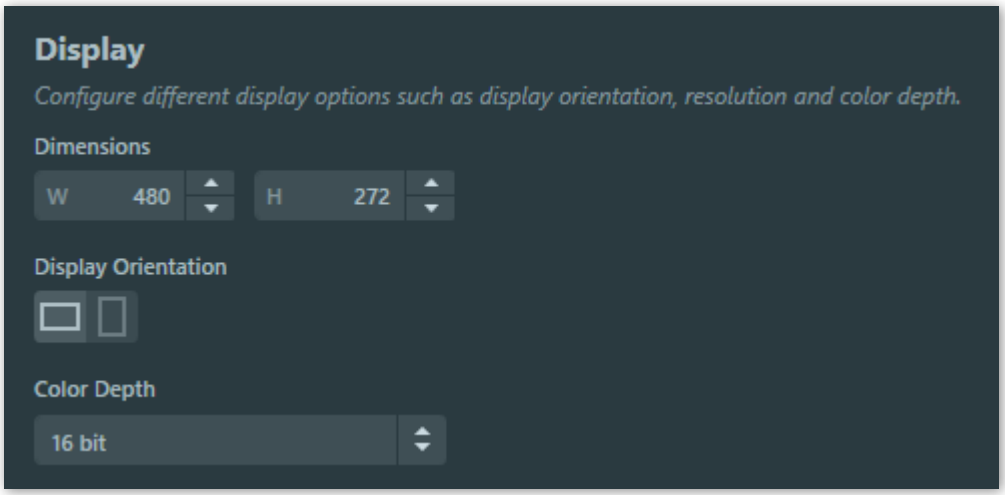
Selected Language
This dropdown contains the languages configured in the Texts view, and selects which language should be displayed at startup of the project.

Skin
This dropdown sets which of the two built-in styles to use for widgets that support setting a style, options are 'Blue' or 'Dark'. *If a widget has been configured with a style from the 'Blue' skin, and the skin is changed to 'Dark', the widget will automatically switch to a corresponding style in the 'Dark' skin.*

Startup Screen This dropdown contains all the screens that have been added to the project, and selects which screen to display at startup of the project.

Display

In this section the settings for the Display can be configured.



Display settings in Config View

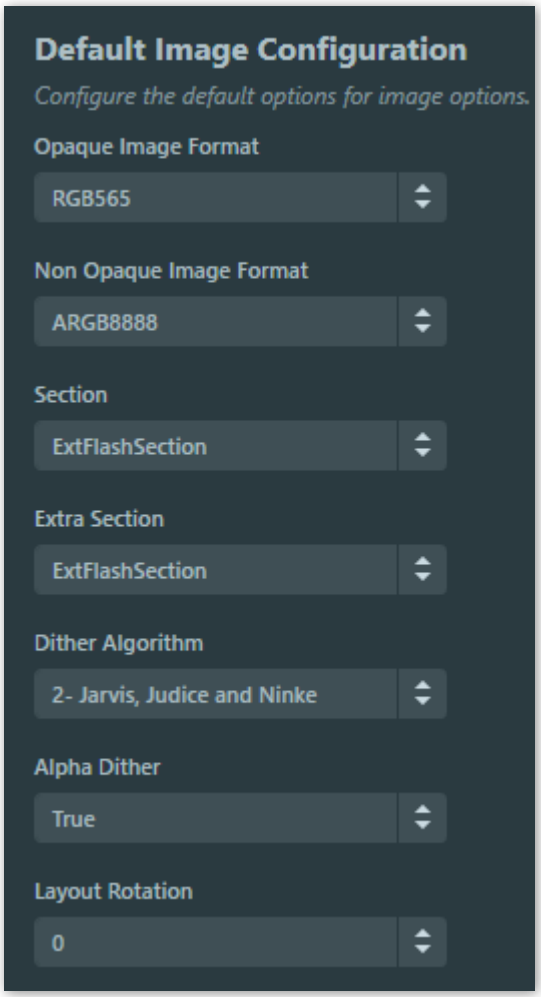
Dimensions
The size of the display can be set through the *W(width)* and *H(height)* properties, however if the size has been configured by the Application Template, configuration of the size will be disabled.

Display Orientation
The orientation of the display can toggled between landscape and portrait, this will also affect how images used in a project are converted to cpp files.

Color Depth
This dropdown contains the color depths that are available to a project. These are determined by the Application Template used to create a project.

Default Image Configuration

In this section the default configuration used for images in a project can be set. These settings will affect all images in the project, unless they are overwrite in the Images view



Default Image Configuration settings in Config View

Opaque Image Format

This dropdown sets which format images that have only opaque pixel data should be generated with. The available image formats in this dropdown depend on the selected color depth of the project.

Non Opaque Image Format

This dropdown sets which format images that have non-opaque pixel data should be generated with. The available image formats in this dropdown depend on the selected color depth of the project.

Section

This dropdown sets the location where image data should be stored on the target hardware. The available sections in this dropdown depend on the Application Template that the project was created with.

Extra Section

When using L8 image formats you can choose to store the color table in a different section using this dropdown. The available sections in this dropdown depend on the Application Template that the project was created with.

Dither Algorithm

This dropdown sets the dithering algorithm used for images.

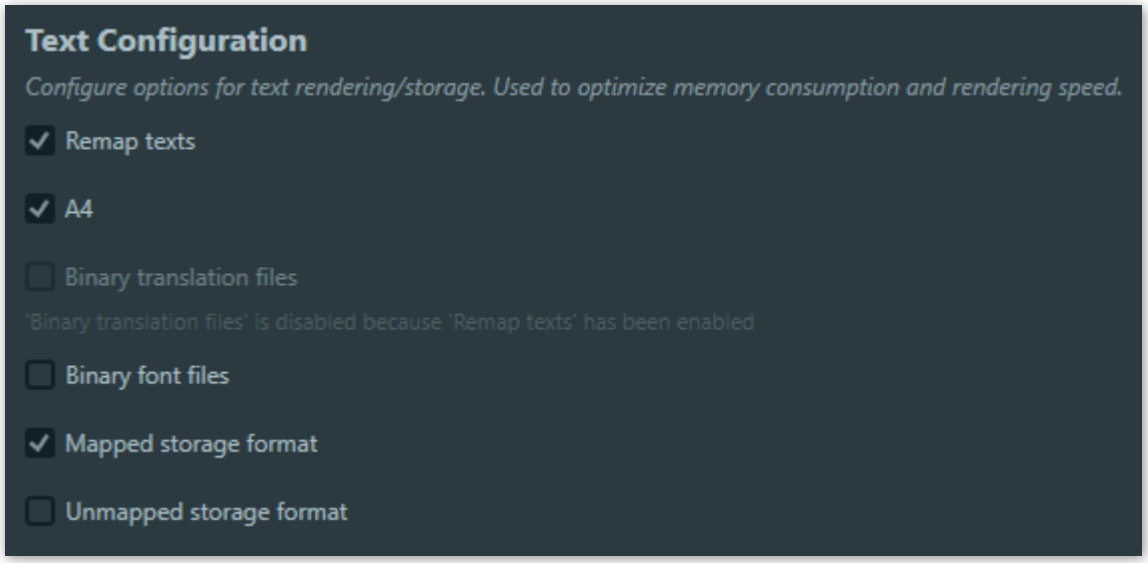
- No dither: *no dithering is applied to the image. This is the setting with the highest performance since no alteration is made. However, depending on the image, the quality of the photo may also degrade visually at lower color depths.*
- Floyd-Steinberg: *diffuses the error to neighboring pixels, resulting in fine-grained dithering but sacrificing sharpness.*
- Jarvis, Judice and Ninke: *diffuses the error to pixels one step further away compared to Floyd-Steinberg, resulting in coarser dithering but a sharper image. The slowest of the 3 error-diffusion dithering algorithms.*
- Stucki: *based on minimized average error dithering but faster and cleaner.*

Alpha Dither
This dropdown sets whether or not to use the dither algorithm through the alpha channel.

Layout Rotaion
This dropdown specifies the rotation of the image data when generated. If the screen orientation is changed, use this to correctly render images in the new orientation.

Text Configuration

In this section options for text rendering and storage can be configured by checking the boxes that fit your project's needs.



Text Configuration settings in Config View

Remap texts
This option defines whether or not translations that are identical should be remapped. Remapping texts will combine identical translations and suffixes across all languages, typographies and alignments, resulting in a reduced footprint.
This option is mutually exclusive with the option 'Binary translation files'

A4
This option defines whether or not the horizontal pixel data of glyphs byte aligned into an A4 format.
This only affects typographies that are configured as 4bpp

Binary translation files
This option defines whether or not the translations in a project should be moved into binary files that can be loaded at runtime.
This option is mutually exclusive with the option 'Remap texts'

Binary font files
This option defines whether or not the font files in a project should be moved into binary files that can be loaded at runtime.

Mapped storage format
This option defines if the font files in a project should be output in mapped storage format.

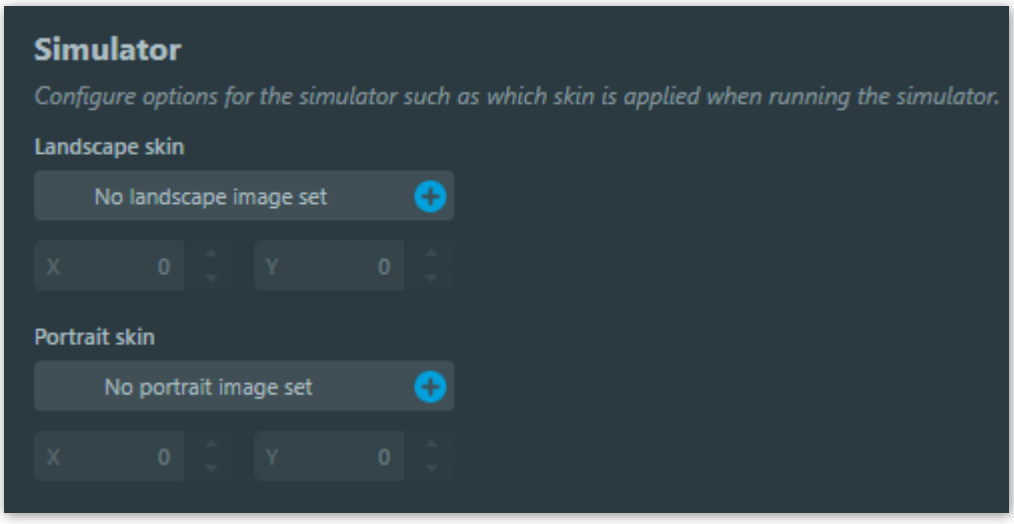
Unmapped storage format
This option defines if the font files in a project should be output in unmapped storage format.

FURTHER READING

Text and Fonts

Simulator

In this section, a skin can be added to the Simulator when running it. A skin can be added for both landscape and portrait orientated projects. The *X* and *Y* properties determine the position of the simulator on the skin.



Simulator settings in Config View

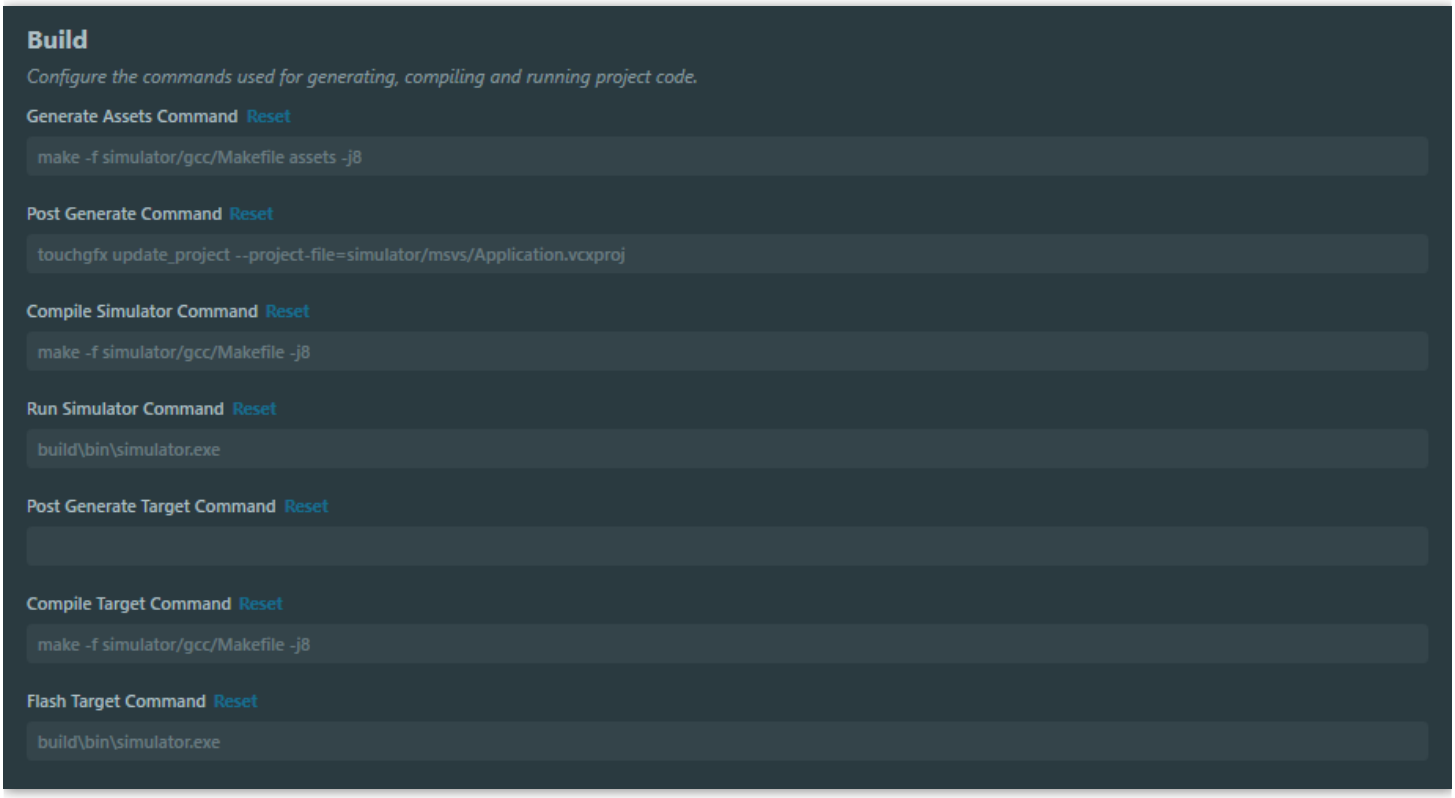
Below is a demonstration of a simulator running with a skin.
When a simulator runs with a skin, the standard window controls are hidden. To close the simulator press **ESC**

Simulator running with a skin.

Build

In this section the commands that are executed when pressing 'Run Simulator', 'Run Target' and 'Generate Code', can be overwritten.

To overwrite a command, simply write any command in the text boxes. To reset a command, if it has been overwritten, press the blue 'Reset' label next to the name of the Command.



Build settings in Config View

Generate Assets Command

This command is usually set up to generate text and image assets, and is executed after the TouchGFX Designer has generated the code.

Post Generate Command

This command is usually used to update various project files. The built-in `touchgfx update_project` commandline tool supports updating the following project files:

- Visual Studio (.vcxproj)
- Keil (.uvprojx)
- IAR (.ewp & .ipcf)
- CubeIDE (.project & .cproject)
- CubeMX (.ioc)

However, any command that needs to be executed after code generation can be written here.

Compile Simulator Command

This command executes the compilation of a project for the simulator, usually by executing the Makefile generated by the TouchGFX Designer.

Run Simulator Command

This command executes the startup of the simulator.exe.

Post Generate Target Command
This command is usually used to update various project files mostly CubeMX (.ioc) project files.

However, any command that needs to be executed after code generation can be written here.

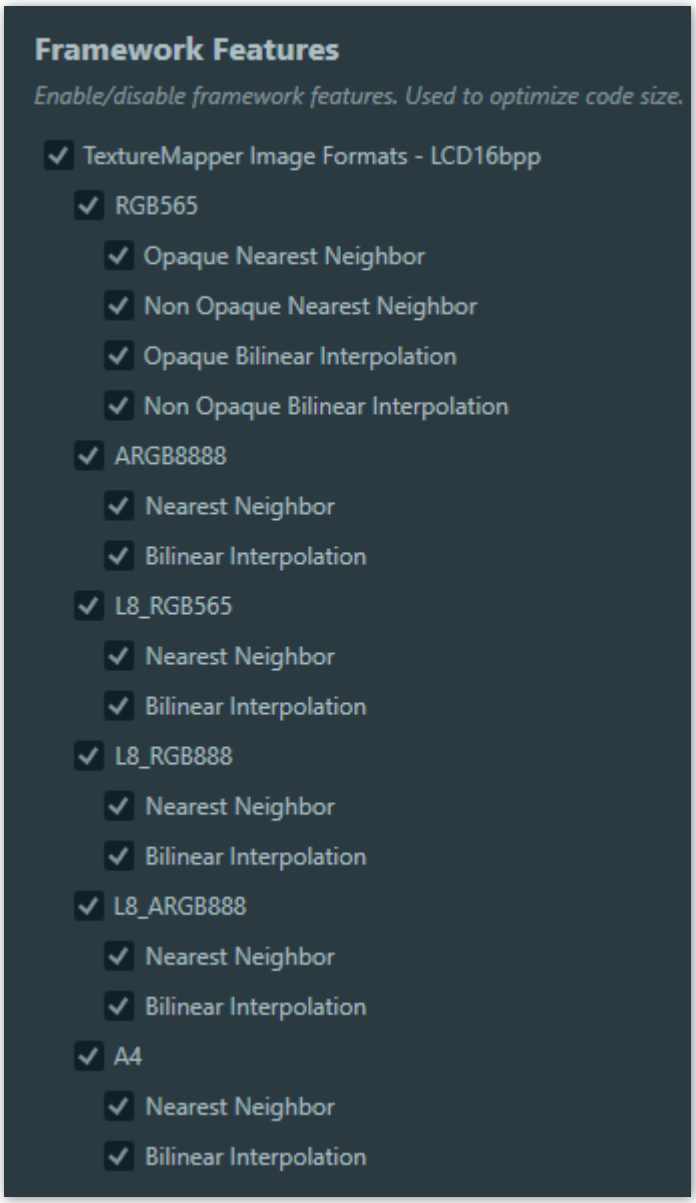
Compile Target Command
This command executes the compilation of a project for the target hardware.

Flash Target Command
This command executes the flashing of a project to the target hardware.

Framework Features

In this section features in the framework, specifically which image formats the TextureMapper widget supports, can be enabled/disabled. This can be used for optimizing the code size a project takes up on hardware.

The available options displayed in this section depend upon which color depth has been chosen for the project. In the image below the image formats for a 16 bit color depth TextureMapper is shown.



Framework Features settings in Config View

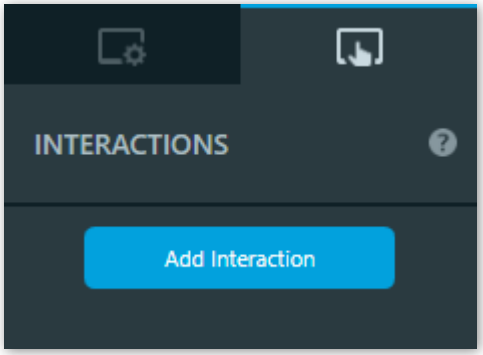
Version: 4.16

Interactions

Interactions allow you to configure an action to take place when a trigger happens. An interaction in TouchGFX Designer is built up of a **trigger** and an **action**:

- A *Trigger* is what will start the interaction - what needs to happen in our application for the Action to take place.
- An *Action* is what will happen after a Trigger has been emitted. This is where you can decide what happens in your application, when your defined trigger conditions have been met.

To add an interaction, go to the Interactions tab for any Screen or Custom Container and press the blue button labelled "Add Interaction" as shown in the image below.



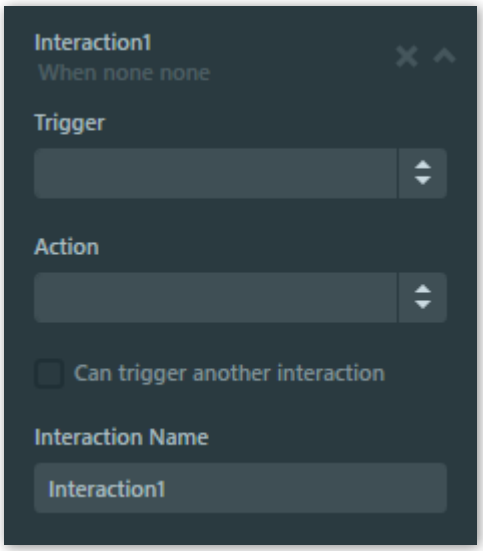
Interactions tab in TouchGFX Designer

After pressing the "Add Interaction" button, the interaction in the image below will be added. The top of the Interaction view consists of the name given to the Interaction, a dynamic description, a button with a cross icon, and a button with a chevron icon.

The dynamic description is modified based on the Trigger and Action that have been selected. Since no Trigger and Action have been selected in the image below, the dynamic description is "When none none".

The button with a cross icon will delete the Interaction with a confirmation popup.

The button with a chevron icon will collapse the Interaction view, making the overview of interactions more manageable.



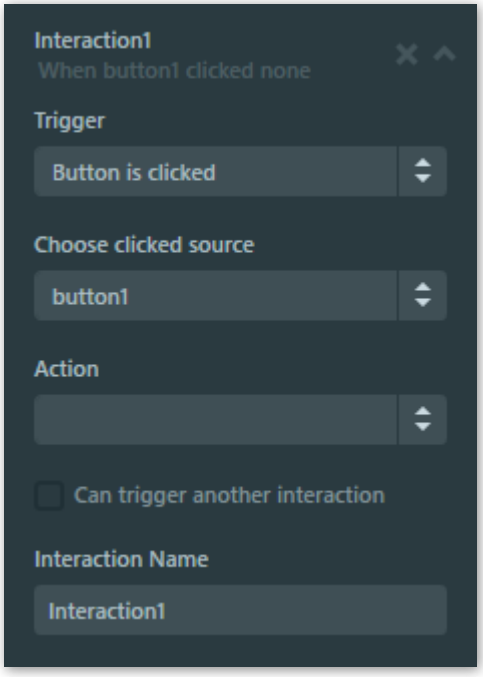
New Interaction in the Interactions tab

Triggers

The Trigger dropdown is populated based on what widgets have been added to a Screen or Custom Container.
An empty Screen will only have two Triggers available: **Hardware button is clicked** and **Screen is entered**.

Adding a widget will add the Triggers associated with it. For example, adding a Button widget will add the **Button is clicked** Trigger to the Screen or Custom Container.

Some Triggers, like **Button is clicked**, require a component to be selected as shown in the image below. However, if there is only one widget matching the Trigger, that widget will be auto-selected.



Button is Clicked Trigger selected on Interaction

As can be seen in the image above, after selecting the Trigger, the dynamic description has been updated from "When none none" to "When button1 clicked none".

Actions

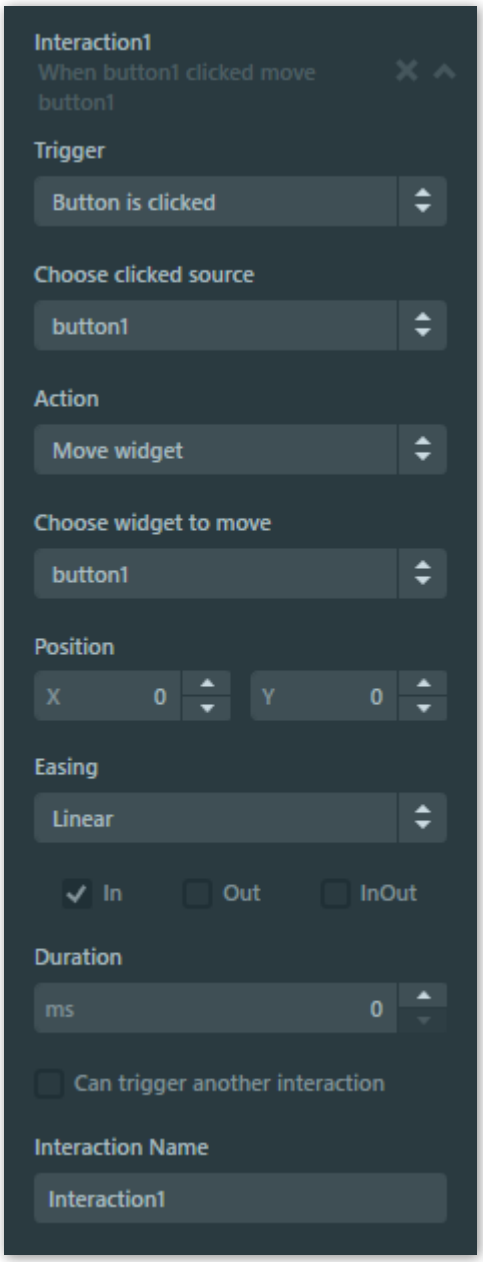
The Action dropdown is populated based on what widgets have been added to a Screen or Custom Container.
An empty Screen will only have four Actions available:

- **Call new virtual function**
- **Change screen**
- **Execute C++ code**
- **Wait for**

Adding a widget will add the Actions associated with it. Adding a Button widget will add the following Actions:

- **Move widget**
- **Fade widget**
- **Hide widget**
- **Show widget**

Some Actions, like **Move widget**, require a component to be selected as shown in the image below. However, if there is only one widget matching the Action, that widget will be auto-selected. Selecting the action **Move widget** also adds more properties relevant to moving a widget.



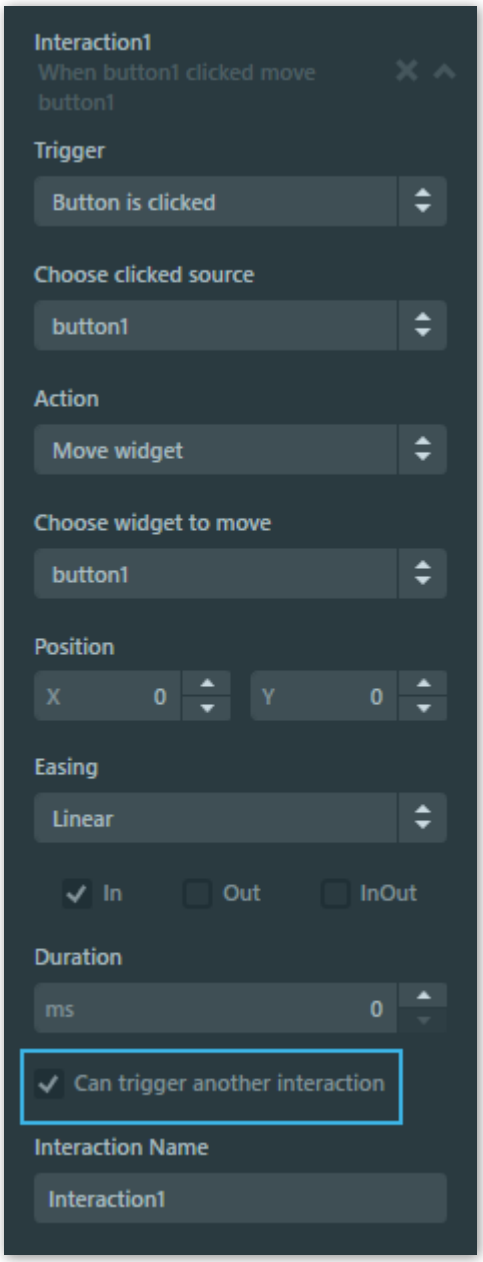
Move widget action selected on Interaction

As can be seen above, after selecting the Action, the dynamic description has been updated from "When button1 clicked none" to "When button1 clicked move button1".

Chaining Interactions

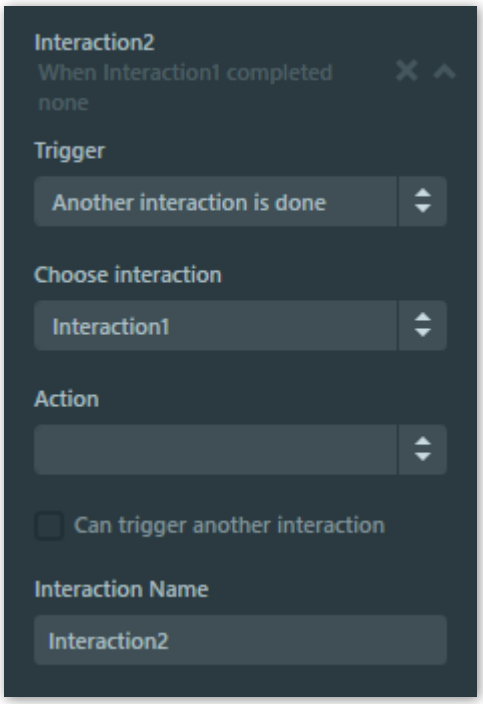
Interactions can also be triggered by another Interaction upon completing its Action.

To enable this behaviour, the checkbox labeled *Can trigger another interaction* needs to be enabled as shown in the image below:



Can trigger another interaction enabled on Interaction

After enabling this behaviour, other Interactions can trigger on the Interaction, as shown in the image below:



Interaction triggering on another Interaction

Version: 4.16

Custom Triggers and Actions

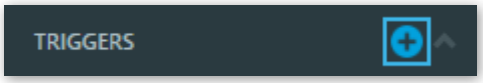
With TouchGFX Designer it is possible to define your own interaction components with Custom Triggers and Actions. Each Screen in your application can contain a collection of actions (these are simply void methods in C++) that you can call from within the TouchGFX Designer as well as in code, while custom containers can also have a collection of triggers (which is equal to a callback in C++) which your application can react to. In this article, we will go through this functionality to learn the possibilities this gives us to create more clean and dynamic TouchGFX applications.

Custom Triggers

Custom Containers have the ability to create Custom Triggers. These are generated as C++ callbacks and can be emitted or reacted to from the Interaction system, or from User Code.

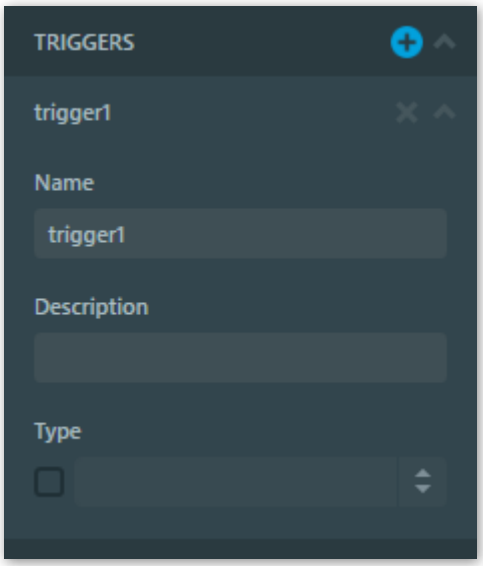
Adding Custom Triggers

Adding a Custom Trigger is done via the properties tab of a Custom Container, by clicking the blue button with a plus icon in the **TRIGGERS** section highlighted in the image below.



Adding a Custom Trigger

When the Custom Trigger has been created further options become available as shown in the image below.



Editing a Custom Trigger

Name

The name specified here will be used for reference within the Interaction system and in the generated code files, as shown in the code examples below:

CustomContainer1Base.hpp

```
class CustomContainer1Base : public touchgfx::Container
{
public:
    CustomContainer1Base();
    virtual ~CustomContainer1Base() {}
    virtual void initialize();

    /*
     * Custom Trigger Callback Setters
     */
    void setTrigger1Callback(touchgfx::GenericCallback<>& callback)
    {
        this->trigger1Callback = &callback;
    }

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }

    /*
     * Custom Trigger Emitters
     */
    virtual void emitTrigger1Callback()
    {
        if (trigger1Callback && trigger1Callback->isValid())
        {
            this->trigger1Callback->execute();
        }
    }

private:

    /*
     * Custom Trigger Callback Declarations
     */
    touchgfx::GenericCallback<>* trigger1Callback;

};
```

CustomContainer1Base.cpp

```
CustomContainer1Base::CustomContainer1Base() :
    trigger1Callback(0)
```

```
{
    setWidth(250);
    setHeight(250);
}

void CustomContainer1Base::initialize()
{

}
```

Description

The text written here, will be used in the Interaction system and can be seen when hovering over the Trigger when selecting it in the Interaction system. If a description has not been specified a standard description will be created as shown in the Emitting Custom Triggers from Interactions section.

Type

Enabling Type will allow for creating triggers that return a value given a specific type. The type can either be selected from a list, or by writing the type manually. Shown below is the code generated when selecting `bool`.

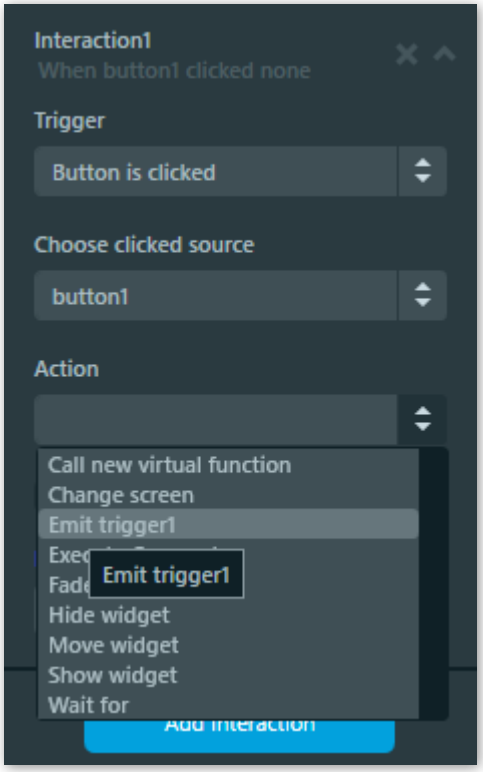
CustomContainer1Base.hpp

```
class CustomContainer1Base : public touchgfx::Container
{
public:
    CustomContainer1Base();
    virtual ~CustomContainer1Base() {}
    virtual void initialize();

    /*
     * Custom Trigger Callback Setters
     */
    void setTrigger1Callback(touchgfx::GenericCallback<bool>& callback)
    {
        this->trigger1Callback = &callback;
    }

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }

    /*
     * Custom Trigger Emitters
     */
    virtual void emitTrigger1Callback(bool value)
    {
        if (trigger1Callback && trigger1Callback->isValid())
        {
```

```
            this->trigger1Callback->execute(value);
        }
    }

private:

    /*
     * Custom Trigger Callback Declarations
     */
    touchgfx::GenericCallback<bool>* trigger1Callback;

};
```

Emitting Custom Triggers from Interactions

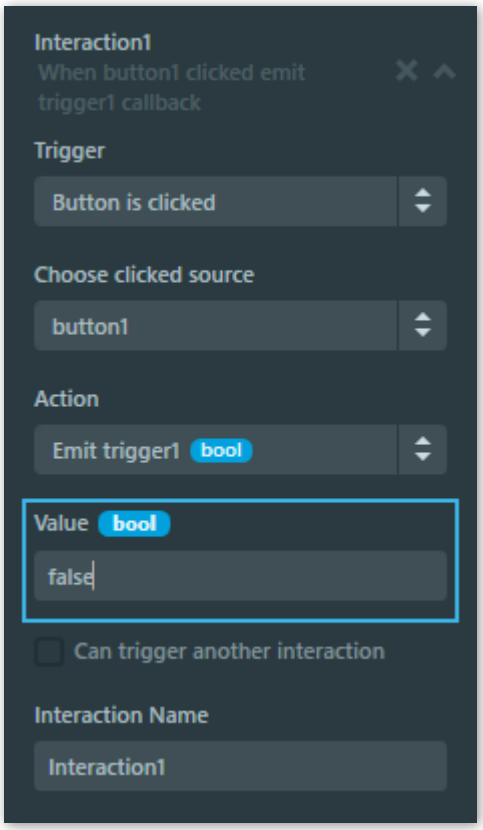
A Custom Trigger can be emitted by using the Interaction system. Simply go to the Interaction tab of the Custom Container that owns the Custom Trigger, create a new Interaction and select the Custom Trigger as the action as shown in the image below.



Emitting a Custom Trigger from an Interaction

As shown in the above picture, whenever the button, that has been added to the Custom Container, is clicked the Custom Trigger is emitted.

If the Custom Trigger has type enabled, the parameter value or variable must be specified as shown in the image below, where `bool` has been specified as the Type.



Specifying a parameter on a Custom Trigger emitted from an Interaction

Emitting Custom Triggers from User Code

Custom Triggers can also be emitted from the User Code class file that inherits from the generated Custom Container. In the generated Custom Container the method below is generated, for a CustomTrigger named "trigger1".

CustomContainer1Base.hpp

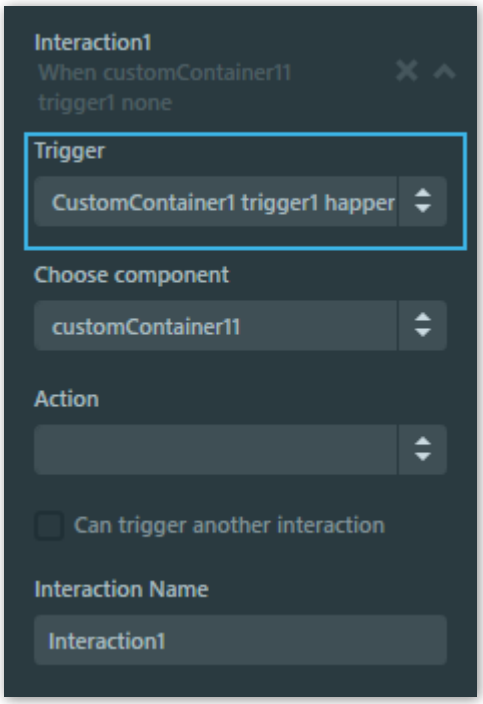
```
virtual void emitTrigger1Callback(bool value)
{
    if (trigger1Callback && trigger1Callback->isValid())
    {
        this->trigger1Callback->execute(value);
    }
}
```

This function can be overwritten or called in the User Code class file that inherits from it.

Reacting to Custom Triggers from Interactions

If a Custom Container with a Custom Trigger is added to a Screen, the Custom Trigger can be used as Trigger on an Interaction on the Screen, as shown in the Image below.

The naming scheme of the Custom Trigger when selecting it as Trigger in an Interaction is: `<Custom Container Name> <Custom Trigger name> happens`.



Selecting Custom Trigger as Trigger on an Interaction

After selecting the Trigger, if there are multiple instances of the same Custom Container containing the Custom Trigger, the component needs to be selected. However if there is only one instance, it will be auto selected.

Reacting to Custom Triggers from User Code

A Custom Trigger can also be reacted to from User Code by implementing the callback as shown in the following code example, where a Custom Container with a Custom Trigger name "trigger1" has been added to a Screen. In the User Code class file that inherits from the Screen, the following highlighted lines have been added to implement the Callback/Custom Trigger.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();

private:
    /*
     * Callback Declarations
     */
    touchgfx::Callback<Screen1View> customContainer1Trigger1Callback;
```

```
    /*
     * Callback Handler Declarations
     */
    void customContainer1Trigger1CallbackHandler();
};
```

Screen1View.cpp

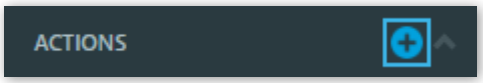
```
#include <gui/screen1_screen/Screen1View.hpp>

Screen1View::Screen1View():
    customContainer1Trigger1Callback(this, &Screen1View::customContainer1Trigger1CallbackH
{
    customContainer1.setTrigger1Callback(customContainer1Trigger1Callback);
}

void Screen1View::setupScreen()
{
    Screen1ViewBase::setupScreen();
}

void Screen1View::tearDownScreen()
{
    Screen1ViewBase::tearDownScreen();
}

void Screen1View::customContainer1Trigger1CallbackHandler()
{
    //Your code here.
}
```

Custom Actions

Screens and Custom Containers have the ability to create Custom Actions. These are generated as virtual C++ methods, and can be executed from the Interaction system, or from User Code. The implementation and behaviour of a Custom Action, can either be configured within the Interaction system, or by overwriting the generated C++ methods in the User Code class file.

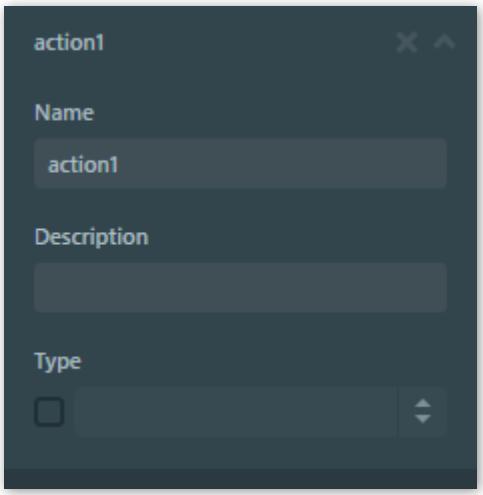
Adding Custom Actions

Adding a Custom Trigger is done via the properties tab of a Screen or Custom Container, by clicking the blue button with a plus icon in the **ACTIONS** section highlighted in the image below.



Adding a Custom Action

When the Custom Action has been created further options become available as shown in the image below.



Editing a Custom Action

Name

The name specified here will be used for further reference within the Interaction system and in the generated code files, as shown in the code examples below.

CustomContainer1Base.hpp

```
class CustomContainer1Base : public touchgfx::Container
{
public:
    CustomContainer1Base();
    virtual ~CustomContainer1Base() {}
    virtual void initialize();

    /*
     * Custom Actions
     */
    virtual void action1();

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }

private:

};
```

CustomContainer1Base.cpp

```
CustomContainer1Base::CustomContainer1Base()
{
    setWidth(250);
    setHeight(250);
}

void CustomContainer1Base::initialize()
{

}

void CustomContainer1Base::action1()
{

}
```

Description

The text written here, will be used in the Interaction system and can be seen when hovering over the Action when selecting it in the Interaction system. If a description has not been specified a standard description will be created as shown in the Calling Custom Action from Interactions section, that follows this scheme: `Call <Name> on <Screen or Custom Container Name>`.

Type

Enabling Type will allow for creating actions that require a parameter given a specific type. The type can either be selected from a list, or by writing the type manually. Shown below is the code generated when selecting `bool`.

CustomContainer1Base.hpp

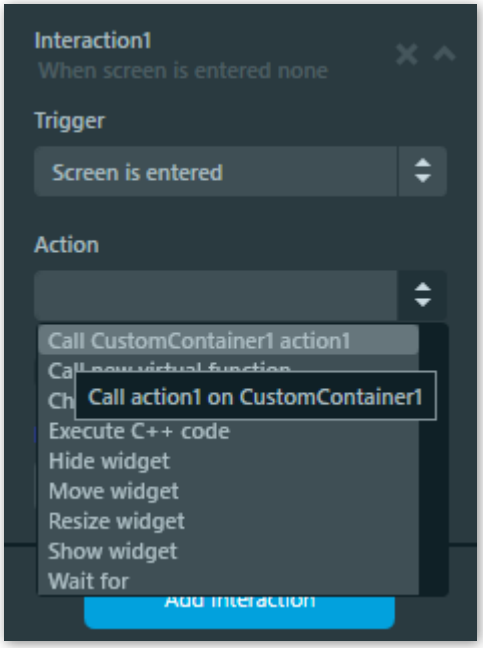
```
/*
 * Custom Actions
 */
virtual void action1(bool value);
```

CustomContainer1Base.cpp

```
void CustomContainer1Base::action1(bool value)
{

}
```

Calling Custom Action from Interactions

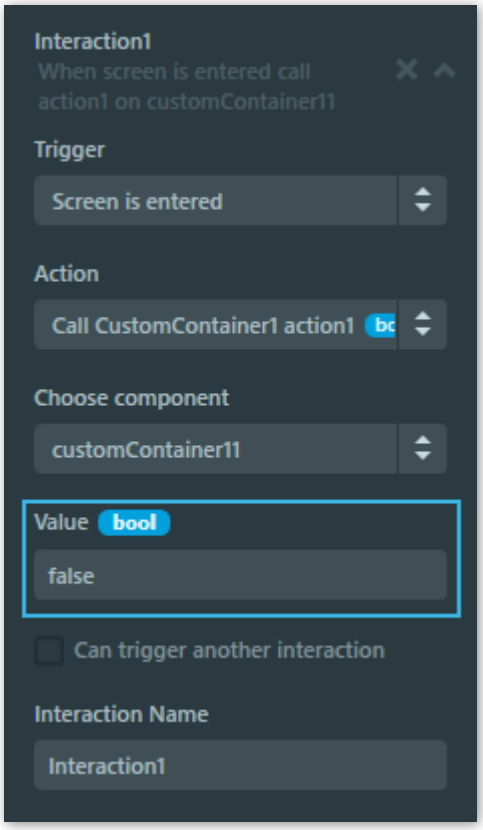
When a Custom Action has been created the action can be executed from within the Interaction system. If the Action is created on a Custom Container and the Custom Container is added to a Screen, the Action can be executed from the screens Interaction tab as shown in the image below.



Executing a Custom Action from an Interaction

After selecting the action, if there are multiple instances of the same Custom Container, the component needs to be selected. However if there is only one instance, it will be auto selected.

If the Custom Action has type enabled, the parameter value or variable must be specified as shown in the image below, where `bool` has been specified as the Type.



Specifying a parameter on a Custom Action executed from an Interaction

Calling Custom Action from User Code

Custom Actions can also be called directly from User Code. In the following code example a Custom Container with a Custom Action named "action1" is created. The Custom Container has been added to a Screen resulting in the generated code below.

Screen1ViewBase.cpp

```
Screen1ViewBase::Screen1ViewBase()
{

    customContainer11.setXY(50, 11);

    add(customContainer11);
}

void Screen1ViewBase::setupScreen()
{
    customContainer11.initialize();
}
```

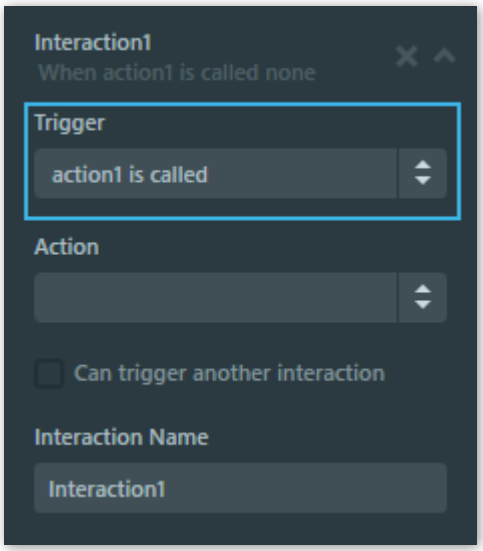
In the User Code class file `Screen1View` that inherits from `Screen1ViewBase` the Custom Action "action1" can be called as shown below.

Screen1View.hpp

```
Screen1View::Screen1View()
{
    customContainer11.action1();
}

void Screen1View::setupScreen()
{
    Screen1ViewBase::setupScreen();
}

void Screen1View::tearDownScreen()
{
    Screen1ViewBase::tearDownScreen();
}
```

Adding Behaviour to Custom Actions from Interactions

Behaviour can be added to Custom Actions by using the Interaction system. Simply go to the Interaction tab of the Screen or Custom Container that owns the Custom Action, create a new Interaction and select the Custom Action as the trigger as shown in the image below.



Selecting a Custom Action as Trigger on an Interaction

Then any Action that is defined in the Interaction, will be executed whenever the Custom Action is called.

Adding Behaviour to Custom Actions from User Code

Custom Actions can also implement its behaviour by overwriting the Action in the User Code class `CustomContainer1` that inherits from `CustomContainer1Base` as shown below.

CustomContainer1.hpp

```
class CustomContainer1 : public CustomContainer1Base
{
public:
    CustomContainer1();
    virtual ~CustomContainer1() {}

    virtual void initialize();

    void action1();

protected:
};
```

CustomContainer1.cpp

```
CustomContainer1::CustomContainer1()
{

}

void CustomContainer1::initialize()
{
    CustomContainer1Base::initialize();
```

```
}

void CustomContainer1::action1()
{
    //Your code here
}
```

FURTHER READING

- **Tutorial 5: Creating Custom Triggers and Actions**

Version: 4.16

Custom Containers

When creating applications you might need a widget that is not found in the standard widget set included in TouchGFX.

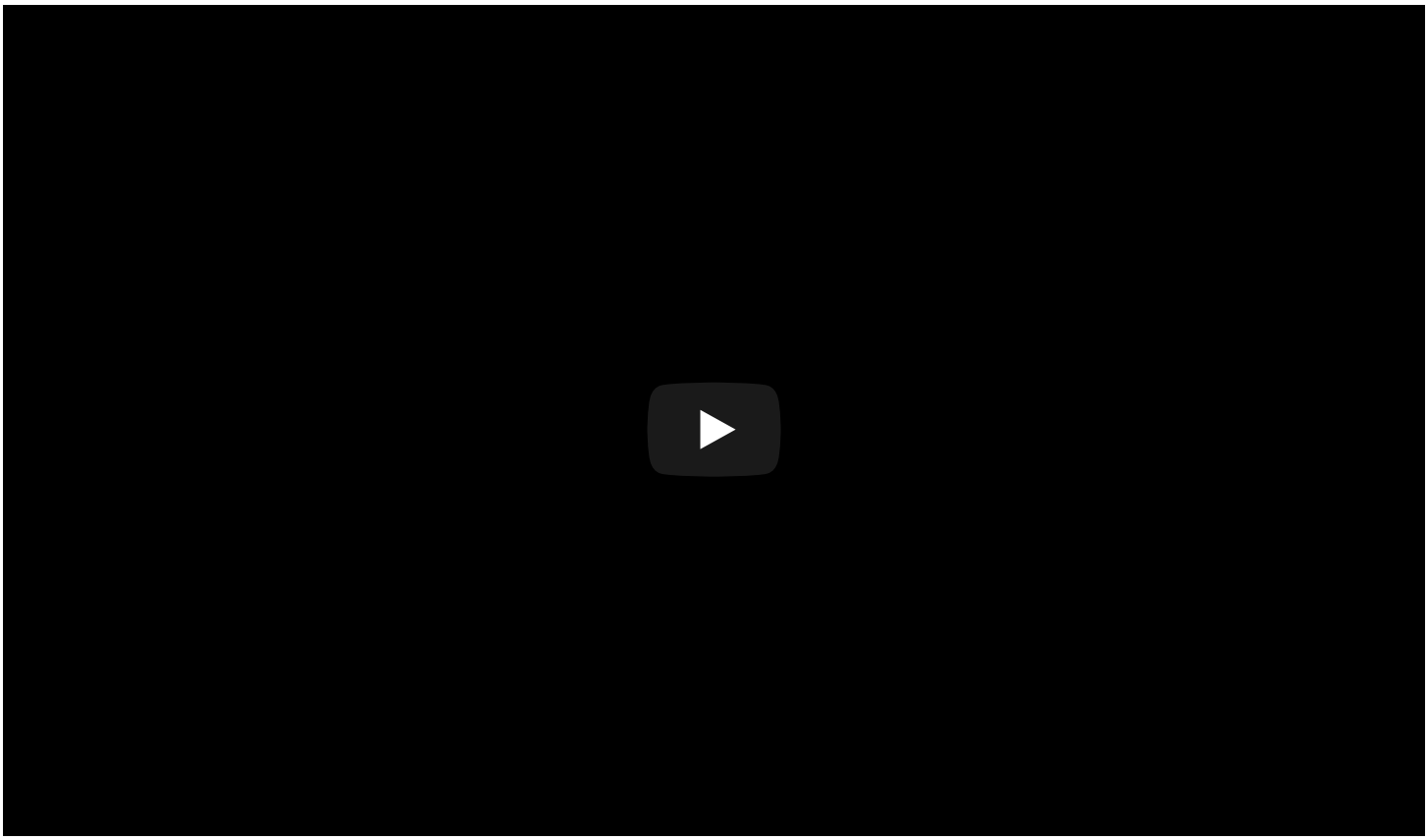
One way of creating your own widgets is using custom containers. A custom container is an object that contains other existing widgets and combines the visual appearance and behaviours of these widgets. It is not dissimilar to the classic composite design pattern and we also refer to the contained widgets as the children of the container.

Drawing performance of custom containers will in general be very high. It will utilize the underlying drawing mechanisms of TouchGFX and will determine which parts of a container and the children needs to be redrawn automatically. This means that you do not need to worry about drawing performance when using containers.

However, there can be times where you need to reduce the footprint of a widget and in these scenarios, the more advanced approach called Custom Widget might be preferable.

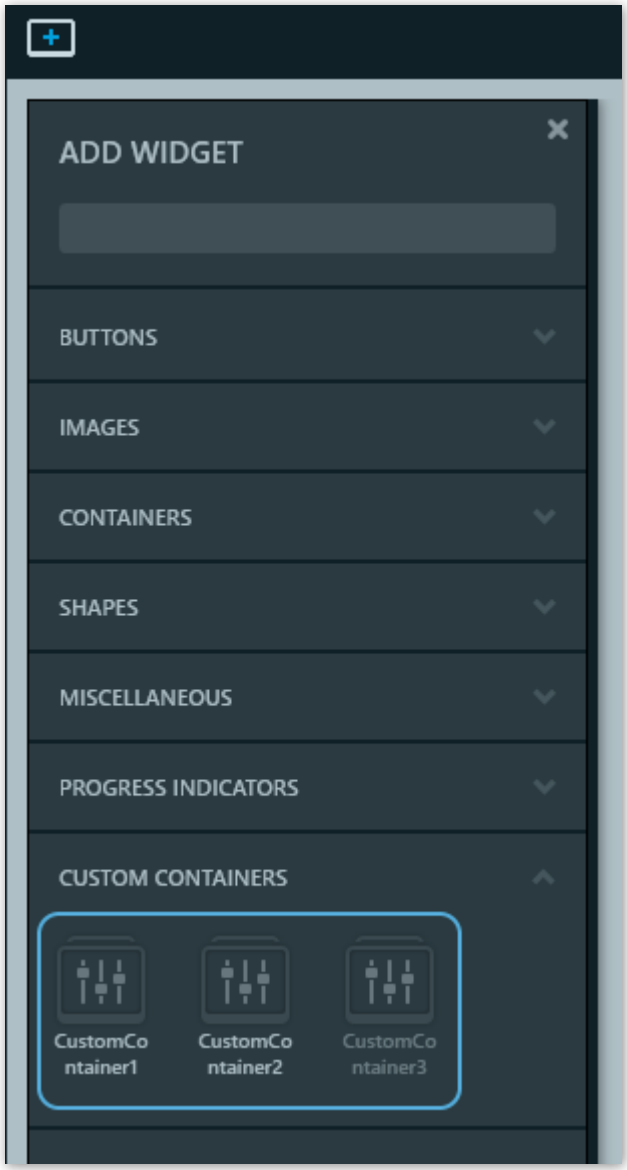
In TouchGFX Designer

If you want to create and use custom containers in TouchGFX Designer, we give a general introduction to how you can use them in your projects in the video below:



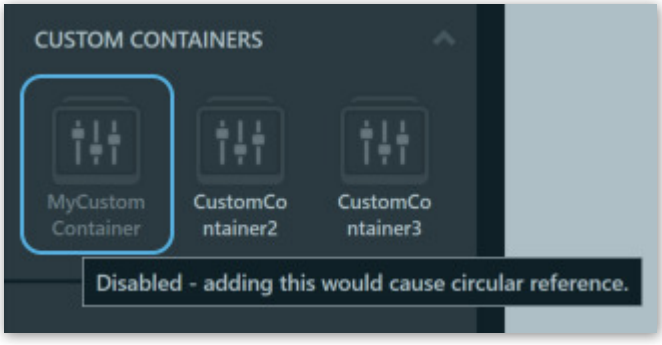
Composite Custom Containers

It is possible to create a custom container that is composed of other custom containers. This can be a powerful way to construct components made up of smaller components. You can do this by adding instances of an already defined custom container found in the Widgets tab:



Inserting instances of custom containers

Note that TouchGFX Designer will help you to avoid inserting instances that would result in a circular reference such as adding a custom container instance to the definition of itself:



Potential circular reference disables the ability to insert an instance

Custom Triggers and Actions

One of the powerful aspects to a custom container is the ability to define custom triggers (callbacks) and custom actions (methods). This means that you can define integral behaviour to your custom container so it becomes more than just a reusable collection of widgets and enables communication with the rest of your application.

> **FURTHER READING**
>
> Read more about this functionality in the **Custom Triggers and Actions** section.

In Code

In this section we will create a custom container in code. The steps are as follows:

- Create a class that extends the touchgfx::Container class
- Declaring all children of the container as member variables
- Setting the width and height of the container
- Setting up each of the children
- Adding each of the children to the hierarchy, in the right order
- Implementing the desired behaviour via methods and callbacks

We will start from scratch and build upon the code until we end up with a simple fully functional custom container.

Create a class that extends the touchgfx::Container class

Start by creating a `MyCustomContainer.hpp` header file with the code below. Use C++ inheritance to gain access to the methods and members of `touchgfx::Container` (remember to include the header file for `Container.hpp`):

MyCustomContainer.hpp

```
#include <gui/common/FrontendApplication.hpp>
#include <touchgfx/containers/Container.hpp>

class MyCustomContainer : public touchgfx::Container
{
public:
```

```
    MyCustomContainer();
    virtual ~MyCustomContainer() {}
    virtual void initialize();

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }

private:

};
```

Declaring all children of the container as member variables

Declare which widgets are going to compose your custom container in the header file. In this example we will just add a box `myBox` and a button `myButton`.

MyCustomContainer.hpp

```
#include <gui/common/FrontendApplication.hpp>
#include <touchgfx/containers/Container.hpp>

class MyCustomContainer : public touchgfx::Container
{
public:
    MyCustomContainer();
    virtual ~MyCustomContainer() {}
    virtual void initialize();

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }

    /*
     * Member Declarations
     */
    touchgfx::Box myBox;
    touchgfx::Button myButton;

private:

};
```

Setting the width and height of the container

Create a cpp file `MyCustomContainer.cpp` which includes the header file we just created. Use the `setWidth()` and `setHeight()` methods in the constructor to set whatever size you want for the custom container:

MyCustomContainer.cpp

```
#include <gui/include/containers/MyCustomContainer.hpp>

MyCustomContainer::MyCustomContainer()
{
    setWidth(250);
    setHeight(250);
}

void MyCustomContainer::initialize()
{

}
```

Setting up each of the children

Now we need to set up the properties for each widget in the constructor:

MyCustomContainer.cpp

```
#include <gui/include/containers/MyCustomContainer.hpp>

MyCustomContainer::MyCustomContainer()
{
    setWidth(250);
    setHeight(250);

    myBox.setPosition(0, 0, 250, 250);
    myBox.setColor(touchgfx::Color::getColorFrom24BitRGB(255, 255, 255));

    myButton.setXY(40, 95);
    myButton.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID), touchgf
}

void MyCustomContainer::initialize()
{

}
```

Adding each of the children to the hierarchy, in the right order

Use the `add()` method in the constructor to add the widgets as children of the custom container:

MyCustomContainer.cpp

```
#include <gui/containers/MyCustomContainer.hpp>

MyCustomContainer::MyCustomContainer()
{
    setWidth(250);
    setHeight(250);

    myBox.setPosition(0, 0, 250, 250);
    myBox.setColor(touchgfx::Color::getColorFrom24BitRGB(255, 255, 255));

    myButton.setXY(40, 95);
    myButton.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID), touchg

    add(myBox);
    add(myButton);
}

void MyCustomContainer::initialize()
{

}
```

Implementing the desired behaviour via methods and callbacks

To add some behaviour to our custom container, we can define some methods and callbacks in `MyCustomContainer.hpp`. In this example we define a method `doSomething()` whose sole purpose is to emit the callback `somethingHappened`:

MyCustomContainer.hpp

```
#include <gui/common/FrontendApplication.hpp>
#include <touchgfx/containers/Container.hpp>

class MyCustomContainer : public touchgfx::Container
{
public:
    MyCustomContainerBase();
    virtual ~MyCustomContainerBase() {}
    virtual void initialize();

    /*
     * Callback Setters
     */
```

```
    void setSomethingHappenedCallback(touchgfx::GenericCallback<>& callback)
    {
        somethingHappenedCallback = &callback;
    }

    /*
     * Methods
     */
    virtual void doSomething();

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
    }

    /*
     * Callback Emitters
     */
    virtual void emitSomethingHappenedCallback()
    {
        if (somethingHappenedCallback && somethingHappenedCallback->isValid())
        {
            somethingHappenedCallback->execute();
        }
    }

    /*
     * Member Declarations
     */
    touchgfx::Box myBox;
    touchgfx::Button myButton;

private:

    /*
     * Callback Declarations
     */
    touchgfx::GenericCallback<>* somethingHappenedCallback;

};
```

Then to add the behaviour to our method and callback, implement them in the `MyCustomContainer.cpp` file. For this simple surface level example, we will simply emit the `somethingHappened` callback, but you can customize this as you want:

MyCustomContainer.cpp

```
#include <gui/containers/MyCustomContainer.hpp>

MyCustomContainer::MyCustomContainer()
{
```

```
    setWidth(250);
    setHeight(250);

    myBox.setPosition(0, 0, 250, 250);
    myBox.setColor(touchgfx::Color::getColorFrom24BitRGB(255, 255, 255));

    myButton.setXY(40, 95);
    myButton.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID), touchgf

    add(myBox);
    add(myButton);
}

void MyCustomContainer::initialize()
{

}

void MyCustomContainer::doSomething()
{
    MyCustomContainer::emitSomethingHappenedCallback();
}
```

FURTHER READING

For a more detailed example on how to create and use a custom container, refer to **Tutorial 4: Creating a Scroll Wheel with custom behavior**.

Version: 4.16

Caching Bitmaps

In this section we will discuss the bitmap cache in TouchGFX. The bitmap cache is a dedicated RAM buffer where bitmaps can be stored (or cached) by the application. If a bitmap is cached, TouchGFX will automatically use the RAM cache as pixel source when drawing the bitmap.

Bitmap caching can be beneficial in many cases. Reading data from RAM is often faster than reading from flash (especially when using the Texturemapper because it uses non-linear memory access), so caching to RAM can increase the performance of your UI. Be aware that caching from internal flash to external RAM can reduce performance. Caching to RAM also allows you to use the flash for other purposes like log files while showing your UI, because bitmaps will be read from RAM (in some cases writing to a flash requires it to be non-memory mapped). It can also be useful when you need to modify the pixels of a bitmap and therefore need the bitmap to be in modifiable memory.

For performance reasons, TouchGFX requires all graphics data stored in external flash to be directly accessible (through a pointer), without going through a driver layer. This means that TouchGFX cannot render directly from a non-memory mapped flash (like an SD-card). To overcome this limitation the bitmap cache provides a mechanism for caching some or all of the bitmap data in RAM during power-up. Bitmap caching is therefore useful when you need to store your bitmaps on slow external storage like a USB-disk or SD-card.

Setup the Bitmap Cache

In order to use the bitmap caching feature, you need to first provide a bitmap cache configuration to TouchGFX, and secondly (in some cases) to provide a hardware specific implementation of the `BlockCopy` function for reading data from your external storage.

Bitmap Cache Configuration

The bitmap cache configuration consists of a pointer to a buffer and the size of the buffer. These two values must be provided to TouchGFX in the call to touchgfx_generic_init or Bitmap::registerBitmapDatabase. This call is normally found in the BoardConfiguration.cpp file:

BoardConfiguration.cpp (extract)

```
// Place cache start address in SDRAM at address 0xC0008000;
uint16_t* cacheStartAddr = (uint16_t*)0xC0008000;
uint32_t cacheSize = 0x300000; //3 MB, as example
HAL& hal = touchgfx_generic_init<STM32F4HAL>(dma, display, tc, DISPLAY_WIDTH, DISPLAY_HEI
```

In the above example a 3 MB buffer in external memory is passed to TouchGFX as bitmap cache. The address is selected by the application programmer. In the next example we just declare an array and just pass the address and size of the array. The specific location of the array will depend on your linker script. This method is most often used when creating a (small) bitmap cache in internal RAM:

BoardConfiguration.cpp (extract)

```
// Define an array for the bitmap cache
uint16_t cache[128*1024]; //256 KB cache
HAL& hal = touchgfx_generic_init<STM32F4HAL>(dma, display, tc, DISPLAY_WIDTH, DISPLAY_HEI
```

Enabling Bitmap cache with TouchGFX Generator

If you are using CubeMX and TouchGFX Generator, enabling and configuring of the bitmap cache should be done in TouchGFXHAL.cpp. First the default created Bitmap cache database needs to be removed, hereafter a new cache is set based on the memory area provided.

TouchGFXHAL.cpp (extract)

```
void TouchGFXHAL::initialize()
{
    /* Initialize TouchGFX Engine */
    TouchGFXGeneratedHAL::initialize();

    uint16_t* cacheStartAddr = (uint16_t*)0xC0008000;
    uint32_t cacheSize = 0x300000; //3 MB, as example

    touchgfx::Bitmap::removeCache();
    touchgfx::Bitmap::setCache(cache, sizeof(cache), 0);
}
```

If you need to cache all your bitmaps, of course the size of the cache must be large enough to contain all your bitmap data. Note: There is a small amount of memory used for bookkeeping (8 bytes x number of bitmaps in the application), so you must allocate slightly more memory than actually needed for the raw pixel data. This amount depends on the number of bitmaps in your application, but a few kilobytes of additional memory is usually enough.

BlockCopy Copies Data from Flash to the Cache

When you cache a bitmap, TouchGFX copies the pixels from the original location to the bitmap cache using the `BlockCopy` function in the HAL class.

If your bitmaps are stored in normal addressable flash (like internal flash or a memory mapped external flash like a QSPI-flash), you do not need to do anything. The built-in implementation works

fine.

On the other hand, if your bitmaps are stored in flash that is not addressable, e.g. a filesystem or non-memory mapped flash, then the standard copy method is not sufficient and you need to provide an updated version that is able to read from your specific flash storage.

Read more about this topic Using Non-Memory Mapped flash for storing images section.

Cache Operations

The bitmap caching operations are all placed in the `Bitmap` class:

`Bitmap` caching method	Description
`bool Bitmap::cache(BitmapId id)`	This method caches a bitmap. The bitmap is only cached if enough unused memory is available in the cache. Returns true if the bitmap was cached. Caching an already cached bitmap does not do any work.
`bool Bitmap::cacheReplaceBitmap(BitmapId out, BitmapId in)`	This method replaces a bitmap (out) in the cache with another bitmap (in). The method will only succeed if the bitmap to be replaced is already cached and if the bitmaps have the same size (in bytes).
`bool Bitmap::cacheRemoveBitmap(BitmapId id)`	This method removes a bitmap from the cache. The memory used by the bitmap can be used for caching of another bitmap afterwards.
`void Bitmap::clearCache()`	This method removes all the cached bitmaps from the cache.
`void Bitmap::cacheAll()`	This method caches all bitmaps. It can not be used if the amount of RAM allocated for the cache (or available) is less than the total size of the bitmaps.

Cache Strategies

When the amount of RAM that you can allocate for your bitmap cache is less than the total size of the bitmaps you can not cache all the bitmaps during startup. You can e.g. select to cache only the bitmaps needed for the first screen. When you change between your screens you can remove some or all of the cached bitmaps and cache the bitmaps needed for the next screen. This is exemplified in the next section.

Cache Bitmap on a Screen Basis

Your application user interface is composed of a set of Views. The Views probably all use some bitmaps. A simple strategy for caching is to cache all the bitmaps used by a View in the `View::setupScreen` method and clear the cache in the

`View::tearDownScreen` method:

Screen1View.cpp (extract)

```
void Screen1View::setupScreen()
{
    //ensure background is cached
    Bitmap::cache(BITMAP_SCREEN2_ID);
    //cache some icons
    Bitmap::cache(BITMAP_ICON10_ID);
    Bitmap::cache(BITMAP_ICON11_ID);
    Bitmap::cache(BITMAP_ICON12_ID);
}

void Screen1View::tearDownScreen()
{
    //Remove all bitmaps from the cache
    Bitmap::clearCache();
}
```

The memory requirement for the cache is the size of the bitmaps used by the screen with the biggest use of bitmaps. The drawback of this method is that if two `Views` both use a bitmap, the bitmap will be erased from the cache on exit from the first `View` and cached again on entry to the second `View`.

The Bitmap::cacheRemoveBitmap can be used to selective uncache bitmaps and thus reduce this overhead. The drawback of the cacheRemoveBitmap is that the cache memory will be fragmented.

Another general drawback of caching is that when you change your UI (e.g. adding a button) you may need to update the caching code to include the new bitmap.

Replace the Background Bitmaps in the Cache

If your application has a set of minor bitmaps (e.g. icons) and some large full screen "background" bitmaps another strategy can be advised:

Cache all the small bitmaps prior to entering the first screen. A good place to do this is in the FrontendApplication constructor. Also cache the background bitmap for the first screen:

```
FrontendApplication::FrontendApplication(Model& m, FrontendHeap& heap)
    : touchgfx::MVPApplication(),
```

```
      transitionCallback(),
      frontendHeap(heap),
      model(m)
{
    //cache some icons
    Bitmap::cache(BITMAP_ICON10_ID);
    Bitmap::cache(BITMAP_ICON11_ID);
    Bitmap::cache(BITMAP_ICON12_ID);

    //cache first background
    Bitmap::cache(BITMAP_SCREEN1_ID);
    backgroundBitmapCached = BITMAP_SCREEN1_ID; //remember ID in a variable
}
```

In the `View::setupScreen` method replace the cached background bitmap with the required bitmap:

```
Screen1View::setupScreen()
{
    //ensure background is cached
    Bitmap::cacheReplaceBitmap(backgroundBitmapCached, BITMAP_SCREEN1_ID);
    backgroundBitmapCached = BITMAP_SCREEN1_ID; //remember new ID of cached bitmap
}
```

```
void Screen1View::tearDownScreen()
{
    //nothing cache related
}
```

The memory requirement for the cache using this strategy is the size of the cached bitmaps and one background bitmap. Compared to the previous method the code is simpler to maintain as the views have less code. The performance is better as we move less bitmaps in and out of the cache.

The cacheReplaceBitmap operation is preferable to the cacheRemoveBitmap method as it does not fragment the memory.

Cache Memory Management

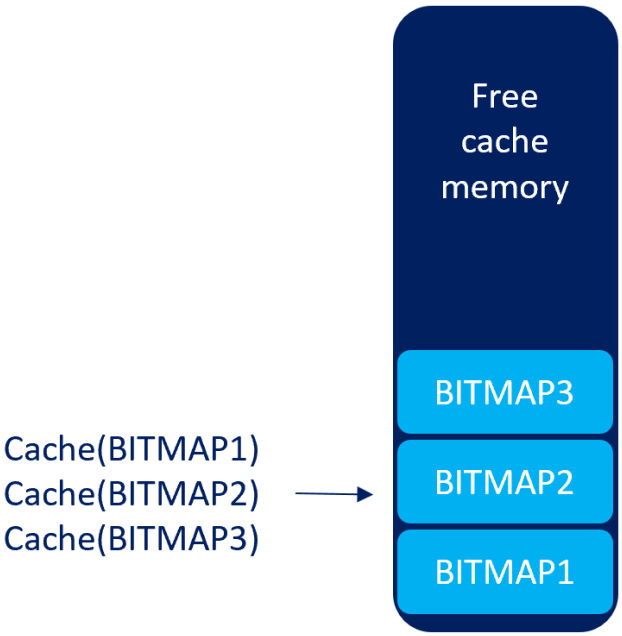
In order to get the full effect of the bitmap caching it is necessary to understand the internal operations of the cache.

The cache is implemented as a stack. New bitmaps are cached after the previously cached bitmaps. Memory used by a bitmap is marked as "free" when the bitmap is removed from the cache, but the memory is not immediately useable unless the removed bitmap was on top of the stack. If the bitmap was in "the middle" of the cache a compacting operation is performed the next time Bitmap::cache is

called to reclaim the memory. This "costly" method can be avoided if you do not call Bitmap::cache with a "hole" in the cache.

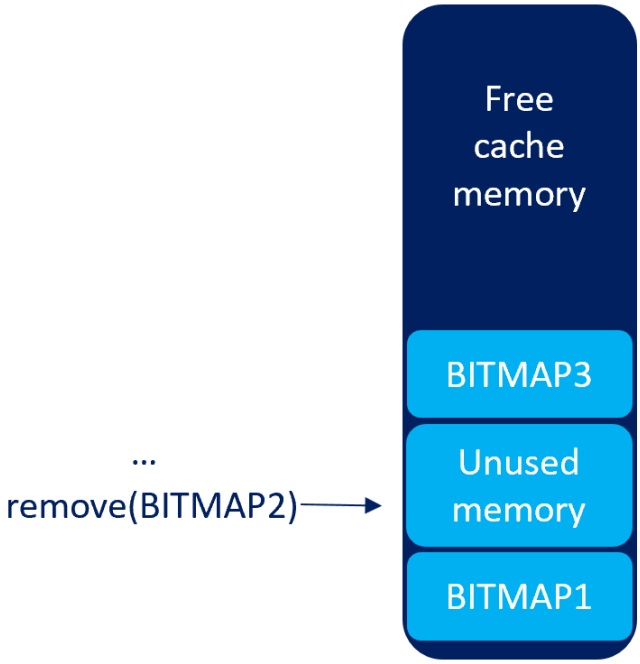
The drawings below illustrates the principles:

1. Caching allocates on top of the previously allocated bitmaps:



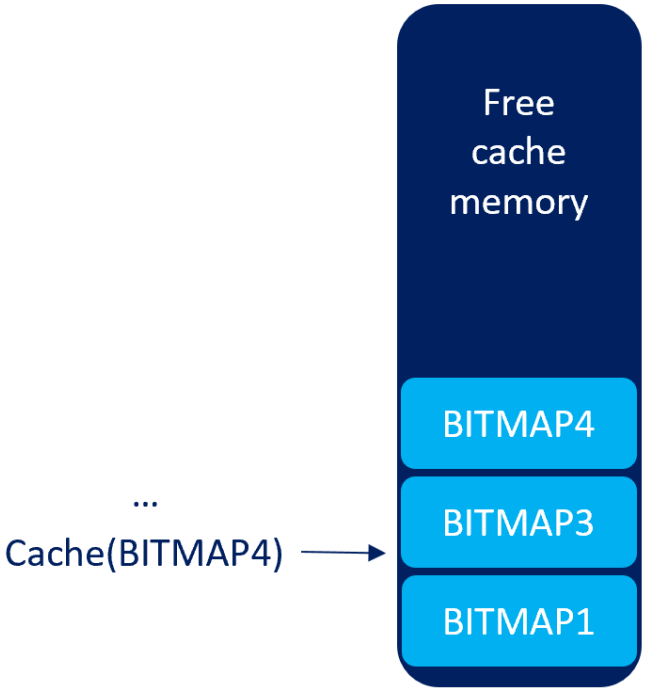
Allocation sequence of bitmaps in memory

2. Removal marks the memory unused:



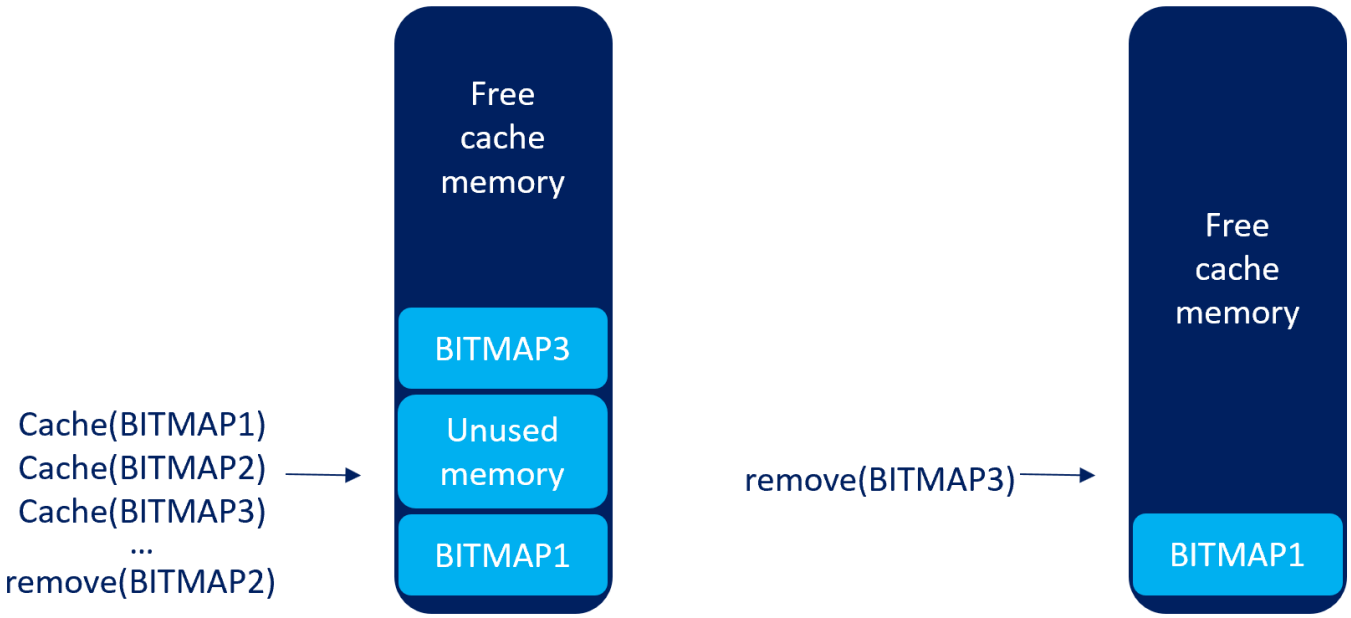
Unused memory in cache after removal of cached bitmap

3. Allocating the next bitmap compacts the cache and allocates on the top:



The cache reclaims unused memory before caching a bitmap

4. When you remove the topmost (last allocated) bitmap, the memory is freed immediately along with any free memory just below it:



Topmost bitmap cache removal

The next cache operation will in this case not involve a compact.

This animation shows the whole sequence for this code:

```
Bitmap::cache(BITMAP_BITMAP1_ID);
Bitmap::cache(BITMAP_BITMAP2_ID);
Bitmap::cache(BITMAP_BITMAP3_ID);
...
```

```
Bitmap::cacheRemoveBitmap(BITMAP_BITMAP2_ID);
...
Bitmap::cache(BITMAP_BITMAP4_ID);
...
Bitmap::cacheRemoveBitmap(BITMAP_BITMAP3_ID);
Bitmap::cacheRemoveBitmap(BITMAP_BITMAP4_ID);
```


Caching and uncaching bitmaps

Version: 4.16

Custom Widgets

When creating applications you may need widgets that are not part of the TouchGFX distribution. TouchGFX have a few ways in which you can create your own graphical elements. The simplest way is to use the Custom Container approach, where you combine already existing widgets into your own. This article, however, deals with a more advanced approach where you can essentially create a widget that has full control of the framebuffer and thus is able to draw precisely what you want.

When to use the custom widget approach

Creating custom widgets is not the most typical way to create your own widget. The custom container approach is preferable if it suits your needs, but sometimes this approach is not enough. When you need full control of the framebuffer you need to use the custom widget approach.

A few examples of widgets that could and should be created using the custom widget approach are:

- A sepia filter
- A mandelbrot fractal widget
- A widget that shows data from a file, for example a gif animation.

In TouchGFX Designer

TouchGFX Designer does not currently support the creation of custom widgets. As a result, you will need to write the code for the custom widget manually (refer to the remainder of this article on how to do this) and then insert the widget in the user code portion of your View.

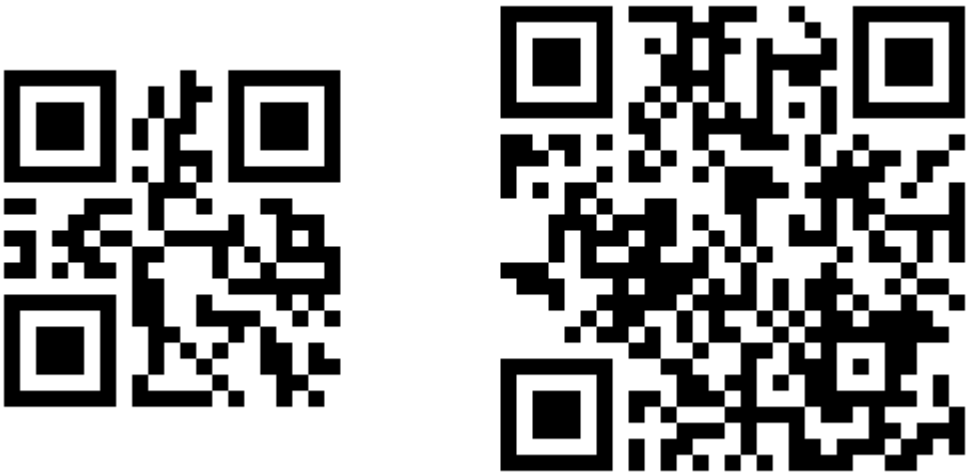
In code

You create your own custom widget by extending the `Widget` class. Doing this requires a bit more effort on the user's side, but will also give full control of all pixels drawn by the widget. Your custom widget will not necessarily utilize any existing widgets but instead define how it should be drawn by specifying colors of pixels. The custom widget approach will in general also have a smaller memory footprint, which in some cases is of great importance.

One example of a custom widget is a QR code widget. This particular widget makes for a good example, so we will walk through how to create it in the following section. In this example, a QR code

widget is an NxN matrix of black and white pixels. The size of the widget and the color of each pixel will be determined at run time and depend on the information in a QR code data object.

Here are two examples of how a QR code widget might look:



QR code widget examples

CAUTION

It is possible to create the qr code widget with the custom container approach by having n*n black or white boxes as children of the container. However, this will take up a lot of memory, and will probably not be as efficient or flexible.

TIP

The standard `touchgfx::Button` (supplied with the framework) is created as a custom widget, not a custom container. This saves memory per button on your screen.

Your own custom widget

In order to create your widget that extends the `Widget` class, you need to describe two things.

- The way your widget is drawn
- The part of your widget that is solid

Partial drawing

Any widget, and therefore also your custom widget, needs to support partial drawing. This means that your widget should be able to draw only a part of itself.

This is due to two points. It is often not necessary to redraw the entire screen but only parts of it. The algorithms of TouchGFX might split up the drawing of a screen into multiple partial drawings to minimize the global number of pixels drawn. This partial drawing of a screen, might then ask a widget to only draw itself partially.

As an example our QR code widget needs to be able to support drawing only the highlighted part of itself.

QR code widget partial drawing

The way to do this in your code is by overriding the `draw` method:

```
virtual void draw(const touchgfx::Rect& invalidatedArea) const
{
    //run through the pixels of the invalidated area
    //draw a black pixel if the qr data object has a value at this position
    //draw a white pixel otherwise
}
```

The `invalidatedArea` is the part of the widget that needs to be updated. In our QR code example the invalidated area is the highlighted area. The area is expressed in coordinates relative to the top left corner of the widget.

CAUTION

It is the responsibility of the widget to draw inside the invalidated area. If you draw outside the invalidated area you will not get overall correct behaviour of your screens.

TIP

The `draw` method is `const` because the optimized drawing algorithm might split up the drawing of a widget into multiple calls to draw. The `const` assures that the widget is not moved, updated etc. in between these multiple `draw` executions.

Solid area

Furthermore, a widget needs to be able to report how large a portion of itself is solid. Solid means that you can not see the things that are behind it on the screen. For instance, a standard red box is completely solid, whereas an image with an alpha value of 50% is completely non solid; you can see everything behind it.

A solid widget will enable TouchGFX to speed up the drawing process. Since we do not have to draw the widgets below the solid widget, fewer pixels have to be drawn.

To report the solid area in your code, override the `getSolidRect` method.

```
virtual Rect getSolidRect() const;
```

Our QR code widget will be completely solid. You will not be able to see anything behind the black and white pixels. Therefore, we let the method return a rectangle the full size of the widget:

```
virtual Rect getSolidRect() const
{
    return touchgfx::Rect(0, 0, getWidth(), getHeight());
}
```

Example source code

We end up with an application and a widget looking like this - depending on the data supplied:

QR code widget screenshot

The complete code of the widget is shown below:

gui/include/gui/common/QRCodeWidget.hpp

```
#ifndef QR_CODE_WIDGET_HPP
#define QR_CODE_WIDGET_HPP

#include <touchgfx/widgets/Widget.hpp>

class QRCodeWidget : public touchgfx::Widget
{
public:
    QRCodeWidget();

    virtual void draw(const touchgfx::Rect& invalidatedArea) const;
    virtual touchgfx::Rect getSolidRect() const;

    void setQRCodeData(QRCodeData* data);
    void setScale(uint8_t s);

private:
    void updateSize();

    QRCodeData* data;
    uint8_t scale;
};
#endif
```

In the header file, we define the widget as an extension of the `touchgfx::Widget` class. We override the `draw` and `getSolidRect` methods to define how our widget is drawn. We declare methods for setting the data of the QR code and setting the scaling factor of the QR code.

gui/src/common/QRCodeWidget.cpp

```
#include <gui/common/QRCodeWidget.hpp>

QRCodeWidget::QRCodeWidget() :
    data(0),
    scale(1)
{
}

void QRCodeWidget::draw(const touchgfx::Rect& invalidatedArea) const
{
    if (!data)
    {
        return;
    }

    touchgfx::Rect absolute = getAbsoluteRect();

    uint16_t* framebuffer = touchgfx::HAL::getInstance()->lockFrameBuffer();

    for (int y = invalidatedArea.y; y < invalidatedArea.bottom(); y++)
    {
```

```
        for (int x = invalidatedArea.x; x < invalidatedArea.right(); x++)
        {
            framebuffer[absolute.x + x + (absolute.y + y) * touchgfx::HAL::DISPLAY_WIDTH]
              data->at(x / scale, y / scale) ? 0x0000 : 0xffff;
        }
    }

    touchgfx::HAL::getInstance()->unlockFrameBuffer();
}

touchgfx::Rect QRCodeWidget::getSolidRect() const
{
    return touchgfx::Rect(0, 0, getWidth(), getHeight());
}

void QRCodeWidget::setQRCodeData(QRCodeData* qrCode)
{
    data = qrCode;
    updateSize();
}

void QRCodeWidget::setScale(uint8_t s)
{
    scale = s;
    updateSize();
}

void QRCodeWidget::updateSize()
{
    if (data)
    {
        setWidth(data->getSize() * scale);
        setHeight(data->getSize() * scale);
    }
}
```

The source file defines the `draw` method. This method steps through each of the pixels in the invalidated area and determines the color of the framebuffer based on the contents of the data object and the scaling factor.

The `getSolidRect` method reports the widget as completely solid.

⚠ CAUTION

Note that we locked and unlocked the framebuffer as part of our `draw` method. This is to make sure that the DMA is done drawing when we start drawing.

The code also uses a small class/interface to access the data of the QR code:

```
class QRCodeData
{
public:
    uint8_t getSize();
    bool at(uint8_t x, uint8_t y);
};
```

FURTHER READING

Download and check out the **QRCode** and **Lens** widgets.

THINGS TO TRY

- Modify the QR code widget such that white pixels are completely transparent.
- Create a custom widget that displays a sepia filter, a mandelbrot fractal, a gif image or something else.
- Consider which widgets are most easily done using custom containers and which widgets are most easily realized using the custom widget approach.

Modifying standard widgets/containers

The source code for the standard widgets are included when installing TouchGFX. These can also be modified to fit with the needs of an application. The source code for the standard widgets and containers is located in the folder:

```
touchgfx\framework\source\touchgfx
```

In order to maintain backwards compatibility, the core library contains compiled versions of the standard widgets and containers. It is therefore not necessary to compile these files in your project.

CAUTION

Modifying the standard widgets/containers directly is discouraged

The source code is primarily intended as inspiration and a way to learn about the inner workings of TouchGFX widgets. If you want something to behave differently than the standard implementation, it is possible to achieve this by either subclassing or creating custom containers, which is also the recommended approach.

The reason for this recommendation is two-fold:

- First, you will make it more difficult to upgrade to newer TouchGFX versions, since you must manually merge any changes you did.
- Second, you risk breaking the standard widgets and containers that are dependent on specific behavior.

Therefore, if it is necessary to modify a standard widget/container, we recommend you take a copy of it, rename it and use that instead of directly modifying the source code.

That being said, you are free to do whatever you judge is right. If you add the source code for a standard widget into your project, your linker will choose this version instead of the one in the core library. As a result, you have access to modification of the standard widgets by adding the source code to your compilation.

Version: 4.16

Canvas Widgets

Canvas Widgets and the Canvas Widget Renderer are a powerful and versatile add-on to TouchGFX which provides nice smooth, anti-aliased drawing of geometric shapes using relatively little memory while maintaining high performance. However, rendering geometrical shapes must be seen as a quite expensive operation and can easily strain the microcontrollers resources if not used carefully.

The Canvas Widget Renderer (hereafter referred to as CWR) is a general graphics API, providing optimized drawing for primitives, automatically eliminating most superfluous drawings. CWR is used by TouchGFX for drawing complex geometric shapes. Geometric shapes are defined by Canvas Widgets. TouchGFX comes with a number of supported Canvas Widgets but just like normal widgets you can make your own custom Canvas Widget to match your needs. Where a Canvas Widget defines the geometric shape of a figure to be drawn by the CWR, the actual color of each pixel inside the figure is defined by an associated Painter class. Again, TouchGFX comes with a number of Painters but you can make your own custom Painters to match your needs.

Using CanvasWidgets

Other widgets in TouchGFX have their sizes set automatically. A bitmap widget, for example, will automatically get the width and height of the contained bitmap. It is therefore enough to use `setXY()` on the bitmap widget to place the bitmap on the display.

Canvas Widgets do not have a default size which can be determined automatically and set initially. Care must be taken to not only position, but also size the widget correctly, otherwise the width and height of the Canvas Widget will be zero, and nothing will be drawn on the display.

So, instead of using `setXY()`, use `setPosition()` to place and size the canvas widget. See also Custom Canvas Widgets below for an example on how to create and use a custom canvas widget.

Once the position and size of the Canvas Widget has been set, a geometrical shape can be drawn inside it. The coordinate system will have (0, 0) in the upper left corner of the widget (not the display), the X axis stretches to the right and the Y axis stretches downwards.

Canvas widgets are also supported in TouchGFX Designer, and makes the usage simple and has automatic memory allocation.

Available CanvasWidget based widgets in TouchGFXDesigner:

- Line

- Circle
- Shape
- LineProgress
- CircleProgress

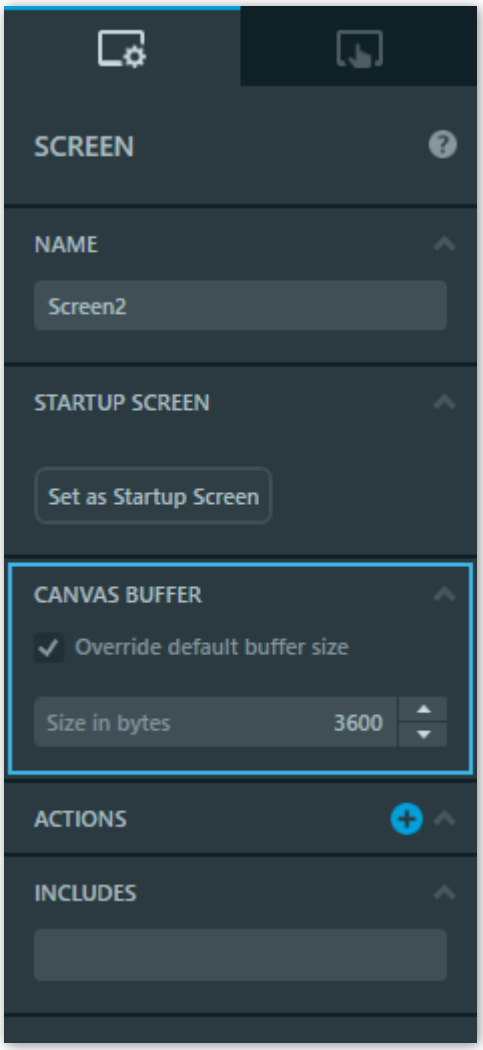
Using these widgets via TouchGFX Designer, makes placement and size adjustment much easier by showing how the widget will look at run time.

Memory Allocation and Usage

To produce nice anti-aliased complex geometrical shapes additional memory is required. For this CWR has to have a special allocated memory buffer that is used during rendering. CWR, as the rest of TouchGFX, has no dynamic memory allocation.

Memory Allocation in TouchGFX Designer

When adding a widget to the canvas of a Screen, a memory buffer is automatically generated. The size of the buffer is based upon the width of the Screen with the following formula (Width × 3) × 5. This is however not always the ideal buffer size for all scenarios. Therefore the buffer size can be overridden has shown in the image below.



Canvas buffer size being overridden in Screen properties

Memory Allocation in User Code

The memory buffer can be allocated and set up in `target/main.cpp` and `simulator/main.cpp` or be setup and allocated per Screen.

```
static const uint16_t CANVAS_BUFFER_SIZE = 3600;
static uint8_t canvasBuffer[CANVAS_BUFFER_SIZE]
```

A static const defining the size of the memory buffer, and the actual memory buffer can be defined in the beginning of the `main.cpp` or `ScreenView.hpp`

Then in either the `main()` method of `main.cpp` or `setupScreen()` method of `ScreenView.cpp` the following line setting up the buffer can be added.

```
CanvasWidgetRenderer::setupBuffer(canvasBuffer, CANVAS_BUFFER_SIZE);
```

The amount of CWR memory needed depends on the maximum size of the shapes that are to be drawn in the application. You can, however, reserve less memory than the maximum shape requires. To

handle this situation, the CWR splits up the drawing of shapes into smaller frame buffer parts resulting in slightly longer rendering time, as shapes in these cases will sometimes have to be rendered more than once. It is possible to investigate the memory consumption closer and fine-tune it when running in simulator mode. Simply add the following function call to your main.cpp:

```
CanvasWidgetRenderer::setWriteMemoryUsageReport(true);
```

Now whenever a draw operation finishes, CWR will report (print in the console) how much memory was required. For canvas_widget_example this could be “CWR requires 3604 bytes” (for the first draw operation) followed by “CWR requires 7932 bytes (4328 bytes missing)” (for the second draw operation). Even though it appears that CWR does not have enough memory (4328 bytes missing in this case) the application runs fine. This is because CWR detects that too little memory is available to complete the complex draw operation in a single run. Instead, it splits the draw operation into two separate draw operations and the shape will be drawn just fine but will require more time to render.

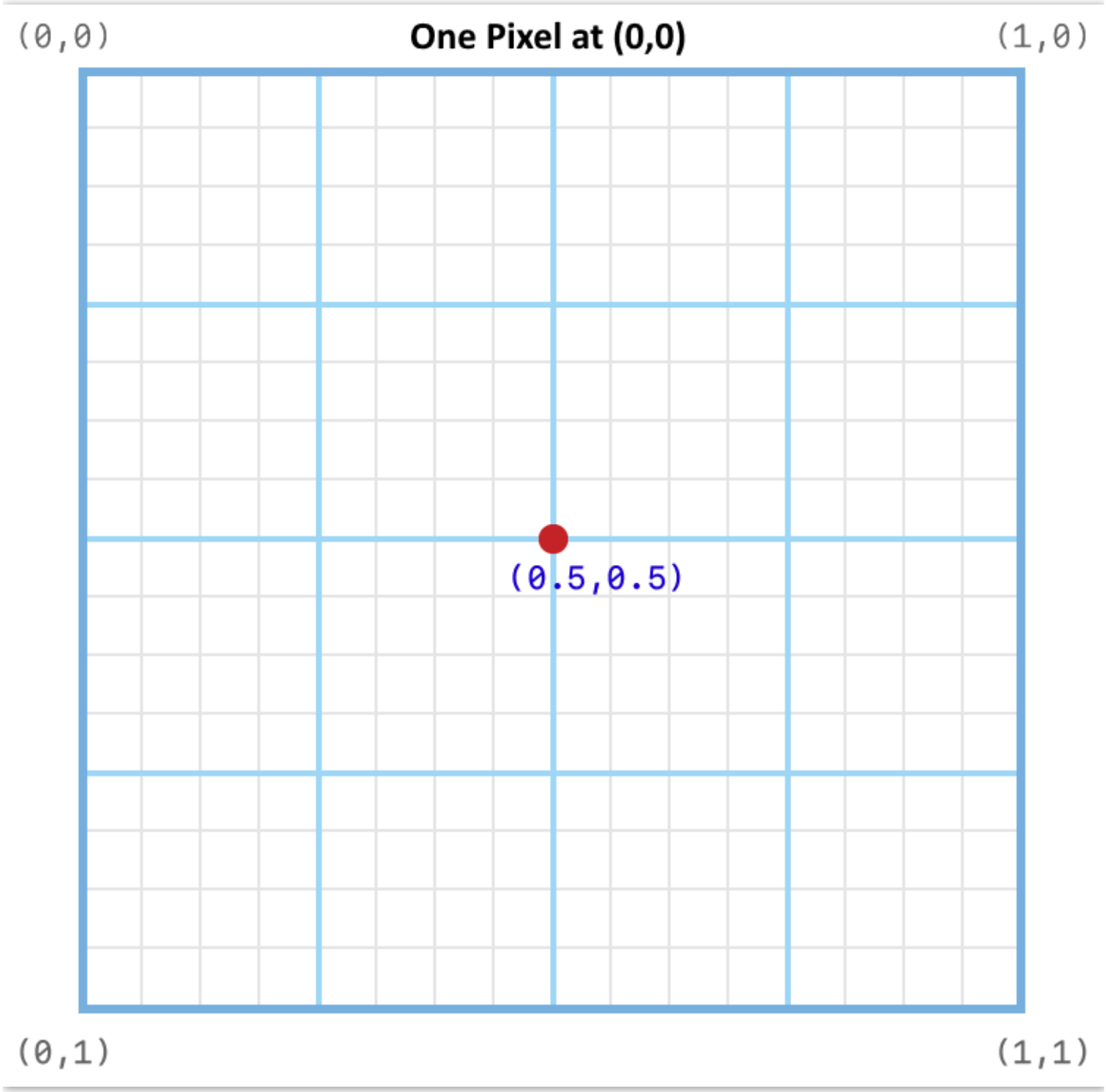
Setting the correct memory buffer size is therefore a trade off between memory and performance (rendering time). A good starting value is usually around 3000, but using the above technique, a better value can often be determined. If the shape is too complex and the allocated memory buffer is too small, part of the shape will not be drawn (some vertical pixel lines will be skipped) and it is possible that nothing is drawn at all. In any case rendering time will increase a lot.

This means that if you want your application to render the CWR drawing at maximum speed you need to allocate the requested amount of memory. But if you can go with a slower rendering timer it is perfectly okay to reduce the memory buffer.

The CWR Coordinate System

The coordinate system in TouchGFX is normally used to address pixels for positioning bitmaps on the display. Bitmaps, texts and other graphic elements are all placed in a coordinate system, where (0,0) is the upper left hand pixel, the x-axis extends to the right and the y-axis extends downwards. In CWR it is not enough to be able to address pixels using integers, though this might be enough in special cases, this is far from enough in general. To demonstrate this, consider a circle with line width 1, which must fit precisely inside a box of 5 by 5 pixels. The center of this circle must be at (2.5, 2.5) and the radius must be 2, so fractions are required for the center coordinate. Similarly, if the circle should fit inside a box of 6 by 6 pixels, the center must be at (3, 3) and the radius must be 2.5, so here fractions are required for the radius.

This new way of addressing coordinates for drawing graphics, means that the center of the pixel at (0,0) has CWR coordinate (0.5, 0.5). Hence, the box containing the pixel in the upper left corner of the screen has the following outline: (0,0) -> (1,0) -> (1,1) -> (0,1) -> (0,0).



CWR coordinate system for pixel at (0,0)

Though this might seem confusing at first, it quickly becomes very natural. Where the coordinate system for bitmaps address the pixels, the same coordinate for Canvas Widgets address the gap just before and above the pixel.

Custom Canvas Widgets

Implementing a custom Canvas Widget requires an implementation of a new class with the following functions:

```
virtual bool drawCanvasWidget(const Rect& invalidatedArea) const;
virtual Rect getMinimalRect() const;
```

The `drawCanvasWidget()` must draw whatever the custom widget needs to draw and `getMinimalRect()` should return the actual rectangle in the Widget which contains the geometrical shape.

> **NOTE**
>
> **The reason for having `getMinimalRect()` is that a geometrical shape can be moved around inside its widget and it is often impractical to resize and reposition the widget whenever the shape changes to only invalidate the smallest possible area.**
> **A dummy implementation of `getMinimalRect()` could simply `return rect;`, that is the size of the widget, but that would cause the entire area covered by the canvas widget to be redrawn, and not just the part of the canvas widget containing the geometrical shape. Very often, the geometrical shape occupies only a small part of the canvas widget.**

Canvas Widgets all use the Canvas class, which encapsulates the Canvas Widget Renderer as described above. CWR has many optimizations applied automatically, though awareness of your geometrical shape in relation to the invalidated area, and avoiding unnecessary drawing outside the invalidated area, is always a good way to boost performance.

A rough implementation of a diamond shaped square inside a 10x10 box could look something like this:

```
class Diamond10x10 : public CanvasWidget
{
public:
  virtual Rect getMinimalRect() const
  {
    return Rect(0,0,10,10);
  }
  virtual bool drawCanvasWidget(const Rect& invalidatedArea) const
  {
    Canvas canvas(this, invalidatedArea);
    canvas.moveTo(5,0);
    canvas.lineTo(10,5);
    canvas.lineTo(5,10);
    canvas.lineTo(0,5);
    return canvas.render(); // Shape is automatically closed
  }
};
```

> **NOTE**
>
> **Again, be careful that `getMinimalRect()` returns to correct rectangle, or the graphics on screen might be wrong.**

In order to see the Diamond10x10 on the display, the color must be set using `Diamond10x10::setPainter()` inherited from CanvasWidget. Also, the Diamond10x10 must be placed and sized correctly. This could look similar to this:

In the header file declare

```
Diamond10x10 box;
PainterRGB565 myPainter;
```

and in the code you should have something like this:

```
myPainter.setColor(Color::getColorFrom24BitRGB(0xFF, 0x0, 0x0));
box.setPosition(100,100,10,10);
box.setPainter(myPainter);
add(box);
```

Painters

A Painter defines a coloring scheme to fill a Canvas Widget object. TouchGFX comes with a set of predefined painter classes, but custom painters can easily be implemented.

In order to implement a custom Painter, care must be taken to never write outside the frame buffer. Such a bug in a custom Painter can result in serious crashes. Here is an example of a custom Painter which we will use to paint an object red, only function `renderNext()` needs to be implemented. See AbstractPainter for more information.

```
class Red : public AbstractPainterRGB565
{
public:
    virtual bool renderNext(uint8_t &red, uint8_t &green, uint8_t &blue, uint8_t &alpha)
    {
        red = 0xFF;
        green = 0x00;
        blue = 0x00;
        alpha = 0xFF;
    }
};
```

To paint the box object from above red, put this in the header file:

```
Diamond10x10 box;
Red redPaint;
```

and put this in the code:

```
box.setPosition(100,100,10,10);
box.setPainter(redPaint);
add(box);
```

Please note that it is possible to override more methods to create special painters e.g. `renderInit()`, however, TouchGFX has some generic painters which covers most uses.

Version: 4.16

Dynamic Bitmaps

This section explains how to use Dynamic Bitmaps. Recall that standard bitmaps are compiled into the application and therefore must be available at compile time. The bitmaps are converted from e.g. PNG files and stored in an internal format together with size and format information.

It is also possible to create a bitmap in RAM at runtime. This is called a *dynamic bitmap*. A dynamic bitmap can be used just as the static bitmaps that are compiled into the application. This means that you can use a dynamic bitmap with e.g. the Image and Button widgets.

Dynamic Bitmap Configuration

When you create a dynamic bitmap the RAM for the pixels is allocated from the bitmap cache. You must therefore configure a bitmap cache before you can create dynamic bitmaps.

See the article on bitmap caching for configuration instructions.

It is required to define the maximum number of Dynamic Bitmaps used in your application. This maximum is passed to TouchGFX together with the bitmap cache address and size. Here we configure a bitmap cache with up to 4 dynamic bitmaps:

BoardConfiguration.cpp (extract)

```
// Place cache start address in SDRAM at address 0xC0008000;
uint16_t* cacheStartAddr = (uint16_t*)0xC0008000;
uint32_t cacheSize = 0x300000; //3 MB, as example
HAL& hal = touchgfx_generic_init<STM32F4HAL>(dma, display, tc, DISPLAY_WIDTH, DISPLAY_HEI
```

Using a Dynamic Bitmap Example

To use the dynamic bitmap we need a widget to show it. So insert an Image widget in the view (in code or in the Designer):

```
class TemplateView : public View
{
private:
    Image image;
}
```

Create the dynamic bitmap in setupScreen. Here we use the 16bpp format RGB565. If your framebuffer is e.g 24 bit use RGB888. To create a transparent bitmap, use the format ARGB8888:

```
void TemplateView::setupScreen()
{
    BitmapId bmpId;

    //Create (16bit) dynamic bitmap of size 100x150
    const int width = 100;
    const int height = 150;
    bmpId = Bitmap::dynamicBitmapCreate(100, 150, Bitmap::RGB565);

    //set all pixels white
    if (bmpId != BITMAP_INVALID)
    {
       memset(Bitmap::dynamicBitmapGetAddress(bmpId), 0xFF, width*height*2);
    }

    //Make Image widget show the dynamic bitmap
    image.setBitmap(Bitmap(bmpId));

    //Position image and add to View
    image.setXY(20, 20);
    add(image);
    ...
}
```

If you want to load your image from a file you can replace the call to memset with your loader code. See the article Loading Images At Runtime

Dynamic Bitmap Operations

The dynamic bitmap operations are all placed in the Bitmap class.

Creating a Dynamic Bitmap

The following method creates a dynamic bitmap with the width, height and bitmap format specified. The bitmap is only created if enough unused memory is available. The method returns BITMAP_INVALID if the bitmap was not created.

```
static BitmapId Bitmap::dynamicBitmapCreate(const uint16_t width, const uint16_t height, E
```

Deleting a Dynamic Bitmap

This method deletes a dynamic bitmap.

```
static bool Bitmap::dynamicBitmapDelete(BitmapId id)
```

Get the address of the pixels in a Dynamic Bitmap

The following method returns the address of the dynamic bitmap. This method is used by file loaders to copy image data into the bitmap.

```
static uint8_t* dynamicBitmapGetAddress(BitmapId id)
```

Set the solid area of a Dynamic Bitmap

The following method sets the solid rectangle of a dynamic bitmap.

```
static bool dynamicBitmapSetSolidRect(BitmapId id, const Rect& solidRect)
```

Read more about the "solid area" concept in the Custom Widgets article.

By default the solid area is set to be the whole bitmap for non-transparent formats like RGB565 and RGB888. It is set to empty for transparent formats like ARGB8888.

Version: 4.16

Binary Fonts

This section describes how to use binary fonts in TouchGFX. The first section contains some in-depth information about the font and text system in TouchGFX that can be beneficial to understand when working with binary fonts. The second section explains how to use binary fonts.

Binary fonts can be used as an alternative to the traditional way of compiling and linking font information in to your application (the .cpp files in `generated/fonts/src`). The main advantages of this approach is to get a smaller application binary and get flexibility in providing different sets of fonts with your device. For example you can pack the Chinese font with devices going to China, and the Japanese font with devices going there. The drawback of this approach is that the whole binary font needs to be loaded to RAM (or memory-mapped flash) which can be a problem if the font is large.

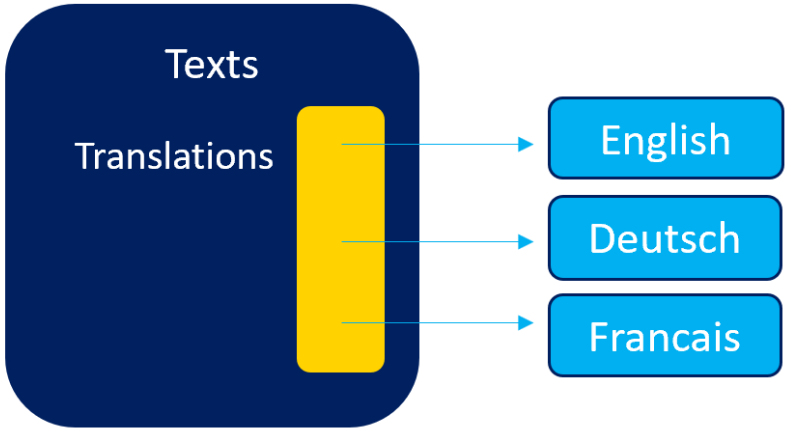
The main advantage of the normal principle of linking fonts into the application is that the application will always automatically contain the updated texts and typographies used in the application. This is very easy and safe to use. The disadvantage is that fonts can make the application big.

The Font and Text system classes

In the default configuration TouchGFX generates .cpp-files for all texts and fonts used in the application. These files are compiled and linked into the application together with the generated UI and your application code.

When you show a text on the UI with e.g. a TextArea, you reference the text with a TextId. This TextId is used by the Widgets to find the actual letters in the text. The Widgets will access the texts through the *touchgfx::Texts* class `framework/include/touchgfx/Texts.hpp`.

The Text class contains a pointer array with a pointer to a translation table for each language in the application. A translation table is in principle a collection of all strings used in that language:



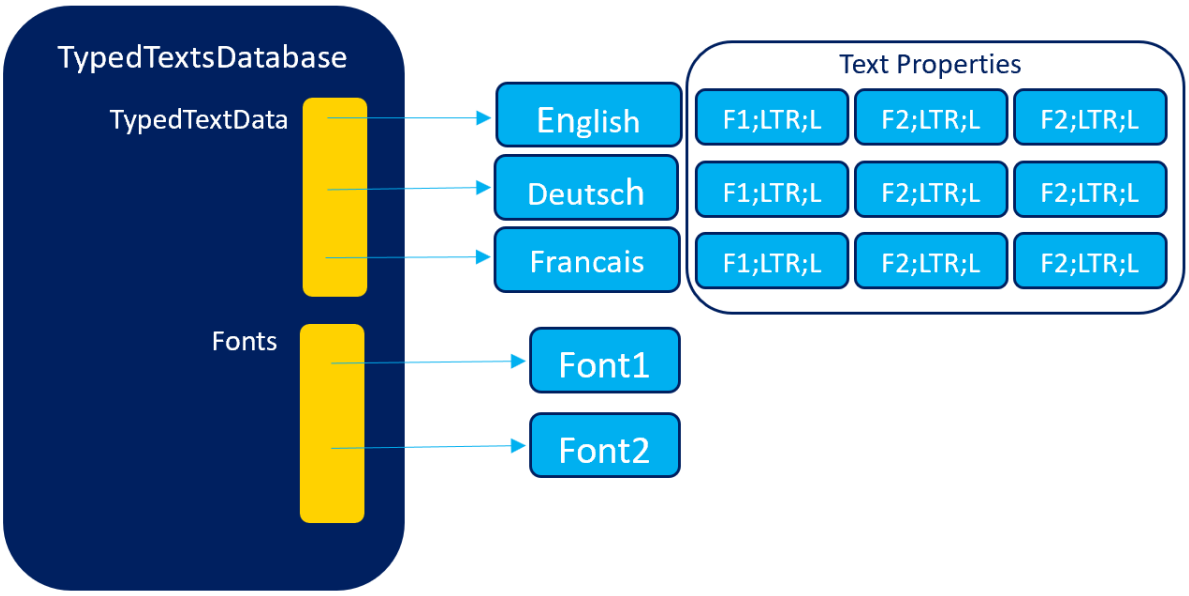
Mapping texts to specific languages

This table allows TouchGFX to find a given text in the selected language.

The tables are regenerated whenever you change a text in TouchGFX Designer and generate your application.

Before we can draw on the screen we need to know which font to use for the text. This mapping between texts and fonts is controlled by the TypedTextDatabase class (`generated/texts/include/texts/TypedTextDatabase.hpp`).

In the texts tab in TouchGFX Designer you can specify a typography, writing order (Left-to-right or Right-to-left), and an alignment for each text (Left, Right, Center). The typography, order, and alignment can be different for each translation of the text. This information is compiled into a table specific for each language. This makes it easy for TouchGFX to find out what font to use for a given text, how to align it, and how to write it.



typography information is specific to a language

In the above figure the TypedTextData table has pointers to three arrays. One for each language in the application. Each of the arrays has 3 elements, one for each text in the system. Each elements describes a font, a reading order, and the alignment. We see that in this example the texts use the same font in the three languages. The Fonts table has two pointers because there are two fonts in the application.

When TouchGFX is about to draw a text on the screen, it looks up the TypedTextData for the given text. This data contains the font index, letter order (LTR/RTL), and the horizontal alignment (Left, Right, Center) of the text as specified in the Excel sheet. TouchGFX uses the font index in the TypedTextData (F1 or F2) to lookup the correct font for the text.

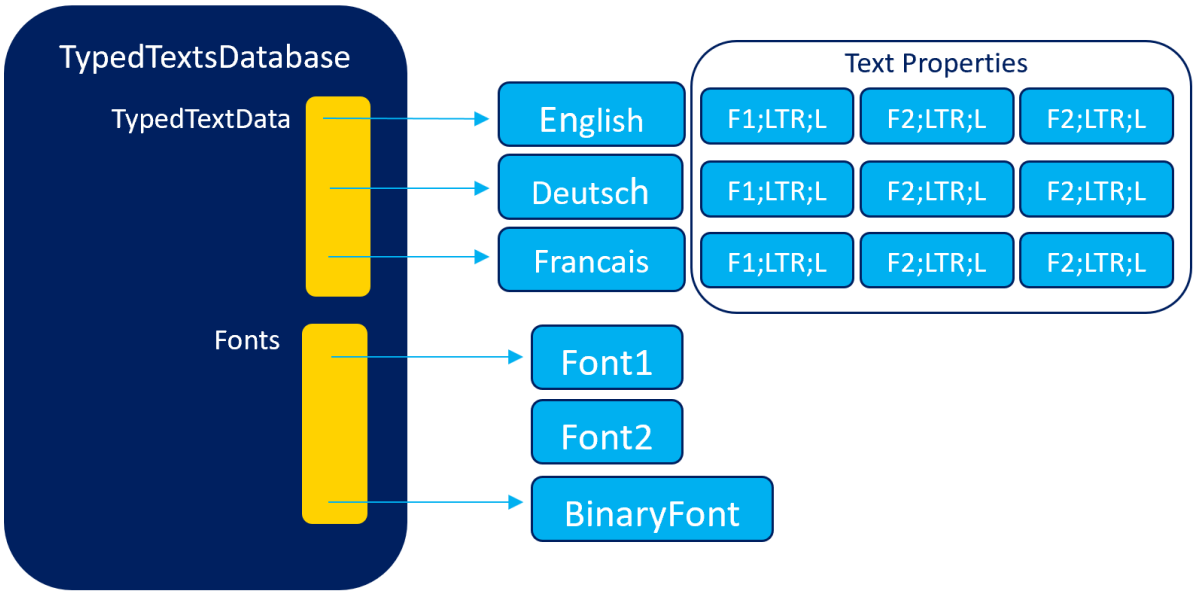
All this happens automatically when the fonts are compiled into your application.

Using Binary Fonts

When an application is using many letters in many different fonts the size of the application can grow substantially.

To relieve this problem TouchGFX allows applications to use binary fonts. These fonts are not linked into the application but are stored separately from the application as files. These files are loaded and provided to TouchGFX by the application at runtime. The application can e.g. load the font from an external storage like an sd-card or maybe download the font from the Internet.

When the application has loaded the font, it can ask TouchGFX to install the Binary Font in the font system:

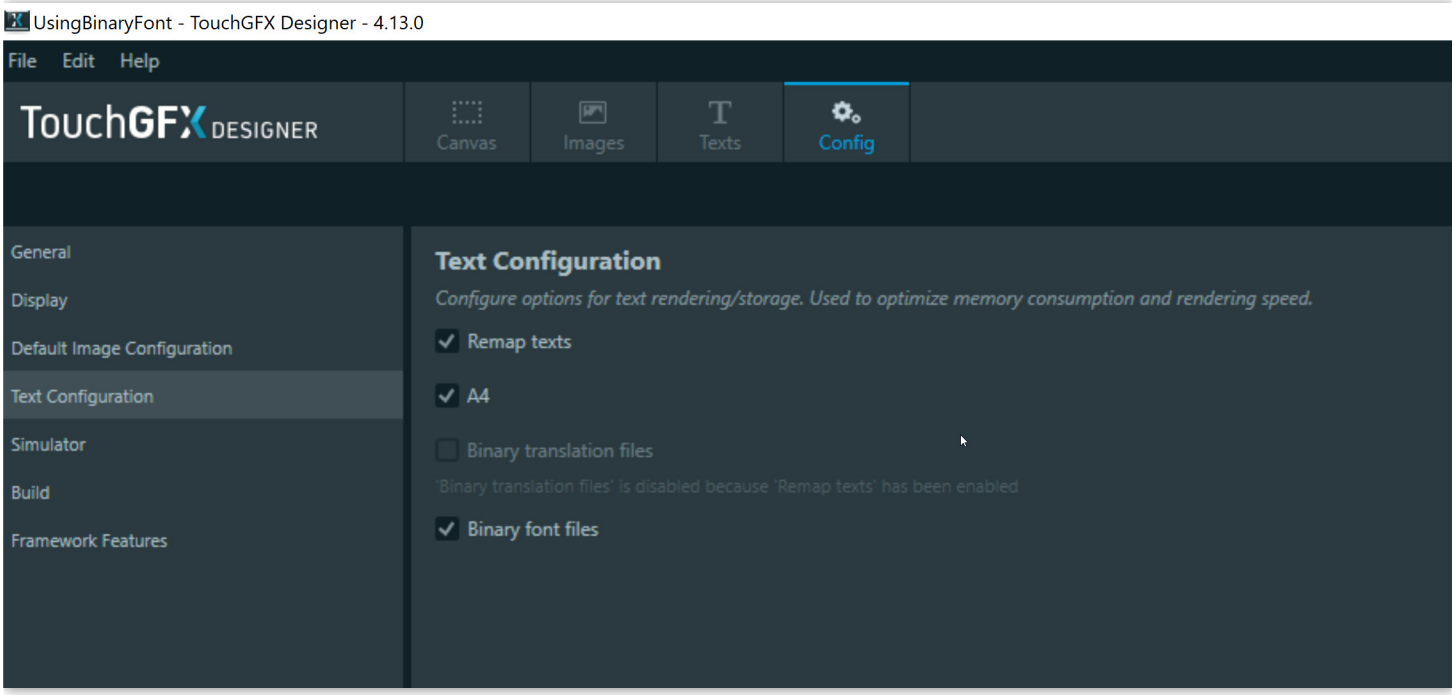


Installing a binary font in the font table

Here the built-in Font2 is replaced by the Binaryfont loaded by the application. The linked-in Font2 is hereafter not used by TouchGFX.

Configuring the Font converter to generate Binary Fonts

The font converter must be configured to generate binary fonts. This is easily done in TouchGFXDesigner. Go to the Config tab, select "Text Configuration", and click "Binary font files":



Selecting Binary Fonts

When you regenerate the code, TouchGFX will generate binary fonts in the `generated/fonts/bin/` folder, and empty fonts in the normal files in `generated/fonts/src/`

Manual Configuration

If you are not using TouchGFX Designer you can still generate binary fonts. Change the option "binary_fonts" to "yes" in the text_configuration section in the *application.config* file in your project.

application.config

```
  "text_configuration": {
    "remap": "yes",
    "a4": "yes",
    "binary_translations": "no",
    "binary_fonts": "yes",
    "framebuffer_bpp": "16"
  }
```

When you generate assets the next time, the binary fonts will be in the `generated/fonts/bin` folder.

Installing the binary font

Before TouchGFX can use a binary font it must be copied from the file or other storage. The font must be made available in addressable memory like RAM or QSPI flash (where it can be accessed through a pointer).

When the application has loaded the binary font to memory, it can install the font in TouchGFX. Now TouchGFX will use that font and not the compiled font. The binary needs to be installed before the text is used, but it does not need to be done immediately after booting. The *FrontendApplication::FrontendApplication(Model& m, FrontendHeap& heap)* constructor in `FrontApplication.cpp` can be used to install fonts. This constructor is executed before anything is drawn.

Here is an example:

FrontendApplication.cpp

```
//read the file into this array in internal RAM
uint8_t fontdata[10000];

//binary font object using the data
BinaryFont bf;

FrontendApplication::FrontendApplication(Model& m, FrontendHeap& heap)
    : FrontendApplicationBase(m, heap)
{
    //read the binary font from a file
    FILE* font = fopen("generated/fonts/bin/Font_verdana_20_4bpp.bin", "rb");
    if (font)
    {
        //read data from the file
        fread(fontdata, 1, 10000, font);
        fclose(font);

        //initialize BinaryFont object in bf using placement new
        new (&bf) BinaryFont((const struct touchgfx::BinaryFontData*)fontdata);

        //replace application font 'DEFAULT' with the binary font
        TypedTextDatabase::setFont(DEFAULT, &bf); //verdana_20_4bpp
    }
}
```

The exact code for opening a file and reading data will depend on your file system and operating system. The basic steps are to make the font data available in memory, initialize a BinaryFont object with a pointer to the data, and finally pass the BinaryFont object to TouchGFX.

After the call to *setFont* TouchGFX will use the binary font to draw text on the screen.

Resetting a font

Sometimes you want to go back to the original font compiled into your application after using a binary font. For example if you are changing language, and want to use the default font. The `resetFont()` function in TypedTextDatabase will reset the font pointer to the built-in font:

```
//reset to original font
TypedTextDatabase::resetFont(DEFAULT);
```

After this call, the application can reuse the memory occupied by the binary font to allocate a new font or for other purposes.

Version: 4.16

Font Caching

This section describes how to use the font cache to handle binary fonts in TouchGFX.

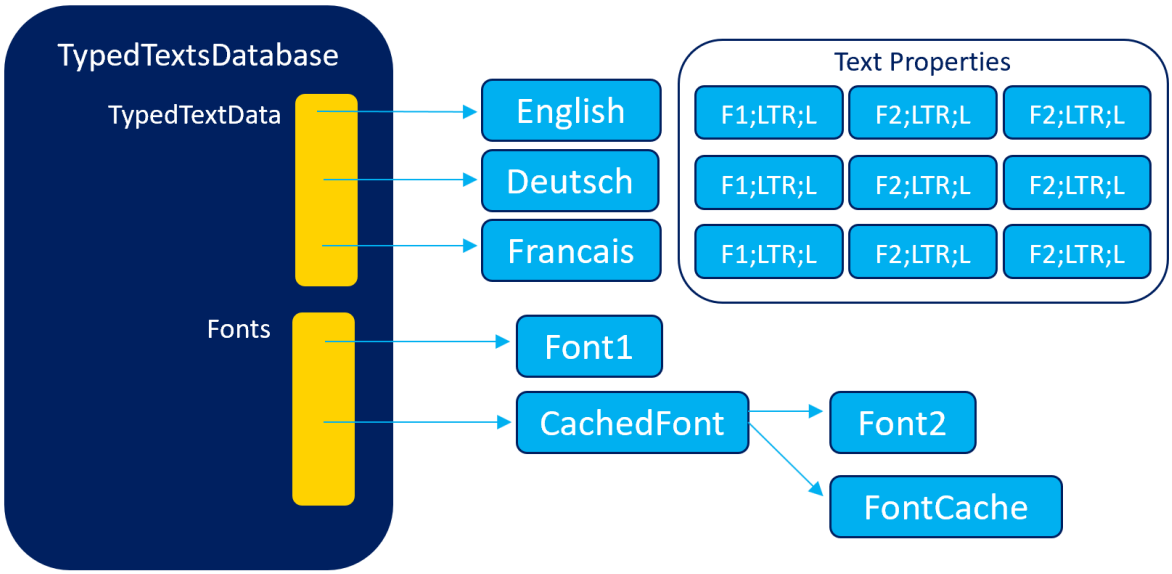
Read first about binary fonts.

Font Caching

Recall that using binary fonts require the whole font to be loaded to memory. This can in some cases be undesireable if the font is large, e.g. with large Chinese fonts.

Font caching allows an application to load from external memory only the letters required to show a string. This means that the whole font does not need to reside in the addressable flash or RAM, but can be stored on a larger file system.

In the drawing below, the compiled-in font, Font2, has been replaced by the font cache. When TouchGFX is drawing a text, that uses Font2, TouchGFX will find the pointer to the `CachedFont` object in the font table. This special font will lookup the letters in the FontCache object.



Using a cached font

The CachedFont is setup with a pointer to the linked-in font (Font2 above). When TouchGFX asks the CachedFont for a specific letter, the cachedFont will first look in the normal Font it is replacing (Font2). This font may be an empty font, but can also be a "normal" font containing a selection of some often used letters. If the font does not contain the required letter, the CachedFont will look into the FontCache to see if the letter has been loaded from the file system.

This principle limits the amount of letters that must be cached, as we do not need to cache letters already found in the normal font.

Using the Font Cache in application code

Before the application can install a CachedFont it must also create a FontCache, a memory buffer, and a file system reader object:

Screen1View.cpp

```
uint8_t fontdata[5120]; //Memory buffer for the font cache, 5Kb
FontCache fontCache;
CachedFont cachedFont;  //Cached Font object
FileDataReader reader;  //Filesystem reader object
```

The FontCache must be linked to the buffer and the reader:

Screen1View.cpp

```
//setup the font cache with buffer and size; and file reader object
fontCache.setMemory(fontdata, 5120);
fontCache.setReader(&reader);
```

Now the application can setup the font cache, initialize the CachedFont and pass it to TouchGFX.

The font cache requires a TextId to initialize a CachedFont object. The TextId is used to lookup the font that the CachedFont must point to. This secures that you are replacing the font used by the text that you have on your display:

Screen1View.cpp

```
//initialize the cachedFont object to the font used by T_SINGLEUSEID1
TypedText text = TypedText(T_SINGLEUSEID1);
fontCache.initializeCachedFont(text, &cachedFont);

//replace the linked in font in TouchGFX with cachedFont
TypedTextDatabase::setFont(DEFAULT, &cachedFont);
```

The code above can be put anywhere in the application. If the cached font is only used in a specific view, this view can be a good place to insert the code.

Caching Letters

The font cache is still empty. Before we can show any letters they must be read from the font cache. This is done by passing an array of unicodes (a string) to the font cache. In this example we just pass the text from T_SINGLEUSEID1.

Screen1View.cpp

```
//cache the glyphs used by the text T_SINGLEUSEID1
Unicode::UnicodeChar* str = const_cast<Unicode::UnicodeChar*>(text.getText());
bool b = fontCache.cacheString(text, str);
```

The font cache will load the letters found in the `str` array through the reader object. The read unicodes will be linked to the font that is used by the TextId `text` argument.

The application is responsible for configuring the reader object to load from the correct file.

Caching Ligatures

For languages that convert sequences of unicodes to other unicodes before displaying (e.g. Arabic and Devanagari) the above method is not good. It caches the original unicodes and not the unicodes that are displayed after conversion. This method will convert the given unicodes and cache the required unicodes (after conversion):

Screen1View.cpp

```
//cache the glyphs used by the text T_SINGLEUSEID1 after conversion
Unicode::UnicodeChar* str = const_cast<Unicode::UnicodeChar*>(text.getText());
bool b = fontCache.cacheLigatures(cachedFont, text, str);
```

Memory Usage

The font cache can calculate the current amount used memory:

Screen1View.cpp

```
touchgfx_printf("Memory usage %d\n", fontCache.getMemoryUsage());
```

Caching GSUB Tables

Some fonts use a GSUB table while rendering. These are only a few fonts for eastern languages, e.g. Devanagari fonts. The GSUB tables allow the font system to reorder characters and substitute sequences of characters for other "combination" characters.

The Font Cache can load this GSUB table from the file system. If it is not loaded, the font is not displayed correctly as the GSUB table is then not available to the text rendering system.

The GSUB table is loaded by supplying an extra argument when initializing the cached font:

Screen1View.cpp

```
//initialize the cachedFont and load the GSUB table
text = TypedText(T_SINGLEUSEID1);
fontCache.initializeCachedFont(text, &cachedFont, true);
```

Implementing the Font File Reader

The FileDataReader class used in the above example code is not included in TouchGFX as it is dependant on the operating system you are using.

Here is an example for normal "stdio" compatible file systems.

Screen1View.cpp

```
class FileDataReader : public FontDataReader
{
public:
    virtual ~FileDataReader() { }
    virtual void open()
    {
        fp = fopen("Font_verdana_20_4bpp.bin", "rb");
        if (!fp)
        {
            touchgfx_printf("Unable to open font file!!!\n");
        }
    }
    virtual void close()
    {
        fclose(fp);
    }
    virtual void setPosition(uint32_t position)
    {
        fseek(fp, position, SEEK_SET);
    }
    virtual void readData(void* out, uint32_t numberOfBytes)
    {
```

```
        fread(out, numberOfBytes, 1, fp);
    }
private:
    FILE* fp;
};
```

The FileDataReader class implements the FontDataReader interface from FontCache.hpp:

FontCache.hpp

```
class FontDataReader
{
public:
    virtual ~FontDataReader() { }
    virtual void open() = 0;
    virtual void close() = 0;
    virtual void setPosition(uint32_t position) = 0;
    virtual void readData(void* out, uint32_t numberOfBytes) = 0;
};
```

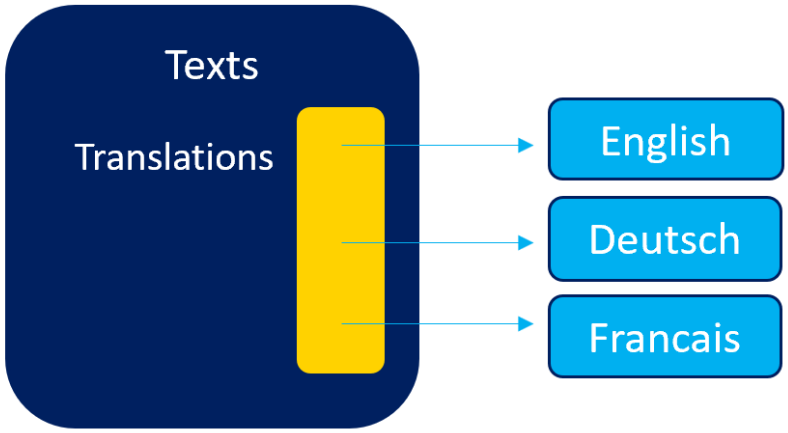
Version: 4.16

Binary Translations

This section describes how to use binary translations in TouchGFX. Normally text translations are compiled into the application. This principle is efficient and easy to use.
Binary translations keep the text translation out of the application. The binary translations are generated in separate binary files which can be programmed to flash or for example stored on an sdcard. This gives more flexibility to application developers when handling a large number of translations.

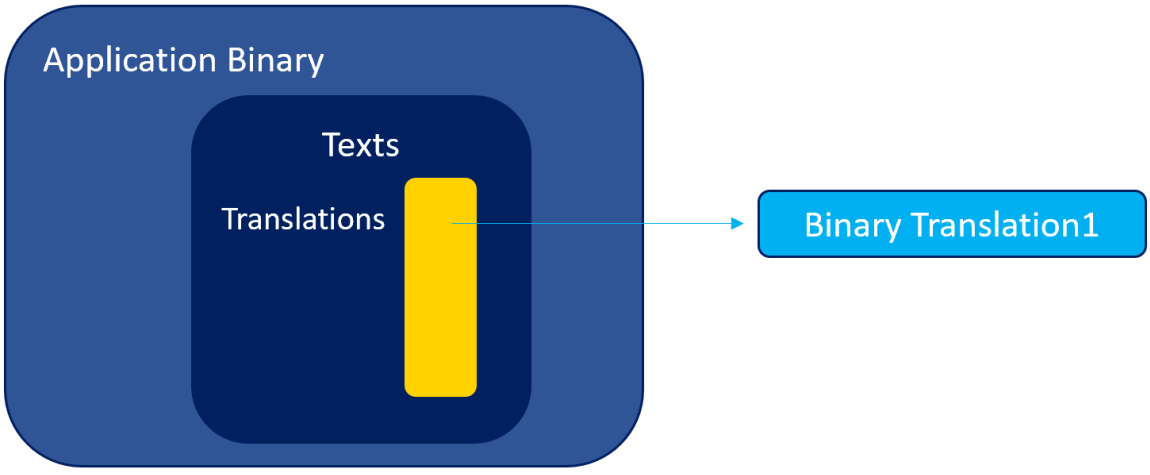
Translations

The Text class in TouchGFX contains a pointer array with a pointer to a translation table for each language in the application. A translation table is in principle a collection of all strings used in that language:



Mapping texts to specific languages

This table allows TouchGFX to find a given text in the selected language.

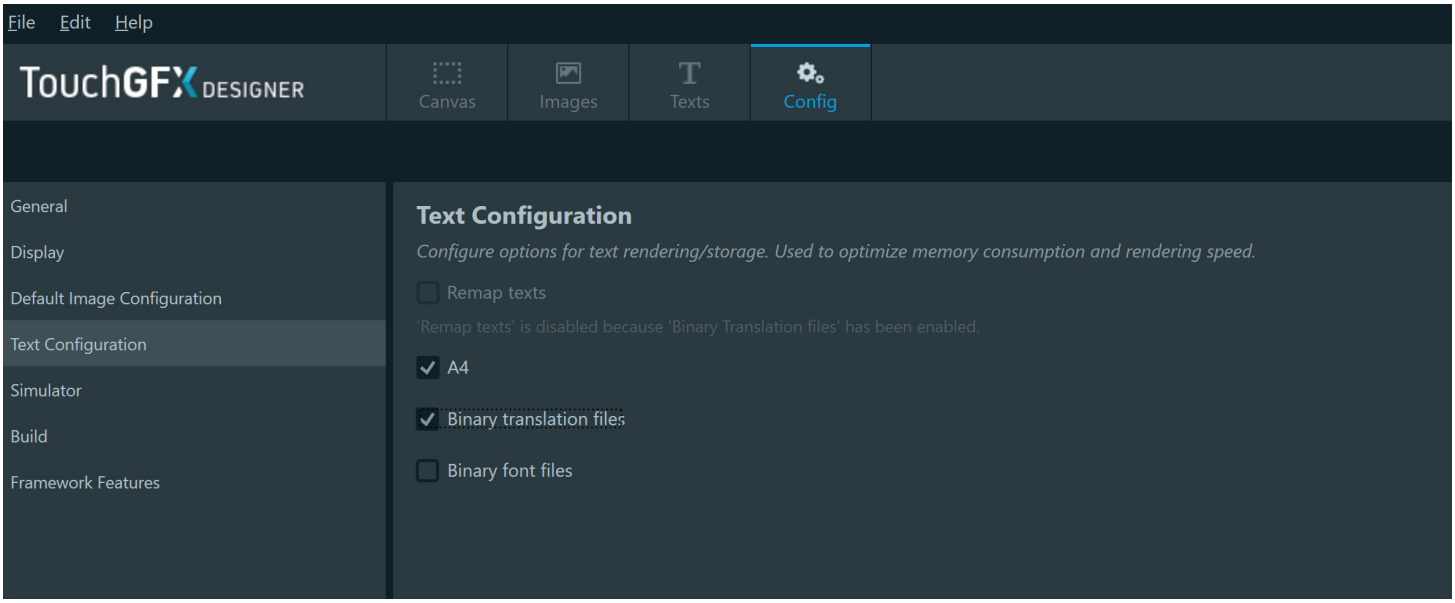


Mapping to a binary translation

When using binary translations at runtime you change the mapping from the compiled-in translation to a binary translation. The binary translations must be available in addressable memory (flash or RAM). It can for example be loaded from a disk or written to the flash during production.

Configuring the Text converter

The TouchGFX text converter can be configured to generate binary translations. This is easily done in the TouchGFX Designer 4.13. Go to the Config tab, select "Text Configuration", and click "Binary translation files"



Enabling binary translation

When you generate the code the next time, the binary translations will be in the generated/texts/binary folder.

The translations compiled into the application are empty when binary translations are generated. Therefore no texts are shown unless you load binary translations.

Installing a binary translation

When the application has the binary translation in memory, it can install it in TouchGFX. Now TouchGFX will use that translation. Depending on the application this can be done in different places or time (The *FrontendApplication::FrontendApplication(Model& m, FrontendHeap& heap)* constructor in *gui/src/common/FrontApplication.cpp* can e.g. be used).

Here is an example where we read a translation for English from a filesystem and install it as language "GB".

```
//read the translation into this global array
uint8_t translation[10000];
...

//read the translation from a file, or change array to a pointer that points
//to the translation data in internal or addressable external flash
FILE* file = fopen("generated/texts/binary/LanguageGb.bin", "rb");
if (file)
{
    //read data from file
    fread(translation, 1, 10000, file);
    fclose(file);

    //replace empty translation with the binary data
    Texts::setTranslation(GB, translation);

    //always change language to get TouchGFX changed from the
    //empty translation compiled in to the binary translation
    Texts::setLanguage(GB);
}
```

Note, it is necessary to change language after installing a translation. Otherwise TouchGFX will still use the previous translation. Read about changing language here.

It is also necessary to force a redraw of the current screen, or change screen, to see the new texts (if you are loading translations for the language that is used on the display). TouchGFX does not redraw anything automatically.

The current screen can be redrawn by invalidating the root container:

```
container.invalidate();
```

This will force a redraw. In some cases it is better to change the screen to get everything set up again from scratch (e.g. to recalculate TextArea sizes). You can change the screen by creating an interaction in TouchGFX Designer with the "Change Screen" action. See this article.

Version: 4.16

Backend Communication

In most applications, the UI needs to be connected to the rest of your system somehow, and send and receive data. This could be interfacing with hardware peripherals (sensor data, A/D conversions, serial communication, ...) or interfacing with other software modules.

This article describes the recommended solutions for implementing this connection.

The first method is a "quick-and-dirty" approach, primarily intended for prototyping, whereas the second method is an architecturally sound way of properly connecting the UI with the remaining components in a real world application.

In the end of this article, we link to examples of using both methods.

The Model Class

All TouchGFX applications have a Model class, which apart from storing UI state information is also intended to function as the interface to your surrounding system. By this we are referring to both hardware peripherals but also communicating with other OS tasks in your system. It is normally not a good design to access other software modules or hardware in the individual View classes.

FURTHER READING

To learn more about the Model: **MVP pattern**

The Model class is well suited for placing any such interface code because:

1. The Model class has a `tick()` function which is automatically called every frame and can be implemented to look for and react to events from other sub-modules.
2. The Model class has a pointer to your currently active Presenter, in order to be able to notify the UI of incoming events.

System Interfacing

There are two ways of interfacing with the surrounding system, either by sampling directly from the GUI Task, or by sampling from a Secondary Task

Sampling from GUI Task

The best way of interfacing with surrounding system depends on how frequently you need to sample, how time consuming it is and how time critical it is.

If your requirements in those regards are lenient, the simplest approach is to just sample the surrounding system directly in the `Model::tick` function.

If the sampling occurs less frequently than your frame rate (typically around 60Hz), you can just add a counter and only do the sampling every Nth tick. When done this way, your sampling operation must be somewhat fast (typically 1ms or less), otherwise your frame rate will begin to suffer, since the sampling is done in the context of the GUI task and will delay drawing the frame.

Sampling from Secondary Task

Alternatively, if it is not desirable to place the interaction with the surrounding system directly within the context of the GUI task, you can create a new OS task responsible for doing the sampling.

You can configure this task to run at the exact time intervals you need for your specific scenario. Also depending on your needs this new task can have a lower or higher priority than the GUI task.

If it has a higher priority, then you are guaranteed that it runs at exactly the times you have specified, regardless of what the GUI task is doing. This has the drawback that if it is a CPU consuming process it might impact the frame rate of the UI.

If on the other hand the sampling is not time critical, you can assign the task a lower priority than the GUI task, such that the UI frame rate is never affected by the sampling of the surrounding system. The GUI task will sleep a lot while rendering (e.g. when waiting for a DMA-based pixel transfer to complete) so lower priority tasks will be allowed to run quite frequently and this is therefore sufficient for the vast majority of applications.

If you use the secondary task approach, we recommend that you take advantage of the inter-task messaging system that is provided by your RTOS. Most, if not all, RTOSes have a queue/mail mechanism which allows you to send data (typically user-defined C structs, byte arrays or simple integers) from one task to another. In order to communicate new data to the GUI task, set up a mailbox or message queue for the UI task, and send the data to the GUI task using this messaging system. You can then `Model::tick` poll the mailbox of the GUI task to check if any new data has arrived. In case, read the data and update the UI accordingly.

Propagating Data to UI

Regardless of whether you use Sampling from GUI Task or Sampling from Secondary Task, the `Model::tick` function is the place where the GUI Task becomes aware of the new data to be shown in

the UI. Apart from acting as an interface to your surrounding system, recall from earlier that the Model class is also responsible for holding state data, so there might be some state variables that need to be updated too.

Let us consider a simple example where a temperature sensor is attached to the system, and that the current temperature is to be shown in the UI. In preparation, we will augment the Model class to support this:

Model.hpp

```
class Model
{
public:
  // Function that allow your Presenters to read current temperature.
  int getCurrentTemperature() const { return currentTemperature; }

  // Called automatically by framework every tick.
  void tick();
  ...
private:
  // Variable storing last received temperature;
  int currentTemperature;
  ...
};
```

With the above, your `Presenters` are able to ask the model about the current temperature, allowing a Presenter to set this value in the UI (the View) when entering a screen that displays the temperature. What we need to do now is to be able to update the UI again when new temperature information is received. To do this we take advantage of the fact that the Model has a pointer to your currently active Presenter. The type of this pointer is an interface (`ModelListener`) which you can modify to reflect the application-specific events that are appropriate:

ModelListener.hpp

```
class ModelListener
{
public:
  // Call this function to notify that temperature has changed.
  // Per default, use an empty implementation so that only those
  // Presenters interested in this specific event need to
  // override this function.
  virtual void notifyTemperatureChanged(int newTemperature) {}
};
```

Now that we have this interface hooked up, the remaining this is to do the actual sampling of incoming "*new temperature*" events `Model::tick`

Model.cpp

```
void Model::tick()
{
  // Pseudo-code for sampling data
  if (OS_Poll(GuiTaskMBox))
  {
    // Here we assume that you have defined a "Message" struct containing type and data,
    // along with some event definitions.
    struct Message msg = OS_Read(GuiTaskMBox);
    if (msg.eventType == EVT_TEMP_CHANGED)
    {
       // We received information that temperature has changed.
       // First, update Model state variable
       currentTemperature = msg.data;

      // Second, notify the currently active Presenter that temperature has changed.
      // The modelListener pointer points to the currently active Presenter.
      if (modelListener != 0)
      {
        modelListener->notifyTemperatureChanged(currentTemperature);
      }
    }
  }
}
```

The approach above ensures two things:

1. The `currentTemperature` variable is always up-to-date so that your Presenter can at any time obtain the current temperature.
2. The `Presenter` is immediately notified of temperature changes and can take appropriate action.

One advantage of the MVP pattern is that you achieve a separated handling of notifications depending on what screen you are currently on. Assume for instance that a temperature changed event occurs while displaying some kind of settings menu *(e.g. MainMenuPresenter/MainMenuView is active)* where the current temperature is of no relevance.

Since the `notifyTemperatureChanged` function has a default empty implementation, this notification is simply disregarded by the `MainMenuPresenter`. On the other hand, if you have a `TemperatureControlPresenter` you can in this Presenter override the `notifyTemperatureChanged` function and inform the View that it should display an updated temperature:

TemperatureControlPresenter.hpp

```
class TemperatureControlPresenter : public ModelListener
{
public:
  // override the empty function.
```

```
  virtual void notifyTemperatureChanged(int newTemperature) {
    view.setTemp(newTemperature);
  }
};
```

The View class `TemperatureControlView`, must of course implement the `setTemp` method.

Transmitting Data from UI to Surrounding System

The reverse direction where data/events are transferred from the UI to the surrounding system, is done through the Model in much the same way. Continuing the example from before if we need to add the ability to configure a new target temperature, we would add the following to the Model:

Model.hpp

```
void setNewTargetTemperature(int newTargetTemp)
{
  // Pseudo-code for sending an event to a task responsible for controlling temperature.
  struct Message msg;
  msg.eventType = EVT_SET_TARGET_TEMP;
  msg.data = newTargetTemp;
  OS_Send(SystemTaskMBox, &msg);
}
```

In case the user sets a new target temperature in the UI, the View can inform the Presenter which holds a pointer to the Model object and is therefore able to call the `setNewTargetTemperature` function.

Examples

The following examples are full demos configured for specific demo boards, however much of the code demonstrated can be reused for other demo boards and custom hardware.

From GUI Task

A working example for STM32F746 showing how to sample a button and controlling a LED directly in the Model class. The example uses the MVP architecture to transfer values and events between the two views and the Model class. The Model class samples a button and updates the LED to match the state of the application.

A working example for STM32F429 showing how to sample a button in the Model class. The example uses the MVP architecture to transfer the button event to the View.

From Other Task

A working example for STM32F469 showing how to sample an analog input in separate thread. The example uses the MVP architecture to transfer the analog value to the View.

A working example showing intertask communication and propagation to and from the UI. Use this as inspiration for your own setup. The example communicates between the backend system implemented in C code and the C++ TouchGFX GUI. The example runs on the STM32F746G-DISCO board on top of FreeRTOS.

From Multiple tasks

This working example was demonstrated at the TouchGFX webinar "Integration with your hardware" from the 28th of May 2018.

The application was designed for the STM32F769-DISCO board and interacts with an LED and the USER BUTTON to show how to integrate both C code and hardware peripherals into your TouchGFX application.

The application configures the button in GPIO mode. Behavior is to sample the state of the button in btntask.c and pass a message through the GUI message queue if the button is pressed down. This allows us to advance the animation in the application by keeping the button pressed.

The application uses three FreeRTOS tasks. One for the GUI, one for each peripheral (LED and USER Button).

From Task and External Interrupt Line

This working example was demonstrated at the TouchGFX webinar "Integration with your hardware" from the 28th of May 2018.

The application was designed for the STM32F769-DISCO board and interacts with an LED and the USER BUTTON to show how to integrate both C code and hardware peripherals into your TouchGFX application.

This application configures the button in EXTI mode (external interrupt line 0). Behavior is to receive an interrupt when the button is pressed down after which the interrupt is cleared. This does not allow the same behavior as in GPIO, but instead we'll be single stepping the animation because a message is only sent through the gui message queue whenever an interrupt is received.

The application uses two FreeRTOS tasks. One for the GUI, one for the LED. (The Button task from Multiple tasks demo remains active in this application to exemplify that the peripheral interaction code has been moved into an interrupt handler).

Version: 4.16

Mixins

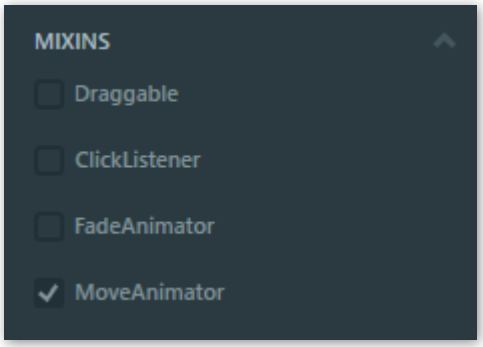
A Mixin is a class that extends the functionality of a widget to, for example, be able to animate movement or a change in their alpha value. The Move Animator and Fade Animator mixins are the basis of TouchGFX Designer Interactions being able to generate code that animates movement and alpha change. These mixins can be added to a widget either through TouchGFX Designer or manually in User Code.

Move Animator

The Move Animator mixin makes the widget capable of animating a movement from its current position to a specified end position. The movement in both the X and Y direction can be described by supplying EasingEquations.

In TouchGFX Designer, the mixin can be applied by enabling it in the properties for the given widget in the "Mixins" section, as shown in the image below.

The Move Animator mixin will automatically be applied to a widget if an Interaction that moves the widget has been created.



Move Animator mixin enabled in TouchGFX Designer

Enabling the Move Animator mixin changes the declaration signature of the generated widget as seen below, where a Box has had the Move Animator mixin enabled.

```
touchgfx::MoveAnimator< touchgfx::Box > box;
```

Using Move Animator in User Code

When a widget has had the Move Animator mixin applied to it, the widget now has the capability of animating its movement from one position to another. In this section a demonstration of how to use

this new functionality is shown.

After enabling the Move Animator mixin in TouchGFX Designer on a Box widget, the method `startMoveAnimation` becomes available for use. This methods takes five arguments in the following order

- endX: the X position relative to its parent that the widget should move to.
- endY: the Y position relative to its parent that the widget should move to.
- duration: the time in ticks the movement in the X and Y axis should take.
- xProgressionEquation: the EasingEquation that should be used for the movement in the X axis.
- yProgressionEquation: the EasingEquation that should be used for the movement in the Y axis.

Below an example of a movement to the coordinates X: 0, X: 0 over a duration of 40 ticks, using a linear EasingEquation in both X and Y axis.

```
    box.startMoveAnimation(0, 0, 40, EasingEquations::linearEaseNone, EasingEquations::lir
```

FURTHER READING

- **API Reference for the available EasingEquations**
- **Graphical demonstations of EasingEquations**

Callback Implementation in User Code

When a Move Animator mixin has completed an animation, a callback is emitted. In this section a demonstration of how to implement this callback is shown.

After enabling the Move Animator mixin in TouchGFX Designer on a Box widget, the next step is to add declarations for a callback and a function to handle the event in the Screen header class file that inherits from the base class where the Box widget is located.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    // Declaring callback handler for move animation ended on a Box
    void boxMoveAnimationEndedHandler(const touchgfx::MoveAnimator<Box>& comp);

protected:
    // Declaring callback type of MoveAnimator<Box>
    Callback <Screen1View, const touchgfx::MoveAnimator<Box>&> boxMoveAnimationEndedCallba
```

```
};
```

Then the callback declaration and function to handle the event need to be bound to the view object.

Screen1View.cpp

```
Screen1View::Screen1View() :
  // In constructor for callback, bind to this view object and bind which function to hand
  boxMoveAnimationEndedCallback(this, &Screen1View::boxMoveAnimationEndedHandler) { }
```

Next step is to tell the Box widget which callback to use when its move animation has ended, this is done in `setupScreen()` to ensure that the callback is set every time the screen is entered.

Screen1View.cpp

```
void Screen1View::setupScreen()
{
  // Add the callback to box
  box.setMoveAnimationEndedAction(boxMoveAnimationEndedCallback);
}
```

Last step is to implement the function, `boxMoveAnimationEndedHandler`, that handles the callback. For good practice we check that the Box which move animation has ended is actually the 'box'

Screen1View.cpp

```
void Screen1View::boxMoveAnimationEndedHandler(const touchgfx::MoveAnimator<touchgfx::Box>
{
    if (&b == &box)
    {
        //Implement what should happen when move animation on 'box' has ended here.
    }
}
```

API reference

FURTHER READING

- **API reference for the MoveAnimator class**

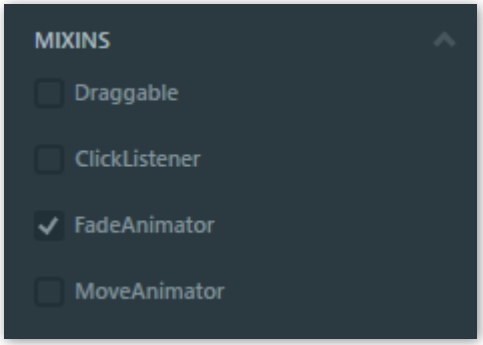
Fade Animator

The Fade Animator mixin makes the widget capable of animating its alpha value to fade from its current alpha value to a specified end alpha value. The rate of fading can be described by supplying an EasingEquation.

> **NOTE**
>
> **Only widgets that implement an alpha value support the Fade Animator mixin. This means all the widgets that can contain other widgets, like the Container, do not support the Fade Animator mixin.**

In TouchGFX Designer, the mixin can be applied by enabling it in the properties for the given widget in the "Mixins" section, as shown in the image below.

The Fade Animator mixin will also automatically be applied to a widget if an Interaction that fades the widget over a duration larger than zero has been added.



Fade Animator mixin enabled in TouchGFX Designer

Enabling the Fade Animator mixin changes the declaration signature of the generated widget as seen below, where a Box has had the Fade Animator mixin enabled.

```
touchgfx::FadeAnimator< touchgfx::Box > box;
```

Using Fade Animator in User Code

When a widget has had the Fade Animator mixin applied to it, the widget now has the capability of animating its alpha value from one setting to another. In this section a demonstration of how to use this new functionality is shown.

After enabling the Fade Animator mixin in TouchGFX Designer on a Box widget, the method `startFadeAnimation` becomes available for use. This method takes three arguments in the following order:

- endAlpha: the alpha value the widget should be when animation is completed.
- duration: the time in ticks the animation to the new alpha value setting should take.

- alphaProgressionEquation: the EasingEquation that should be used for the rate of change to the alpha value.

Below an example of an alpha value change to 0 over a duration of 40 ticks, using a linear EasingEquation.

```
    box.startFadeAnimation(0, 0, 40, EasingEquations::linearEaseNone);
```

FURTHER READING

- **API Reference for the available EasingEquations**
- **Graphical demonstations of EasingEquations**

Callback Implementation in User Code

When a Fade Animator mixin has completed an animation, a callback is emitted. In this section a demonstration of how to implement this callback is shown.

After enabling the Fade Animator mixin in TouchGFX Designer on a Box widget, the next step is to add declarations for a callback and a function to handle the event in the Screen header class file that inherits from the base class where the Box widget is located.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    // Declaring callback handler for fade animation ended on a Box
    void boxFadeAnimationEndedHandler(const touchgfx::FadeAnimator<Box>& comp);

protected:
    // Declaring callback type of FadeAnimator<Box>
    Callback <Screen1View, const touchgfx::FadeAnimator<Box>&> boxFadeAnimationEndedCallba
};
```

Then the callback declaration and function to handle the event need to be bound to the view object.

Screen1View.cpp

```
Screen1View::Screen1View() :
  // In constructor for callback, bind to this view object and bind which function to hand
  boxFadeAnimationEndedCallback(this, &Screen1View::boxFadeAnimationEndedHandler) { }
```

Next step is to tell the Box widget which callback to use when its move animation has ended, this is done in `setupScreen()` to ensure that the callback is set every time the screen is entered.

Screen1View.cpp

```
void Screen1View::setupScreen()
{
  // Add the callback to box
  box.setFadeAnimationEndedAction(boxFadeAnimationEndedCallback);
}
```

Last step is to implement the function, `boxFadeAnimationEndedHandler`, that handles the callback. For good practice we check that the Box which fade animation has ended is actually the 'box'

Screen1View.cpp

```
void Screen1View::boxFadeAnimationEndedHandler(const touchgfx::FadeAnimator<touchgfx::Box>
{
    if (&b == &box)
    {
        //Implement what should happen when fade animation on 'box' has ended here.
    }
}
```

API reference

FURTHER READING

- **API reference for the FadeAnimator class**

ClickListener

The Click Listener mixin makes the widget capable of responding to touch input by extending the widget with a callback.

In TouchGFX Designer, the mixin can be applied by enabling it in the properties for the given widget in the "Mixins" section, as shown in the image below.

Click Listener mixin enabled in TouchGFX Designer

Enabling the Click Listener mixin changes the declaration signature of the generated widget as seen below, where a Box has had the Click Listener mixin enabled.

```
touchgfx::ClickListener< touchgfx::Box > box;
```

Callback Implementation in User Code

When a Click Listener mixin receives a touch event, a callback is emitted. In this section a demonstration of how to implement this callback is shown.

After enabling the Click Listener mixin in TouchGFX Designer on a Box widget, the next step is to add declarations for a callback and a function to handle the event in the Screen header class file that inherits from the base class where the Box widget is located.

The callback should declare three things: which class type to bind to, which widget the callback originates from and the type of event that occurs. In this example it is `Screen1View`, `const Box&` and `const ClickEvent&`

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    // Declaring callback handler for Box
    void boxClickHandler(const Box& b, const ClickEvent& e);

protected:
    // Declaring callback type of box and clickEvent
    Callback<Screen1View, const Box&, const ClickEvent&> boxClickedCallback;
};
```

Then the callback declaration and function to handle the event need to be bound to the view object.

Screen1View.cpp

```
Screen1View::Screen1View() :
    // In constructor for callback, bind to this view object and bind which function to h
    boxClickedCallback(this, &Screen1View::boxClickHandler) { }
```

Next step is to tell the Box widget which callback to use when it is touched, this is done in `setupScreen()` to ensure that the callback is set every time the screen is entered.

Screen1View.cpp

```
void Screen1View::setupScreen()
{
    // Add the callback to box
    box.setClickAction(boxClickedCallback);
}
```

Last step is to implement the function, `boxClickHandler`, that handles the callback. For good practice we check that the Box which initiated the callback is actually the 'box'

Screen1View.cpp

```
void Screen1View::boxClickHandler(const Box& b, const ClickEvent& evt)
{
    if (&b == &box)
    {
        //Implement what should happen when 'box' is touched/clicked here.
    }
}
```

API reference

FURTHER READING

- **API reference for the ClickListener class**

Draggable

The Draggable mixin makes the widget capable of being dragged around via touch input.

In TouchGFX Designer, the mixin can be applied by enabling it in the properties for the given widget in the "Mixins" section, as shown in the image below.

Draggable mixin enabled in TouchGFX Designer

Enabling the Draggable mixin changes the declaration signature of the generated widget as seen below, where a Box has had the Click Listener mixin enabled.

```
touchgfx::Draggable< touchgfx::Box > box;
```

API reference

FURTHER READING

- **API reference for the Draggable class**

Version: 4.16

Texts and Fonts

Fonts and texts are a very important aspect of modern graphical user interfaces. It is important to be able to display high quality anti-aliased texts in all the languages that your application supports.

TouchGFX supports the creation and modification of texts and typographies through the Texts View of TouchGFX Designer. The TouchGFX Designer outputs text and typography configurations into a spreadsheet located at `assets/texts/texts.xlsx`. This spreadsheet, along with font files are fed to the font- and text-converter tools, producing generated C++ code files, that TouchGFX can render.

This article introduces the text and font converter tools and explains how to use the generated texts in an application through code and TouchGFX Designer.

Texts and Typographies

The texts, translations and typographies in a TouchGFX application are stored in the `assets/texts/texts.xlsx` spreadsheet. This spreadsheet consists of two sheets. One defining the typographies used in the application and one defining the texts and all their translations. This spreadsheet is commonly referred to as the "Text Database".

The typographies can be edited in TouchGFX Designer with the Texts View, which allows for real-time editing and handling of texts and translations. It is, however, possible to edit the typographies and texts directly in the texts.xlsx spreadsheet. This is in general not recommended, at least not during development. If using an external resource for translation it can however be easier to share the spreadsheet instead of requiring it to be done in TouchGFX Designer. If editing the spreadsheet during development, the TouchGFX Designer must be closed while editing as it locks the texts.xlsx file while open.

FURTHER READING

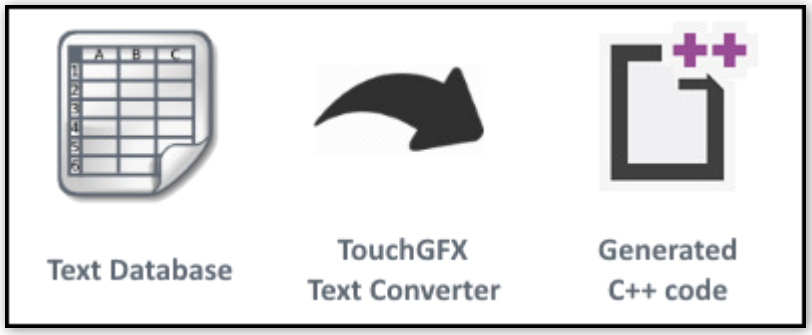
To learn more about how to create and edit typographies, texts, translations and languages go to **Texts View**

NOTE

For Glyph Bitmap Distribution Format fonts (.bdf), not all font sizes can be rendered with the font. If the given size in the typography sheet does not match with the given font, the font convert utility will report the supported font sizes. Updating the size in the Typography Sheet to one of the supported sizes will solve the problem.

The Text Converter

The text converter is the tool that converts the text information in the text database to an internal C++ format used by TouchGFX applications. The tool is an integrated part of the build tool-chain and will be executed automatically when building the simulator. The text converter is not executed if the text database has not been updated since the last build.



NOTE

The output directory of the text converter is `generated/texts/` .

The text converter converts all the texts specified in the text database into the text format used by TouchGFX. The format is wrapped in an object called `TypedText` . A `TypedText` in TouchGFX is a combined entity of the text contents itself and the typography of the text. The typography contains, the font and font size of the text and the bits per pixel (bpp) used in anti aliasing the glyphs of the font.

The text converter generates a file called `generated/texts/include/texts/TextKeysAndLanguages.hpp` . This file contains an enum `TEXTS` that references all texts in the text database.

Notice that all entries in the enum are generated from the text id stated in each row in the text database, but with a T_ prepended and converted to uppercase. These enum values are used in applications to initialize TypedTexts.

The `TextKeysAndLanguages.hpp` also contains an enum `LANGUAGES` that specifies all the languages that are present in the text database. The naming is the same as in the language column in the text database.

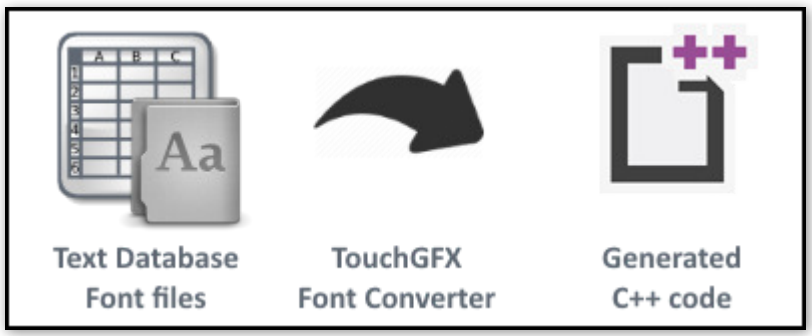
generated/texts/include/texts/TextKeysAndLanguages.hpp

```
/* DO NOT EDIT THIS FILE */
/* This file is autogenerated by the text-database code generator */

#ifndef TEXT_KEYS_AND_LANGUAGES_HPP
```

```
#define TEXT_KEYS_AND_LANGUAGES_HPP


typedef enum {
    GB,
    DE,
    NUMBER_OF_LANGUAGES
} LANGUAGES;


typedef enum {
    T_TEMPERATURE_READOUT,
    T_TEMPERATURE_HEADLINE,
    NUMBER_OF_TEXT_KEYS
} TEXTS;

#endif /* TEXT_KEYS_AND_LANGUAGES_HPP */
```

The Font Converter

The font converter is a tool that combines the information in font files with information in the text database and generates the characters needed by the application. The output format is an internal C++ format used by TouchGFX applications. The tool is an integrated part of the build tool-chain and will be executed automatically when building the simulator.



The Font Converter accepts

- TrueType (.ttf)
- OpenType (.otf)
- Glyph Bitmap Distribution Format (.bdf).

Simply place the font in the `assets/fonts/` folder and the font will be available for reference in TouchGFX Designer *(If the font is added while TouchGFX Designer is running, it must be restarted to update the available fonts)*.

In TouchGFX Designer, it is also possible to use the fonts installed in Windows, selecting any of these fonts will automatically add them to the `assets/fonts/` folder

The Font Converter supports kerning by using the kerning information in the supplied font.

> **NOTE**
>
> - **Using TouchGFX does not in any way provide licenses for commercial use of any TrueType, OpenType or Bitmap fonts.**
> - **The output directory of the font converter is generated/fonts/.**

Character Memory Optimization

TouchGFX is optimized for low memory consumption. By analysing the characters used for a specific typography, the number of generated characters (in internal C++ format) are minimized to the characters that are actually used by the application.

Text memory consumption is also optimized by compacting texts that use common suffixes by enabling the option to remap texts in the Text Configuration.

Wildcards

It is possible to use runtime values as part of texts. This is possible by use of wildcards in the texts. These are specified in the given format `<*>`, where the * represents an optional helping text which will not be included in the resulting text. It is possible to have up to two wildcards in one text.

All translations for a given text must contain the same number of wildcards. The wildcard values are inserted at runtime in the application C++ code.

Example of wildcard usage: **The temperature is <insert_temperature>°**

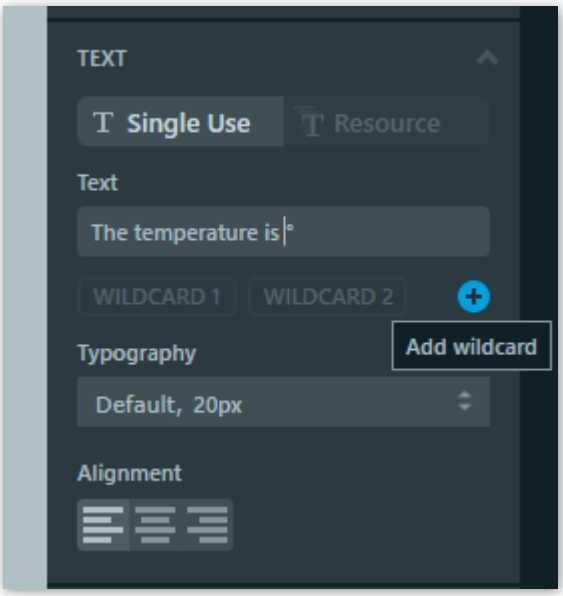
One detail to notice is that due to the character memory optimization (see section above) the only characters that are generated for a specific typography are the ones used in texts having this typography. To force the font generator to include certain characters, you can use "*Wildcard Characters*" and "*Character Ranges*" for each typography.

The wildcard format `<*>` can be escaped by using backslash notation like this: **\<not a wildcard\>**. This will result in the literal text "*<not a wildcard>*" being used in the application.

Using Wildcards in TouchGFX Designer

In TouchGFX Designer, wildcards can be added to regular TextAreas. Effectively this now makes the TextArea widget cover the functionality previously covered by the TextAreaWithOneWildcard/TextAreaWithTwoWildcards widgets, although there is no changes to how the code is generated in TouchGFX.

In TouchGFX Designer you can add Wildcards to TextAreas by either using the usual syntax `<*>`, or by simply clicking the *Add Wildcard* button in properties for the selected TextArea. A well-known example is adding a temperature reading to a TextArea, which could say *The temperature is °*. In this case it could be an outdoor temperature reading. Here we want to insert a Wildcard that not only displays a static number, but also updates according to temperature readings. The Wildcard will be added to the current position of the in-text caret:



Adding a wildcard to a Text Area widget

Now our text in properties will display *The temperature is <value>°*, while our text on canvas displays *The temperature is °*:

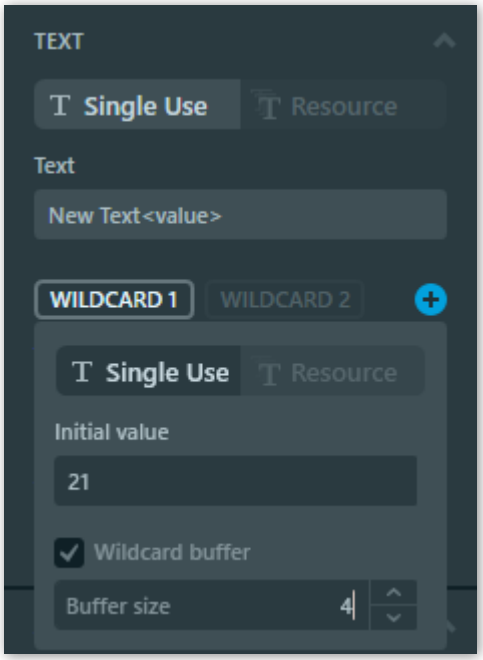
Text Area widget with wildcard in TouchGFX Designer

To set up the specific wildcard you can click the corresponding Wildcard button (in this example Wildcard 1), which allows editing the Wildcard we just added.

Here you can choose how you will update the wildcard. Either with predefined resource texts or by dynamic run-time created texts. In both cases, you can update the text at run-time. For the latter you

need a Wildcard buffer for storing the dynamic text. Such a buffer is created by selecting the *Wildcard Buffer* check mark. In this case you also need to specify a size (number of characters) of the buffer. If you want to be memory efficient, you need to match the specified size as closely as possible with your actual needed text size. Remember to add one extra space for the string termination ('\0').

You can also set an initial value for the Wildcard, enabling you to see how the final TextArea could look with a temperature reading. Setting an initial value will either create a hard coded Single Use text in the Text Database or if you have selected to use Wildcard buffer insert it into the Wildcard buffer:



Wildcard settings in TouchGFX Designer

Using Wildcards in User Code

Wildcards can also be added and updated via User Code as shown in the code example below, where a `Unicode::UnicodeChar` array is managed and updated.

gui/include/gui/some_screen/SomeView.hpp

```
#include <touchgfx/widgets/TextAreaWithWildcard.hpp>
...
class SomeView : public View<SomePresenter>
{
  TextAreaWithOneWildcard txt;
  Unicode::UnicodeChar txtBuffer[10];
}
```

gui/src/some_screen/SomeView.cpp

```
#include <texts/TextKeysAndLanguages.hpp>
#include <touchgfx/Color.hpp>
```

```
void SomeView::setupScreen()
{
  txt.setTypedText(TypedText(T_TEMPERATURE_READOUT));
  txt.setXY(10, 20);
  txt.setColor(Color::getColorFrom24BitRGB(0xFF, 0xFF, 0xFF))
  txt.setWildcard(txtBuffer);
  add(txt);

  updateTxt(5);
}

void SomeView::updateTxt(int newValue)
{
  Unicode::snprintf(txtBuffer, 10, "%d", newValue);
  txt.invalidate();
}
```

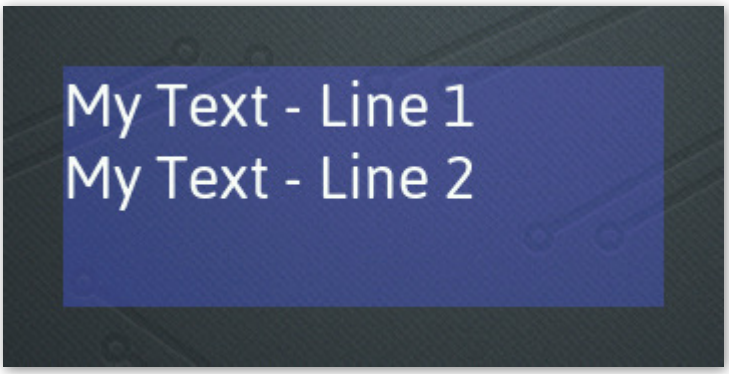
Text Placement

As for all TouchGFX widgets a TextArea is placed on the screen by specifying a position (X and Y) and a dimension (width and height). This is easily done via TouchGFX Designer in the widgets properties, However the rendering of text in TouchGFX Designer is not always 100% acurrate compared to how the text is rendered by TouchGFX.

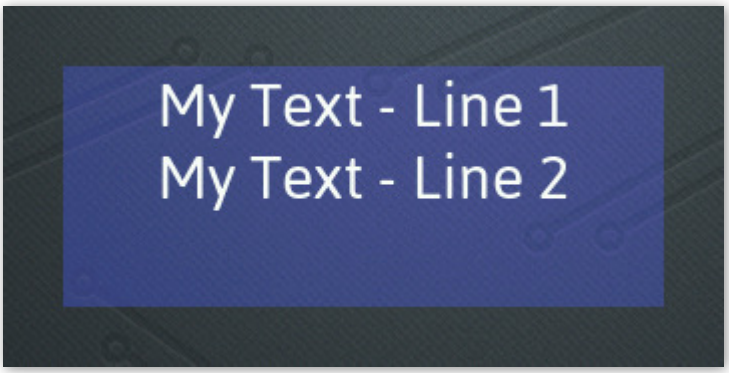
There are also a few more details and possibilites to be aware of when dealing with texts, decribed in this section.

Alignment

The text inside the TextArea is aligned according to the alignment specified for the chosen text entry in the text database. The text is aligned with respect to the area of the TextArea. In the following screenshots the area of the TextArea is highlighted in blue.



Left aligned text



Center aligned text

Right aligned text

These settings can be set in TouchGFX Designer Texts View.

Setting the Correct Width and Height of a TextArea

A TextArea is able to adjust its width and height according to the currently selected text. This is done by calling the `TextArea::resizeToCurrentText()` method.

> **NOTE**
>
> `resizeToCurrentText()` **is called automatically when instantiating a TextArea with a new TypedText if the width and height are not set.**

When using center/right aligned text you most often do not want to resize the width and height because your text needs to be centered/right aligned in a fixed area. In this case set the width and height manually. This can be done by calling `TextArea::setPosition(x, y, width, height)`, `TextArea::setWidth(width)` and `TextArea::setHeight(height)`.

If your width and/or height is too small to fit the text, the text will be clipped to the area as can be seen below.

Text cut off by the bounds of the TextArea widget size

Setting the Correct X and Y for a TextArea

To place a TextArea at the correct X and Y position, you need to be aware of the fact that the font used will have some extra spacing above the characters to allow for large characters. This makes it a bit hard to place a TextArea according to a Y position for the upper left corner, since you do not know the exact spacing above your text. One way of placing a text is to specify the position where you believe it should be and then fine tune the position by inspecting the placement in the simulator. This is most often a fairly simple task but it has to be redone if you change the font or font size later on.

A more robust way of doing it is to use text baseline. The baseline is the line upon which most letters "sit" and below which descenders (characters like p and j) extend.

Baseline for text

To set a text baseline use the `TextArea::setBaselineY(y)` or `TextArea::setXBaselineY(x, y)`. For these methods you do not specify the upper left corner of the TextArea but instead the baseline of the first text line. This will take the font size and spacing into account and set the Y position of the TextArea accordingly.

The baseline functionality is not available in TouchGFX Designer, since TextArea widget placement is easily done via TouchGFX Designer Canvas, and can therefore only be used in User Code.

NOTE

The TextArea needs to have its TypedText set before calling setBaselineY since it relies on the font. Also be aware that you need to call `setBaselineY` again if you change the TextAreas TypedText to one with a different font or font size.

Automatic Wrapping of Long Text Lines

Sometimes a TextArea needs to contain a text which is very long. By default, such text is simply written as a single line and all text that does not fit inside the TextArea is simpy cut off. If instead the text should be wrapped at spaces and re-flowed to fill several lines, simply call:

```
myTextArea.setWideTextAction(WIDE_TEXT_WORDWRAP); // Default is WIDE_TEXT_NONE
```

Available Wide Text Actions

- `WIDE_TEXT_NONE`: Do nothing, simply cut the text in the middle of any character that extends beyond the width of the TextArea.
- `WIDE_TEXT_WORDWRAP`: Wrap between words, ellipsis anywhere "Very long t...".
- `WIDE_TEXT_WORDWRAP_ELLIPSIS_AFTER_SPACE`: Wrap between words, ellipsis anywhere only after space "Very long ...".
- `WIDE_TEXT_CHARWRAP`: Wrap between any two characters, ellipsis anywhere, as used in Chinese.
- `WIDE_TEXT_CHARWRAP_DOUBLE_ELLIPSIS`:Wrap between any two characters, double ellipsis anywhere, as used in Chinese.

FURTHER READING

API Reference for the WideTextAction enum

This will probably make the TextArea need more vertical space. This can either be achieved by increasing the height of the TextArea in the Designer or it can be done in user code as follows.

```
myTextArea.setWidth(200);
myTextArea.resizeHeightToCurrentText(); // Will set height by wrapping text at 200px long
myTextArea.invalidate();
```

Remember to call `myTextArea.invalidate()` before resizing `myTextArea` if you are decreasing the text area size. If not, you will still see part of the old text area, since it is not covered by the new smaller text area.

Switching Language

TouchGFX supports multi language interfaces. The current language used in the interface can be changed by calling the static method `Texts::setLanguage`:

```
Texts::setLanguage(GB);
```

The value *GB* is found in the `LANGUAGES` enum in the `TextKeysAndLanguages.hpp` as shown in the example in The Text Converter section.

After this call, invalidate all widgets that display texts (or simply invalidate the entire screen) and they will display texts in the newly selected language.

In TouchGFX Designer

You can switch between languages, enabling testing for all translations. This is done from General section of the Config view. Here you simply change the startup language of the application by changing the *Selected Language*. Languages will need to be created and translated before they are selectable in the Config view.

Version: 4.16

Languages and Characters

TouchGFX enables internationalized and localized applications.

TouchGFX does this by supporting a wide range of languages and characters and by understanding text layout mechanisms, such as writing direction and contextual shaping.

Languages

The languages supported are the languages of the Unicode basic multilingual plane with the restriction that only Left-to-Right or Right-to-Left writing systems are supported. This implies that languages such as Arabic, Chinese, English and many more are supported, maybe with a few limitations. Urdu and Burmese are examples of languages that are currently not supported.

Characters

The encoding of characters is based on the Unicode standard. 16-bit unicodes are supported, i.e. Unicodes from 0x0000 to 0xFFFF are supported. Some languages may use the Private Use Area from 0xE000-0xE3FF for special characters in a given font (e.g. Devanagari).

Writing Direction

TouchGFX supports Left-to-Right (LTR) and Right-to-Left (RTL) writing systems. There is no built-in support for Top-to-Bottom writing systems.

It should be noted that RTL does not mean that text is written backwards (compared to LTR). It means that WORDS are written starting from the right towards the left. For Arabic and Hebrew, this is the correct setting. "TouchGFX" will not be written "XFGhcuoT" but the direction of words (or collection of workds) can be controlled using the RTL/LTR setting.

The handling of LTR and RTL writing inside TouchGFX applications respects mixing of the two to some degree. This is known as bidirectional script support. A subset of the official rules for bidirectional writing is supported by TouchGFX. This means that for example "10:45", "3.14159", "STMicroelectronics TouchGFX" and others are recognized and written fully LTR even in an RTL text.

For RTL text, some parts of the text must thus be written LTR. This text is found and collected; all characters that are non-RTL letters are collected. Characters such as color (:), dot (.), comma (,), space

( ) will also "tie together" two consecutive LTR parts. This is what makes sure that "10:45" is handled as a single LTR entity whereas "Mark:" (ending in a color) will get the colon to the left as Arabic and Hebrew speaking countries would expect, i.e. "<some Arabic message> :Mark" where the colon is on the left side in the RTL text.

Please note that numbers used in the Latin character set (0-9), as well as numbers used in the Arabic character set, are all handled as LTR characters to make sure that numbers show up properly on the display.

It should also be noted that the writing direction is very important when a text contains a mix of LTR entities and RTL entities. Also note, that it cannot be determined if a text is RTL or LTR by examining the characters in the text. If a text contains first a hebrew word (RTL) and then an english word (LTR), the output on display will depend on the writing direction of the text. If the text is set to be RTL the output would look something like this: "English werbeH" because the entire text is RTL so the first word must be written to the far right, but if the text is set to be LTR the output would look something like this: "werbeH English" because the text should start with the first word at the far left. The RTL versus LTR setting cannot be determined automatically because an English text may contain Hebrew words, just like a Hebrew text may contain English words.

Another important issue regarding RTL text is the automatic swapping of parenthesis characters. These are (,), {, }, [,], <, >. All these are automatically swapped with the opposite character to ensure that text looks correct. Please note that there is no automatic conversion from Latin numbers to Arabic numbers. This must be done by the user before displaying the text, should this be desired.

Contextual Shaping

Certain scripts will select a different form of one or more characters/glyphs depending on the context of the character. As an example the Arabic alphabet has different contextual forms for the letters in the alphabet, depending on the position of the letter inside the word. TouchGFX supports such contextual shaping of languages by implementing a simplied set of rules for combining characters. Also, some diacritics are placed using custom logic to determine the vertical position - this is particularly true for Arabic, Thai and Devanagari.

List of Supported Languages

It is difficult to provide an exhaustive list of all supported languages. In general, standard glyphs without special re-ordering or positioning rules are supported. Some languages, such as Hindi (Devanagari) and Arabic, with special rules have been included in TouchGFX.

Left-to-Right Languages

Simple languages using latin characters

In general, simple languages using characters and glyphs that do not require special re-ordering or positioning are supported. These languages include, but is not limited to, these:

- Bosnian, Bulgarian, Croatian, Czech, Danish, Dutch, English, Estonian, Finnish, French, German, Hungarian, Italian, Latvian, ithuanian, Norwegian, Polish, Portuguese, Romanian, Serbian, Slovenian, Slovak, Spanish, Swedish, Turkish, Ukrainian

Simple languages using special character sets

Some languages still follow simple positioning rules, but use a different set of characters and glyphs. These are also support and include, but is not limited to, these:

- Chinese, Greek, Japanese, Russian

Other

- Thai has limited support. Tone marks are positioned (vertically) using TouchGFX rules.
- Hindi (Devanagari) has limited support. Some characters may be placed slightly wrong, but text should not be unreadable.

Right-to-Left Languages

Simple languages using special character sets

- Hebrew, Indonesian, Khazakh

Languages with different ligatures for different forms (isolated, initial, middle, final)

- Arabic (Sequences of more than four characters are not recognized and converted to one ligature. These are: Sallallahou Alayhe Wasallam, Jallajalalouhou and Rial Sign). Some diacritics may be placed slightly incorrect.
- Farsi
- Malay (ڬ "Keheh with dot above" only supported in isolated form)

Unsupported Languages

The following languages are known to be unsupported because they rely on extensive use of ligatures, digraphs and vertical positioning:

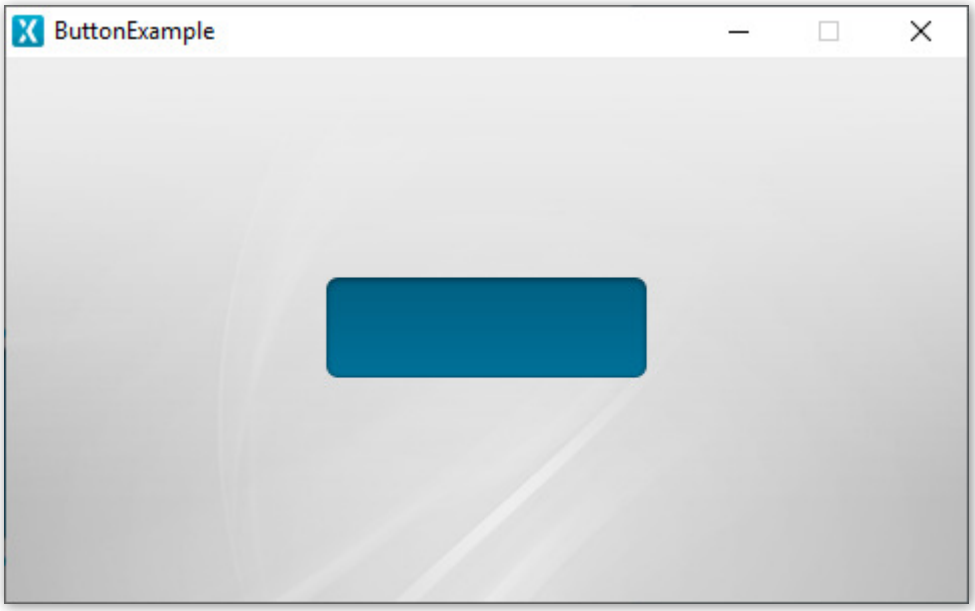
- Urdu, Burmese

Version: 4.16

Button

A Button in TouchGFX is a widget that is aware of touch events and can send a callback when the Button is released. Each state, pressed and released, is associated with an image.

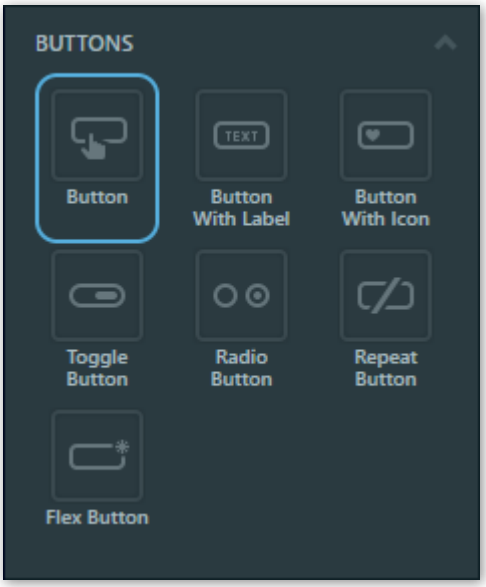
The Button can be replicated with the FlexButton. A FlexButton is a more configurable button that takes up a bit more RAM in exchange for flexibility.



Button running in the simulator (pressed state)

Widget Group

The Button can be found in the Buttons widget group in TouchGFX Designer.



Button in TouchGFX Designer

Properties

The properties for a Button in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a Button is determined by the size of the selected images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Released Image* and *Pressed Image* specify the images assigned to the pressed and released states from the Designer skin library or the Project folder.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by a Button are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Button is clicked	A button has been clicked.

Performance

A Button is composed of two images and is dependent on image drawing. Therefore, a Button is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Button.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase()
{
    buttonName.setXY(155, 106);
    buttonName.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID), touch

    add(buttonName);
```

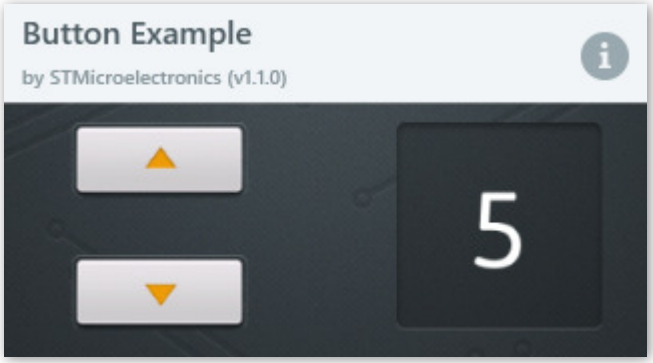
```
}

void Screen1ViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
{
    if (&src == &buttonName)
    {
        //Interaction name
        //When buttonName clicked calls the new virtual function "functionName()" set by t
        functionName();
    }
}
```

TIP

You can use these functions and the others available in the Button class in user code. Remember to force a redraw by calling `buttonName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the Button, try creating a new application within TouchGFX Designer with the following UI template:



Button Example UI template in TouchGFX Designer

To further explore the callback handler, most of the TouchGFX Designer examples use the Button for its trigger ability "button is clicked".

API Reference

FURTHER READING

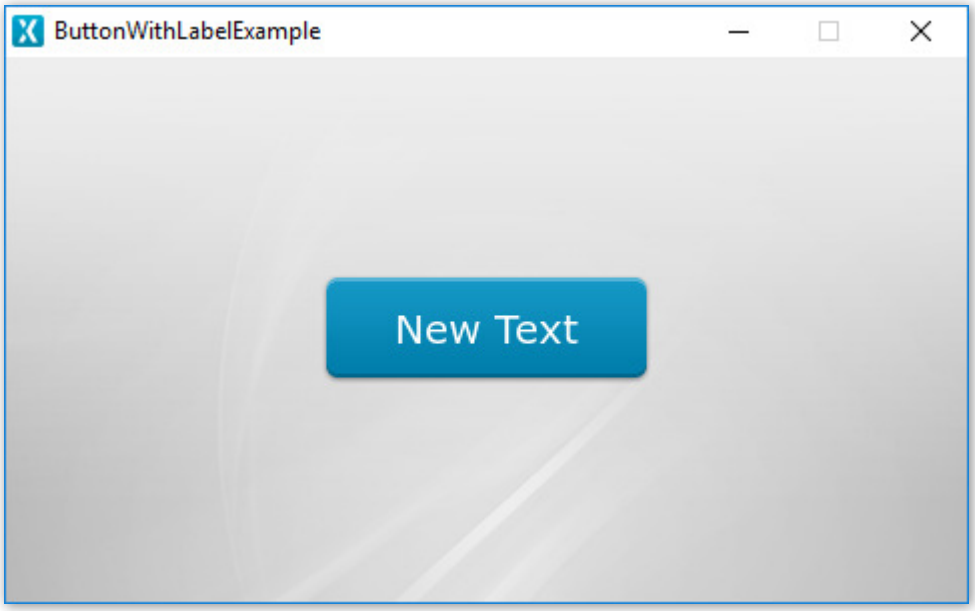
- **API reference for the Button class**

Version: 4.16

ButtonWithLabel

A ButtonWithLabel in TouchGFX is a widget that is aware of touch events and can send a callback when the ButtonWithLabel is released. Each state, pressed and released, is associated with an image and a text.

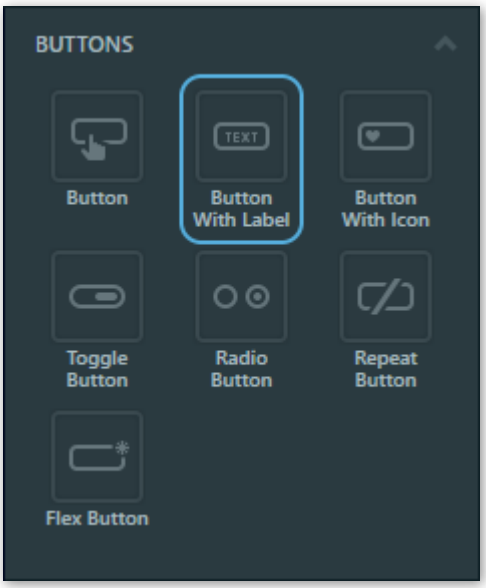
The ButtonWithLabel can be replicated with the FlexButton. A FlexButton is a more configurable button that takes up a bit more RAM in exchange for flexibility.



ButtonWithLabel running in the simulator (pressed state)

Widget Group

The ButtonWithLabel can be found in the Buttons widget group in TouchGFX Designer.



Properties

The properties for the ButtonWithLabel are described in the following sections.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a ButtonWithLabel is determined by the size of the selected images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Text	*Single Use* and *Ressource* specify the type of text: unique or from a known ressource. When Single Use is selected: *Text* specifies the content of the text to be displayed. *Typography* specifies the format of the text. *Alignment* specifies the horizontal alignment of the text relative to the widget. When Ressource is selected: *Ressource ID* specifies the ressource to retrieve the text from. For more details on text configuration, refer to the Using texts and fonts section.
Text Appearance	*Released Color* and *Pressed Color* specify the color of the Text in the pressed and released states. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.* *Text Rotation* specifies the angle in degrees of rotation of the text. There are four possible angles : 0, 90, 80 and 270 degrees.

Property Group	Property Descriptions
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use*.
Image	*Released Image* and *Pressed Image* specify the images assigned to the pressed and released states fron the Designer skin library or the Project folder.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the ButtonWithLabel are described in the following sections.

Actions

Specific widget action	Description
Set label color type	Set the color of the text.

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description

Trigger	Description
Button is clicked	A ButtonWithLabel has been clicked.

Performance

A ButtonWithLabel is composed of two images and text, and is dependent on image and text drawing. Text drawing is very similar to general image drawing (though due to the nature of text characters, a significant amount of alpha blending takes place). Therefore, the ButtonWithLabel is considered a fast widget on most platforms.

For more details on text drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ButtonWithLabel.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <texts/TextKeysAndLanguages.hpp>
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase() :
    buttonCallback(this, &Screen1ViewBase::buttonCallbackHandler)
{
    buttonWithLabelName.setXY(155, 106);
    buttonWithLabelName.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_
    buttonWithLabelName.setLabelText(touchgfx::TypedText(T_SINGLEUSEID1));
    buttonWithLabelName.setLabelColor(touchgfx::Color::getColorFrom24BitRGB(255, 255, 255)
    buttonWithLabelName.setLabelColorPressed(touchgfx::Color::getColorFrom24BitRGB(255, 25
    buttonWithLabelName.setLabelRotation(TEXT_ROTATE_0);
    buttonWithLabelName.setAction(buttonCallback);

    add(buttonWithLabelName);
}

void Screen1ViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
{
    if (&src == &buttonWithLabelName)
```

```
    {
        //InteractionName
        //When buttonName clicked calls the new virtual function "functionName()" set by t
        functionName();
    }
}
```

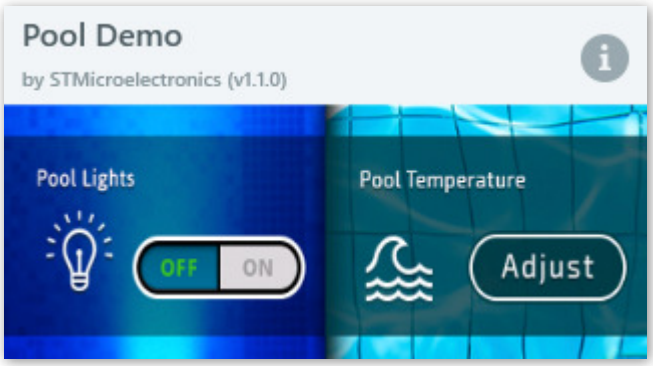
TIP

You can use these functions and the others available in the ButtonWithLabel class in user code. Remember to force a redraw by calling `buttonWithLabelName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ButtonWithLabel, try creating a new application within TouchGFX Designer with the following UI templates:

Custom Trigger Action Example UI template in TouchGFX Designer



Pool Demo UI template in TouchGFX Designer

API Reference

FURTHER READING

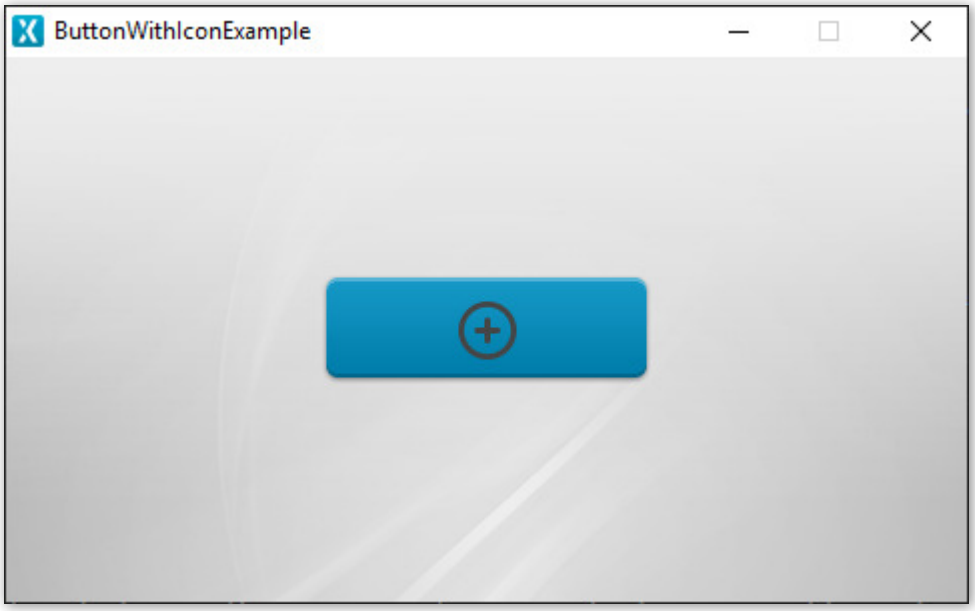
- **API reference for the ButtonWithLabel class**

Version: 4.16

ButtonWithIcon

A ButtonWithIcon in TouchGFX is a widget that is aware of touch events and can send a callback when the ButtonWithIcon is released. Each state, pressed and released, is associated with an image and an icon.

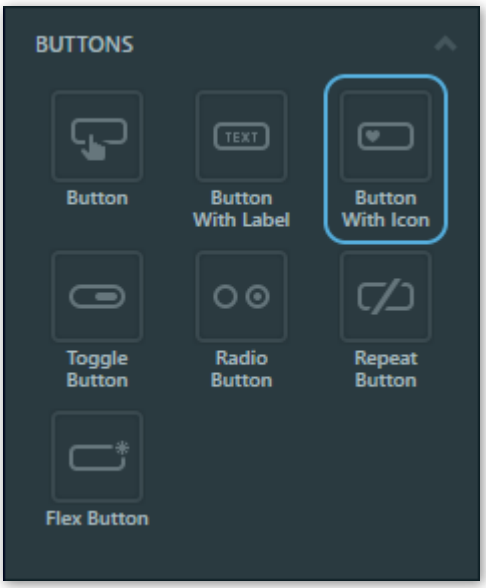
The ButtonWithIcon can be replicated with the FlexButton. A FlexButton is a more configurable button that takes up a bit more RAM in exchange for flexibility.



ButtonWithIcon running in the simulator

Widget Group

The ButtonWithIcon can be found in the Buttons widget group in TouchGFX Designer.



Properties

The properties for a ButtonWithIcon in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a ButtonWithIcon is determined by the size of the selected images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Image	*Button Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.* *Released Image* and *Pressed Image* specify the images assigned to the pressed and released states from the Designer skin library or the Project folder.
Icon	*Icon Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.* *Released Icon* and *Pressed Icon* specify the images assigned to the pressed and released states of the icon from the Designer skin library or the Project folder.
Icon Location	*X* and *Y* specify the top left corner of the icon relative to its parent. *W* and *H* specify the width and height of the icon, based on the selected image.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

Property Group	Property Descriptions
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the ButtonWithIcon are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Button is clicked	A ButtonWithIcon has been clicked.

Performance

The ButtonWithIcon is composed of four images and is dependent on image drawing. Therefore, a ButtonWithIcon is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ButtonWithIcon.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase() :
    buttonCallback(this, &Screen1ViewBase::buttonCallbackHandler)
{
    buttonWithIconName.setXY(155, 106);
    buttonWithIconName.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID
    buttonWithIconName.setIconXY(71, 16);
    buttonWithIconName.setAction(buttonCallback);

    add(buttonWithIconName);
}

void Screen1ViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
{
    if (&src == &buttonWithIconName)
    {
        //InteractionName
        //When buttonName clicked calls the new virtual function "functionName()" set by t
        functionName();
    }
}
```

TIP

You can use these functions and the others available in the ButtonWithIcon class in user code. Remember to force a redraw by calling `buttonWithIconName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ButtonWithIcon, try creating a new application within TouchGFX Designer with the following UI template:

Transition Example UI template in TouchGFX Designer

API Reference

FURTHER READING

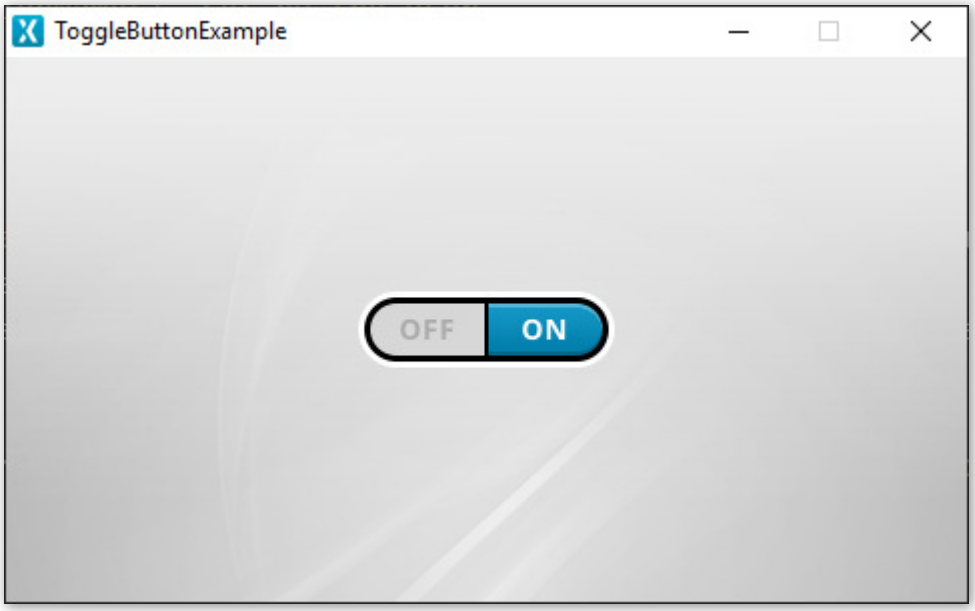
- **API reference for the ButtonWithIcon class**

Version: 4.16

ToggleButton

A ToggleButton in TouchGFX is a widget that is aware of touch events and can send a callback when the ToggleButton is clicked. Each state, pressed and released, is associated with an image. A ToggleButton is a Button specialization that swaps the two bitmaps when clicked to emulate switching between two states.

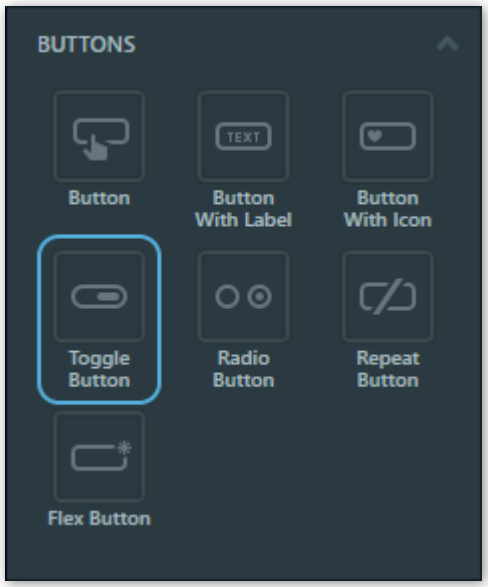
The ToggleButton can be replicated with the FlexButton. A FlexButton is a more configurable button that takes up a bit more RAM in exchange for flexibility.



ToggleButton running in the simulator

Widget Group

The ToggleButton can be found in the Buttons widget group in TouchGFX Designer.



ToggleButton in TouchGFX Designer

Properties

The properties for a ToggleButton in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a ToggleButton is determined by the size of the selected images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Released Image* and *Pressed Image* specify the images assigned to the pressed and released states from the Designer skin library or the Project folder.

Property Group	Property Descriptions
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the ToggleButton are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Button is clicked	A button has been clicked.

Performance

The ToggleButton is composed of two images and is dependent on image drawing. Therefore, a ToggleButton is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ToggleButton.

Screen1ViewBase.cpp

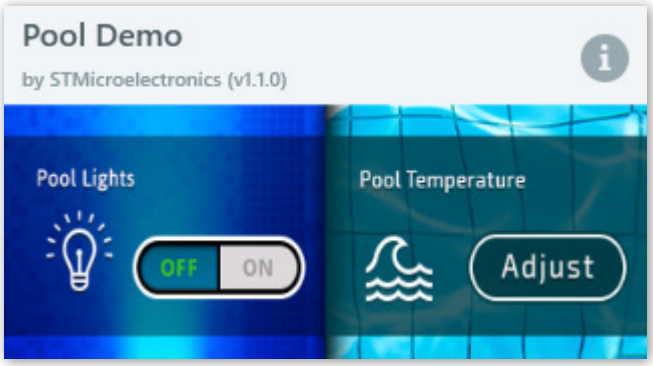
```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

creen1ViewBase::Screen1ViewBase() :
    buttonCallback(this, &Screen1ViewBase::buttonCallbackHandler)
{
    toggleButtonName.setXY(176, 117);
    toggleButtonName.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_TOGGLEBARS_TOGGLE_ROUND_LARGE
    toggleButtonName.setAction(buttonCallback);

    add(ToggleButtonName);
}

void Screen1ViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
{
    if (&src == &toggleButtonName)
    {
        //InteractionName
        //When buttonName clicked calls the new virtual function "functionName()" set by t
        functionName();
    }
}
```

TIP

You can use these functions and the others available in the ToggleButton class in user code. Remember to force a redraw by calling `toggleButtonName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ToggleButton, try creating a new application within TouchGFX Designer with the following UI template:



Pool Demo UI template in TouchGFX Designer

API Reference

FURTHER READING

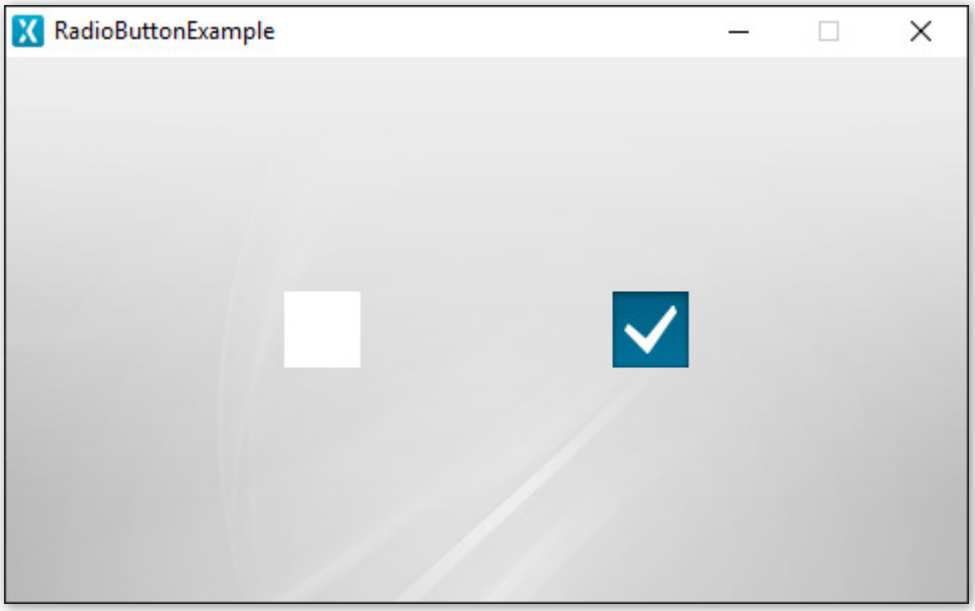
- **API reference for the ToggleButton class**

Version: 4.16

RadioButton

A RadioButton in TouchGFX is a widget that is aware of touch events and can send a callback when the RadioButton is clicked. A radio button consists of four images, corresponding to a selected or unselected button during a pressed or released state. RadioButtons can be added to a RadioButtonGroup which handles the deselection of radio buttons when a new selection is made.

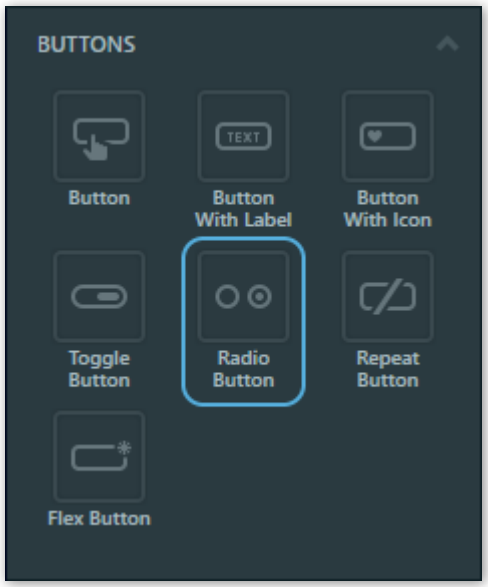
The RadioButton can be replicated with the FlexButton. A FlexButton is a more configurable button that takes up a bit more RAM in exchange for flexibility.



RadioButton running in the simulator

Widget Group

The RadioButton can be found in the Buttons widget group in TouchGFX Designer.



RadioButton in TouchGFX Designer

Properties

The properties for the RadioButton are described in the following sections.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a RadioButton is determined by the size of the selected images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Selection	*Selected* specifies the initial selection state of the button. *Deselectable* specifies the ability to deselect the button by pressing it while in the selected state.
Group	*Group* specifies the name of the group this button will be assigned to. Selection and deselection behaviour is contained within these RadioButtonGroups.

Property Group	Property Descriptions
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Released Image* and *Pressed Image* specify the images assigned to the pressed and released states from the Designer skin library or the Project folder.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the RadioButton are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Radio Button is selected	A RadioButton has been deselected.

Trigger	Description
Radio Button is deselected	A RadioButton has been selected.

Performance

The RadioButton is composed of four images and is dependent on image drawing. Therefore, a RadioButton is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a RadioButton.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

creen1ViewBase::Screen1ViewBase() :
    radioButtonSelectedCallback(this, &Screen1ViewBase::radioButtonSelectedCallbackHandler
{
    radioButtonName.setXY(136, 114);
    radioButtonName.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_CHECK_BUTTONS_CHECK_MARK_INACT
    radioButtonName.setSelected(false);
    radioButtonName.setDeselectionEnabled(true);

    add(radioButtonName);
    radioButtonGroupName.add(radioButtonName);

    radioButtonGroupName.setRadioButtonSelectedHandler(radioButtonSelectedCallback);
}

void Screen1ViewBase::radioButtonSelectedCallbackHandler(const touchgfx::AbstractButton& s
{
    if (&src == &radioButtonName)
    {
        //InteractionName
        //When buttonName clicked calls the new virtual function "functionName()" set by t
```

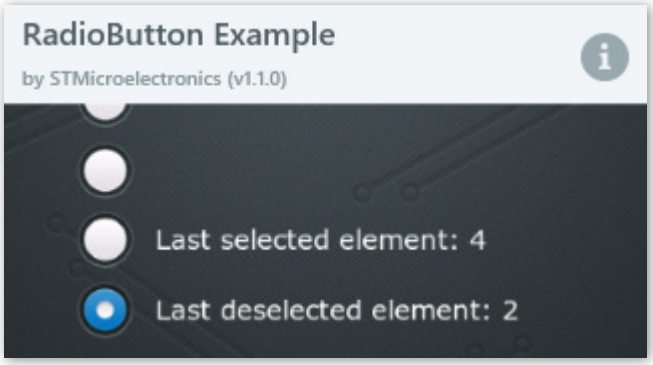
```
        functionName();
    }
}
```

TIP

You can use these functions and the others available in the RadioButton class in user code. Remember to force a redraw by calling `radioButtonName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the RadioButton, try creating a new application within TouchGFX Designer with the following UI template:



RadioButton Example UI template in TouchGFX Designer

API Reference

FURTHER READING

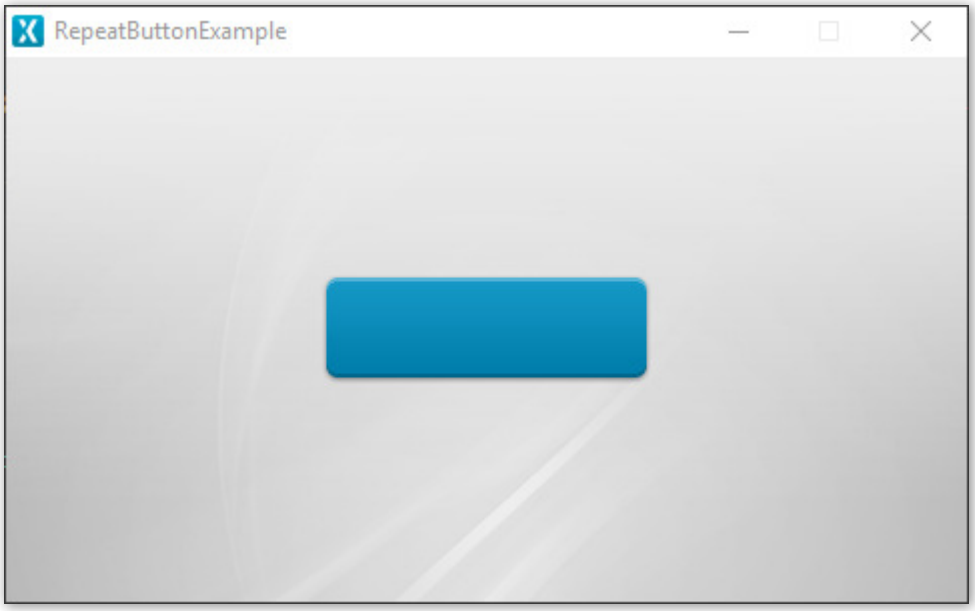
- **API reference for the RadioButton class**

Version: 4.16

RepeatButton

A RepeatButton in TouchGFX is a widget that is aware of touch events and can send a callback when the RepeatButton is pressed. The button activates its pressed action immediately, then after a given delay, then repeatedly after an interval. Each state, Pressed and Released, is associated with an image.

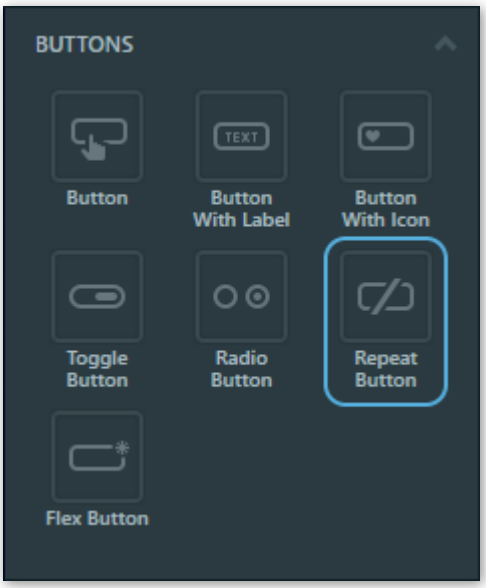
The RepeatButton can be replicated with the FlexButton. A FlexButton is a more configurable button that takes up a bit more RAM in exchange for flexibility.



RepeatButton running in the simulator

Widget Group

The RepeatButton can be found in the Buttons widget group in TouchGFX Designer.



Properties

The properties for a RepeatButton in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a RepeatButton is determined by the size of the selected images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Released Image* and *Pressed Image* specify the images assigned to the Pressed and Release states from the Designer skin library or the Project folder.
Settings	*Delay* specifies the time (ms) to wait when the button is pressed before starting the loop of triggers. *Interval* specifies the time (ms) in between every trigger. *Designer accepts inputs of milliseconds and converts them into ticks.*
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the RepeatButton are described in the following sections.

Actions

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Button is clicked	A button has been clicked.

Performance

A RepeatButton is composed of two images and is dependent on image drawing. Therefore, a RepeatButton is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a RepeatButton.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase() :
    buttonCallback(this, &Screen1ViewBase::buttonCallbackHandler)
{
    repeatButtonName.setXY(155, 106);
    repeatButtonName.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_BUTTONS_ROUND_EDGE_SMALL_ID),
    repeatButtonName.setDelay(12);                    // Set at 200 (ms) in Designer
    repeatButtonName.setInterval(20);                 // Set at 333 (ms) in Designer
    repeatButtonName.setAction(buttonCallback);

    add(repeatButtonName);
}

void Screen1ViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
{
    if (&src == &repeatButtonName)
    {
        //InteractionName
        //When repeatButtonName clicked calls the new virtual function "functionName()" se
        functionName();
    }
}
```

TIP

You can use these functions and the others available in the RepeatButton class in user code. Remember to force a redraw by calling `repeatButtonName.invalidate()` if you change the appearance of the widget.

API Reference

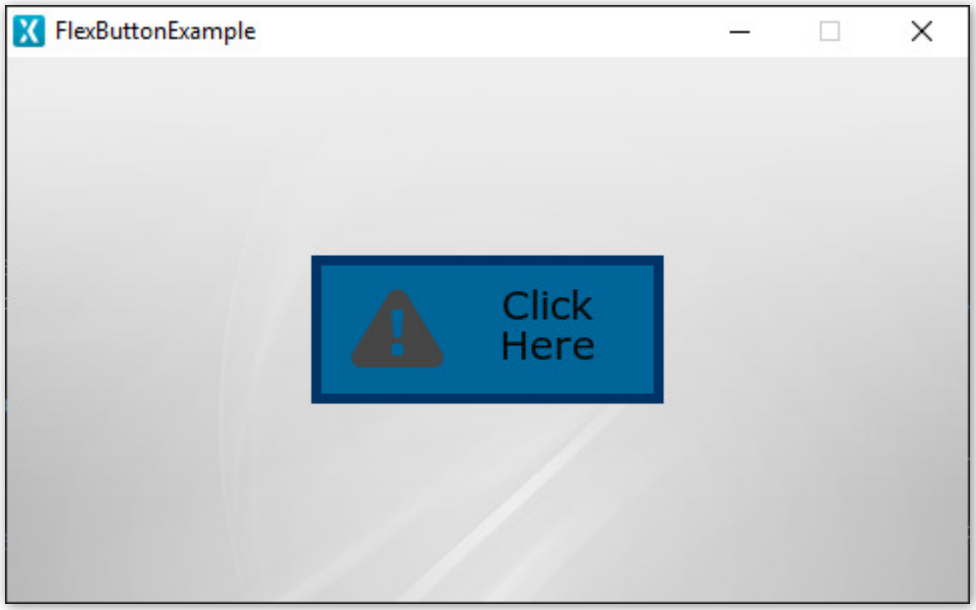
FURTHER READING

- **API reference for the RepeatButton class**

Version: 4.16

FlexButton

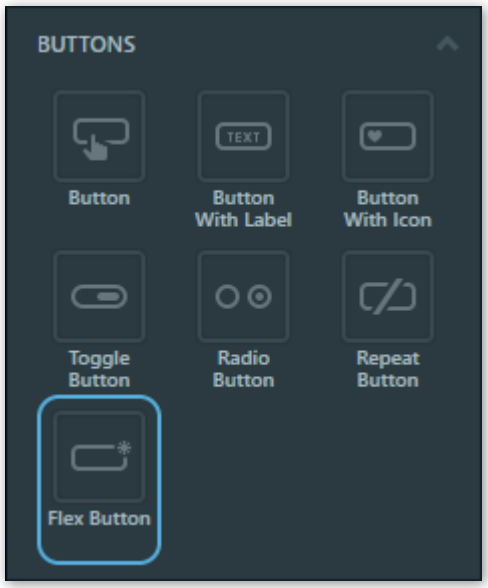
A FlexButton in TouchGFX is a widget that is aware of touch events and can send a callback when the FlexButton is triggered. The FlexButton is adaptable to the needs of the user. It can combine the behaviour and appearance of other button types but takes up a bit more RAM as a tradeoff. This will, however, in most cases be an insignificant amount. The FlexButton can be composed of a maximum of 4 visual elements: BoxWithBorder, Icon, Text and Image.



FlexButton running in the simulator (combining BoxWithBorder, Icon and Text elements)

Widget Group

The FlexButton can be found in the Buttons widget group in TouchGFX Designer.



FlexButton in TouchGFX Designer

Properties

The properties for a FlexButton in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Trigger	*Click* , *Touch* , *Toggle* and *Repeat* specify which action triggers the button callback.
Visual Elements	*Image* , *Box With Border* , *Text* and *Icon* specify which elements make up the widgets visual appearance.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the FlexButton are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Button is clicked	A button has been clicked.

Performance

A FlexButton is potentially composed of up to two Boxes, four Images and one Text, and relies on image and text drawing. Text drawing is very similar to general image drawing (though due to the nature of text characters, a significant amount of alpha blending takes place). Therefore, the FlexButton is considered a fast widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a FlexButton. The code corresponds to the FlexButton shown at the start of this section, combining the behavior and appearance of the BoxWithBorder, Icon and Text elements.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase() :
    buttonCallback(this, &Screen1ViewBase::buttonCallbackHandler)
```

```
{
                            // Box with Border behavior and appearance
    flexButtonName.setBoxWithBorderPosition(0, 0, 176, 74);
    flexButtonName.setBorderSize(5);
    flexButtonName.setBoxWithBorderColors(touchgfx::Color::getColorFrom24BitRGB(0, 102, 15
                            // Text behavior and appearance
    flexButtonName.setText(TypedText(T_SINGLEUSEID1));
    flexButtonName.setTextPosition(30, 12, 176, 74);
    flexButtonName.setTextColors(touchgfx::Color::getColorFrom24BitRGB(10, 10, 10), touchg
                            // Icon behavior and appearance
    flexButtonName.setIconBitmaps(Bitmap(BITMAP_BLUE_ICONS_ALERT_32_ID), Bitmap(BITMAP_BLU
    flexButtonName.setIconXY(20, 17);
                            // Widget
    flexButtonName.setPosition(152, 99, 176, 74);
    flexButtonName.setAction(flexButtonCallback);

    add(flexButtonName);
}

void Screen1ViewBase::flexButtonCallbackHandler(const touchgfx::AbstractButtonContainer& s
{
    if (&src == &flexButtonName)
    {
        //InteractionName
        //When FlexButtonName clicked calls the new virtual function "functionName()" set
        functionName();
    }
}
```

TIP

You can use these functions and the others available in the FlexButton class in user code. Remember to force a redraw by calling `flexButtonName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the FlexButton, try creating a new application within TouchGFX Designer with the following UI template:

FlexButton Example UI template in TouchGFX Designer

API Reference

FURTHER READING

- API reference for the AbstractButton class

Version: 4.16

Image

An Image in TouchGFX draws the pixel data from an associated image file. The image file must be imported into the project before usage.

The size of an Image is defined by the associated image file and cannot be altered at runtime. If you need the image shown to be of a different size you need to resize the associated imported image. This is due to performance reasons.

If you need to resize an image at runtime use ScalableImage. Note that the performance of drawing a scaled image is much lower than a non-scaled image.

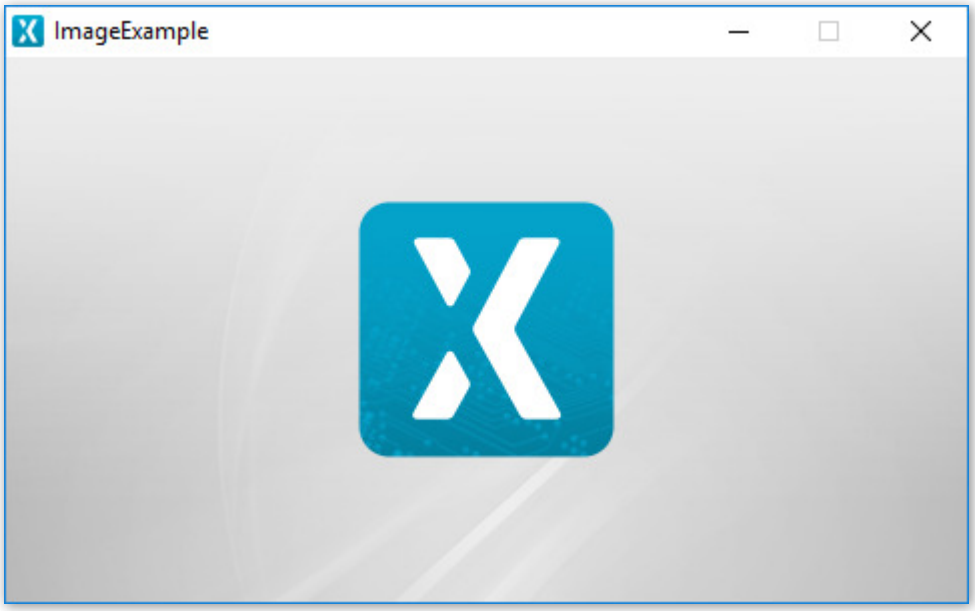


Image running in the simulator

Widget Group

The Image can be found in the Images widget group in TouchGFX Designer.

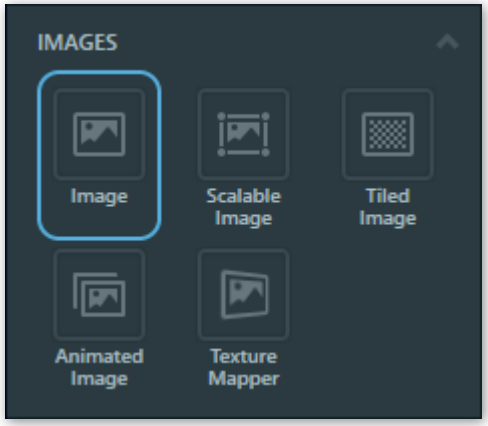


Image in TouchGFX Designer

Properties

The properties for a Image in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. The size of the widget is determined by the size of the associated image. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Image* specifies the associated image. Select either from the imported images in the Project tab or from the set of free TouchGFX images in the Skins tab.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixin	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by an Image in TouchGFX Designer.

Actions

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

An Image does not emit any triggers.

Performance

The Image is dependent on image drawing and is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the view base class we can see how TouchGFX Designer sets up an Image.

Screen1ViewBase.cpp

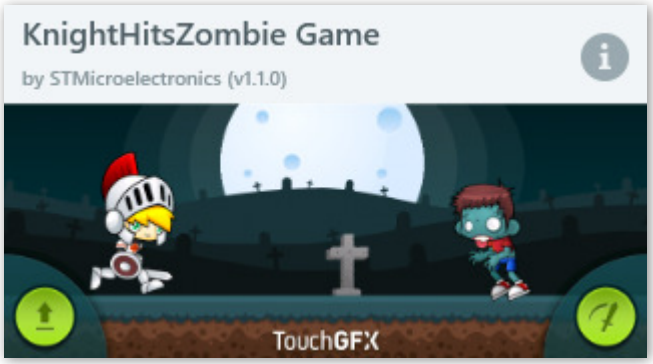
```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase()
{
    imageName.setXY(0, 0);
    imageName.setBitmap(Bitmap(BITMAP_STM32_LOGO_ID));

    add(imageName);
}
```

TIP

- You can use these functions and the others available in the Image class in user code. Remember to force a redraw by calling `imageName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the Image, try creating a new application within TouchGFX Designer with one of the following UI templates:



KnightHitsZombie Game UI template in TouchGFX Designer

API Reference

FURTHER READING

- **API reference for the Image class**

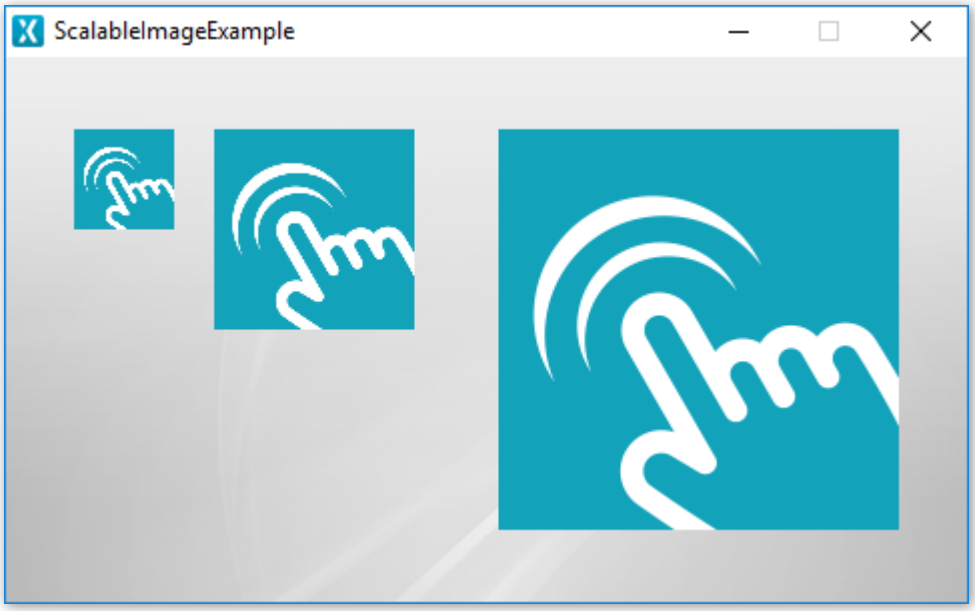
Version: 4.16

ScalableImage

ScalableImage is a widget capable of drawing a scaled version of a bitmap. Simply change the width/height of the widget to resize the image. The quality of the scaled image depends of the rendering algorithm used. The rendering algorithm can be changed dynamically.

NOTE

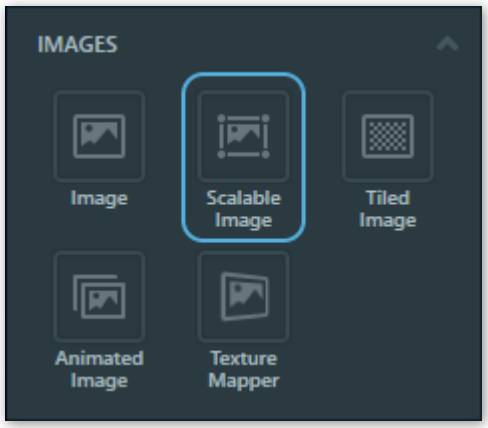
- **This widget has a significant effect on the MCU load.**
- **This widget does not support 1 bit per pixel color depth.**



ScalableImage running in the simulator

Widget Group

The ScalableImage can be found in the Images widget group in TouchGFX Designer.



ScalableImage in TouchGFX Designer

Properties

The properties for a ScalableImage in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Scaling Algorithm* specifies the algorithm used for scaling the chosen image. *Image* specifies which image the widget should use.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by a ScalableImage in TouchGFX Designer.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A ScalableImage does not emit any triggers.

Performance

A ScalableImage heavily depends upon the MCU for scaling the image. Therefore, the ScalableImage is considered a demanding widget on most platforms.

For more details on drawing performance, read the read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how the Designer sets up a ScalableImage.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase()
{
    scalableImage.setBitmap(touchgfx::Bitmap(BITMAP_IMAGE_ID));
    scalableImage.setPosition(246, 36, 200, 200);
    scalableImage.setScalingAlgorithm(touchgfx::ScalableImage::NEAREST_NEIGHBOR);

    add(scalableImage);
}

void Screen1ViewBase::setupScreen()
```

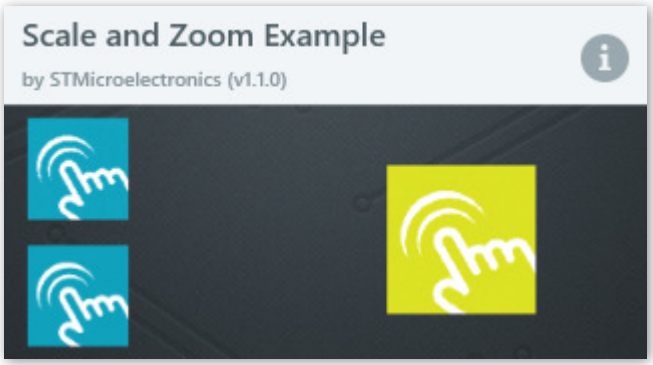
```
{

}
```

TIP

You can use these functions and the others available in the ScalableImage class in user code. Remember to force a redraw by calling `scalableImage.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ScalableImage, try creating a new application within TouchGFX Designer with one of the following UI templates:



Scale and Zoom Example UI template in TouchGFX Designer

API Reference

FURTHER READING

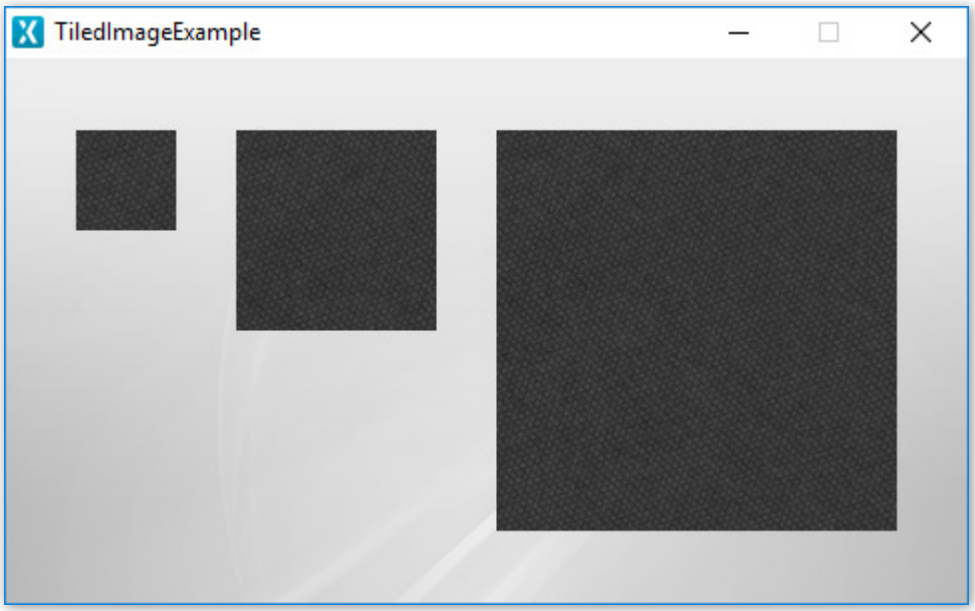
- **API reference for the ScalableImage class**

Version: 4.16

TiledImage

Description

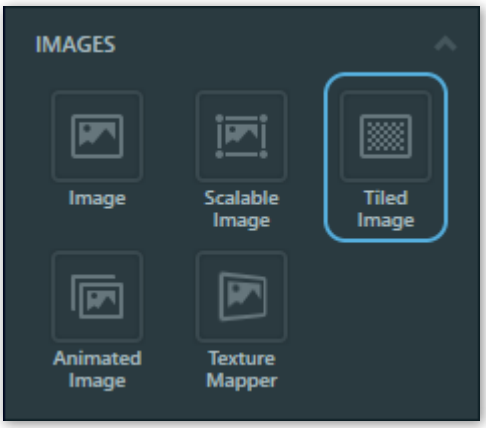
A TiledImage is a simple widget capable of showing a tiled bitmap. This means that when TiledImage is larger than the provided bitmap, the bitmap is repeated horizontally and vertically. The bitmap can be alpha-blended with the background and have areas of transparency.



TiledImage running in the simulator

Widget Group

The TiledImage can be found in the Images widget group in TouchGFX Designer.



TiledImage in TouchGFX Designer

Properties

The properties for a TiledImage in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Image* Specifies the image that should be used within the widget. *An image with a repeating pattern is recommended.*
Offset	*X* and *Y* specify the offset of the image where the tile drawing should start.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the TiledImage are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A TiledImage does not emit any triggers.

Performance

A TiledImage is dependent on image drawing, and is considered a fast performing widget on most platforms.
A TiledImage redraws the same image multiple times to cover the area of the widget. Therefore, small source images result in a greater number of image draws.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a TiledImage.

mainViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"

mainViewBase::mainViewBase()
{

    tiledImage.setBitmap(touchgfx::Bitmap(BITMAP_BLUE_TEXTURES_IRONGRIP_ID));
    tiledImage.setPosition(35, 36, 50, 50);
    tiledImage.setOffset(0, 0);
```

```
    add(tiledImage);
}

void mainViewBase::setupScreen()
{

}
```

> **TIP**
>
> You can use these functions and the others available in the TiledImage class in user code. Remember to force a redraw by calling `tiledImage.invalidate()` if you change the appearance of the widget.

User Code

The following code example shows how to animate movement into a TiledImage by continuously adjusting the offset in the `handleTickEvent()`:

mainView.cpp

```
#include <gui/main_screen/mainView.hpp>

mainView::mainView()
{

}

void mainView::setupScreen()
{
    mainViewBase::setupScreen();
}

void mainView::tearDownScreen()
{
    mainViewBase::tearDownScreen();
}

void mainView::handleTickEvent()
{
    int x = tiledImage.getXOffset();
    int y = tiledImage.getYOffset();
    tiledImage.setOffset(x + 1, y + 1);
    tiledImage.invalidate();
}
```

API Reference

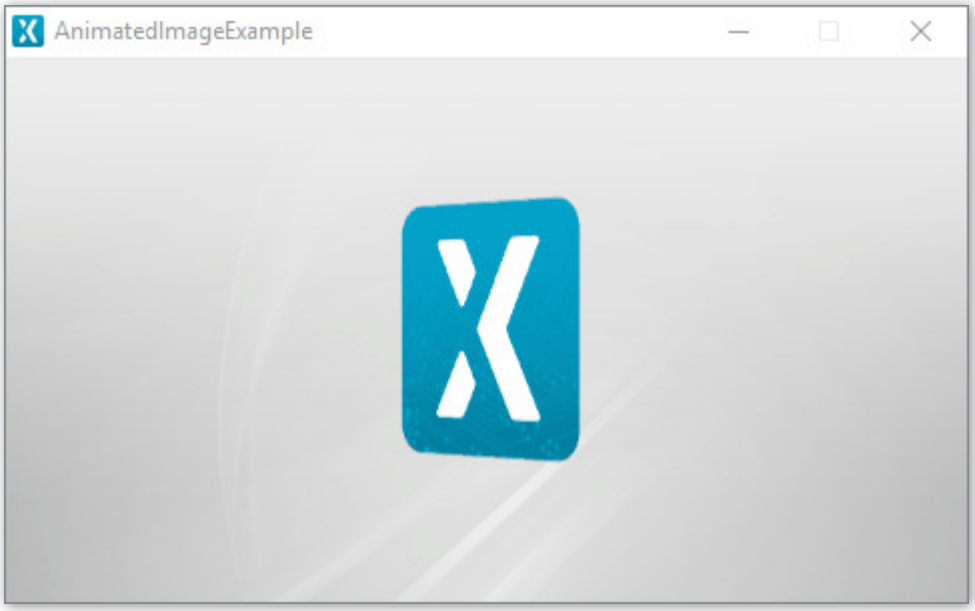
FURTHER READING

- API reference for the TiledImage class

Version: 4.16

AnimatedImage

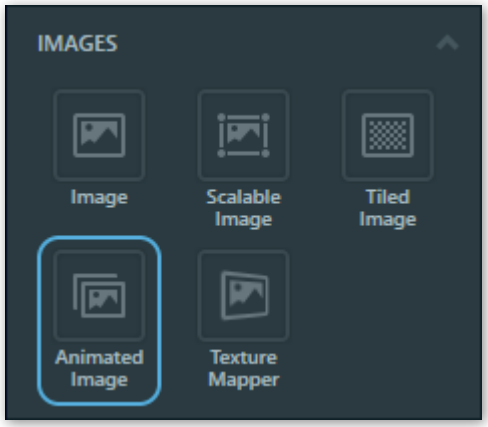
An AnimatedImage is capable of running an animation from start to end using a range of images sharing a common identifier. It is capable doing a single animation or looping the animation until stopped or paused.



AnimatedImage running in the simulator

Widget Group

The AnimatedImage can be found in the Images widget group in TouchGFX Designer.



AnimatedImage in TouchGFX Designer

Properties

The properties for a AnimatedImage in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a AnimatedImage is taken from the size of the associated images and cannot be altered except by changing the images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Image	*First Image* and *Last Image* specify the first and last images in the range of images used for the animation. *The images used must have an identifier e.g. img_01.png, img_02.png, img_03.png, img_04.png, img_05.png, img_06.png, img_07.png, etc.*
Animation	*Start on load* specifies if the animation should start as soon as the screen is loaded. *Reverse Animation* specifies if the images used for the animation should be run in reverse order. *Loop Animation* specifies if the animation should run continuously. *Update Interval* specifies the the amount of time that will pass between each image in the animation.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the AnimatedImage are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Animation is done	An AnimatedImage has completed its animation.

Performance

An AnimatedImage is dependent on image drawing, and is considered a fast performing widget on most platforms.
An AnimatedImage draws images according to the *Update Interval*. Therefore, a lower *Update Interval* results in more image draws.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up an AnimatedImage.

mainViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"
```

```
mainViewBase::mainViewBase()
{

    image.setXY(0, 0);
    image.setBitmap(touchgfx::Bitmap(BITMAP_BLUE_BACKGROUNDS_MAIN_BG_TEXTURE_480X272PX_ID)

    animatedImage.setXY(0, -104);
    animatedImage.setBitmaps(BITMAP_BUTTERFLY_01_ID, BITMAP_BUTTERFLY_72_ID);
    animatedImage.setUpdateTicksInterval(2);
    animatedImage.startAnimation(false, true, true);

    add(image);
    add(animatedImage);
}

void mainViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the AnimatedImage class in user code. Remember to force a redraw by calling `animatedImage.invalidate()` if you change the appearance of the widget.

User Code

The following code example shows how to set up the callback of an AnimatedImage when an animation is done:

mainView.hpp

```
#ifndef MAINVIEW_HPP
#define MAINVIEW_HPP

#include <gui_generated/main_screen/mainViewBase.hpp>
#include <gui/main_screen/mainPresenter.hpp>

class mainView : public mainViewBase
{
public:
    mainView();
    virtual ~mainView() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
protected:
    /*
     * Callback Declarations
     */
```

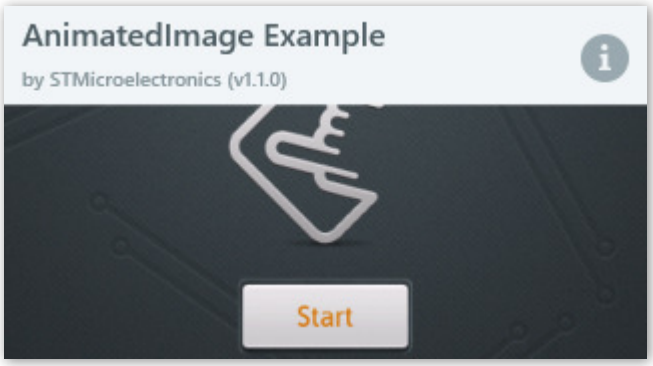
```
    touchgfx::Callback<mainView, const touchgfx::AnimatedImage&> animatedImageAnimationDon

    /*
     * Callback Handler Declarations
     */
    void animatedImageAnimationDoneCallbackHandler(const touchgfx::AnimatedImage& src);
};

#endif // MAINVIEW_HPP
```

mainView.cpp

```
#include <gui/main_screen/mainView.hpp>

mainView::mainView():
    animatedImageAnimationDoneCallback(this, &mainView::animatedImageAnimationDoneCallback
{

}

void mainView::setupScreen()
{
    mainViewBase::setupScreen();
    animatedImage.setDoneAction(animatedImageAnimationDoneCallback);
}

void mainView::tearDownScreen()
{
    mainViewBase::tearDownScreen();
}

void mainView::animatedImageAnimationDoneCallbackHandler(const touchgfx::AnimatedImage& sr
{
    if (&src == &animatedImage)
    {
        //execute code whenever the animation of animatedImage stops
    }
}
```

TouchGFX Designer Examples

To further explore the AnimatedImage, try creating a new application within TouchGFX Designer with one of the following UI templates:



AnimatedImage Example UI template in TouchGFX Designer

API Reference

FURTHER READING

- **API reference for the AnimatedImage class**

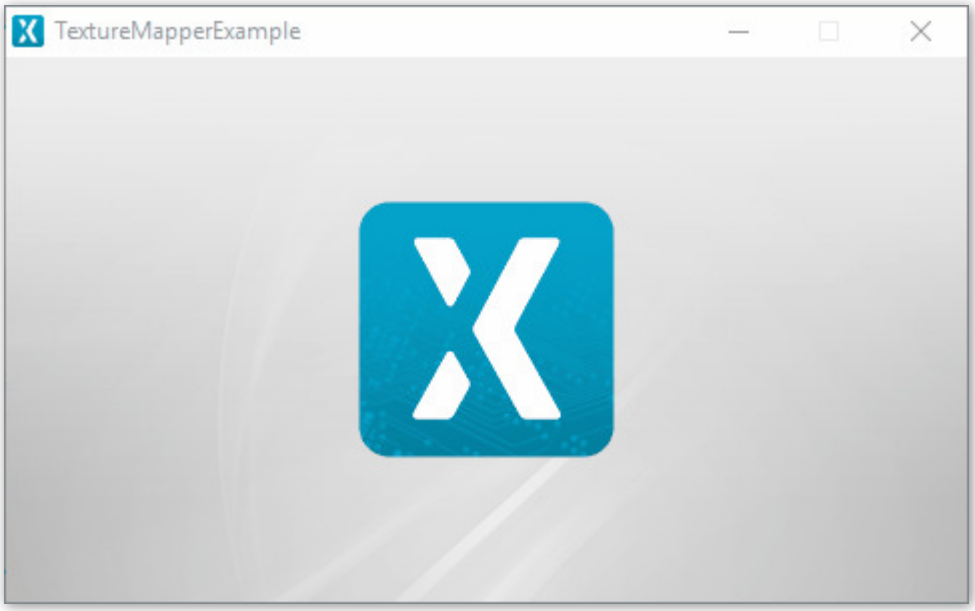
Version: 4.16

TextureMapper

A TextureMapper is a widget capable of drawing a transformed image, that can be freely scaled and rotated around an adjustable origin. Perspective impression is also achieved by applying a virtual camera, where the amount of perspective is adjustable.

NOTE

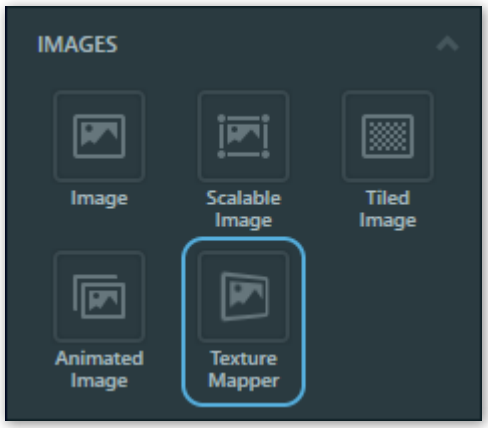
- **This widget has a significant effect on the MCU load.**
- **This widget does not support 1 bit per pixel color depth.**



TextureMapper running in the simulator

Widget Group

The TextureMapper can be found in the Images widget group in TouchGFX Designer.



TextureMapper in TouchGFX Designer

Properties

The properties for a TextureMapper in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen*. *Animation Texture Mapper* specifies if the TextureMapper should be generated as an AnimationTextureMapper.
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use*.
Image	*Image* specifies the image that should be transformed. *Lock Image to Center* specifies if the image position should be locked to the center of the widget. *If the Texture Mapper is resized at run time, this option does not maintain a centered position for the image.*. *X* and *Y* specify the top left corner of the image to be transformed within the widget.
Angle & Scale	*X Angle*, *Y Angle* and *Z Angle* specify the rotation transformation of the image within the widget. *Angles are in radians.* *Scale* specifies the scale transformation of the image in the widget.

Property Group	Property Descriptions
Origo	*Lock Origo to Center* specifies if the rotation point of the image is locked to the center of the widget. *If the Texture Mapper is resized at run time, this option does not maintain a centered origo position*. *X Origo*, *Y Origo* and *Z Origo* specify the point at which the image within the widget be rotated and scaled. For more details on the intricacies of this, refer to the Origo & Camera section.
Camera	*Camera Distance* specifies the distance of the virtual camera. *This changes the amount of perspective when the image is rotated.*
Appearance	*Rendering Algorithm* specifies the algorithm used to render the image within the widget. *The options are Nearest-neighbour and Bilinear Interpolation*. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Origo & Camera

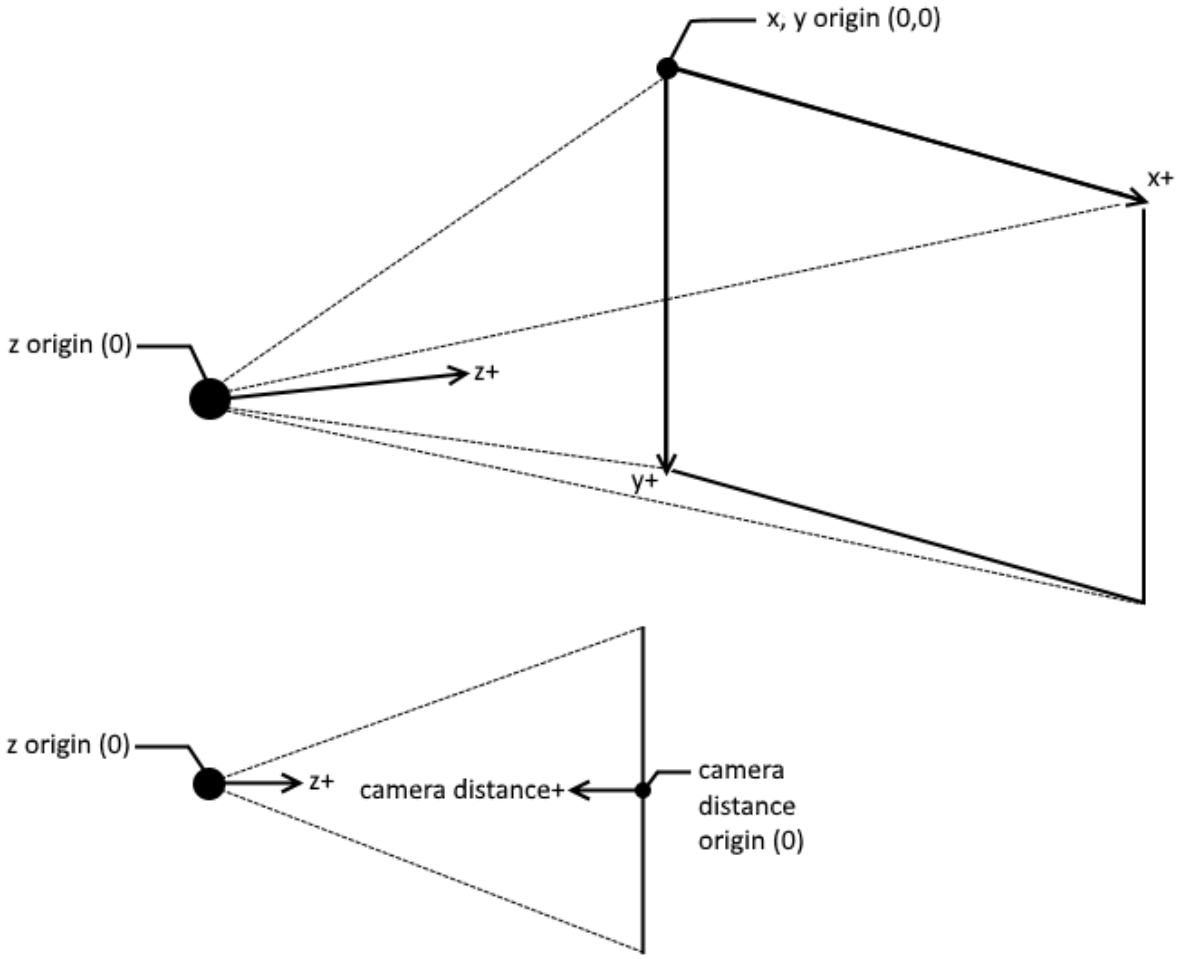
Origo determines the location around which the transformation of the selected image should take place. The coordinate properties *X Origo* and *Y Origo* is in relation to the width and height of the TextureMapper and not in relation to the width and height of the chosen image.

The coordinate property *Z Origo* is in relation to the *Camera Distance*. If the *Camera Distance* is set to 1000, and the image should rotate around it's own axis the *Z Origo* should also be set to 1000.

To lock the transformation location in the center of the TextureMapper, put a check mark in the checkbox with the label *Lock Origo to Center*. This will lock the *X Origo* and *Y Origo* properties to the center of the TextureMapper and lock the *Z Origo* to the value of the *Camera Distance*.

The *Camera Distance* changes the amount of perspective that is shown when the image is rotated. The closer the *Camera Distance* is, the greater the FOV (field of view) becomes, and therefore the

percieved amount of perspective increases.



Coordinate system used for the origo and camera distance in Texture Mapper

Interactions

The actions and triggers supported by the TextureMapper are described in the following sections.

> **NOTE**
>
> **If a rotation or scale interaction is applied to a TextureMapper, that has a duration or delay greater than zero, it will be generated as a AnimationTextureMapper.**

Actions

Widget specific action	Description
Rotate TextureMapper	Rotate the TextureMapper around its *Origo* in x-, y- and z-axis, either relative to its current orientation or to a specific angle.

Widget specific action	Description
Scale TextureMapper	Scale the TextureMapper either relative to its current size or to a specific size.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A TextureMapper does not emit any triggers.

Performance

A TextureMapper heavily depends upon the MCU for scaling and rotating the image. Therefore, a TextureMapper is considered a demanding widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a TextureMapper.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase() :
    interaction1Counter(0)
{
```

```
    textureMapper.setXY(150, 46);
    textureMapper.setBitmap(touchgfx::Bitmap(BITMAP_BLUE_LOGO_TOUCHGFX_LOGO_ID));
    textureMapper.setWidth(180);
    textureMapper.setHeight(180);
    textureMapper.setBitmapPosition(26.000f, 26.000f);
    textureMapper.setScale(1.000f);
    textureMapper.setCameraDistance(1000.000f);
    textureMapper.setOrigo(90.000f, 90.000f, 1000.000f);
    textureMapper.setCamera(90.000f, 90.000f);
    textureMapper.updateAngles(-0.500f, -0.500f, -0.500f);
    textureMapper.setRenderingAlgorithm(touchgfx::TextureMapper::BILINEAR_INTERPOLATION);

    add(textureMapper);
}

void Screen1ViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the TextureMapper class in user code. Remember to force a redraw by calling `textureMapper.invalidate()` if you change the appearance of the widget.

User Code

If the Texture Mapper is setup to be a AnimationTextureMapper, there are two callbacks that can be setup:

- `setTextureMapperAnimationStepAction` is invoked every time the current animations have performed a step.
- `setTextureMapperAnimationEndedAction` is invoked when all animations have ended.

The following two pieces of code demonstrate how to set up these two callbacks:

Screen1View.hpp

```
class Screen1View
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
private:
    /*
```

```
     * Callback Declarations
     */
    touchgfx::Callback<Screen1View, const touchgfx::AnimationTextureMapper&> textureMapper
    touchgfx::Callback<Screen1View, const touchgfx::AnimationTextureMapper&> textureMapper

    /*
     * Callback Handler Declarations
     */
    void textureMapperStepActionCallbackHandler(const touchgfx::AnimationTextureMapper& sr
    void textureMapperAnimationEndedCallbackHandler(const touchgfx::AnimationTextureMapper
};
```

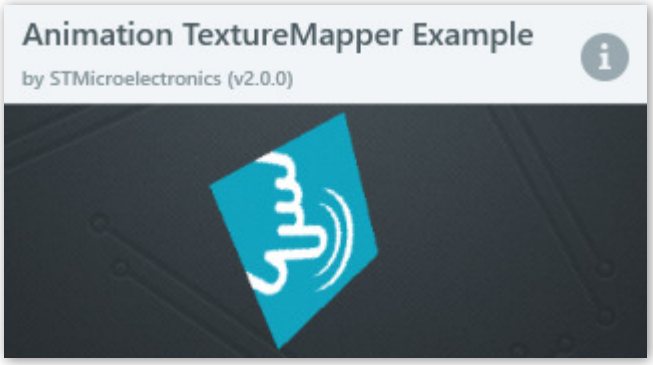
Screen1View.cpp

```
#include <gui/screen1_screen/Screen1View.hpp>

Screen1View::Screen1View() :
  textureMapperStepActionCallback(this, &Screen1View::textureMapperStepActionCallbackHandl
  textureMapperAnimationEndedCallback(this, &Screen1View::textureMapperAnimationEndedCallb
{
  textureMapper.setTextureMapperAnimationStepAction(textureMapperStepActionCallback);
  textureMapper.setTextureMapperAnimationEndedAction(textureMapperAnimationEndedCallback);
  add(textureMapper);
}

void Screen1View::textureMapperStepActionCallbackHandler(const touchgfx::AnimationTextureM
{
    if (&src == &textureMapper)
    {
        //execute code whenever the animation running in AnimationTextureMapper steps
    }
}

void Screen1View::textureMapperAnimationEndedCallbackHandler(const touchgfx::AnimationText
{
    if (&src == &textureMapper)
    {
        //execute code whenever the animation running in AnimationTextureMapper ends
    }
}
```

TouchGFX Designer Examples

To further explore the TextureMapper, try creating a new application within TouchGFX Designer with one of the following UI templates:

TextureMapper Example UI template in TouchGFX Designer



Animation TextureMapper Example UI template in TouchGFX Designer

API Reference

FURTHER READING

- **API reference for the TextureMapper class**
- **API reference for the AnimationTextureMapper class**

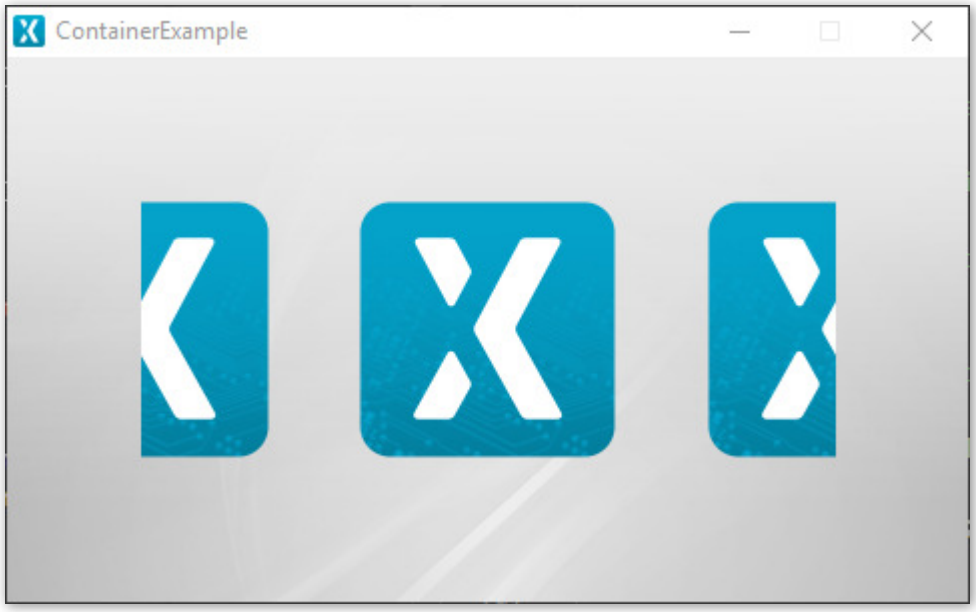
Version: 4.16

Container

A Container is a component in TouchGFX that can contain child nodes.

To read more about the fundamental nature of the concept of Containers, read the Widgets and Containers page.

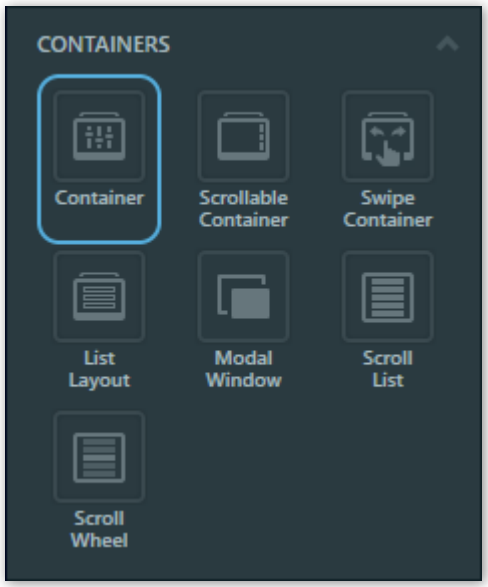
It is also possible to generate a Container as a CachableContainer. A CacheableContainer can render its content to a dynamic bitmap. This is explained more in detail on the Achieving Better Performance with CacheableContainer page.



Container widget running in the simulator

Widget Group

The Container can be found in the Containers widget group in TouchGFX Designer.



Container widget in TouchGFX Designer

Properties

The properties for a Container in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Caching	*Cachable* specifies if the Container should be generated as a CachableContainer.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by a Container in TouchGFX Designer.

Actions

Widget specific action	Description
Resize widget	Resize a widget.

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A Container does not emit any triggers.

Performance

A Container itself does not have any impact on performance and is entirely dependent on its children. Therefore, the Container is considered a very fast widget on most platforms.

In certain cases, using a CachableContainer to cache UI elements in a dynamic bitmap can significantly improve performance throughout an application. This is explained more in detail in the Achieving Better Performance with CacheableContainer article.

For more general details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Container.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase()
{
    container1.setPosition(67, 11, 347, 250);

    image1.setXY(109, 61);
    image1.setBitmap(touchgfx::Bitmap(BITMAP_BLUE_LOGO_TOUCHGFX_LOGO_ID));
    container1.add(image1);

    add(container1);
}
```

TIP

You can use these functions and the others available in the Container class in user code. Remember to force a redraw by calling `container1.invalidate()` if you change the appearance of the widget.

API Reference

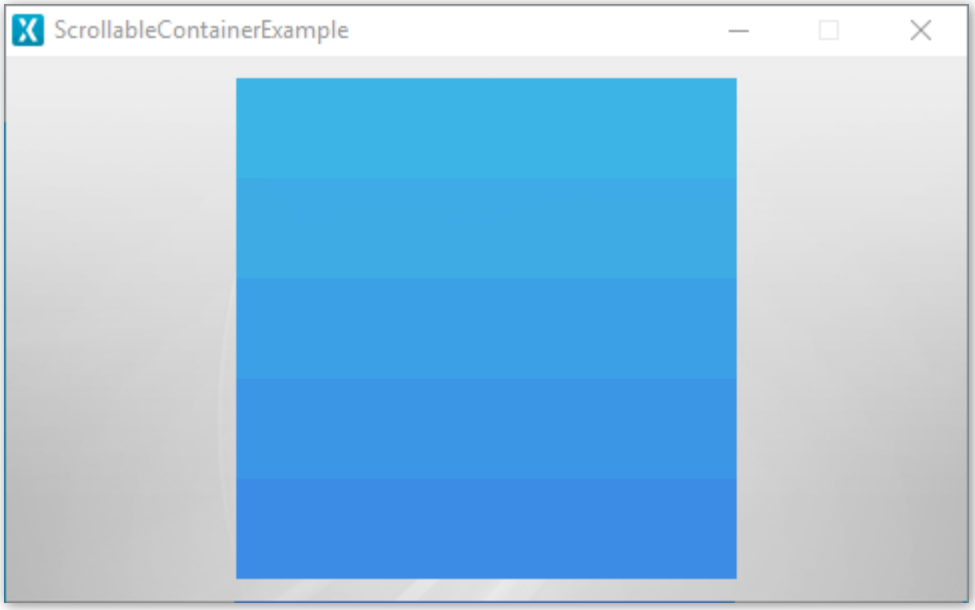
FURTHER READING

- **API reference for the Container class**
- **API reference for the CachableContainer class**

Version: 4.16

ScrollableContainer

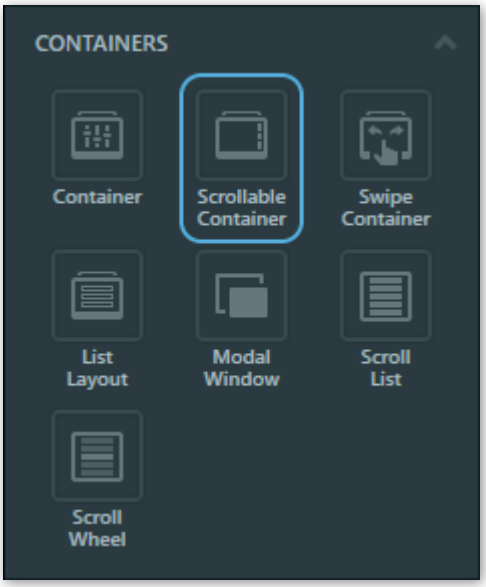
A ScrollableContainer is a Container that allows its content to be scrolled both vertically and horizontally.



ScrollableContainer running in the simulator

Widget Group

The ScrollableContainer can be found in the Containers widget group in TouchGFX Designer.



ScrollableContainer in TouchGFX Designer

Properties

The properties for a ScrollableContainer in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Scrolling	*Enable horizontal scroll* specifies if horizontal scrolling is enabled. *Enable vertical scroll* specifies if vertical scrolling is enabled. *Show scrollbars* specifies if the scrollbars should always should be visible. *Show scrollbars while scrolling* specifies if the scrollbars should only be visible when the content is being scrolled. *If 'Show scrollbars' is enabled this option is disregarded*\. *Scrollbars Color* specifies the color of the scrollbars. *Scrollbars Alpha* specifies the transparency of the scrollbars. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the ScrollableContainer are described in the following sections.

Actions

Widget specific action	Description
Resize widget	Resize the widget.

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A ScrollableContainer does not emit any triggers.

Performance

A ScrollableContainer is a Container type, and does not per default appear in the draw chain apart from the scrollbar rendering. Therefore, the performance is mostly dependent on the drawing performance of the children.

For more general details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ScrollableContainer.

Screen1ViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <touchgfx/Color.hpp>

mainViewBase::mainViewBase()
{
    scrollableContainer.setPosition(115, 11, 250, 250);
```

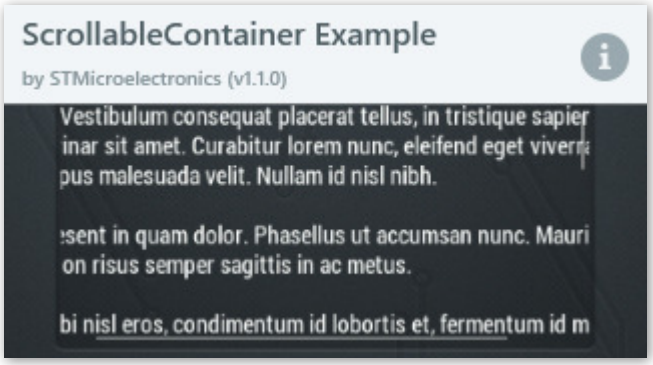
```
    scrollableContainer.enableHorizontalScroll(false);
    scrollableContainer.setScrollbarsColor(touchgfx::Color::getColorFrom24BitRGB(0, 0, 0));
    scrollableContainer.setScrollbarsPermanentlyVisible();
    scrollableContainer.setScrollbarsVisible(false);
    scrollableContainer.add(<widget_name>); //add a widget as child

    add(scrollableContainer);
}

void mainViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the ScrollableContainer class in user code. Remember to force a redraw by calling `scrollableContainer.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ScrollableContainer, try creating a new application within TouchGFX Designer with one of the following UI templates:



ScrollableContainer Example UI template in TouchGFX Designer

API Reference

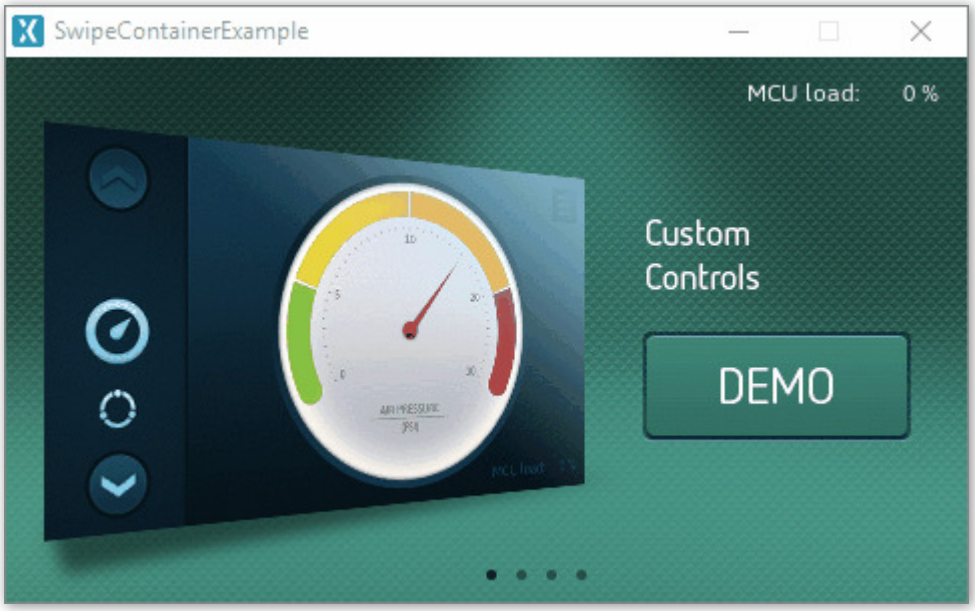
FURTHER READING

- **API reference for the ScrollableContainer class**

Version: 4.16

SwipeContainer

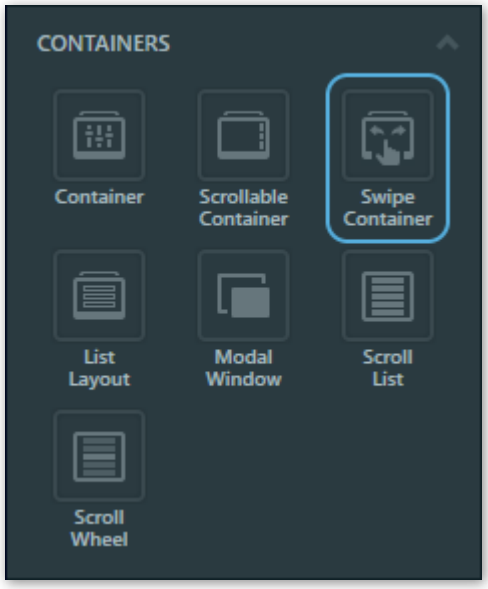
A SwipeContainer in TouchGFX is a specialization of the Container that consists of multiple pages, which can be accessed by swiping between them. The pages in the SwipeContainer can contain other widgets, similar to the Container.



SwipeContainer running in the simulator

Widget Group

The SwipeContainer can be found in the Containers widget group in TouchGFX Designer.



SwipeContainer in TouchGFX Designer

Properties

The properties for the SwipeContainer are described in the following sections.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Pages	*Selected Page* specifies the page shown on the canvas. This page will also be the starting page when the project is running. The + button creates a new page when clicked.
Page indicator	*Show page indicator* specifies the visibility of the page indicator. *X* and *Y* specify the top left corner of the page indicator relative to the top left corner of the widget. *Center horizontally* specifies if the position page indicator should be centered in the x-axis of the widget. *Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.* *Normal Image* and *Highlighted Image* specify the images assigned to the normal and highlighted states of the PageIndicator.
Swipe settings	*Swipe threshold* specifies the distance that has to be swiped by the user before resulting in a page change. *End swipe elastic width* specifies the distance the first and last pages can be swiped beyond the borders of the widget before stopping.

Property Group	Property Descriptions
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the SwipeContainer are described in the following sections.

Actions

Standard widget action	Description
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The SwipeContainer does not emit any triggers.

Performance

A SwipeContainer is a Container type and does not per default appear in the draw chain. Therefore, the performance is mostly dependent on the drawing performance of the children, though the SwipeContainer also does some image drawing in the form of its PageIndicator.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how the TouchGFX Designer sets up the SwipeContainer of two pages with a page indicator centered horizontally.

Screen1ViewBase.cpp

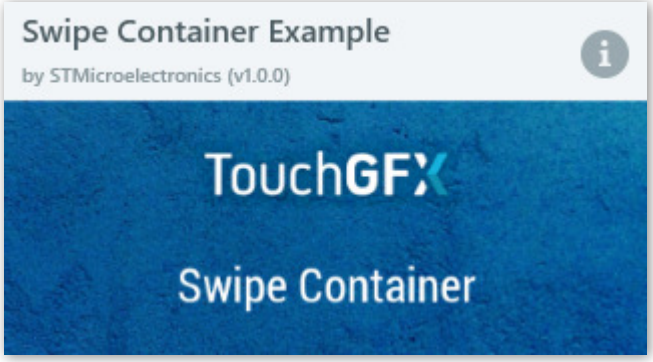
```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase()
{
    swipeContainerName.setXY(15, 10);
    swipeContainerName.setPageIndicatorBitmaps(touchgfx::Bitmap(BITMAP_BLUE_PAGEINDICATOR_
    swipeContainerName.setPageIndicatorXY(210, 0);
    swipeContainerName.setSwipeCutoff(50);
    swipeContainerName.setEndSwipeElasticWidth(50);

    swipeContainerNamePage1.setWidth(450);
    swipeContainerNamePage1.setHeight(250);
    swipeContainerName.add(swipeContainerNamePage1);

    swipeContainerNamePage2.setWidth(450);
    swipeContainerNamePage2.setHeight(250);
    swipeContainerName.add(swipeContainerNamePage2);

    swipeContainerName.setSelectedPage(0);

    add(swipeContainerName);
}
```

TIP

You can use these functions and the others available in the SwipeContainer class in user code. Remember to force a redraw by calling `swipeContainerName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the SwipeContainer, try creating a new application within TouchGFX Designer with the following UI template:



API Reference

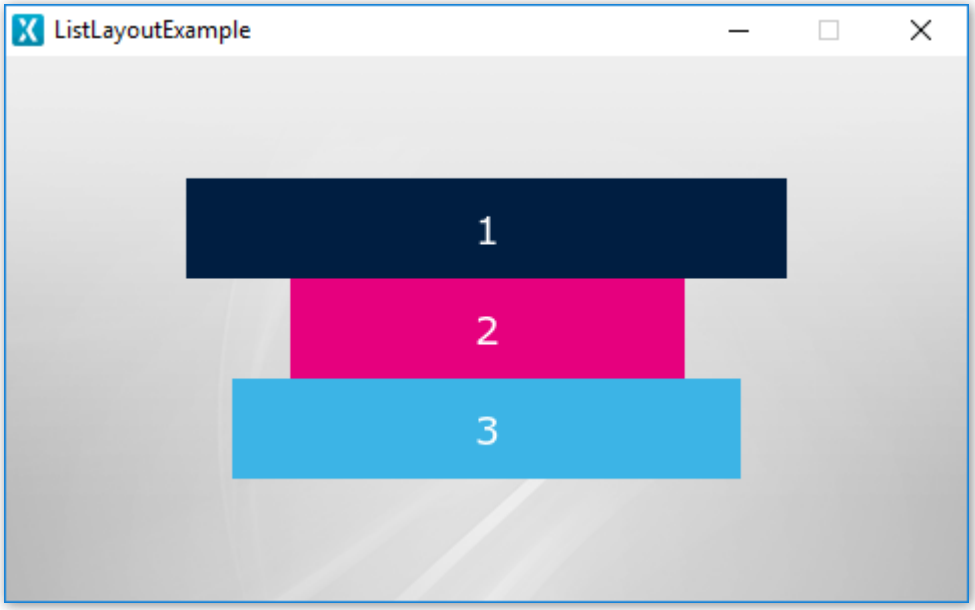
FURTHER READING

- **API reference for the SwipeContainer class**

Version: 4.16

ListLayout

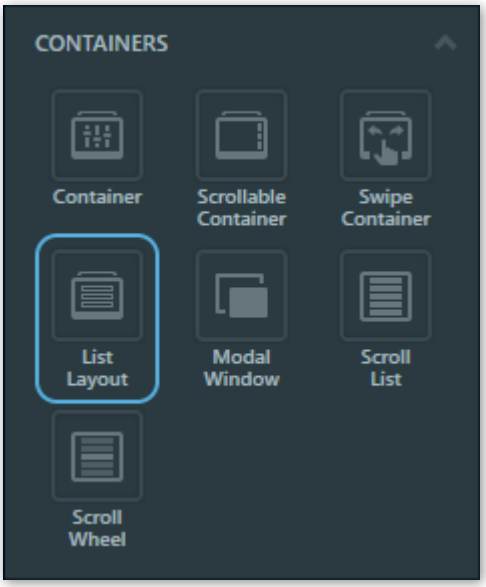
The ListLayout widget is a Container which automatically arranges its children in a list in a given direction. Adding and removing widgets from the ListLayout rearranges the children.



ListLayout running in the simulator

Widget Group

The ListLayout can be found in the Containers widget group in TouchGFX Designer.



ListLayout in TouchGFX Designer

Properties

The properties for a ListLayout in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. The size of the ListLayout amounts to the total size of its children. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Direction	*Direction* specifies the direction of the layout arrangement. Choose between a horizontal layout in the east (right) direction or vertical layout in the south (down) direction.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by a ListLayout in TouchGFX Designer.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A ListLayout does not emit any triggers.

Performance

A ListLayout itself does not have any notable impact on performance and is almost entirely dependent on its children. Therefore, the ListLayout is considered a very fast widget on most platforms.

For more general details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ListLayout.

Screen1ViewBase.cpp

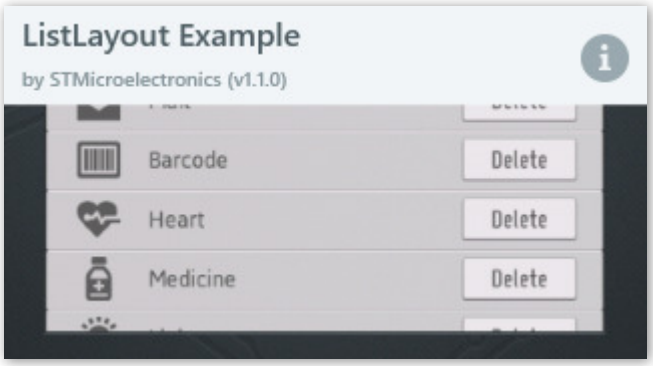
```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase()
{
    listLayout1.setDirection(touchgfx::SOUTH);
    listLayout1.setXY(90, 111);

    box1.setWidth(50);
    box1.setHeight(50);
    box1.setColor(touchgfx::Color::getColorFrom24BitRGB(255, 255, 255));
    listLayout1.add(box1);

    add(listLayout1);
}
```

TIP

You can use these functions and the others available in the ListLayout class in user code. Remember to force a redraw by calling `listLayout1.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ListLayout, try creating a new application within TouchGFX Designer with one of the following UI templates:



ListLayout Example UI template in TouchGFX Designer

API Reference

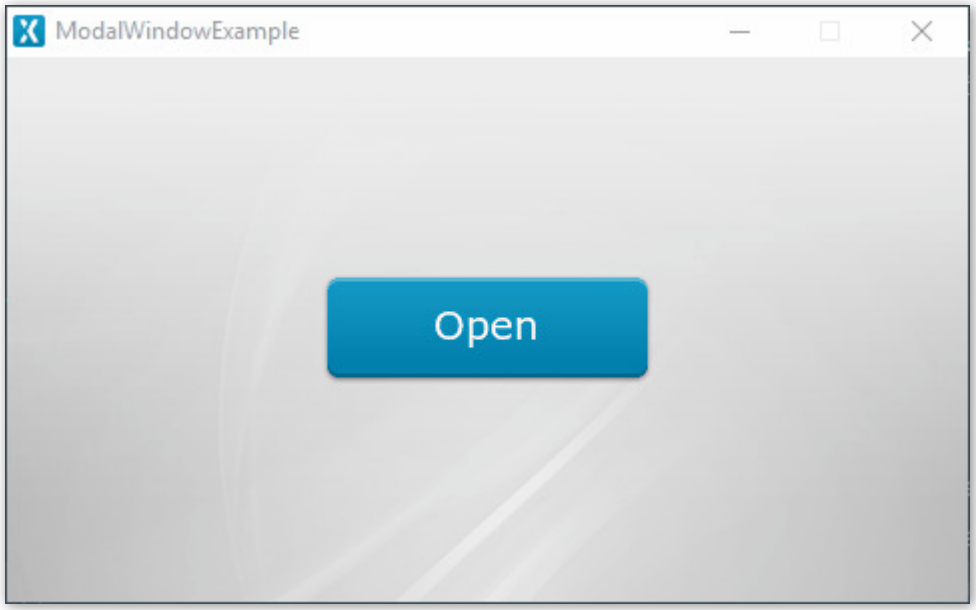
FURTHER READING

- **API reference for the ListLayout class**

Version: 4.16

ModalWindow

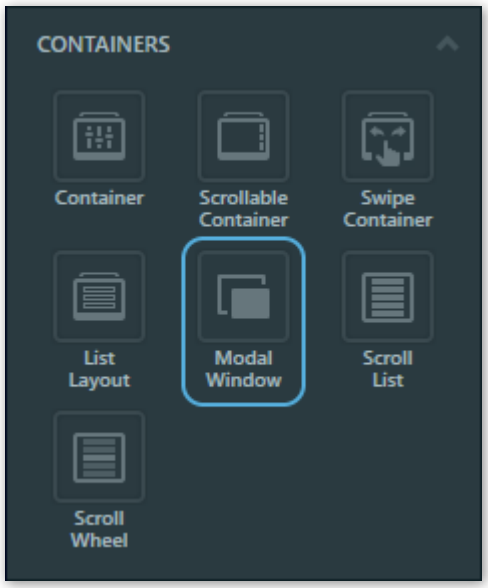
The ModalWindow is a Container type widget that displays a window and blocks touch events to the underlying view and widgets. The ModalWindow consists of a background Image and a Box that acts as a shade over the underlying view and widgets with adjustable alpha. The ModalWindow will fill up the entire screen and should always be added as the last element such that it is always on top of all other elements.



ModalWindow running in the simulator

Widget Group

The ModalWindow can be found in the Containers widget group in TouchGFX Designer.



ModalWindow in TouchGFX Designer

Properties

The properties for a ModalWindow in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Modal Image Location	*X* and *Y* specify the top left corner of the image within the ModalWindow. *W* and *H* specify the width and height of the container within the ModalWindow. *The size of the container within the ModalWindow is taken from the size of the associated image and cannot be altered except by changing the image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Appearance	*Window Image* specifies which image the ModalWindow should use. *Shade Color* specifies the color of the overlay shade. *Shade Alpha* specifies the transparency of the overlay shade. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

Interactions

The actions and triggers supported by the ModalWindow are described in the following sections.

Actions

Standard widget action	Description
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The ModalWindow does not emit any triggers.

Performance

A ModalWindow is a Container type that consists of a Box, a Container and an Image. The ModalWindow does not per default appear in the draw chain. Therefore, the performance is mostly dependent on the childrens drawing performance.

For more general details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ModalWindow.

mainViewBase.cpp

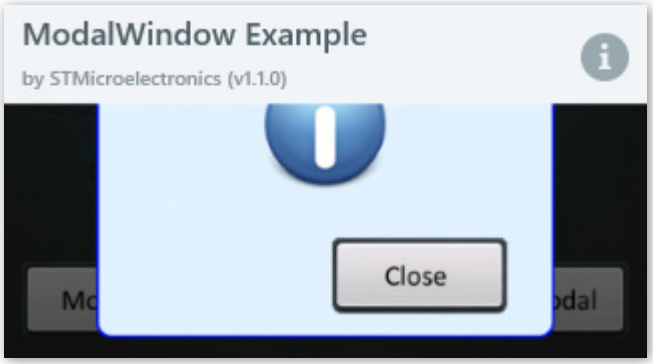
```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <texts/TextKeysAndLanguages.hpp>
#include <touchgfx/Color.hpp>

mainViewBase::mainViewBase() :
    buttonCallback(this, &mainViewBase::buttonCallbackHandler)
{
    modalWindow.setBackground(touchgfx::BitmapId(BITMAP_BLUE_BACKGROUNDS_MAIN_BG_320X240P
    modalWindow.setShadeColor(touchgfx::Color::getColorFrom24BitRGB(0, 0, 0));
    modalWindow.setShadeAlpha(150);
    modalWindow.hide();

    add(modalWindow);
}

void mainViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the ModalWindow class in user code. Remember to force a redraw by calling `modalWindow.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the ModalWindow, try creating a new application within TouchGFX Designer with one of the following UI templates:



ModalWindow Example UI template in TouchGFX Designer

API Reference

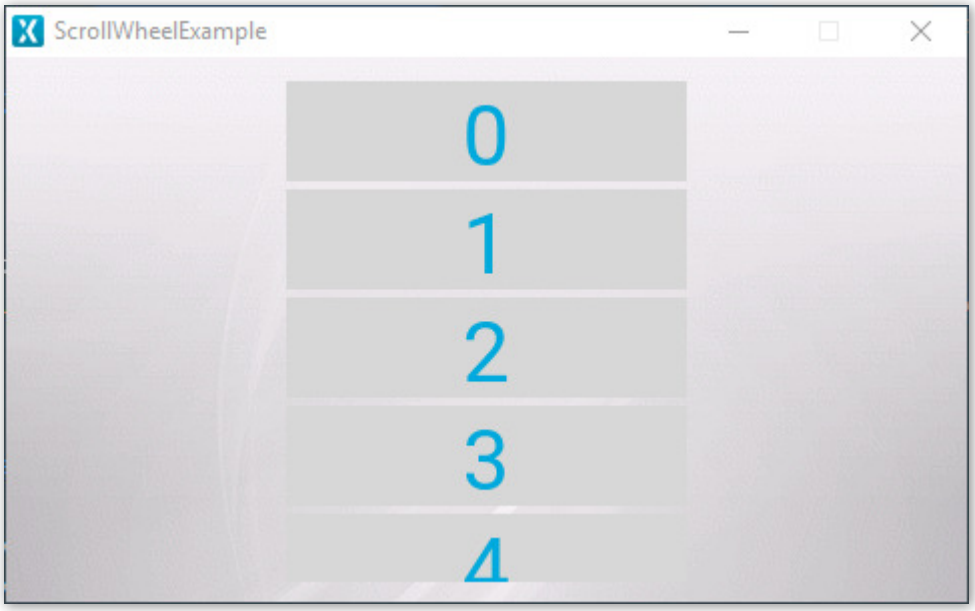
FURTHER READING

- **API reference for the ModalWindow class**

Version: 4.16

ScrollList

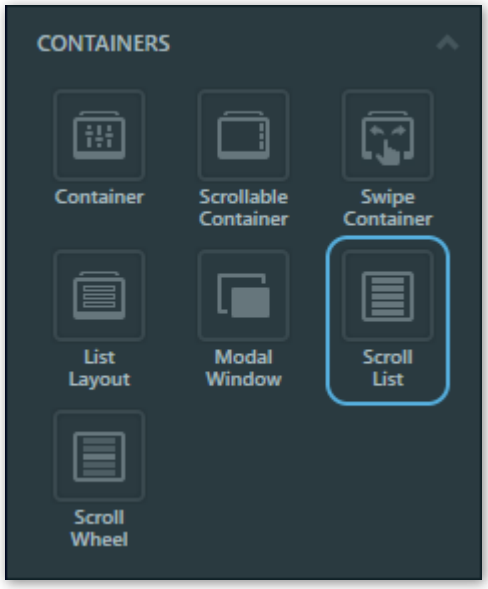
The ScrollList is a scrollable menu consisting of a number of items and a number of widgets, which are dynamically updated as they are scrolled into view. The ScrollList is also able to invoke callbacks when interacting with the items in the ScrollList.



ScrollList running in the simulator

Widget Group

The ScrollList can be found in the Containers widget group in TouchGFX Designer.



ScrollList in TouchGFX Designer

Properties

The properties for a ScrollList in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Type	*Type* specifies if ScrollList is oriented vertically or horizontally
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Item Template	*Item Template* specifies which CustomContainer to use as template. *Number of Items* specifies the number of items present in the ScrollList.
List Appearance	*Circular* specifies if the items in the ScrollList will loop when reaching the end. *Items Snap* specifies if items should snap. *If snapping is false, the items can flow freely. If snapping is true, the items will snap into place such that an item is always in the selected spot.* *Item Margin* specifies the spacing between items. *Padding Before* and *Padding After* specifies the distance offset before and after the visible drawables in the ScrollList.
Animation	*Easing* and *Easing Option* specify which easing equation to use for animations. *Swipe Acc.* and *Drag Acc.* specify the acceleration when scrolling.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Item Templates

The items in a ScrollList are based on a concept called *Item Template* which is a CustomContainer that serves as a base for the graphical elements for the items in the ScrollList. Before creating a ScrollList, a Custom Container should be created to have an Item Template for the ScrollList.

After having created the ScrollList the CustomContainer can be selected under the property *Item Template*. Specifying the *Item Template* results in the ScrollList resizing to fit with the size property that is not in the scrollable direction (width for vertical ScrollLists and height for horizontal ScrollLists) of the selected Custom Container. Changing the other size property (height for vertical and width for horizontal) determines the number of items visible.

Interactions

The actions and triggers supported by the ScrollList are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A ScrollList does not emit any triggers.

Performance

A ScrollList is a Container type, and does not per default appear in the draw chain. Therefore, the performance is wholly dependent on the childrens drawing performance.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ScrollList.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase() :
    updateItemCallback(this, &Screen1ViewBase::updateItemCallbackHandler)
{
    scrollList.setPosition(140, 10, 200, 252);
    scrollList.setHorizontal(false);
    scrollList.setCircular(false);
    scrollList.setEasingEquation(touchgfx::EasingEquations::backEaseOut);
    scrollList.setSwipeAcceleration(10);
    scrollList.setDragAcceleration(10);
    scrollList.setNumberOfItems(20);
    scrollList.setPadding(0, 0);
    scrollList.setSnapping(false);
    scrollList.setDrawableSize(50, 2);
    scrollList.setDrawables(scrollListListItems, updateItemCallback);

    add(scrollList);
}

void Screen1ViewBase::setupScreen()
{
    scrollList.initialize();
    for (int i = 0; i < scrollListListItems.getNumberOfDrawables(); i++)
    {
        scrollListListItems[i].initialize();
    }
}

void Screen1ViewBase::updateItemCallbackHandler(touchgfx::DrawableListItemsInterface* iter
{
    if (items == &scrollListListItems)
    {
        touchgfx::Drawable* d = items->getDrawable(containerIndex);
        TextContainer* cc = (TextContainer*)d;
        scrollListUpdateItem(*cc, itemIndex);
    }
}
```

TIP

You can use these functions and the others available in the ScrollList class in user code. Remember to force a redraw by calling `scrollList.invalidate()` if you change the appearance of the widget.

User Code

After the graphical elements for the ScrollList and its properties are set, user code can be written to update the items in the ScrollList. The header file for the `Screen1ViewBase` class which is generated by TouchGFX Designer is shown below:

ScreenViewBase.hpp

```
class ScreenViewBase : public touchgfx::View
{
public:
    ScreenViewBase();
    virtual ~ScreenViewBase() {}
    virtual void setupScreen();

    virtual void scrollListUpdateItem(CustomContainer& item, int16_t itemIndex)
    {
        // Override and implement this function in Screen
    }

protected:
    FrontendApplication& application() {
         return *static_cast<FrontendApplication*>(Application::getInstance());
    }
    touchgfx::BoxWithBorder boxWithBorder;
    touchgfx::ScrollList scrollList;
    touchgfx::DrawableListItems<CustomContainer, 6> scrollListListItems;
private:
    void updateItemCallbackHandler(DrawableListItemsInterface* items, int16_t containerInd
    touchgfx::Callback<ScreenViewBase, DrawableListItemsInterface*, int16_t, int16_t> upda
};
```

When TouchGFX Designer generates the code for ScrollList, the function `scrollListUpdateItem`, highlighted above, is created for the user to override and update the items in the ScrollList. The function is called each time an item in the ScrollList needs updating, thereby ensuring that an item is updated before it becomes visible. The `scrollListUpdateItem` has two parameters, which are used to identify the item being updated and to update it. The parameter `itemIndex` contains the index value of the item, which is used to identify which item is being updated. The parameter `item` is a reference to a CustomContainer object which is a visible item in the ScrollList. Updating the graphics for the parameter `item` results in an update to the render for a visible item in the ScrollList.

An example integration of `scrollListUpdateItem` is shown below:

Screen1View.hpp

```
#ifndef SCREEN1_VIEW_HPP
#define SCREEN1_VIEW_HPP
```

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <gui/screen1_screen/ScreenPresenter.hpp>

class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();

    virtual void scrollListUpdateItem(CustomContainer& item, int16_t itemIndex);
protected:
};

#endif // SCREEN1_VIEW_HP
```

Screen1View.cpp

```
#include <gui/screen1_screen/Screen1View.hpp>

Screen1View::Screen1View()
{

}

void Screen1View::setupScreen()
{
    Screen1ViewBase::setupScreen();
}

void Screen1View::tearDownScreen()
{
    Screen1ViewBase::tearDownScreen();
}

void Screen1View::scrollListUpdateItem(CustomContainer& item, int16_t itemIndex)
{
    item.setValue(itemIndex);
}
```

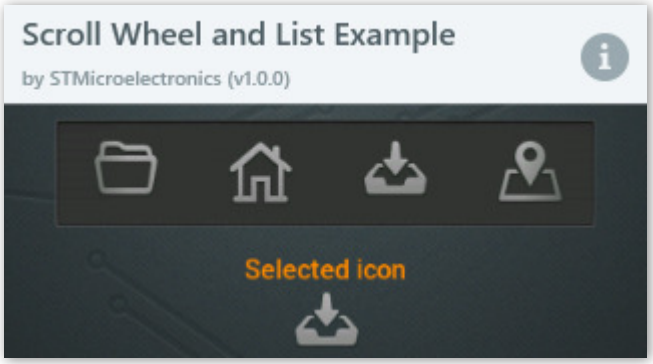
In the header file `Screen1View.hpp`, the `scrollListUpdateItem` function is overidden and then implemented in `Screen1View.cpp`.

The goal of this example is to update the text in the Item Template with the index value of the items which are visible, like the example shown in the beginning of this section. Since the Item Template is based on the CustomContainer, a `setValue` function is created for the CustomContainer. The setValue function is able to take the `itemIndex` parameter and update the text in the item template.

Calling setvalue for the parameter item will cause the items to update their appearance, thereby showing their index value.

TouchGFX Designer Examples

To further explore the ScrollList, try creating a new application within TouchGFX Designer with one of the following UI templates:



ScrollWheel and List Example UI template in TouchGFX Designer

API Reference

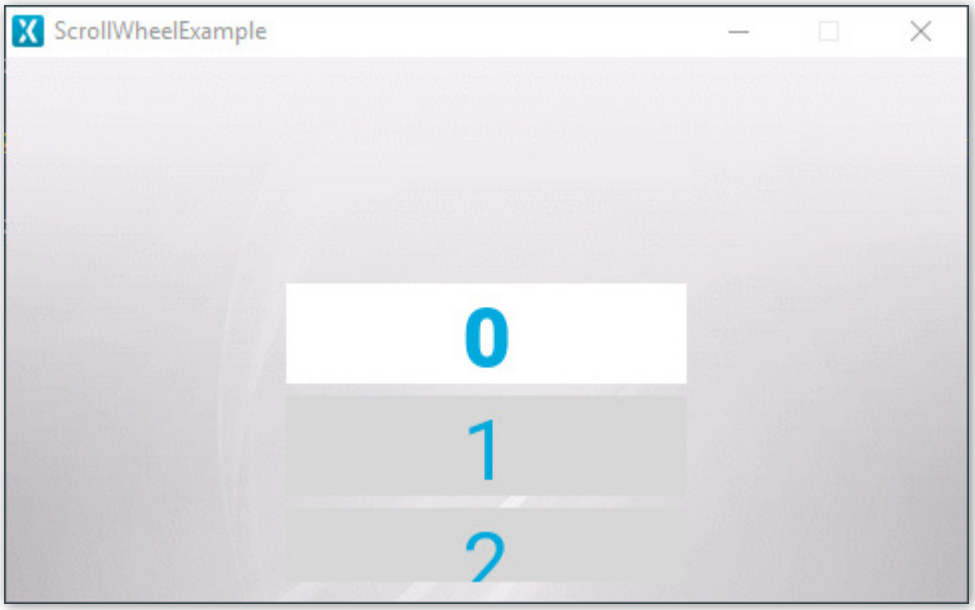
FURTHER READING

- **API reference for the ScrollList class**

Version: 4.16

ScrollWheel

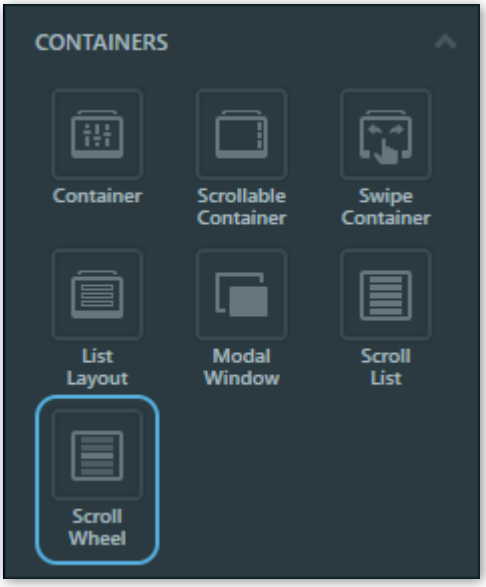
The ScrollWheel is a scrollable menu containing multiple items, which are dynamically updated when scrolling through the items in the wheel, and the item which is selected is moved into focus. Enabling the code to react to interactions with the ScrollWheel, different callbacks can be invoked based on the interaction with the items in the wheel.



ScrollWheel running in the simulator

Widget Group

The ScrollWheel can be found in the Containers widget group in TouchGFX Designer.



ScrollWheel in TouchGFX Designer

Properties

The properties for a ScrollWheel in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Type	*Type* specifies whether the ScrollWheel is oriented vertically or horizontally.
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Item Template	*Item Template* specifies which CustomContainer to use as template. *Number of Items* specifies the number of items present in the ScrollWheel. *Initial Selected Item* specifies which item is selected first. *Use Selected Style Template* specifies whether to use a separate template for the selected item. *Selected Style Template* specifies which CustomContainer to use as selected template.
List Appearance	*Circular* specifies if the items in the ScrollWheel will loop when reaching the end. *Selected Item Offset* specifies the location of the selected item. *Item Margin* specifies the spacing between items. *Extra Size Before* and *Extra Size After* specify the size of the area in which *Selected Style Template* is shown. *Margin Before* and *Margin After* specify the size of the margin before and after the area in which *Selected Style Template* is shown.

Property Group	Property Descriptions
Animation	*Easing* and *Easing Option* specify which easing equation to use for animations. *Swipe Acc.* and *Drag Acc.* specify the acceleration when scrolling.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Item Templates

The items in a ScrollList are based on a concept called *Item Template* which is a CustomContainer that serves as a base for the graphical elements for the items in the ScrollWheel. To highlight the selected item, the ScrollWheel has the option to select an Item Template called *Selected Style Template*, which is only used for the selected item. Before creating a ScrollWheel, a CustomContainer for the Item Template, along with a *Selected Style Template* if enabled, should be created.

After the ScrollWheel is created, the CustomContainer can be selected under the property *Item Template*. When selecting the Custom Container for the *Item Template*, the ScrollWheel resizes to fit with the size property that is not in the scrollable direction (*width for vertical orientation and height for horizontal orientation*) of the selected Custom Container. Changing the other size property (*height for vertical orientation and width for horizontal orientation*) determines the number of items visible.

Interactions

The actions and triggers supported by the ScrollWheel are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A ScrollWheel does not emit any triggers.

Performance

A ScrollWheel is a Container type, and does not per default appear in the draw chain. Therefore, the performance is wholly dependent on the drawing performance of the children.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a ScrollWheel.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase() :
    updateItemCallback(this, &Screen1ViewBase::updateItemCallbackHandler)
{
    scrollWheel.setPosition(140, 10, 200, 252);
    scrollWheel.setHorizontal(false);
    scrollWheel.setCircular(false);
    scrollWheel.setEasingEquation(touchgfx::EasingEquations::backEaseIn);
    scrollWheel.setSwipeAcceleration(10);
    scrollWheel.setDragAcceleration(10);
    scrollWheel.setNumberOfItems(60);
    scrollWheel.setSelectedItemOffset(100);
    scrollWheel.setSelectedItemExtraSize(0, 0);
    scrollWheel.setSelectedItemMargin(0, 0);
    scrollWheel.setDrawableSize(50, 3);
    scrollWheel.setDrawables(scrollWheelListItems, updateItemCallback,
                             scrollWheelSelectedListItems, updateItemCallback);
    scrollWheel.animateToItem(0, 0);

    add(scrollWheel);
}

void Screen1ViewBase::setupScreen()
{
    scrollWheel.initialize();
    for (int i = 0; i < scrollWheelListItems.getNumberOfDrawables(); i++)
```

```
    {
        scrollWheelListItems[i].initialize();
    }
    for (int i = 0; i < scrollWheelSelectedListItems.getNumberOfDrawables(); i++)
    {
        scrollWheelSelectedListItems[i].initialize();
    }
}

void Screen1ViewBase::updateItemCallbackHandler(touchgfx::DrawableListItemsInterface* iter
{
    if (items == &scrollWheelListItems)
    {
        touchgfx::Drawable* d = items->getDrawable(containerIndex);
        TextContainer* cc = (TextContainer*)d;
        scrollWheelUpdateItem(*cc, itemIndex);
    }
    else if (items == &scrollWheelSelectedListItems)
    {
        touchgfx::Drawable* d = items->getDrawable(containerIndex);
        TextCenterContainer* cc = (TextCenterContainer*)d;
        scrollWheelUpdateCenterItem(*cc, itemIndex);
    }
}
```

TIP

You can use these functions and the others available in the ScrollWheel class in user code. Remember to force a redraw by calling `scrollWheel.invalidate()` if you change the appearance of the widget.

User Code

After the graphical elements for the ScrollWheel and its properties are set, user code can be written to update the items in the ScrollWheel. The header file for the Screen1ViewBase class which is generated by the TouchGFX Designer is shown below:

Screen1ViewBase.hpp

```
class Screen1ViewBase : public touchgfx::View
{
public:
    Screen1ViewBase();
    virtual ~Screen1ViewBase() {}
    virtual void setupScreen();

    virtual void scrollWheel1UpdateItem(CustomContainer1& item, int16_t itemIndex)
    {
        // Override and implement this function in Screen1
```

```
    }

    virtual void scrollWheel1UpdateCenterItem(CustomContainer2& item, int16_t itemIndex)
    {
        // Override and implement this function in Screen1
    }

protected:
    FrontendApplication& application() {
        return *static_cast<FrontendApplication*>(Application::getInstance());
    }

    touchgfx::BoxWithBorder boxWithBorder1;
    touchgfx::ScrollWheelWithSelectionStyle scrollWheel1;
    touchgfx::DrawableListItems<CustomContainer1, 6> scrollWheel1ListItems;
    touchgfx::DrawableListItems<CustomContainer2, 2> scrollWheel1SelectedListItems;

private:
    void updateItemCallbackHandler(DrawableListItemsInterface* items, int16_t containerInd
    touchgfx::Callback<Screen1ViewBase, DrawableListItemsInterface*, int16_t, int16_t> upd

};
```

When the TouchGFX Designer generates the code for ScrollWheel, the functions `scrollWheel1UpdateItem` and `scrollWheel1UpdateCenterItem`, highlighted above, is created for the user to override and update the items in the ScrollWheel.

The `UpdateItem` function is always generated for a ScrollWheel and can be implemented to updated the contained items, while the `UpdateCenterItem` function updates the item based on the *Selected Style Template* and is therefore only generated if the usage of a *Selected Style Template* is selected. Other than updating different items, the two functions contain the same parameters used for updating the items in the the ScrollWheel.

The parameter `itemIndex` contains the index value of the item, which is used to identify which item is being updated. The parameter `item` is a reference to a visible item in the ScrollWheel. Updating the appearance for the parameter `item` results in an update to the render for a visible item in the ScrollWheel.

An example implementation of `scrollWheel1UpdateItem` and `scrollWheel1UpdateCenterItem` in the user code files `Screen1View.hpp` and `Screen1View.cpp` is shown below:

Screen1View.hpp

```
#ifndef SCREEN1_VIEW_HPP
#define SCREEN1_VIEW_HPP

#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <gui/screen1_screen/Screen1Presenter.hpp>
```

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();

    virtual void scrollWheel1UpdateItem(CustomContainer1& item, int16_t itemIndex);
    virtual void scrollWheel1UpdateCenterItem(CustomContainer2& item, int16_t itemIndex);
protected:
};

#endif // SCREEN1_VIEW_HPP
```

Screen1View.cpp

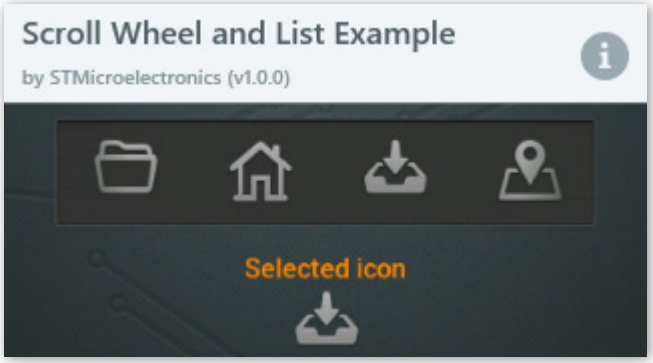
```
#include <gui/screen1_screen/Screen1View.hpp>

Screen1View::Screen1View()
{

}

void Screen1View::setupScreen()
{
    Screen1ViewBase::setupScreen();
}

void Screen1View::tearDownScreen()
{
    Screen1ViewBase::tearDownScreen();
}

void Screen1View::scrollWheel1UpdateItem(CustomContainer1& item, int16_t itemIndex)
{
    item.setIndex(itemIndex);
}

void Screen1View::scrollWheel1UpdateCenterItem(CustomContainer2& item, int16_t itemIndex)
{
    item.setIndex(itemIndex);
}
```

In the header file `Screen1View.hpp`, the `scrollWheel1UpdateItem` and `scrollWheel1UpdateCenterItem` functions are overridden and are then implemented in `Screen1View.cpp`.

The goal of this example is to update the text in the Item Template with the index value of the items which are visible, like the example shown in the beginning of this section. Since both the Item Template and the Selected Style Template are based on CustomContainer, a `setIndex` function is created for both CustomContainers. The `setIndex` function is able to take the `itemIndex` parameter and update the text in the Item Template and the Selected Style Template. Calling the `setIndex` for an item results in an update to the appearance of the visible items thereby showing their index value.

TouchGFX Designer Examples

To further explore the ScrollWheel, try creating a new application within TouchGFX Designer with one of the following UI templates:



ScrollWheel and List Example UI Template in TouchGFX Designer

API Reference

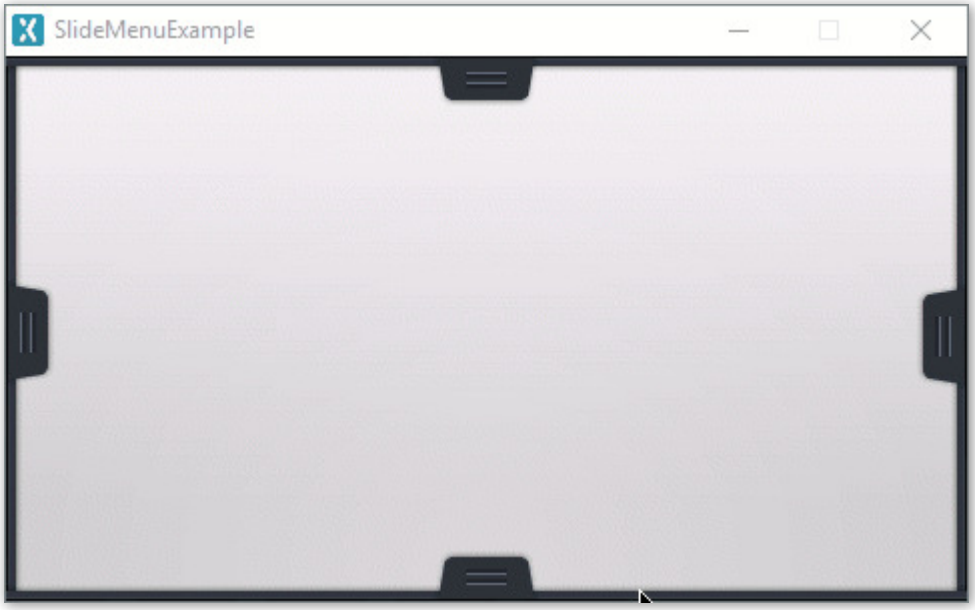
FURTHER READING

- **API reference for the ScrollWheel class**

Version: 4.16

SlideMenu

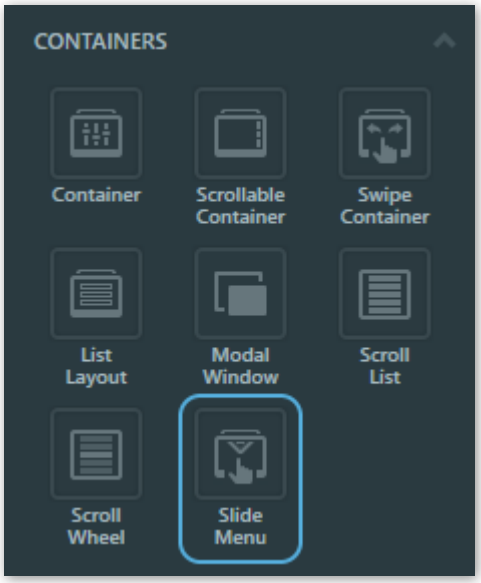
A SlideMenu in TouchGFX is a specialization of the Container that consists of an internal Container, an Image and an optional Button, which can animate between a collapsed and expanded state.



SlideMenu running in the simulator

Widget Group

The SlideMenu can be found in the Containers widget group in TouchGFX Designer.



SlideMenu in TouchGFX Designer

Properties

The properties for the SlideMenu are described in the following sections.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a SlideMenu is determined by the size of its background and button images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Expanding Direction	*Expanding Direction* specifies the direction the SlideMenu should expand.
State	*State* specifies the initial state of the SlideMenu, either collapsed or expanded. *Collapsed: Visible Pixels* specifies the amount of pixels that should be visible when the state is collapsed. *Expanded: Hidden Pixels* Specifices the amount of pixels that should be hidden when the state is expanded. *Expanded Timeout* specifies the amount of time before automatically returning to the collapsed state from the expanded state.
Background	*Background Image* specifies the image to use as background. *Background Location* specifies the x,y coordinate of the background image relative to the widgets' coordinates.
Button	*Use Button* Specifies whether or not to make use of a button to control the state of the widget. *Released Image* specifies the image to use for the buttons' released state. *Pressed Image* specifies the image to use for the buttons' pressed state. *Button Location* specifies the x,y coordinate of the button relative to the widgets' coordinates.

Property Group	Property Descriptions
Animation	*Easing* and *Easing Option* specify which easing equation to use for animations. *Duration* specifies the amount of time the animation should take.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the SlideMenu are described in the following sections.

Actions

Widget specific action	Description
Change State of SlideMenu	Change the state of a SlideMenu to either collapsed or expanded
Reset Timer of SlideMenu	Reset the timer automatically returning the SlideMenu state to collapsed

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
SlideMenu animation ended	A SlideMenu animation from one state to another has ended.

Trigger	Description
SlideMenu state changed	A SlideMenu has had its state changed.

Performance

A SlideMenu is a Container type and does not per default appear in the draw chain. Therefore, the performance is mostly dependent on the drawing performance of the children.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how the TouchGFX Designer sets up a SlideMenu.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"

Screen1ViewBase::Screen1ViewBase()
{
    slideMenuName.setup(touchgfx::SlideMenu::EAST,
        touchgfx::Bitmap(BITMAP_LEFT_SLIDE_MENU_BACKGROUND_ID),
        touchgfx::Bitmap(BITMAP_LEFT_SLIDE_MENU_BUTTON_ID),
        touchgfx::Bitmap(BITMAP_LEFT_SLIDE_MENU_BUTTON_ID),
        0, 0, 49, 111);
    slideMenuName.setState(touchgfx::SlideMenu::COLLAPSED);
    slideMenuName.setVisiblePixelsWhenCollapsed(24);
    slideMenuName.setHiddenPixelsWhenExpanded(0);
    slideMenuName.setAnimationEasingEquation(touchgfx::EasingEquations::circEaseInOut);
    slideMenuName.setAnimationDuration(18);
    slideMenuName.setExpandedStateTimeout(180);
    slideMenuName.setXY(0, 0);

    add(slideMenuName);
}
```

TIP

You can use these functions and the others available in the SlideMenu class in user code. Remember to force a redraw by calling `SlideMenuName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the SlideMenu, try creating a new application within TouchGFX Designer with the following UI template:

SlideMenu Example UI template in TouchGFX Designer

API Reference

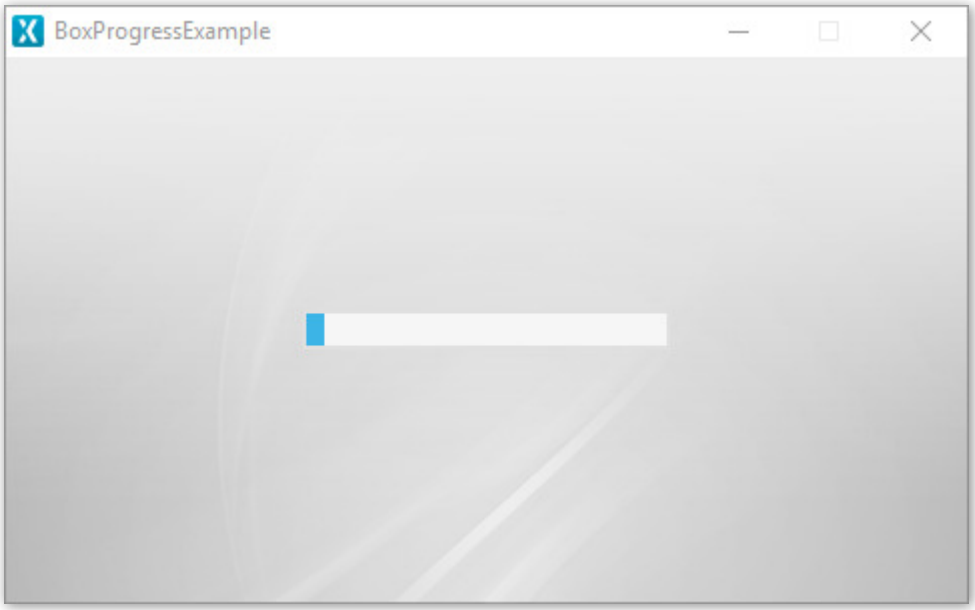
FURTHER READING

- **API reference for the SlideMenu class**

Version: 4.16

BoxProgress

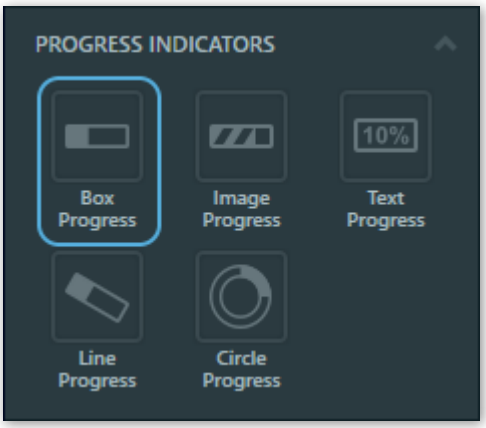
A BoxProgress shows the current progress by using a simple Box as the progress indicator on top of a background Image. The *Color*, the *Alpha* of the box and the *Direction* towards which the box will progress can be configured. It is possible to create a custom background image and change the different parameters of the progress indicator such as the position and the size to fit the custom background image.



BoxProgress running in the simulator

Widget Group

The BoxProgress can be found in the Progress Indicators widget group in TouchGFX Designer.



BoxProgress in TouchGFX Designer

Properties

The properties for a BoxProgress in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a BoxProgress is determined by the size of the selected background image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image & Color	*Background* sets the background image. *Progress* sets the color of the progress box.
Progress Position	*X* and *Y* coordinates specify the top left corner of the progress box relative to the position of the ProgressIndicator. *W* and *H* specify the width and height of the progress box.
Values	*Range Min* and *Range Max* specify the minimum and maximum integer values of the indicator. *Initial* specifies the initial value of the progress indicator.*Steps Total* specifies at what granularity the progress indicator reports new values. For instance, if the progress needs to be reported as 0%, 10%, 20%, ..., 100%, the total value should be set to 10. *Steps Min* specifies the minimum steps the progress indicator shows.
Appearance	*Direction* specifies in which direction the progress indicator should progress. *Alpha* specifies the transparency of the progress box. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

Property Group	Property Descriptions
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the BoxProgress are described in the following sections.

Actions

Widget specific actions	Description
Set value	Set the value of the progress indicator.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The BoxProgress does not emit any triggers.

Performance

A BoxProgress consists of a Box and a background Image. Therefore, the BoxProgress is dependent on image drawing and is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a BoxProgress.

Screen1ViewBase.cpp

```
boxProgress.setXY(148, 126);
boxProgress.setProgressIndicatorPosition(2, 2, 180, 16);
boxProgress.setRange(0, 100);
boxProgress.setDirection(touchgfx::AbstractDirectionProgress::RIGHT);
boxProgress.setBackground(touchgfx::Bitmap(BITMAP_BLUE_PROGRESSINDICATORS_BG_MEDIUM_PROGRE
boxProgress.setColor(touchgfx::Color::getColorFrom24BitRGB(0, 151, 255));
boxProgress.setValue(0);
```

User Code

The following example illustrates how to implement the `handleTickEvent()` function to simulate progress. Running this code creates the application shown at the beginning of this article.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();
protected:
    bool increase = true;
};
```

Screen1View.cpp

```
void Screen1View::handleTickEvent()
{
    int currentValue = boxProgress.getValue();
```

```
    int16_t max;
    int16_t min;
    boxProgress.getRange(min, max);

    if (currentValue == min)
    {
        increase = true;
    }
    else if (currentValue == max)
    {
        increase = false;
    }

    int nextValue = increase == true ? currentValue+1 : currentValue-1;

    boxProgress.setValue(nextValue);
}
```

TIP

You can use these functions and the others available in the BoxProgress class in user code. Remember to force a redraw by calling `boxProgress.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the BoxProgress, try creating a new application within TouchGFX Designer with the following UI template:

ProgressIndicator Example UI template in TouchGFX Designer

API Reference

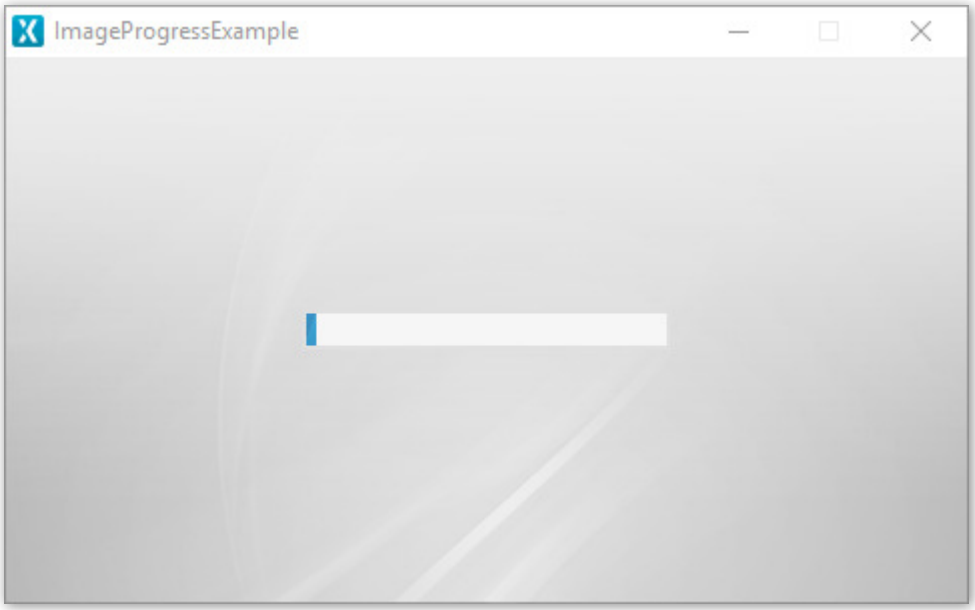
FURTHER READING

- **API reference for the BoxProgress class**

Version: 4.16

ImageProgress

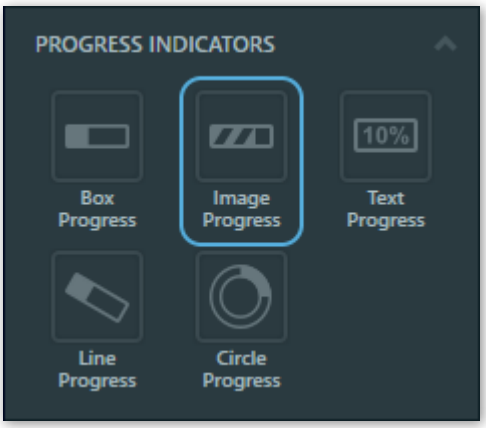
An ImageProgress shows the current progress by using a TiledImage as the progress indicator on top of a background Image. The progress image, the *Alpha* and the *Direction* towards which the image will progress can be configured. It is possible to create a custom background image and change the different parameters of the progress indicator such as the position and the size to fit the custom background image.



ImageProgress running in the simulator

Widget Group

The ImageProgress can be found in the Progress Indicators widget group in TouchGFX Designer.



ImageProgress in TouchGFX Designer

Properties

The properties for an ImageProgress in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of an ImageProgress is determined by the size of the selected background image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Background* sets the background image. *Progress* sets the image used to display progress.
Progress Position	*X* and *Y* coordinates specify the top left corner of the progress image relative to the position of the ProgressIndicator. *W* and *H* specify the width and height of the progress image.
Values	*Range Min* and *Range Max* specify the minimum and maximum integer values of the indicator. *Initial* specifies the initial value of the progress indicator.*Steps Total* specifies at what granularity the progress indicator reports new values. For instance, if the progress needs to be reported as 0%, 10%, 20%, ..., 100%, the total value should be set to 10. *Steps Min* specifies the minimum steps the progress indicator shows.

Property Group	Property Descriptions
Appearance	*Direction* specifies in which direction the progress indicator should progress. *Anchor progress image at zero* specifies if the progress image should be anchored in the 0 point relative to the progress direction, i.e. whether the image is "revealed" or "pulled". *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the ImageProgress are described in the following sections.

Actions

Widget specific actions	Description
Set value	Set the value of the progress indicator.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The ImageProgress does not emit any triggers.

Performance

An ImageProgress consists of a TiledImage and a background Image. Therefore, the ImageProgress is dependent on image drawing and is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up an ImageProgress.

Screen1ViewBase.cpp

```
imageProgress.setXY(148, 126);
imageProgress.setProgressIndicatorPosition(2, 2, 180, 16);
imageProgress.setRange(0, 100);
imageProgress.setDirection(touchgfx::AbstractDirectionProgress::RIGHT);
imageProgress.setBackground(touchgfx::Bitmap(BITMAP_BLUE_PROGRESSINDICATORS_BG_MEDIUM_PROC
imageProgress.setBitmap(BITMAP_BLUE_PROGRESSINDICATORS_FILL_TILING_PROGRESS_INDICATOR_FILL
imageProgress.setValue(0);
imageProgress.setAnchorAtZero(false);
```

TIP

You can use these functions and the others available in the ImageProgress class in user code. Remember to force a redraw by calling `imageProgress1.invalidate()` if you change the appearance of the widget.

User Code

The following example illustrates how to implement the `handleTickEvent()` function to simulate progress. Running this code creates the application shown at the beginning of this article.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
```

```
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();
protected:
    bool increase = true;
};
```

Screen1View.cpp

```
void Screen1View::handleTickEvent()
{
    int currentValue = imageProgress.getValue();
    int16_t max;
    int16_t min;
    imageProgress.getRange(min, max);

    if (currentValue == min)
    {
        increase = true;
    }
    else if (currentValue == max)
    {
        increase = false;
    }

    int nextValue = increase == true ? currentValue+1 : currentValue-1;

    imageProgress.setValue(nextValue);
}
```

TouchGFX Designer Examples

To further explore the ImageProgress, try creating a new application within TouchGFX Designer with the following UI template:

ProgressIndicator Example UI template in TouchGFX Designer

API Reference

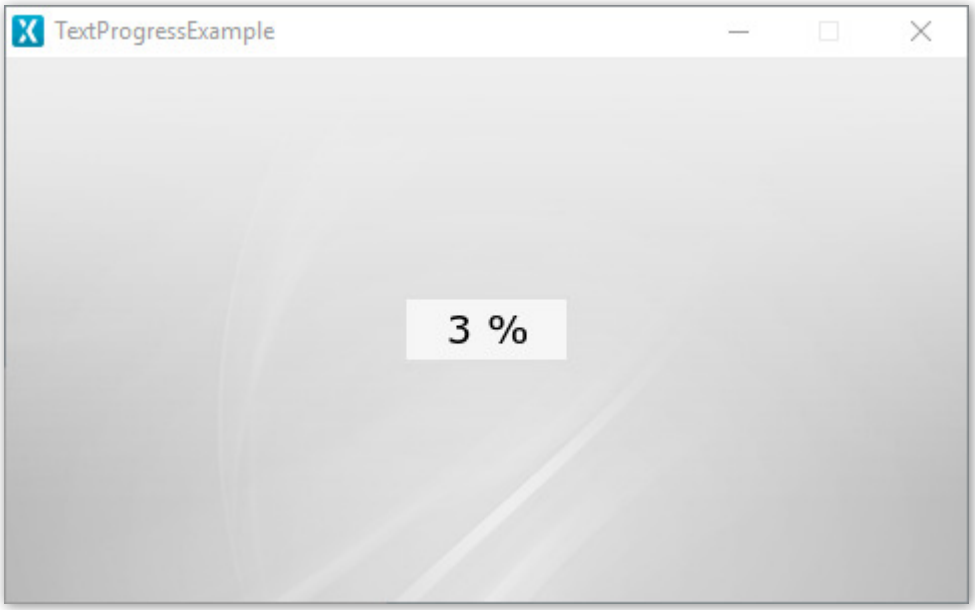
FURTHER READING

- API reference for the ImageProgress class

Version: 4.16

TextProgress

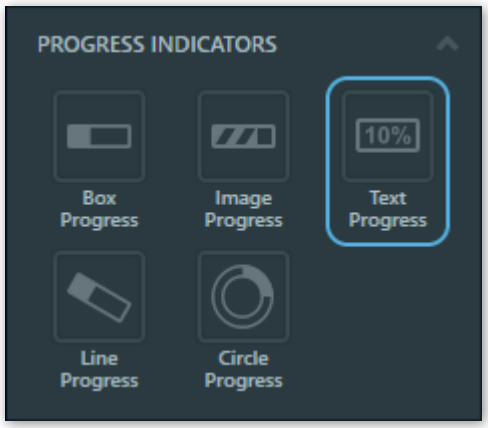
A TextProgress will display progress as a number with a given number of decimals. It shows the current progress by using a TextArea as the progress indicator on top of a background Image. The *Color*, the *Alpha* and the *Text* of the TextArea can be configured. It is possible to create a custom background image and change the different parameters of the progress indicator such as the position and the size to fit the custom background image.



TextProgress running in the simulator

Widget Group

The TextProgress can be found in the Progress Indicators widget group.



TextProgress in TouchGFX Designer

Properties

The properties for a TextProgress in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a TextProgress is determined by the size of the selected background image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Background* specifies the background image.
Text	*Text* specifies the text displayed. The text field is automatically set to use a wildcard "<> %" which means that the wildcard created will be filled with a number that fits the progress configuration. This wildcard is mandatory for the TextProgress to work correctly but any other text can be set before and/or after the wildcard. For more details on text configuration, refer to the Texts and Fonts section.
Text Position & Size	*X* and *Y* specify the top left corner of the progress text relative to its TextProgress parent. *W* and *H* specify the width and height of the progress text.

Property Group	Property Descriptions
Values	*Range Min* and *Range Max* specify the minimum and maximum integer values of the indicator. *Initial* specifies the initial value of the progress indicator.*Steps Total* specifies at what granularity the progress indicator reports new values. For instance, if the progress needs to be reported as 0%, 10%, 20%, ..., 100%, the total value should be set to 10. *Steps Min* specifies the minimum steps the progress indicator shows. *Number of Decimals* specifies the precision required to show progress. The possible values are 0, 1 or 2.
Appearance	*Color* specifies the color of the displayed text. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the TextProgress are described in the following sections.

Actions

Widget specific actions	Description
Set value	Set the value of the progress indicator.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.

Standard widget actions	Description
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The TextProgress widget does not emit any triggers.

Performance

A TextProgress consists of a TextArea and a background Image. Text drawing is very similar to general image drawing (though due to the nature of text characters, a significant amount of alpha blending takes place). Therefore, the TextProgress is considered a fast widget on most platforms.

For more details on text drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a TextProgress.

Screen1ViewBase.cpp

```
textProgress.setXY(198, 119);
textProgress.setProgressIndicatorPosition(0, 0, 84, 34);
textProgress.setRange(0, 100);
textProgress.setColor(touchgfx::Color::getColorFrom24BitRGB(0, 0, 0));
textProgress.setNumberOfDecimals(0);
textProgress.setTypedText(touchgfx::TypedText(T_SINGLEUSEID1));
textProgress.setBackground(touchgfx::Bitmap(BITMAP_BLUE_PROGRESSINDICATORS_BG_MEDIUM_TEXT_
textProgress.setValue(50);
```

User Code

The following example illustrates how to implement the `handleTickEvent()` function to simulate progress. Running this code creates the application shown at the beginning of this article.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();
protected:
    bool increase = true;
    uint8_t counter;
};
```

Screen1View.cpp

```
void Screen1View::handleTickEvent()
{
    counter++;
    if(counter%15 == 0) // Every 0.25 seconds
    {
        int currentValue = textProgress.getValue();
        int16_t max;
        int16_t min;
        textProgress.getRange(min, max);

        if (currentValue == min)
        {
            increase = true;
        }
        else if (currentValue == max)
        {
            increase = false;
        }

        int nextValue = increase == true ? currentValue+1 : currentValue-1;

        counter = 0;
        textProgress.setValue(nextValue);
    }
}
```

TIP

You can use these functions and the others available in the TextProgress class in user code. Remember to force a redraw by calling `textProgress1.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the TextProgress, try creating a new application within TouchGFX Designer with the following UI template:

ProgressIndicator Example UI template in TouchGFX Designer

API Reference

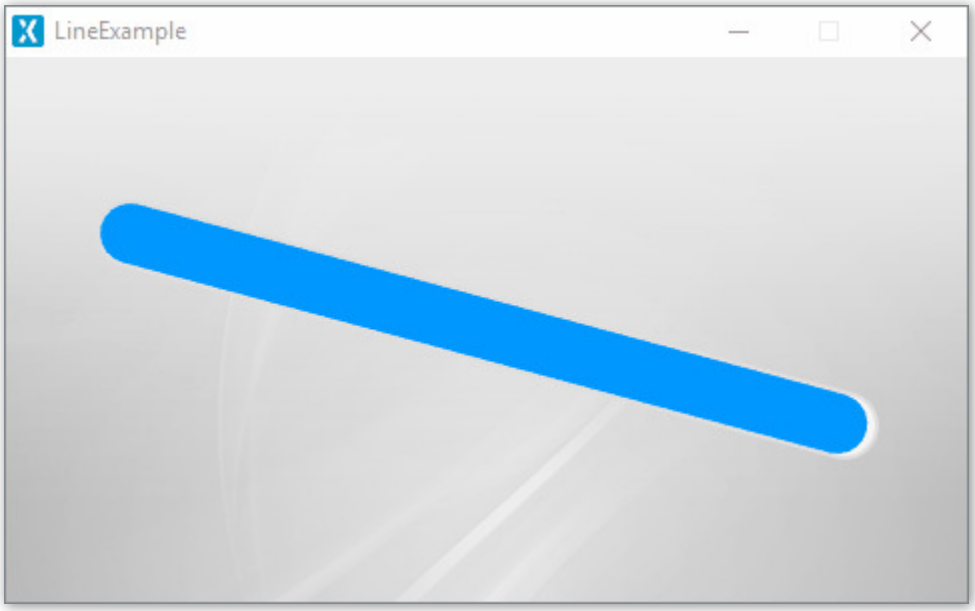
FURTHER READING

- **API reference for the TextProgress class**

Version: 4.16

LineProgress

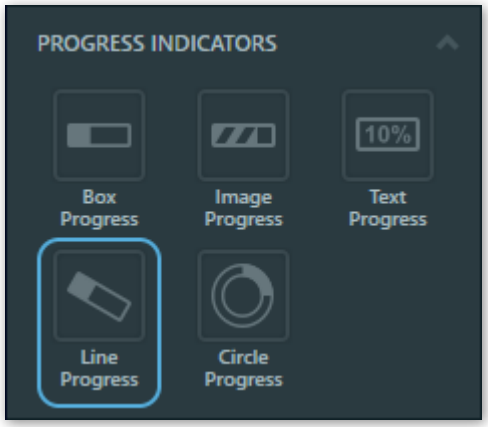
A LineProgress shows the current progress by using a Line as the progress indicator on top of a background Image. The line does not need to be either horizontal or vertical, but can start at any coordinate and finish at any coordinate.



LineProgress running in the simulator

Widget Group

The LineProgress can be found in the Progress Indicators widget group in TouchGFX Designer.



LineProgress in TouchGFX Designer

Properties

The properties for a LineProgress in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a LineProgress is determined by the size of the selected background image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image & Color	*Background* specifies background image. *Image* specifies which image to use as fill for the line. *The image selected will be placed in top left corner of the widget.* *Color* specifies which color to use as fill for the line.
Values	*Range Min* and *Range Max* specify the minimum and maximum integer values of the indicator. *Initial* specifies the initial value of the progress indicator.*Steps Total* specifies at what granularity the progress indicator reports new values. For instance, if the progress needs to be reported as 0%, 10%, 20%, ..., 100%, the total value should be set to 10. *Steps Min* specifies the minimum steps the progress indicator shows.

Property Group	Property Descriptions
Appearance	*Start Position X* and *Start Position Y* specify the coordinate where the line should start. *End Position X* and *End Position Y*specify the coordinate where the line should end. *Line Width* specifies the width of the line. *Cap Style* specifies line ending style. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the LineProgress are described in the following sections.

Actions

Widget specific actions	Description
Set value	Sets the value of a progress indicator to a desired value

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The LineProgress does not emit any triggers.

Performance

A LineProgress consists of an Image and a Line. A Line is a CanvasWidget and is heavily dependent on the MCU for rendering. Therefore, the LineProgress is considered a demanding widget on most platforms.

For more details on CanvasWidget drawing performance, read the UI Component Performance Overview.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a LineProgress.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase()
{
    touchgfx::CanvasWidgetRenderer::setupBuffer(canvasBuffer, CANVAS_BUFFER_SIZE);

    lineProgress.setXY(45, 71);
    lineProgress.setProgressIndicatorPosition(0, 0, 391, 130);
    lineProgress.setRange(0, 100);
    lineProgress.setBackground(touchgfx::Bitmap(BITMAP_BLUE_PROGRESSINDICATORS_BG_LARGE_PR
    lineProgressPainter.setColor(touchgfx::Color::getColorFrom24BitRGB(0, 151, 255));
    lineProgress.setPainter(lineProgressPainter);
    lineProgress.setStart(17, 17);
    lineProgress.setEnd(374, 113);
    lineProgress.setLineWidth(30);
    lineProgress.setLineEndingStyle(touchgfx::Line::ROUND_CAP_ENDING);
    lineProgress.setValue(60);

    add(lineProgress);
}
```

```
void Screen1ViewBase::setupScreen()
{

}
```

TIP

- You can use these functions and the others available in the LineProgress class in user code. Remember to force a redraw by calling `lineProgress.invalidate()` if you change the appearance of the widget.

User Code

The following example illustrates how to implement the `handleTickEvent()` function to simulate progress. Running this code creates the application shown at the beginning of this article.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();
protected:
    bool increase = true;
};
```

Screen1View.cpp

```
void Screen1View::handleTickEvent()
{
    int currentValue = lineProgress.getValue();
    int16_t max;
    int16_t min;
    lineProgress.getRange(min, max);

    if (currentValue == min)
    {
        increase = true;
    }
    else if (currentValue == max)
    {
        increase = false;
    }
```

```
    int nextValue = increase == true ? currentValue+1 : currentValue-1;

    lineProgress.setValue(nextValue);
}
```

TouchGFX Designer Examples

To further explore the LineProgress, try creating a new application within TouchGFX Designer with one of the following UI templates:

ProgressIndicator Example UI template in TouchGFX Designer

API Reference

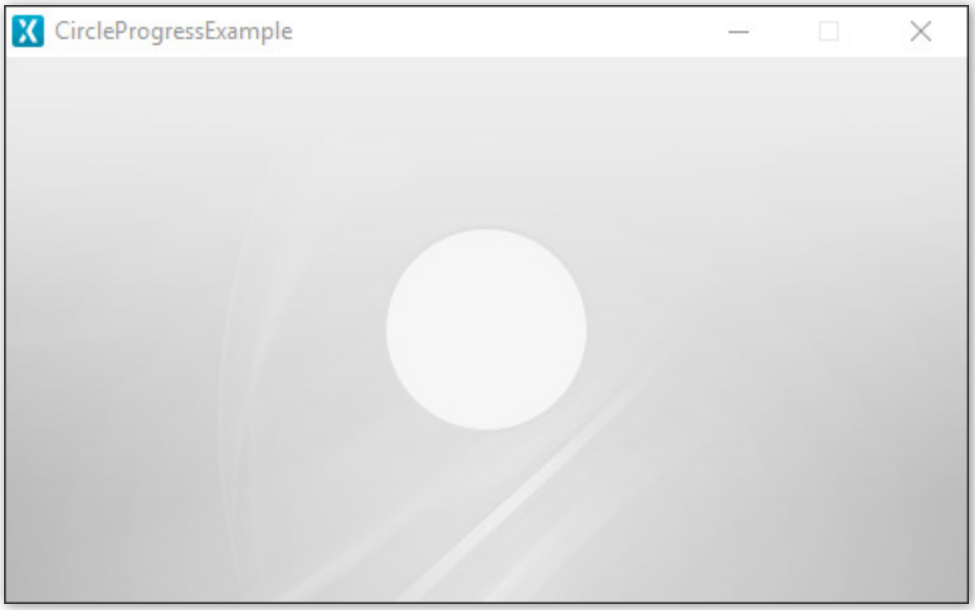
FURTHER READING

- **API reference for the LineProgress class**

Version: 4.16

CircleProgress

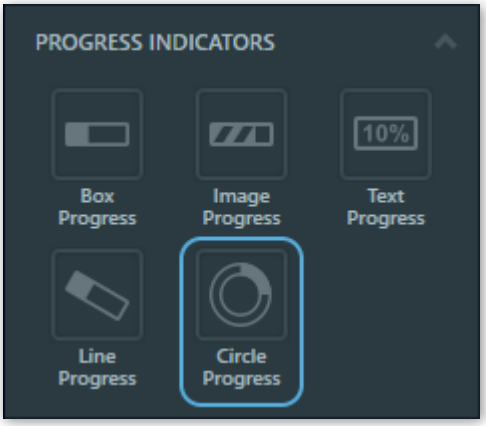
A CircleProgress shows the current progress by using a Circle as the progress indicator on top of a background Image. The *Color*, the *Alpha* and overall parameters regarding the Circle can be configured. It is possible to create a custom background image and change the different parameters of the progress indicator such as the position and the size to fit the custom background image.



CircleProgress running in the simulator

Widget Group

The CircleProgress can be found in the Progress Indicators widget group in TouchGFX Designer.



CircleProgress in TouchGFX Designer

Properties

The properties for a CircleProgress in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a CircleProgress is determined by the size of the selected background image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image & Color	*Background* specifies background image. *Progress* specifies either a color or an image to use as fill for the Circle.
Values	*Range Min* and *Range Max* specify the minimum and maximum integer values of the indicator. *Initial* specifies the initial value of the progress indicator.*Steps Total* specifies at what granularity the progress indicator reports new values. For instance, if the progress needs to be reported as 0%, 10%, 20%, ..., 100%, the total value should be set to 10. *Steps Min* specifies the minimum steps the progress indicator shows.

Property Group	Property Descriptions
Appearance	*Center Position X* and *Center Position Y* specify the center position of the progress circle relative to its CircleProgress parent. *Start & End Angle* specify the start and end angle of the drawn Circle. *Radius* specifies the size of the progress cirlce. *Line Width* specifies the width of the progress circle. If the value is 0, the progress circle is as large as the radius. Otherwise, the width specified will configure the width of the progress circle starting outside and moving inwards. *Cap Style* specifies line ending style. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the CircleProgress are described in the following sections.

Actions

Widget specific actions	Description
Set value	Set the value of the progress indicator.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).

Standard widget actions	Description
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A CircleProgress does not emit any triggers.

Performance

A CircleProgress consists of a Circle and a background Image. The Circle is based on the CanvasWidget and is heavily dependent on the MCU for rendering. Therefore, the CircleProgress is considered a demanding widget on most platforms.

For more details on CanvasWidget drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a CircleProgress.

Screen1ViewBase.cpp

```
circleProgress.setXY(188, 84);
circleProgress.setProgressIndicatorPosition(0, 0, 104, 104);
circleProgress.setRange(0, 100);
circleProgress.setCenter(52, 52);
circleProgress.setRadius(50);
circleProgress.setLineWidth(0);
circleProgress.setStartEndAngle(0, 360);
circleProgress.setBackground(touchgfx::Bitmap(BITMAP_BLUE_PROGRESSINDICATORS_BG_MEDIUM_CI
circleProgressPainter.setBitmap(touchgfx::Bitmap(BITMAP_BLUE_PROGRESSINDICATORS_FILL_MEDI
circleProgress.setPainter(circleProgress1Painter);
circleProgress.setValue(0);
```

User Code

The following example illustrates how to implement the `handleTickEvent()` function to simulate progress. Running this code creates the application shown at the beginning of this article.

Screen1View.hpp

```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();
protected:
    bool increase = true;
};
```

Screen1View.cpp

```
void Screen1View::handleTickEvent()
{
    int currentValue = circleProgress.getValue();
    int16_t max;
    int16_t min;
    circleProgress.getRange(min, max);

    if (currentValue == min)
    {
        increase = true;
    }
    else if (currentValue == max)
    {
        increase = false;
    }

    int nextValue = increase == true ? currentValue+1 : currentValue-1;

    circleProgress.setValue(nextValue);
}
```

TIP

You can use these functions and the others available in the CircleProgress class in user code. Remember to force a redraw by calling `circleProgress1.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the CircleProgress, try creating a new application within TouchGFX Designer with the following UI template:

ProgressIndicator Example UI template in TouchGFX Designer

API Reference

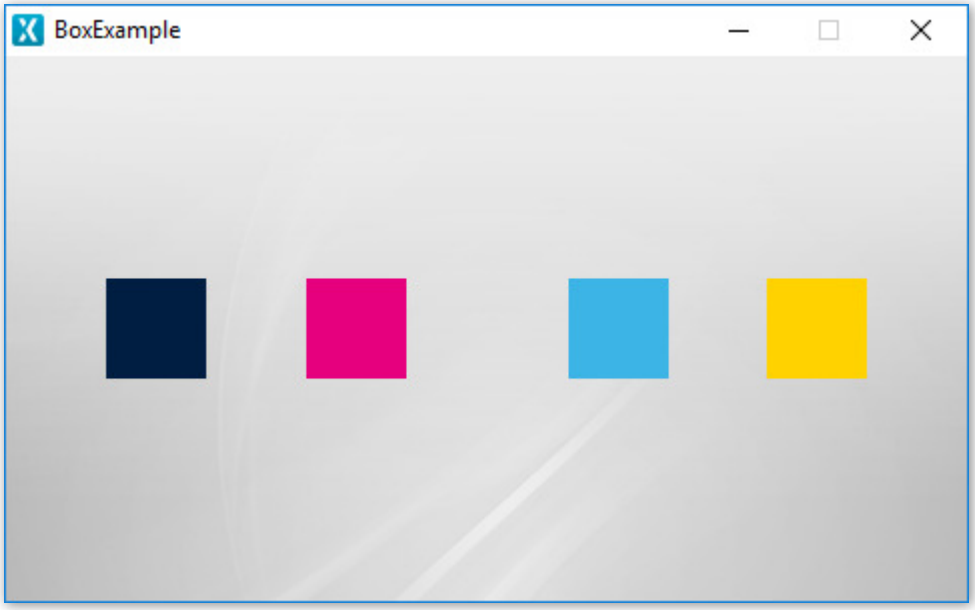
FURTHER READING

- **API reference for the CircleProgress class**

Version: 4.16

Box

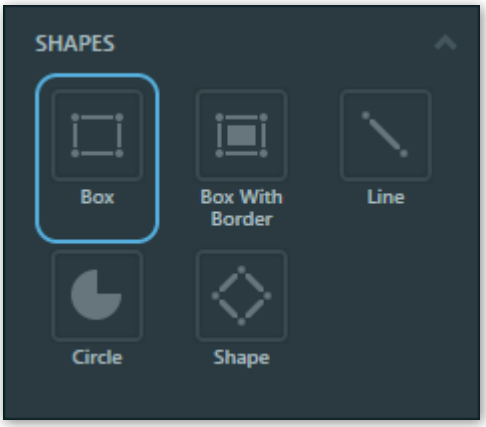
A Box in TouchGFX is a rectangular shaped widget that can be assigned a single color for all contained pixels. The Box can be assigned any size and position.



Box running in the simulator

Widget Group

The Box can be found in the Shapes widget group in TouchGFX Designer.



Box in TouchGFX Designer

Properties

The properties for a Box in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. <InlineNote text="Making the widget invisible also disables interacting with the widget through the screen."
Appearance	*Color* specifies the color of all the pixels contained within the rectangle. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the Box are described in the following sections.

Actions

Widget specific actions	Description
Resize widget	Resize a widget.
Change box color	Change the color of a Box widget.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.

Standard widget actions	Description
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A Box does not emit any triggers.

Performance

A Box is one of the most lightweight widgets in all of TouchGFX because it does not have to read any pixel data or do any complicated calculations. Therefore, the Box is considered a very fast performing widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Box.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase()
{
    boxName.setPosition(260, 133, 294, 99);
    boxName.setColor(touchgfx::Color::getColorFrom24BitRGB(33, 197, 80));

    add(boxName);
}
```

TIP

You can use these functions and the others available in the Box class in user code. Remember to force a redraw by calling `boxName.invalidate()` if you change the appearance of the widget.

API Reference

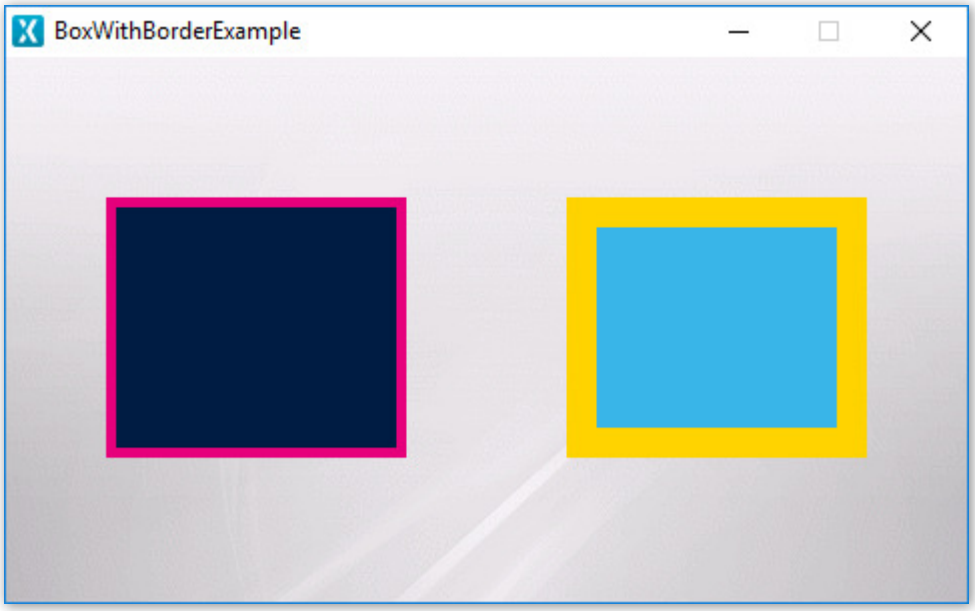
FURTHER READING

- **API reference for the Box class**

Version: 4.16

BoxWithBorder

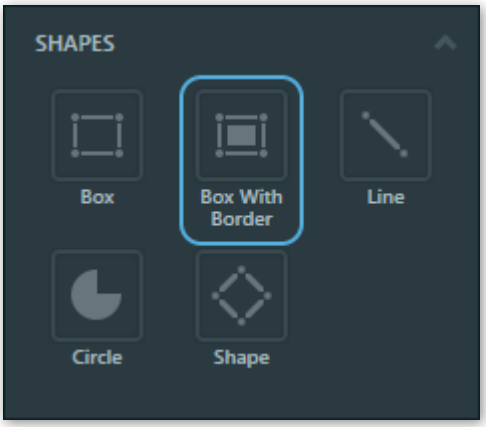
A BoxWithBorder in TouchGFX is a rectangular shaped widget that can be assigned a single color for all contained pixels within a specified border with a separate color and size. The BoxWithBorder can be assigned any size and position.



BoxWithBorder running in the simulator

Widget Group

The BoxWithBorder can be found in the Shapes widget group in TouchGFX Designer.



BoxWithBorder in TouchGFX Designer

Properties

The properties for a BoxWithBorder in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Appearance	*Color* specifies the color of all the pixels contained within the rectangle. *Border Color* specifies the color of the outer border pixels. *Border Size* specifies the size of the outer border. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by a BoxWithBorder in TouchGFX Designer.

Actions

Widget specific actions	Description
Resize widget	Resize a widget.
Change box color	Change the color of a Box widget.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A BoxWithBorder does not emit any triggers.

Performance

A BoxWithBorder is one of the most lightweight widgets in all of TouchGFX because it does not have to read any pixel data or do any complicated calculations. Therefore, the BoxWithBorder is considered a very fast performing widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a BoxWithBorder widget.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase()
{
    boxWithBorderName.setPosition(240, 56, 337, 132);
    boxWithBorderName.setColor(touchgfx::Color::getColorFrom24BitRGB(212, 27, 27));
    boxWithBorderName.setBorderColor(touchgfx::Color::getColorFrom24BitRGB(21, 24, 202));
    boxWithBorderName.setBorderSize(20);

    add(boxWithBorderName);
```

```
}
```

TIP

- You can use these functions and the others available in the BoxWithBorder class in user code. Remember to force a redraw by calling `boxWithBorderName.invalidate()` if you change the appearance of the widget.

API Reference

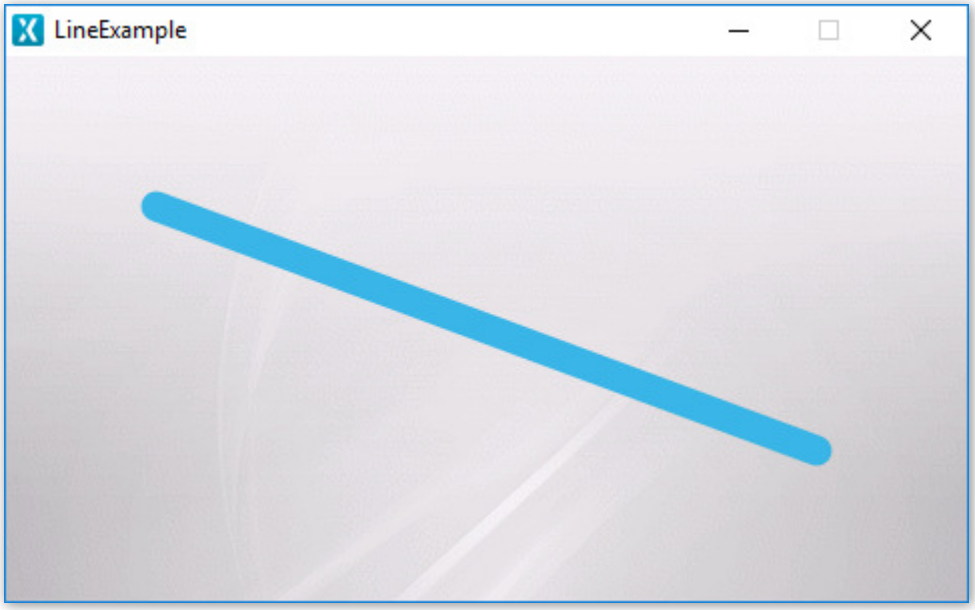
FURTHER READING

- **API reference for the BoxWithBorder class**

Version: 4.16

Line

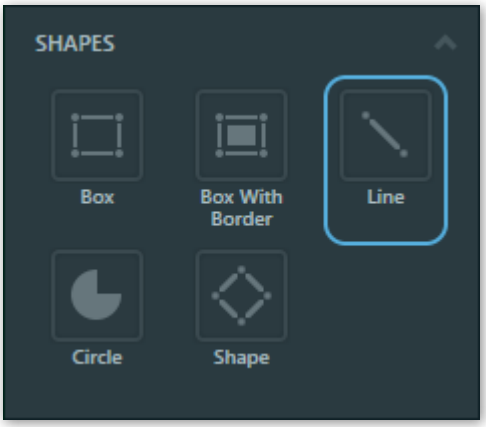
A Line is a widget based on the CanvasWidget capable of drawing a straight line from one point to another. The Line can be filled by a single color or an image using a Painter object.



Line running in the simulator

Widget Group

The Line can be found in the Shapes widget group in TouchGFX Designer.



Line in TouchGFX Designer

Properties

The properties for a Line in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Image & Color	*Image* specifies the image assigned to the line from the Designer skin library or the Project folder. If the image is not chosen, *Color* specifies the color of the line.
Appearance	*Start Position X* and *Start Position Y* specify the start coordinates of the line relative to the top left corner of the widget. *End Position X* and *End Position Y* specify the end coordinates of the line relative to the top left corner of the widget. *Line Width* specifies the width of the line. *Cap Style* specifies the shape of the edges of the line. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the Line are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A Line does not emit any triggers.

Performance

A Line is a CanvasWidget and is heavily dependent on the MCU for rendering. Therefore, the Line is considered a demanding widget on most platforms.

For more details on CanvasWidget drawing performance, read the General UI Component Performance.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Line.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase()
{
    lineName.setPosition(0, 0, 800, 480);
    lineNamePainter.setBitmap(touchgfx::Bitmap(BITMAP_DARK_BACKGROUNDS_MAIN_BG_800X480PX_I
    lineName.setPainter(lineNamePainter);
    lineName.setStart(200, 200);
```

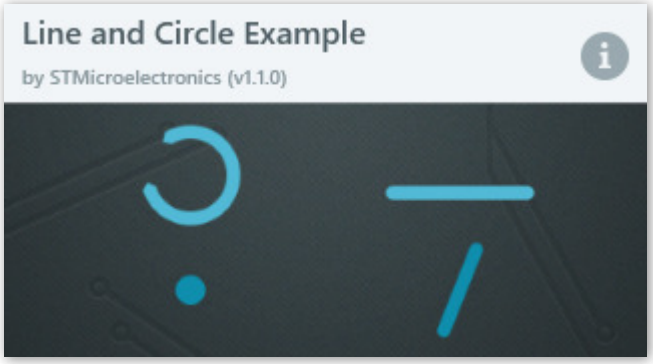
```
    lineName.setEnd(550, 150);
    lineName.setLineWidth(50);
    lineName.setLineEndingStyle(touchgfx::Line::ROUND_CAP_ENDING);

    add(lineName);
}
```

TIP

You can use these functions and the other available in the Line class in user code. Remember to force a redraw by calling `lineName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the Line, try creating a new application within TouchGFX Designer with the following UI template:



Line and Circle Example UI template in TouchGFX Designer

API Reference

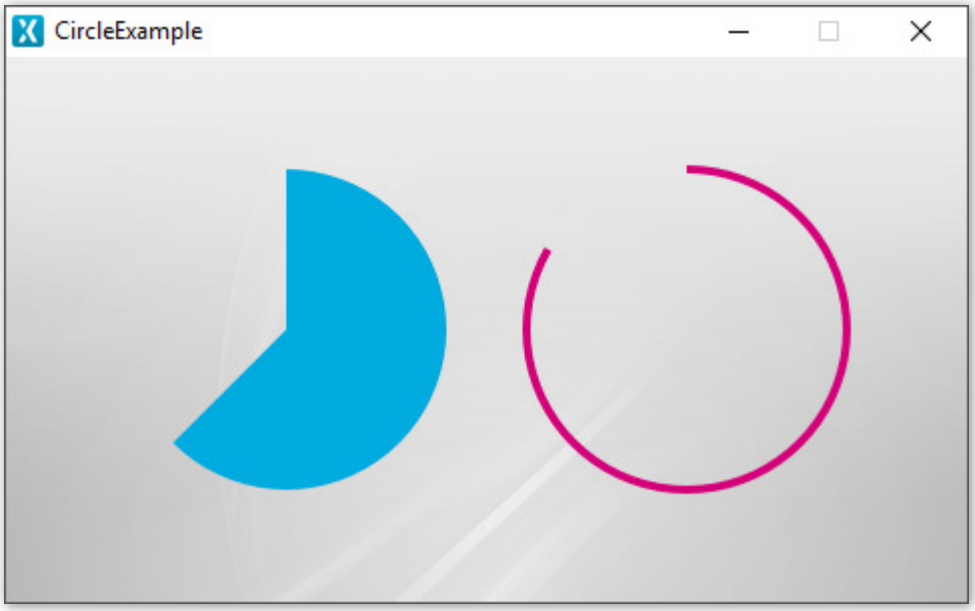
FURTHER READING

- **API reference for the Line class**

Version: 4.16

Circle

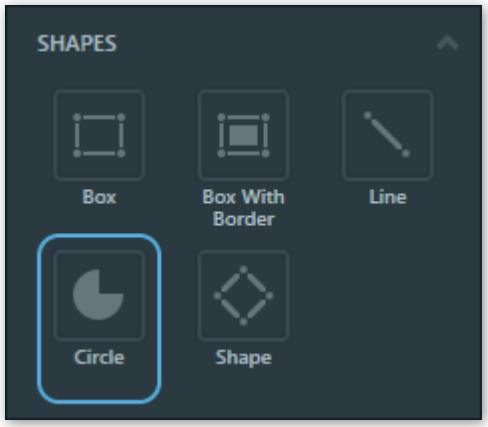
A Circle is a widget based on the CanvasWidget capable of drawing a circle. This circle can be a partial circle, and either filled or outlined. The center, radius, line width, line cap and circle arc can be modified. The Circle can either use an image or a single color as fill.



Circle running in the simulator

Widget Group

The Circle can be found in the Shapes widget group in TouchGFX Designer.



Circle in TouchGFX Designer

Properties

The properties for the Circle are described in the following sections.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Image & Color	*Image* specifies the image used to fill the circle from the Designer skin library or the Project folder. If an image is not chosen, *Color* specifies the color used to fill the circle.
Appearance	*Center Position X* and *Center Position Y* specify the coordinates for the center of the circle, relative to the top left corner of the widget. *Start & End Angle* specify the angles in degrees of the start and ending points of the circle. *Radius* specifies the radius of the circle. *Line Width* specifies the width of the line forming the circumference of the circle. *Set this to 0 to get a filled circle.* *Cap Style* specifies the shape of the edges of the circle. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the Circle are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A Circle does not emit any triggers.

Performance

A Circle is based on the CanvasWidget and is heavily dependent on the MCU for rendering. Therefore, the Circle is considered a demanding widget on most platforms.

For more details on CanvasWidget drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Circle.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <touchgfx/Color.hpp>
```

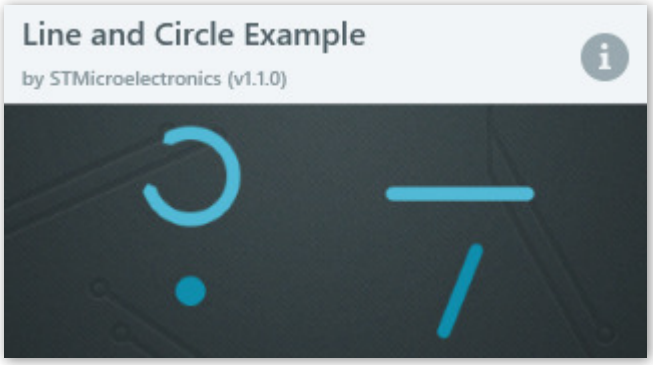
```
Screen1ViewBase::Screen1ViewBase()
{

    touchgfx::CanvasWidgetRenderer::setupBuffer(canvasBuffer, CANVAS_BUFFER_SIZE);

    circleName.setPosition(40, 36, 200, 200);
    circleName.setCenter(100, 100);
    circleName.setRadius(80);
    circleName.setLineWidth(0);
    circleName.setArc(0, 225);
    circleName.setCapPrecision(180);
    circleNamePainter.setColor(touchgfx::Color::getColorFrom24BitRGB(0, 171, 223));
    circleName.setPainter(circleNamePainter);

    add(circleName);
}
```

TIP

You can use these functions and the others available in the Circle class in user code. Remember to force a redraw by calling `circleName.invalidate()` if you change the appearance of the widget.

TouchGFX Designer Examples

To further explore the Circle, try creating a new application within TouchGFX Designer with the following UI template:



Line and Circle Example UI template in TouchGFX Designer

API Reference

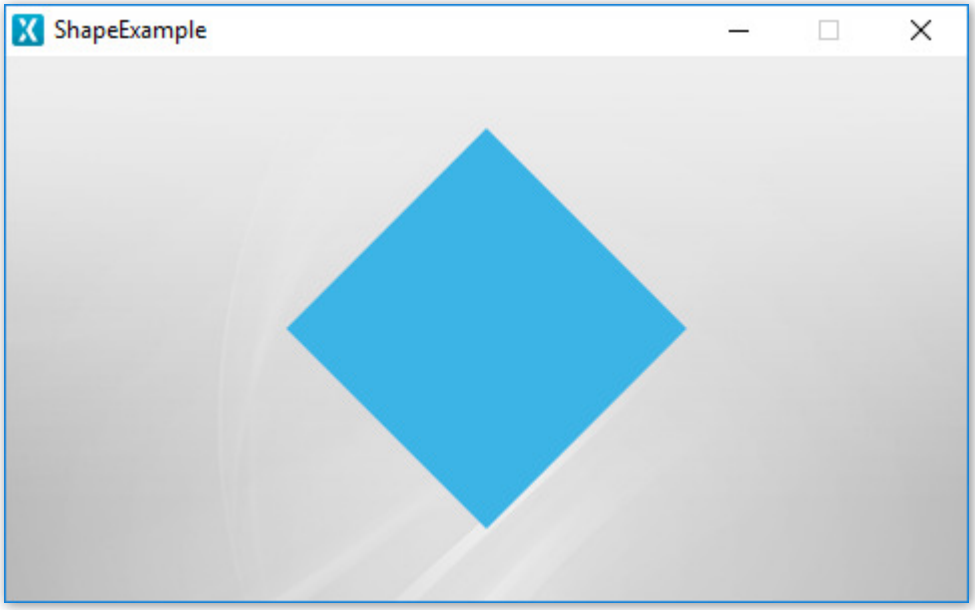
FURTHER READING

- **API reference for the Circle class**
- **API reference for the Canvas class used to draw a Circle**

Version: 4.16

Shape

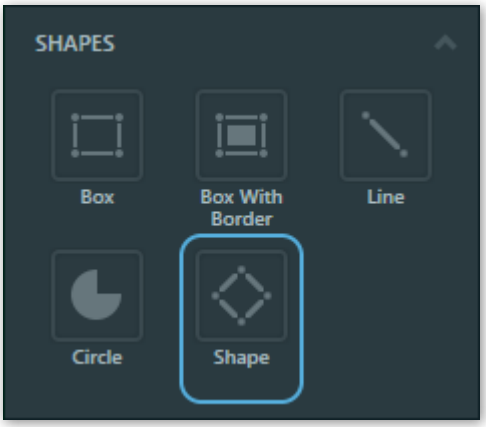
The Shape allows the user to draw any shape with an arbitrary amount of points, while also supporting scaling and rotation.



Shape running in the simulator

Widget Group

The Shape can be found in the Shapes widget group in TouchGFX Designer.



Shape in TouchGFX Designer

Properties

The properties for a Shape in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Image & Color	*Image* or *Color* specifies the color or image to be shown in the Shape.
Transform	*Angle* specifies the rotation of the Shape around its *Origin* point. *Scale X* and *Scale Y* specify the scale of the shape horizontally and vertically from the *Origin* point. *Origin X* and *Origin Y* specify the location from which *Points* originate.
Points	*Points* specify points that create the desired shape. *Each individual point requires an X and Y coordinate.*
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the Shape are described in the following sections.

Actions

Widget specific action	Description
Scale Shape	Scale a Shape, either to a fixed size or relative to its current size
Rotate Shape	Rotate a Shape, either to a fixed angle or relative to its current anlge

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A Shape does not emit any triggers.

Performance

A Shape is a CanvasWidget and is heavily dependent on the MCU for rendering the desired shape. Therefore, a Shape is considered a demanding widget on most platforms.

For more details on CanvasWidget drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Shape.

Screen1ViewBase.cpp

```
#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <touchgfx/Color.hpp>

Screen1ViewBase::Screen1ViewBase()
```

```
{
    touchgfx::CanvasWidgetRenderer::setupBuffer(canvasBuffer, CANVAS_BUFFER_SIZE);

    shape.setPosition(140, 36, 200, 200);
    shape.setOrigin(100.000f, 100.000f);
    shape.setScale(1.000f, 1.000f);
    shape.setAngle(0.000f);
    shapePainter.setColor(touchgfx::Color::getColorFrom24BitRGB(60, 180, 230));
    shape.setPainter(shapePainter);
    const touchgfx::AbstractShape::ShapePoint<float> shapePoints[4] = { { 0.000f, -100.000
    shape.setShape(shapePoints);

    add(shape);
}

void Screen1ViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the Shape class in user code. Remember to force a redraw by calling `shape.invalidate()` if you change the appearance of the widget.

User Code

The following code shows how to set up a shape filled with some random data:

Screen1View.hpp

```
#ifndef SCREEN1VIEW_HPP
#define SCREEN1VIEW_HPP

#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <gui/screen1_screen/Screen1Presenter.hpp>
#include <touchgfx/widgets/canvas/Shape.hpp>
#include <touchgfx/widgets/canvas/PainterRGB888.hpp>

class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();

    /*
     * Member Declarations
```

```
     */
    touchgfx::Shape<100> shape;
    touchgfx::PainterRGB888 shapePainter;
protected:
    void fillData(int maxLength);
};

#endif // SCREEN1VIEW_HPP
```

Screen1View.cpp

```
#include <gui/screen1_screen/Screen1View.hpp>
#include <touchgfx/Color.hpp>

Screen1View::Screen1View()
{
}

void Screen1View::setupScreen()
{
    Screen1ViewBase::setupScreen();
    shape.setPosition(0, 0, 480, 272);
    shape.setOrigin(0.000f, 272.000f);
    shapePainter.setColor(touchgfx::Color::getColorFrom24BitRGB(0, 171, 223));
    shape.setPainter(shapePainter);
    fillData(100);
    add(shape);
}

void Screen1View::tearDownScreen()
{
    Screen1ViewBase::tearDownScreen();
}

void Screen1View::fillData(int maxLength)
{
    float highestX = 0.000f;
    for (int i = 0; i < maxLength - 1; ++i)
    {
        float y = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/272));
        float x = highestX + static_cast <float> (rand()) /( static_cast <float> (RAND_MA
        highestX = x;
        shape.setCorner(i, CWRUtil::toQ5<float>(x), CWRUtil::toQ5<float>(-y));
    }
    shape.setCorner(maxLength - 1, CWRUtil::toQ5<float>(highestX), CWRUtil::toQ5<float>(0
    shape.updateAbstractShapeCache();
}
```

API Reference

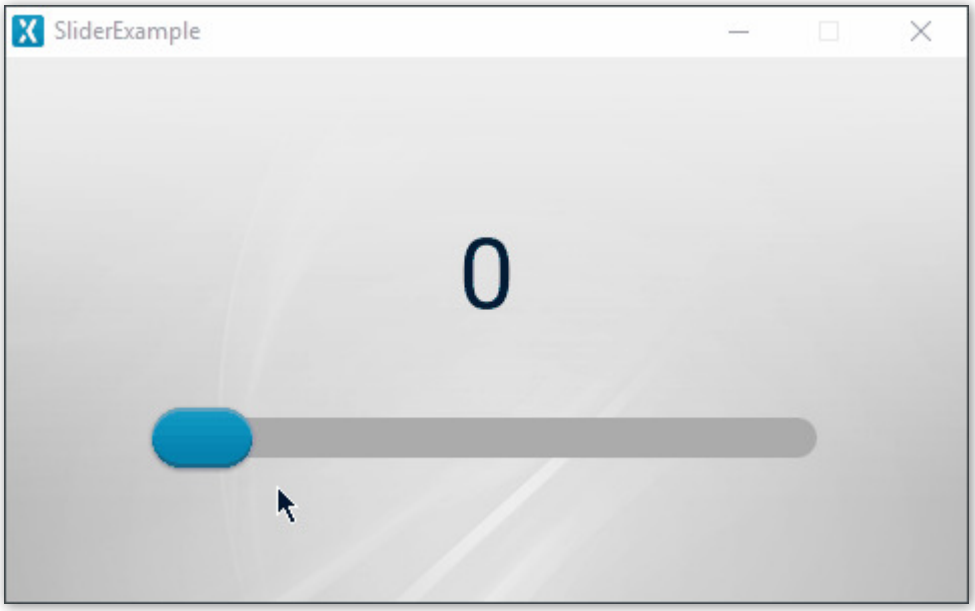
FURTHER READING

- API reference for the Shape class

Version: 4.16

Slider

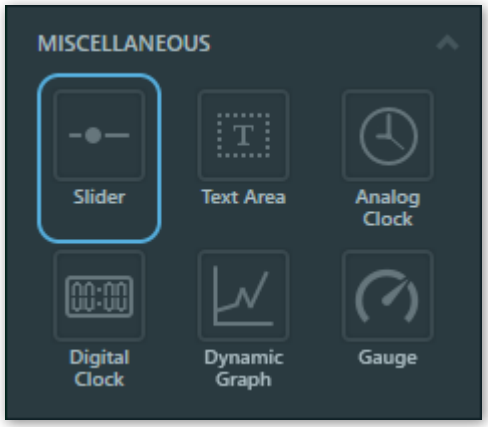
A Slider uses three images to display a slider either in a vertical or horizontal orientation. The indicator image of a Slider can be dragged to modify an internal integer value that is broadcasted through callbacks. The value broadcasted operates on an integer value range *e.g. 0 to 100*.



Slider running in the simulator

Widget Group

The Slider can be found in the Miscellaneous widget group in TouchGFX Designer.



Slider in TouchGFX Designer

Properties

The properties for a Slider in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Type	*Type* specifies whether the Slider should be vertically or horizontally oriented.
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a Slider is taken from the size of the associated images and cannot be altered except by changing the images.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Image	*Background Image* specifies the background image that the indicator slides across. *Background Filled Image* specifies the image filling the area on top of the background image behind the indicator. *Indicator Image* specifies the image that can be dragged to change the value of the slider. *The background image and background filled image must both be the same size.*
Positions	*Background Position X* and *Background Position Y* specify the top left corner position of the *Background Image* and *Background Filled Image*. *Indicator Position Min* and *Indicator Position Max* specify the minimum and maximum positions of the *Indicator Image*. *For a horizontal slider these two values are in the x-axis and for a vertical slider they are in the y-axis.* *Indicator Position Y* specifies the indicator image's position in the y-axis. *If the slider is vertically oriented this value instead adjusts in the x-axis.*

Property Group	Property Descriptions
Values	*Min* and *Max* specifies the internal integer range that is broadcast from the Slider using callbacks. *Start* specifies the initial internal value in the Slider. *This also changes the initial position of the indicator.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the Slider are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Slider adjustment initiated	A Slider has been clicked or dragged.
Slider adjustment confirmed	A Slider indicator is no longer being dragged.
Slider value changed	A Sliders value has changed.

Performance

A Slider consists of three images. Therefore, a Slider is dependent on image drawing and is considered a fast performing widget on most platforms.

For more details on image drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a Slider.

mainViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <texts/TextKeysAndLanguages.hpp>
#include <touchgfx/Color.hpp>

mainViewBase::mainViewBase()
{
    slider.setXY(71, 173);
    slider.setBitmaps(touchgfx::Bitmap(BITMAP_BLUE_SLIDER_HORIZONTAL_MEDIUM_SLIDER2_ROUND_
    slider.setupHorizontalSlider(2, 6, 0, 0, 284);
    slider.setValueRange(0, 100);
    slider.setValue(0);

    add(slider);
}

void mainViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the Slider class in user code. Remember to force a redraw by calling `slider.invalidate()` if you change the appearance of the widget.

User Code

The following code example shows how to set up the three callbacks of a Slider:

- `setStartValueCallback`
- `setNewValueCallback`
- `setStopValueCallback`

mainView.hpp

```
#ifndef MAINVIEW_HPP
#define MAINVIEW_HPP

#include <gui_generated/main_screen/mainViewBase.hpp>
#include <gui/main_screen/mainPresenter.hpp>

class mainView : public mainViewBase
{
public:
    mainView();
    virtual ~mainView() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
protected:

    /*
     * Callback Declarations
     */
    touchgfx::Callback<mainView, const touchgfx::Slider&, int> sliderValueStartedChangeCal
    touchgfx::Callback<mainView, const touchgfx::Slider&, int> sliderValueChangedCallback;
    touchgfx::Callback<mainView, const touchgfx::Slider&, int> sliderValueConfirmedCallbac

    /*
     * Callback Handler Declarations
     */
    void sliderValueStartedChangeCallbackHandler(const touchgfx::Slider& src, int value);
    void sliderValueChangedCallbackHandler(const touchgfx::Slider& src, int value);
    void sliderValueConfirmedCallbackHandler(const touchgfx::Slider& src, int value);
};

#endif // MAINVIEW_HPP
```

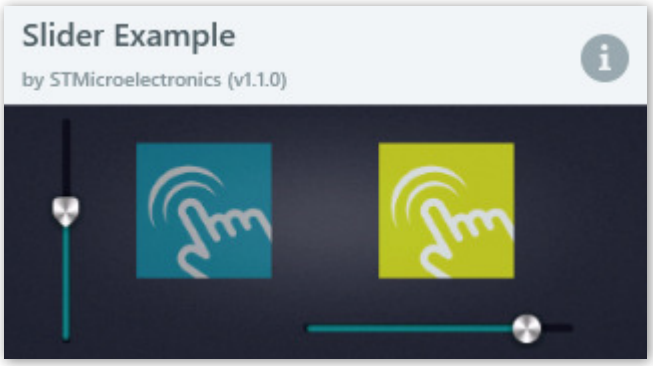
mainView.cpp

```
#include <gui/main_screen/mainView.hpp>

mainView::mainView():
    sliderValueStartedChangeCallback(this, &mainView::sliderValueStartedChangeCallbackHand
    sliderValueChangedCallback(this, &mainView::sliderValueChangedCallbackHandler),
    sliderValueConfirmedCallback(this, &mainView::sliderValueConfirmedCallbackHandler)
{

}
```

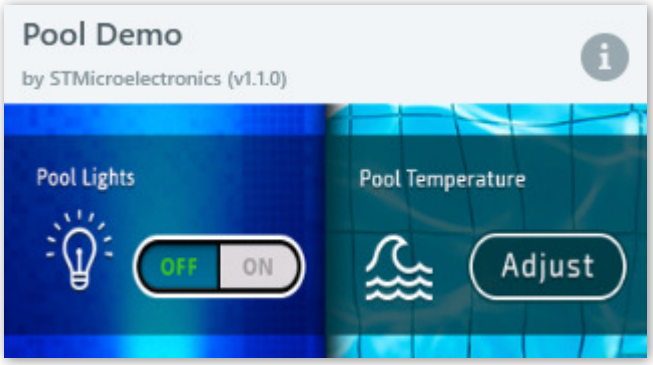
```
void mainView::setupScreen()
{
    mainViewBase::setupScreen();
    slider.setStartValueCallback(sliderValueStartedChangeCallback);
    slider.setNewValueCallback(sliderValueChangedCallback);
    slider.setStopValueCallback(sliderValueConfirmedCallback);
}

void mainView::tearDownScreen()
{
    mainViewBase::tearDownScreen();
}

void mainView::sliderValueStartedChangeCallbackHandler(const touchgfx::Slider& src, int va
{
    if (&src == &slider)
    {
        //execute code whenever the slider starts changing value.
    }
}

void mainView::sliderValueChangedCallbackHandler(const touchgfx::Slider& src, int value)
{
    if (&src == &slider)
    {
        //execute code whenever the value of the slider changes.
    }
}

void mainView::sliderValueConfirmedCallbackHandler(const touchgfx::Slider& src, int value)
{
    if (&src == &slider)
    {
        //execute code whenever the slider stops the changing value.
    }
}
```

TouchGFX Designer Examples

To further explore the Slider, try creating a new application within TouchGFX Designer with one of the following UI templates:



Slider Example UI template in TouchGFX Designer



Pool Demo UI template in TouchGFX Designer

API Reference

FURTHER READING

- **API reference for the Slider class**

Version: 4.16

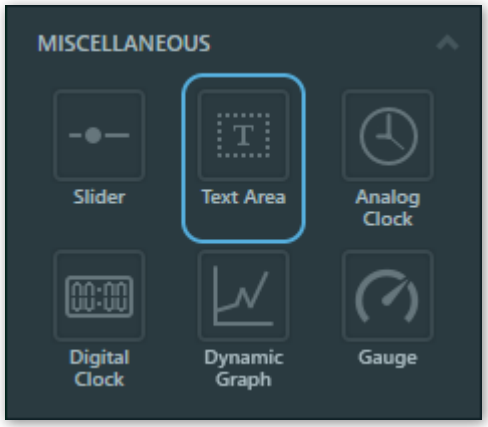
TextArea

A TextArea displays text on the screen. The text of a TextArea can be entirely configured in size, color, custom fonts, dynamic texts etc. For more information on how to handle texts in TouchGFX Designer, read the Texts and Fonts article.

TextArea running in the simulator

Widget Group

The TextArea can be found in the Miscellaneous widget group in TouchGFX Designer.



TextArea in TouchGFX Designer

Properties

The properties for a TextArea in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Auto-size* specifies whether the size of the widget will be automatically set according to the text input. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Text	*Single Use* and *Ressource* specify the type of text: unique or from a known ressource. When Single Use is selected: *Text* specifies the content of the text to be displayed. *Typography* specifies the format of the text. *Alignment* specifies the horizontal alignment of the text. When Ressource is selected: *Ressource ID* specifies the Ressource to retrieve the text from. Up to two wildcards can be created for dynamic text input, which are indicated as '<tag>' where 'tag' can be any string.
Appearance	*Color* specifies the color of the displayed text. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.* *Line Spacing* specifies the space between lines. *Text Rotation* sets the rotation in degrees for the text.

Property Group	Property Descriptions
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Interactions

The actions and triggers supported by the TextArea are described in the following sections.

Actions

Widget specific action	Description
Set text	Set the text of the widget.
Resize widget	Resize the widget.
Set wildcard	Set the wildcard of the widget. A wildcard has to be already added to the TextArea for this action to work.

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

The TextArea does not emit any triggers.

Performance

A TextArea is dependent on text drawing. Text drawing is very similar to general image drawing (though due to the nature of text characters, a significant amount of alpha blending takes place). Therefore, the TextArea is considered a fast performing widget on most platforms.

For more details on text drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up a TextArea.

Screen1ViewBase.hpp

```
touchgfx::TextAreaWithOneWildcard textArea;

/*
 * Wildcard Buffers
 */
static const uint16_t TEXTAREA_SIZE = 20;
touchgfx::Unicode::UnicodeChar textAreaBuffer[TEXTAREA_SIZE];
```

Screen1ViewBase.cpp

```
textArea.setPosition(40, 111, 400, 50);
textArea.setColor(touchgfx::Color::getColorFrom24BitRGB(60, 180, 230));
textArea.setLinespacing(0);
Unicode::snprintf(textAreaBuffer, TEXTAREA_SIZE, "%s", touchgfx::TypedText(T_TOUCHGFXID).g
textArea.setWildcard(textAreaBuffer);
textArea.setTypedText(touchgfx::TypedText(T_SINGLEUSEID1));
```

TIP

You can use these functions and the others available in the TextArea class in user code. Remember to force a redraw by calling `textArea.invalidate()` if you change the appearance of the widget.

User Code

The following example illustrates how to implement the `handleTickEvent()` function to change the text at runtime using a wildcard. Running this code creates the application shown at the beginning of this section.

Screen1View.hpp

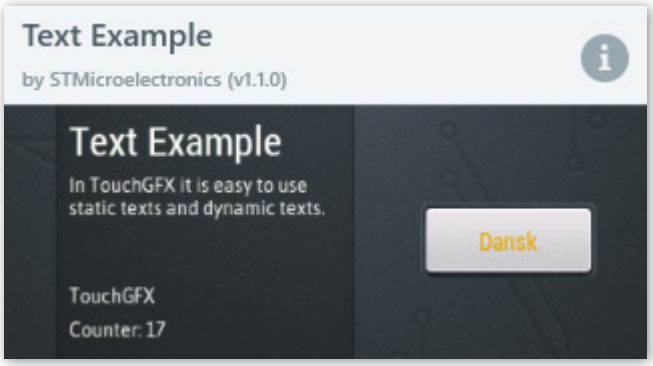
```
class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();

    virtual void handleTickEvent();
protected:
    uint8_t counter;
    bool flag;
};
```

Screen1View.cpp

```
Screen1View::Screen1View():
    counter(0),
    flag(true)
{}

void Screen1View::handleTickEvent()
{
    counter++;
    if(counter%120 == 0) // every 2s
    {
        if(flag)
        {
            Unicode::snprintf(textAreaBuffer, TEXTAREA_SIZE, "%s", touchgfx::TypedText(T_
            flag = false;
        }
        else
        {
            Unicode::snprintf(textAreaBuffer, TEXTAREA_SIZE, "%s", touchgfx::TypedText(T_
            flag = true;
        }
        textArea.invalidate();
        counter = 0;
    }
}
```

TouchGFX Designer Examples

To further explore the TextArea, try creating a new application within TouchGFX Designer with one of the following UI templates:



Text Example UI template in TouchGFX Designer

Arabic Text Example UI template in TouchGFX Designer

API Reference

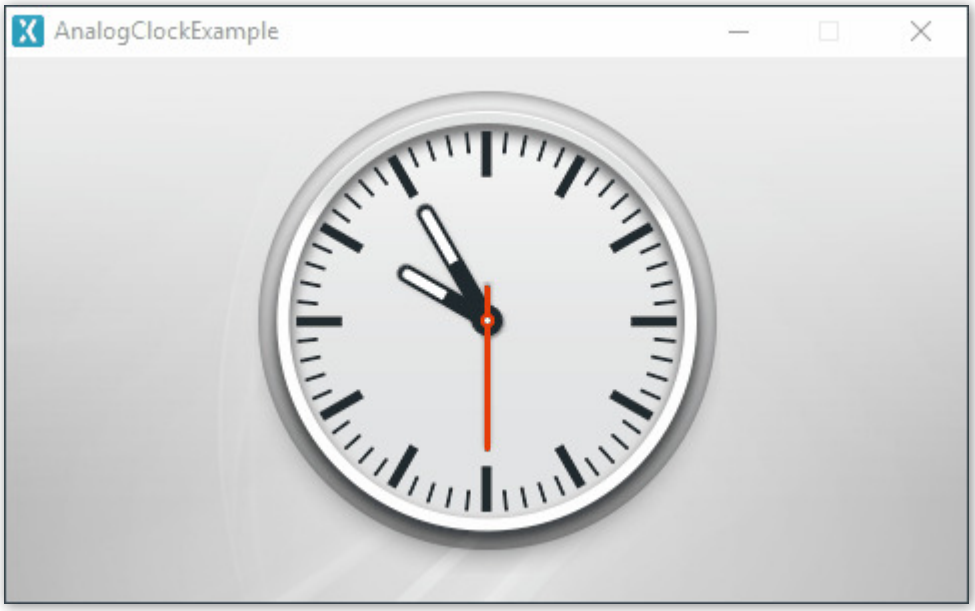
FURTHER READING

- **API reference for the TextArea class**

Version: 4.16

AnalogClock

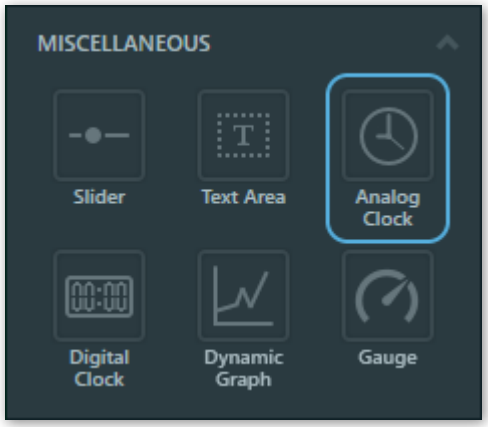
An AnalogClock is a widget that enables the display of a classic analog watch, as opposed to the DigitalClock which displays time with text. The clock uses a background image as the clock face. The hour, minute and second hands are each using an image and rotate around a configurable center.



AnalogClock running in the simulator (sped up footage)

Widget Group

The AnalogClock can be found in the Miscellaneous widget group in TouchGFX Designer.



AnalogClock in TouchGFX Designer

Properties

The properties for a AnalogClock in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *The size of a AnalogClock is taken from the size of the associated image and cannot be altered except by changing the image.* *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Appearance	*Image* specifies the image to be used as background. *Rotation Center X* and *Rotation Center Y* specifies the point at which the clock hands should rotate
Time	*Use Am/Pm* specifies if time should be in 12h or 24h format. *Initial Time* specifies the initial time the clock shows.
Clock Hands	*Clock Hands* specifies which clock hands *(Second, Minute and Hour Hand)* the AnalogClock should show and the order of the hands. Each clock hand can set a *Hand Image* and their rotation point by setting *Rotation Position X* and *Rotation Position Y*. The Minute and Hour Hand have the option to use sweeping hand motion by setting *Sweeping Movement*
Animations	*Animate Clock Hand Movement* specifies if animation of the clock hands are enabled. *Duration* specifies how long the amination is. *Easing* and *Easing Option* specify the easing equation used.

Property Group	Property Descriptions
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Time

The Time property group allows the user to set the initial time of the clock widget and whether or not to use Am/Pm standard.

Choosing Am/Pm also results in a slight code generation change. Instead of using the following function in Analog Clock to initialize the time:
`initializeTime24Hour(uint8_t hour, uint8_t minute, uint8_t second)`

The following function is used when using 12-hour notation:
`initializeTime12Hour(uint8_t hour, uint8_t minute, uint8_t second, bool am)`

To update the time displayed by the clock, one of the following functions can be used.
`setTime24Hour(uint8_t hour, uint8_t minute, uint8_t second)`
`setTime12Hour(uint8_t hour, uint8_t minute, uint8_t second, bool am)`

Clock Hands

In the Clock Hands property group, the user can define which hands to use and their z-order. The hand defined first will be rendered above the others.

Hour, Minute and Second Hands

Each hand needs an image and a rotation position. The rotation position determines the point at which the defined hand image should rotate around itself.

Clock hand properties

The hour and minute hands have the ability to use *Sweeping Movement*. When this option is enabled the hand will no longer do an instantaneous jump from one position to another.

Sweeping movement disabled

Sweeping movement enabled

Animation

The animation section allows the user to define more advanced hand movement. However, if the hour and minute hand have *Sweeping Movement* enabled, they will not animate.

In the following example the animation duration is set to '30', easing is set to 'Bounce' with 'Out' as its easing option:

Example of a clock animation

Interactions

The actions and triggers supported by an AnalogClock are described in the following sections.

Actions

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

An AnalogClock does not emit any triggers.

Performance

An AnalogClock consists of an Image and three TextureMappers, which are MCU resource intensive components. Therefore, an AnalogClock is considered a demanding widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up an AnalogClock.

mainViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"

mainViewBase::mainViewBase()
{
    analogClock.setXY(124, 15);
    analogClock.setBackground(BITMAP_BLUE_CLOCKS_BACKGROUNDS_CLOCK_STANDARD_BACKGROUND_ID,
    analogClock.setupMinuteHand(BITMAP_BLUE_CLOCKS_HANDS_CLOCK_STANDARD_MINUTE_HAND_ID, 7,
    analogClock.setMinuteHandSecondCorrection(false);
    analogClock.setupHourHand(BITMAP_BLUE_CLOCKS_HANDS_CLOCK_STANDARD_HOUR_HAND_ID, 7, 52)
    analogClock.setHourHandMinuteCorrection(false);
    analogClock.setupSecondHand(BITMAP_BLUE_CLOCKS_HANDS_CLOCK_STANDARD_SECOND_HAND_ID, 3,
    analogClock.initializeTime24Hour(10, 10, 0);

    add(analogClock);
}

void mainViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the AnalogClock class in user code. Remember to force a redraw by calling `analogClock.invalidate()` if you change the appearance of the widget.

User Code

The following example shows how to set up clock movement:

mainView.hpp

```
#ifndef MAINVIEW_HPP
#define MAINVIEW_HPP

#include <gui_generated/main_screen/mainViewBase.hpp>
#include <gui/main_screen/mainPresenter.hpp>

class mainView : public mainViewBase
```

```
{
public:
    mainView();
    virtual ~mainView() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();

protected:
    int tickCounter;
    int analogHours;
    int analogMinutes;
    int analogSeconds;
};

#endif // MAINVIEW_HPP
```

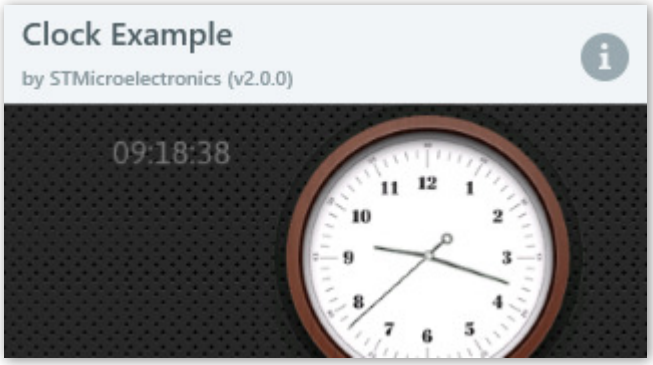
mainView.cpp

```
#include <gui/main_screen/mainView.hpp>

mainView::mainView()
{
}

void mainView::setupScreen()
{
    mainViewBase::setupScreen();
    analogHours = analogClock.getCurrentHour();
    analogMinutes = analogClock.getCurrentMinute();
    analogSeconds = analogClock.getCurrentSecond();
}

void mainView::tearDownScreen()
{
    mainViewBase::tearDownScreen();
}

void mainView::handleTickEvent()
{
    tickCounter++;

    if (tickCounter % 60 == 0)
    {
        if (++analogSeconds >= 60)
        {
            analogSeconds = 0;
            if (++analogMinutes >= 60)
            {
                analogMinutes = 0;
                if (++analogHours >= 24)
```

```
                {
                    analogHours = 0;
                }
            }
        }

        // Update the clocks
        analogClock.setTime24Hour(analogHours, analogMinutes, analogSeconds);
    }
}
```

TouchGFX Designer Examples

To further explore the AnalogClock, try creating a new application within TouchGFX Designer with one of the following UI templates:



Clock Example UI template in TouchGFX Designer

API Reference

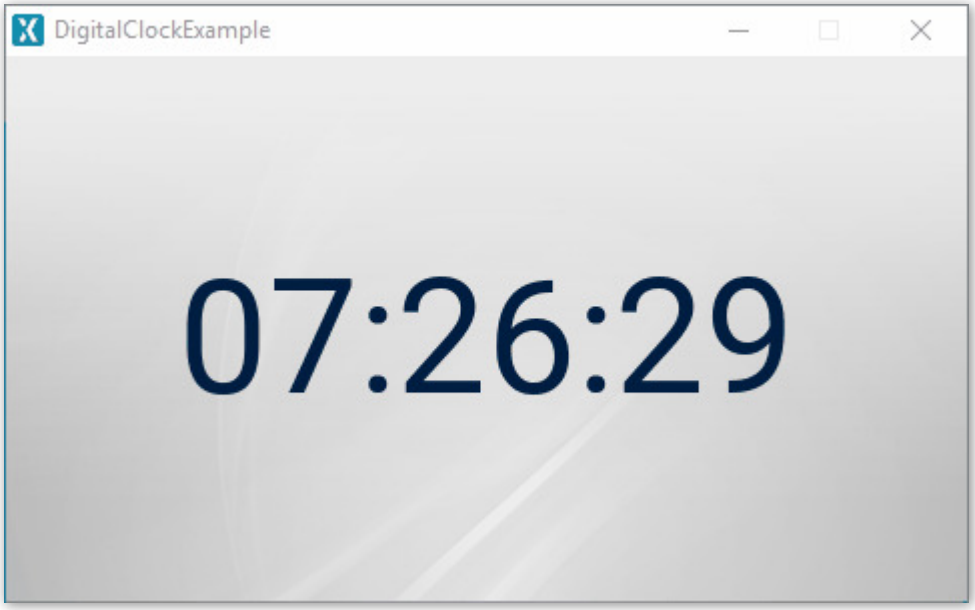
FURTHER READING

- **API reference for the AnalogClock class**

Version: 4.16

DigitalClock

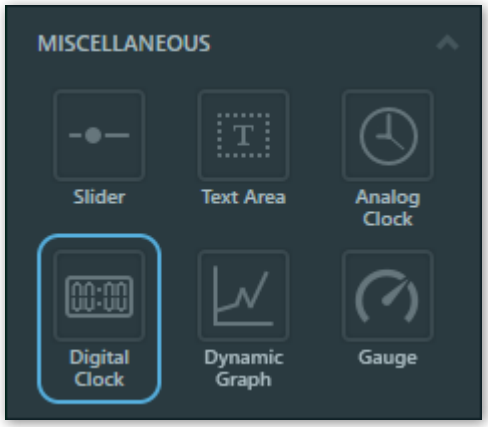
A DigitalClock in TouchGFX is a widget that allows an application to display time with digital text, as opposed to the AnalogClock which displays time using analog clock hands.



DigitalClock running in the simulator (sped up footage)

Widget Group

The DigitalClock can be found in the Miscellaneous widget group in TouchGFX Designer.



DigitalClock in TouchGFX Designer

Properties

Property Group	Property Descriptions

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Time	*Use Am/Pm* specifies if time should be in 12h or 24h format. *Display Leading Zero for Hours* specifies if leading zero for hours should be enabled. *Display Seconds* specifies if showing seconds is enabled. *Initial Time* specifies the initial time the clock shows.
Text	*Single Use* or *Resource* specifies which text resource to use for the DigitalClock. *Text must have a wildcard to function properly.*
Appearance	*Text Color* specifies the color of the text in the DigitalClock. *Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Time

The Time property group is used to adjust how time is displayed in the application by changing different properties. You can choose to use 24-hour time or 12-hour AM/PM by toggling *Use Am/Pm*. Toggling *Display Leading Zero for Hours* specifies how hours below 10 are displayed (e.g. 09:10:00 or 9:10:00) and *Display Seconds* toggles the display of seconds on/off (e.g. 9:10:00 or 9:10).

Choosing Am/Pm also results in a slight code generation change. Instead of using the following function in Analog Clock to initialize the time:
`initializeTime24Hour(uint8_t hour, uint8_t minute, uint8_t second)`

The following function is used when using 12-hour notation:
`initializeTime12Hour(uint8_t hour, uint8_t minute, uint8_t second, bool am)`

To update the time which the clock displays, one of the following functions can be used.
`setTime24Hour(uint8_t hour, uint8_t minute, uint8_t second)`
`setTime12Hour(uint8_t hour, uint8_t minute, uint8_t second, bool am)`

Interactions

The actions and triggers supported by the DigitalClock are described in the following sections.

Actions

Standard widget action	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

A DigitalClock does not emit any triggers.

Performance

A DigitalClock consists of a TextArea, which does not impact performance in any meaningful way. Therefore, a DigitalClock is considered a fast performing widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how TouchGFX Designer sets up the DigitalClock.

mainViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"
#include <texts/TextKeysAndLanguages.hpp>

mainViewBase::mainViewBase()
{
    digitalClock.setPosition(75, 88, 331, 97);
    digitalClock.setColor(touchgfx::Color::getColorFrom24BitRGB(0, 30, 65));
    digitalClock.setTypedText(touchgfx::TypedText(T_SINGLEUSEID2));
    digitalClock.displayLeadingZeroForHourIndicator(true);
    digitalClock.setDisplayMode(touchgfx::DigitalClock::DISPLAY_24_HOUR);
    digitalClock.setTime24Hour(7, 7, 0);

    add(digitalClock);
}

void mainViewBase::setupScreen()
{

}
```

TIP

You can use these functions and the others available in the DigitalClock class in user code. Remember to force a redraw by calling `digitalClock.invalidate()` if you change the appearance of the widget.

User Code

The following example shows how to set up the clock to start.

mainView.hpp

```
#ifndef MAINVIEW_HPP
#define MAINVIEW_HPP

#include <gui_generated/main_screen/mainViewBase.hpp>
#include <gui/main_screen/mainPresenter.hpp>

class mainView : public mainViewBase
{
```

```
public:
    mainView();
    virtual ~mainView() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
    virtual void handleTickEvent();

protected:
    int tickCounter;
    int digitalHours;
    int digitalMinutes;
    int digitalSeconds;
};

#endif // MAINVIEW_HPP
```

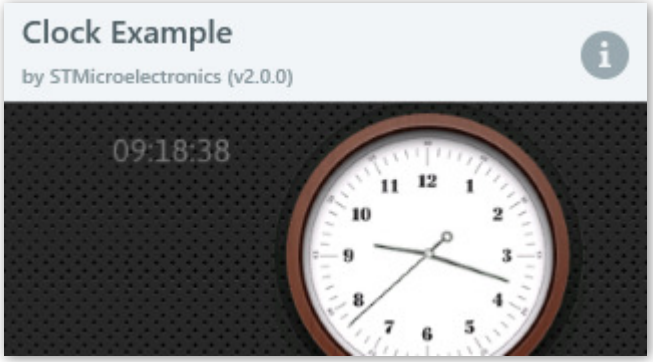
mainView.cpp

```
#include <gui/main_screen/mainView.hpp>

mainView::mainView()
{
}

void mainView::setupScreen()
{
    mainViewBase::setupScreen();
    digitalHours = digitalClock.getCurrentHour();
    digitalMinutes = digitalClock.getCurrentMinute();
    digitalSeconds = digitalClock.getCurrentSecond();
}

void mainView::tearDownScreen()
{
    mainViewBase::tearDownScreen();
}

void mainView::handleTickEvent()
{
    tickCounter++;

    if (tickCounter % 60 == 0)
    {
        if (++digitalSeconds >= 60)
        {
            digitalSeconds = 0;
            if (++digitalMinutes >= 60)
            {
                digitalMinutes = 0;
                if (++digitalHours >= 24)
                {
```

```
                    digitalHours = 0;
                }
            }
        }

        // Update the clock
        digitalClock.setTime24Hour(digitalHours, digitalMinutes, digitalSeconds);
    }
}
```

TouchGFX Designer Examples

To further explore the DigitalClock, try creating a new application within TouchGFX Designer with one of the following UI templates:



Clock Example UI template in TouchGFX Designer

API Reference

FURTHER READING

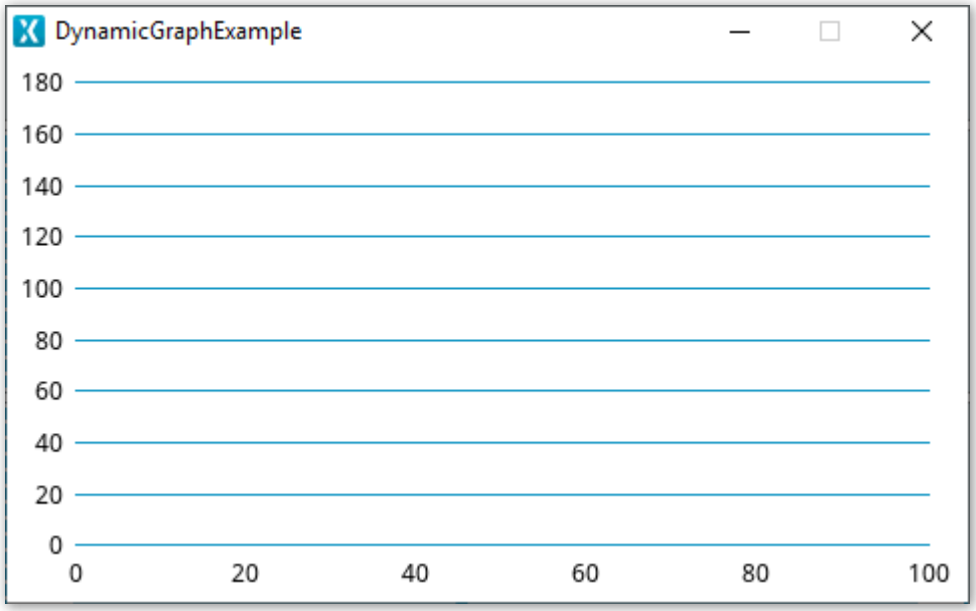
- **API reference for the DigitalClock class**

Version: 4.16

DynamicGraph

A DynamicGraph in TouchGFX is a widget that allows an application to display data points on a monotonous x-axis. The DynamicGraph supports three types of dynamic behavior, that defines what happens when the graph runs out of space on the x-axis. The chosen dynamic behavior also greatly impacts the performance of the DynamicGraph, as the behavior chosen impacts the area needed to be redrawn when inserting data points.

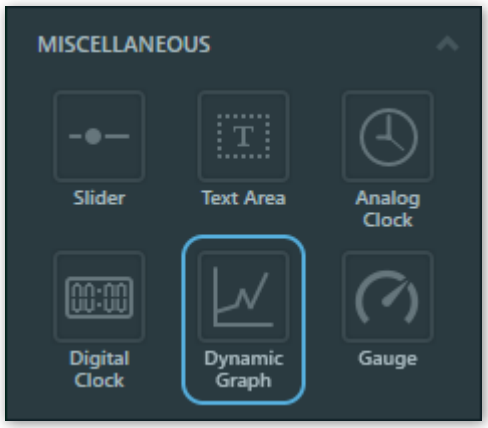
The DynamicGraph, can have its visual appearance defined via, Graph Elements, Grid Lines and Labels



DynamicGraph running in the simulator

Widget Group

The DynamicGraph can be found in the Miscellaneous widget group in TouchGFX Designer.



DynamicGraph in TouchGFX Designer

Properties

The properties for a DynamicGraph in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Graph Area Margin	Margin specifies how much space for graph labels there should be. *Top* specifies the amount of space above the graph area. *Bottom* specifies the amount of space below the graph area. *Left* specifies the amount of space left of the graph area. *Right* specifies the amount of space right of the graph area.
Graph Area Padding	Padding specifies how much room around the graph elements inside the graph area there should be. *Top* specifies the amount of room at the top of the graph area. *Bottom* specifies the amount of room at the bottom of the graph area. *Left* specifies the amount of room left of the graph area. *Top* specifies the amount of room right of the graph area.

Property Group	Property Descriptions
Data Points	*Dynamic Behavior* specifies the behavior of the graph when adding data points, available options are: Wrap And Clear, Scroll and Wrap and Overwrite. *Number of Data Points* specifies the number of values the graph is capable of showing. *Value Range* specifies the minimum and maximum y-axis values the graph is capable of showing. *Level of Precision* specifies how many decimal places the graph is capable of displaying. *Visible Range (index values)* specifies the range of values shown on the x-axis. *Custom Value Mapping* specifies the mapping of the index values of the x-axis to custom values. *Generate Random Values* specifies whether or not random values should be initialized in code generation. *(Random values will always be shown in the Canvas of the TouchGFX Designer)*
Elements	*Area*, *Boxes*, *Diamonds*, *Dots*, *Histogram* and *Line* specify which elements make up the widgets visual appearance. *More than one of each type can be added*.
Vertical Grid Lines	*Major Division* specifies whether or not to enable vertical major division grid lines. *Minor Division* specifies whether or not to enable vertical minor division grid lines. *Can only be enabled if Major Division has been enabled*.
Horizontal Grid Lines	*Major Division* specifies whether or not to enable horizontal major division grid lines. *Minor Division* specifies whether or not to enable horizontal minor division grid lines. *Can only be enabled if Major Division has been enabled*.
X-Axis Labels	*Major Division* specifies whether or not to enable major division labels on the x-axis. *Minor Division* specifies whether or not to enable minor division labels on the x-axis. *Can only be enabled if Major Division has been enabled*.
Y-Axis Labels	*Major Division* specifies whether or not to enable major division labels on the y-axis. *Minor Division* specifies whether or not to enable minor division labels on the y-axis. *Can only be enabled if Major Division has been enabled*.

Property Group	Property Descriptions
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid*.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

Precision

Internally the DynamicGraph stores all data points in a 32 bit integer, therefore to add and display data points with a certain number of digits denoting the degree of accuracy, a level of precision can be defined.

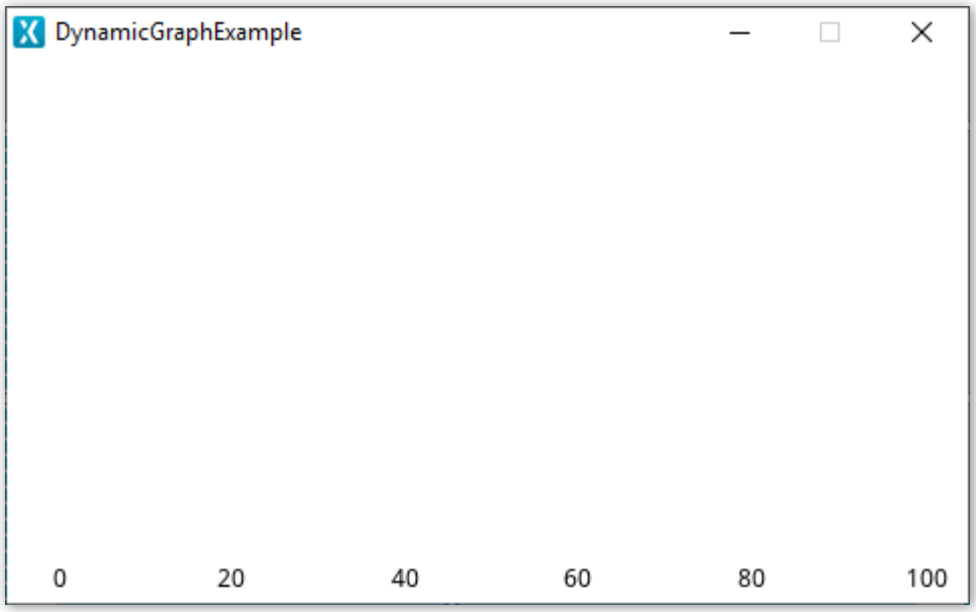
If for example the level of precision is set to 0.1, each data point added to the DynamicGraph, will be multiplied by 10 internally, thereby simulating one digit of precision. However it must be noted that as the level of precision increases, the highest/lowest possible value lowers/increases by the factor of the precision specified.

Level of Precision	Lowest Possible Value	Highest Possible Value
1	-1 Billion	1 Billion
0.1	-100 Million	100 Million
0.01	-10 Million	10 Million
0.001	-1 Million	1 Million
0.0001	-100 Thousand	100 Thousand

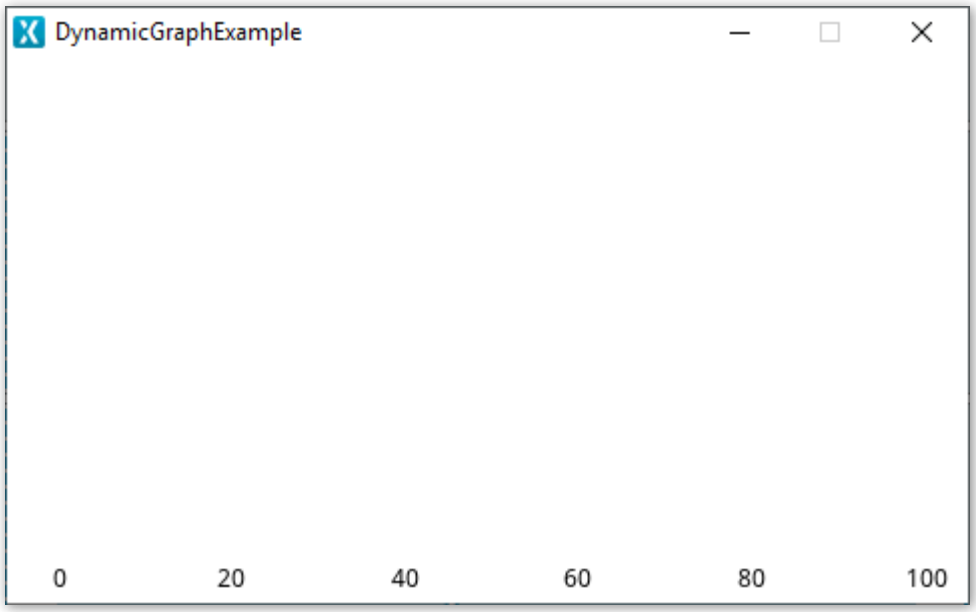
The lowest/highest numbers given above are rough estimations

Dynamic Behavior

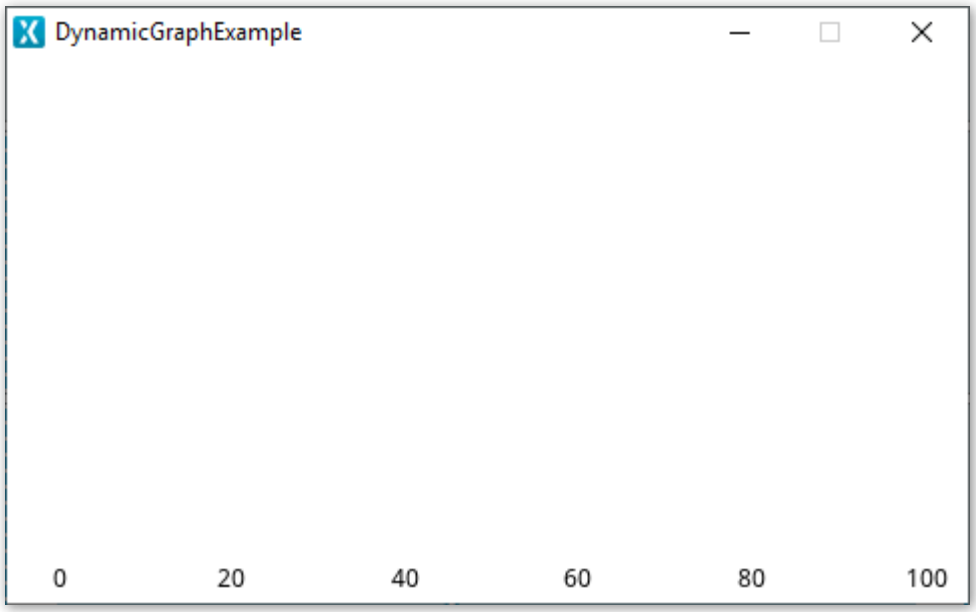
The DynamicGraph supports three types of dynamic behavior, Wrap And Clear, Scroll and Wrap and Overwrite. The selected dynamic behavior specifies what will happen once the graph runs out of space on the x-axis, as can be seen in the three demontrations below.



DynamicGraph Wrap and Clear example



DynamicGraph Scroll example



DynamicGraph Wrap and Overwrite example

Graph Area, Margin and Padding

The DynamicGraph renders all graph elements and grid lines in a Graph Area, incapsulated by padding and margin. If both padding or margin are defined as zero, the Graph Area will follow the the size given to the DynamicGraph.

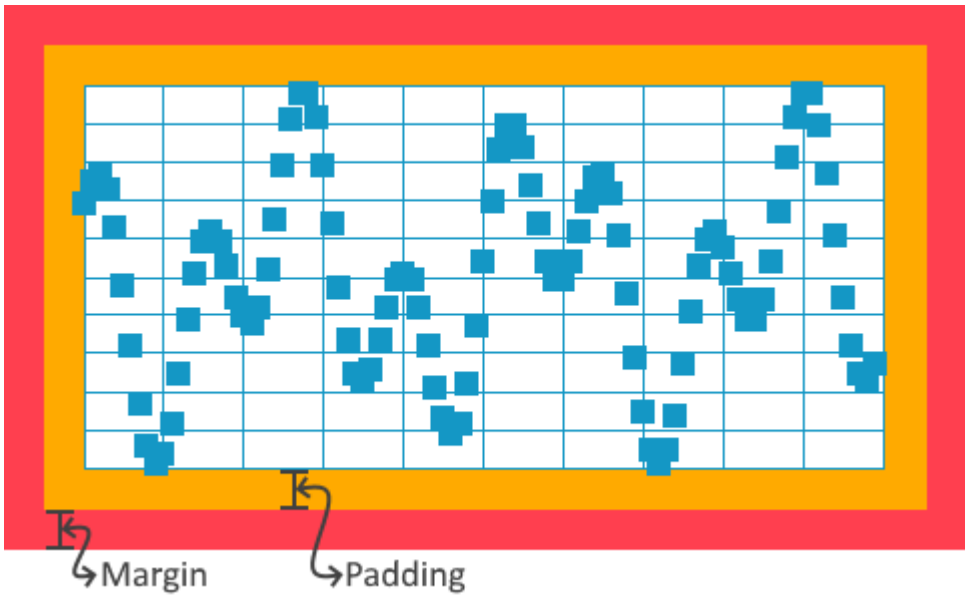
To make space for labels along the x- and y-axis, a margin can be defined. The Margin settings move the graph area that contains the graph elements *(Area, Boxes, Diamonds, etc.)* inside the widget, reserving space for labels along the x- and y-axis.

In the figure below the red area represents a 20 px margin added to right, top, left and bottom.

Depending on their sizes, some elements will not be shown fully if they are positioned close to the edges of the Graph Area, therefore a padding can be defined. The Padding settings add some padding inside the graph area that contains the graph elements *(Area, Boxes, Diamonds, etc.)*, this will allow Grid lines, Boxes, Dots, Diamonds, Histogram and Line elements drawn at the edges of the Graph Area to be drawn fully.

Padding can also be used to create extra space between the labels along the axes and the Graph Area.

In the figure below the orange area represents a 20 px padding added to right, top, left and bottom. It also shows how the Boxes element is allowed to draw into the padded area.



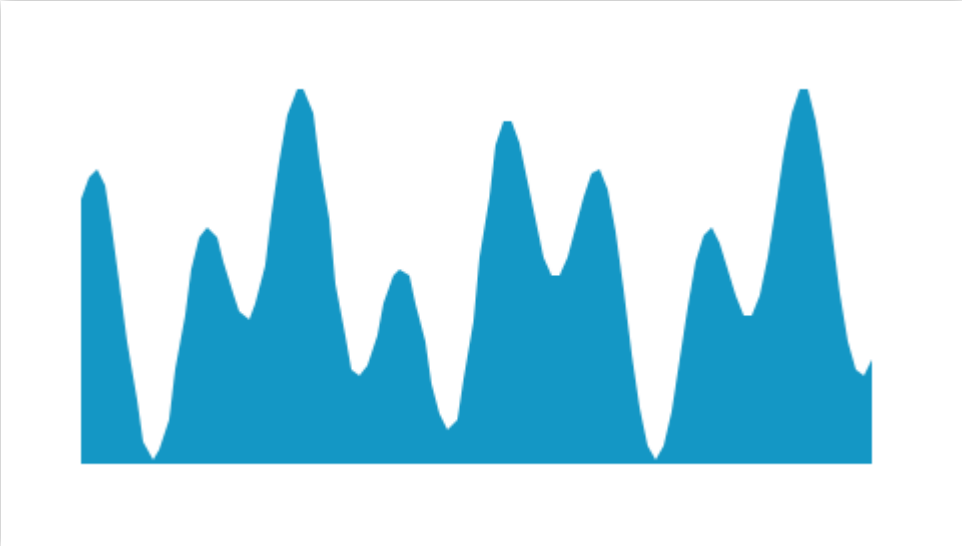
DynamicGraph margin and padding example

Elements

The DynamicGraph has six available element types to display data: Area, Boxes, Diamonds, Dots, Histogram and Line

Area

The Area element will fill the area below the line connecting the data points in the graph.



DynamicGraph Area example

Property	Property Description
Image	Specifies which image to use as fill for the area.
Color	Specifies which color to use as fill for the area.
Baseline	Specifies the base of the area drawn. Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values.
Alpha	Specifies the transparency of the area. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

API Reference

FURTHER READING

- **API reference for the GraphElementArea class**

Boxes

The Boxes element will draw a square box for every data point in graph.

DynamicGraph Boxes example

Property	Property Description
Color	Specifies which color to use as fill for the boxes.
Box Size	Specifies the size of the boxes.
Alpha	Specifies the transparency of the boxes. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

API Reference

FURTHER READING

- **API reference for the GraphElementBoxes class**

Diamonds

The Diamonds element will draw a diamond (a square with the corners up/down/left/right) for every data point in graph.

DynamicGraph Diamonds example

Property	Property Description
Image	Specifies which image to use as fill for the diamonds.
Color	Specifies which color to use as fill for the diamonds.
Diamond Size	Specifies the size of the diamonds
Alpha	Specifies the transparency of the diamonds. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

API Reference

FURTHER READING

- **API reference for the GraphElementDiamonds class**

Dots

The Dots element will draw a circular dot for every data point in graph.

DynamicGraph Dots example

Property	Property Description
Image	Specifies which image to use as fill for the dots.
Color	Specifies which color to use as fill for the dots.
Dot Size	Specifies the size of the dots
Alpha	Specifies the transparency of the dots. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

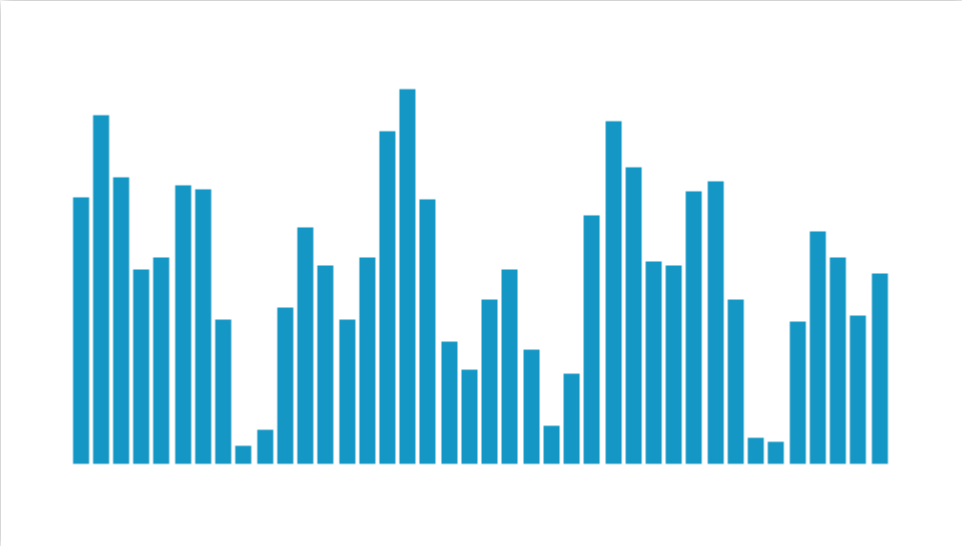
API Reference

FURTHER READING

- **API reference for the GraphElementDots class**

Histogram

The Histogram element is used to draw blocks from the x-axis to the data point in the graph. If more graphs are placed on top of each other, the histogram can be moved slightly to the left/right.



DynamicGraph Histogram example

Property	Property Description
Image	Specifies which image to use as fill for the histogram.
Color	Specifies which color to use as fill for the histogram.
Bar Width	Specifies width of the histogram bars.
Bar Offset	Specifies bar offset along the horizontal axis.
Baseline	Specifies the base of the histogram drawn. Normally, the base is 0 which means that the histogram is drawn below positive y values and above negative y values.
Alpha	Specifies the transparency of the histogram. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

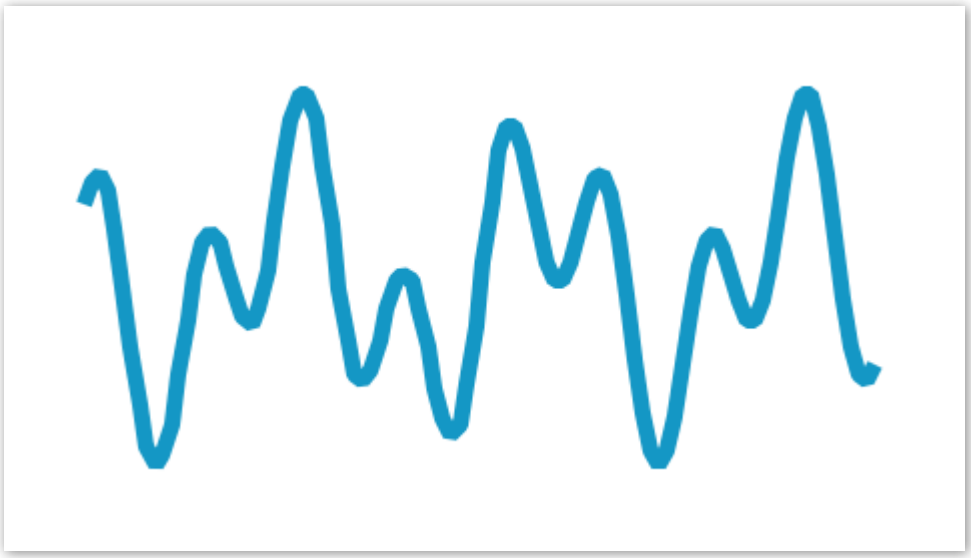
API Reference

FURTHER READING

- **API reference for the GraphElementHistogram class**

Line

The Line element will draw a line with a given thickness through the data points in the graph.



DynamicGraph Line example

Property	Property Description
Image	Specifies which image to use as fill for the line.
Color	Specifies which color to use as fill for the line.
Line Width	Specifies the width of the line
Alpha	Specifies the transparency of the line. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

API Reference

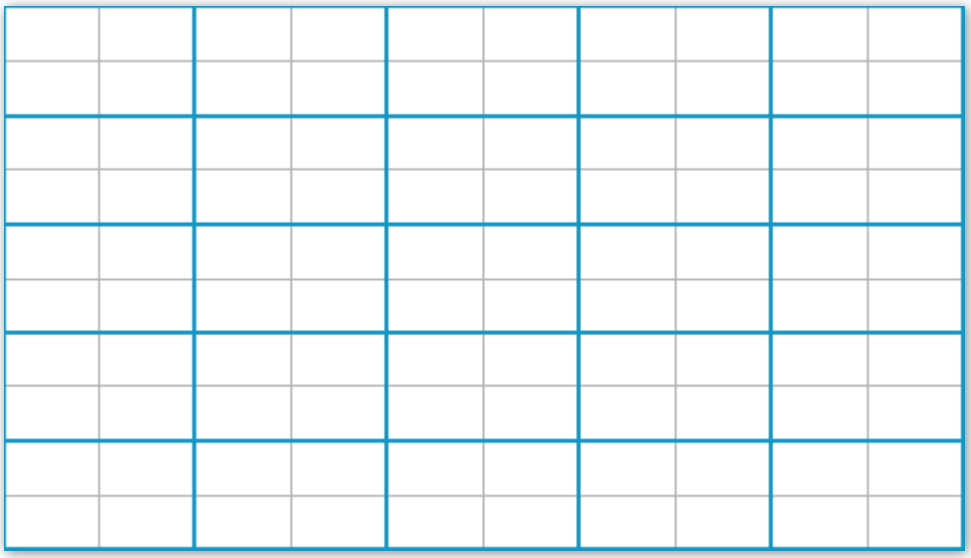
FURTHER READING

- **API reference for the GraphElementLine class**

Grid Lines

The DynamicGraph can have horizontal and vertical grid lines in both major and minor divisions.

Minor divisions are overruled by major divisions, such that minor divisions will not be drawn on locations where major divisions are present.



DynamicGraph Grid Lines example

Property	Property Description
Color	Specifies which color to use as fill for the grid line.
Interval	Specifies the interval at which grid lines should be drawn
Line Width	Specifies the width of the grid line
Alpha	Specifies the transparency of the line. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

API Reference

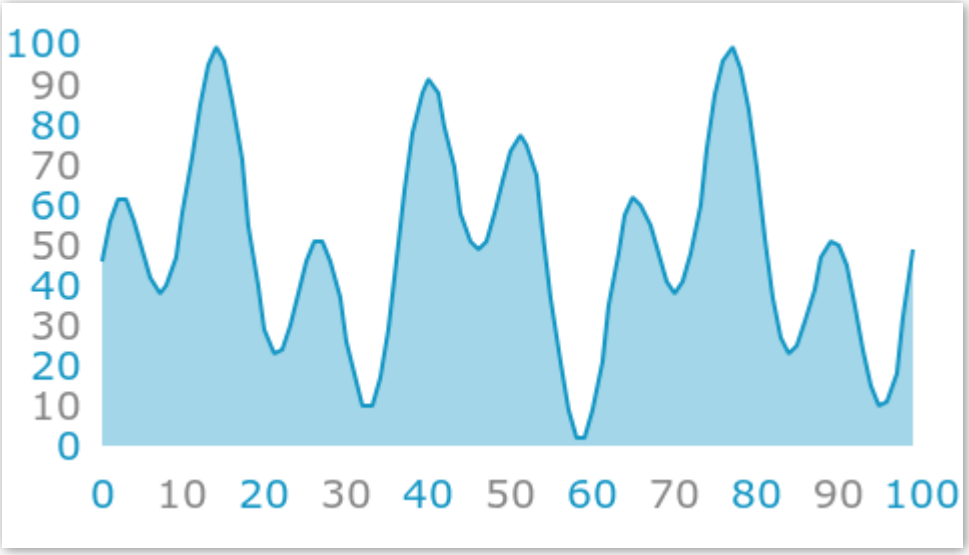
FURTHER READING

- **API reference for the GraphElementGridBase class**
- **API reference for the GraphElementGridX class**
- **API reference for the GraphElementGridY class**

Labels

. The DynamicGraph can have x-axis and y-axis labels displaying the value in both major and minor divisions.

Minor divisions are overruled by major divisions, such that minor divisions will not be drawn on locations where major divisions are present.



DynamicGraph Labels example

Property	Property Description
Position	Specifies the location of the labels. Possible locations for x-axis labels are "Top" and "Bottom" Possible locations for y-axis labels are "Left" and "Right"
Text	*Single Use* and *Ressource* specify the type of text: unique or from a known ressource. When Single Use is selected: *Text* specifies the content of the text to be displayed. *Typography* specifies the format of the labels. *Alignment* specifies the horizontal alignment of the labels. When Ressource is selected: *Ressource ID* specifies the Ressource to retrieve the labels from.
Text Rotation	Specifies the rotation of the labels, possible values are "0", "90", "180" and "270".
Text Color	Specifies which color to use as fill for the labes.
Interval	Specifies the interval at which labels should be drawn
Number of Decimals	Specifies the number of decimals the labels along the axis should show.
Decimal Separator	Specifies whether to use ',' or '.' as the decimal separator.
Alpha	Specifies the transparency of the line. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid.*

API Reference

FURTHER READING

- **API reference for the GraphLabelsBase class**
- **API reference for the GraphLabelX class**
- **API reference for the GraphLabelsY class**

Interactions

The actions and triggers supported by a DynamicGraph in TouchGFX Designer.

Actions

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Graph Clicked	A DynamicGraph has been clicked.
Graph Dragged	A DynamicGraph has been dragged across.

Performance

The DynamicGraph performance will vary, depending upon the setup of the widget.

The chosen dynamic behavior impacts the performance.

- "Wrap And Clear" is a fast performing behavior because it only draws the newest data point added.

- "Wrap And Overwrite" is a fast performing behavior because it only draws the newest data point added.
- "Scroll" is a demanding behavior, since everytime a data point is added, all the previous data points visible also need to be redrawn.

Certain graph elements will be faster to draw.

Boxes and Histogram are the fastest performing graph elements, because they do not have to read any pixel data or do any complicated calculations.

Area, Diamonds, Dots and Line, are CanvasWidgets and are heavily dependent on the MCU for rendering.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how the Designer sets up a DynamicGraph.

Screen1ViewBase.cpp

```
dynamicGraph.setScale(1);
dynamicGraph.setPosition(0, 0, 320, 240);
dynamicGraph.setGraphLabelPadding(0, 0, 0, 0);
dynamicGraph.setGraphPadding(0, 0, 0, 0);
dynamicGraph.setGraphRangeY(0, 100);

dynamicGraphLine1.setScale(1);
dynamicGraphLine1Painter.setColor(touchgfx::Color::getColorFrom24BitRGB(20, 151, 197));
dynamicGraphLine1.setPainter(dynamicGraphLine1Painter);
dynamicGraphLine1.setLineWidth(2);
dynamicGraph.addGraphElement(dynamicGraphLine1);
```

TIP

You can use these functions and the others available in the DynamicGraph class in user code. Remember to force a redraw by calling `dynamicGraph.invalidate()` if you change the appearance of the widget.

User Code

To add data points to the DynamicGraph, the method `addDataPoint()` is used. The following code example shows how to add data points to a DynamicGraph, by overwriting the `handleTickEvent()` method.

Screen1View.hpp

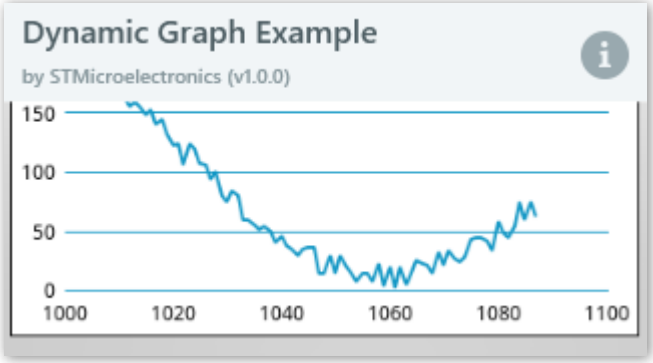
```
class Screen1View
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
protected:
    int tickCounter;
    void handleTickEvent();
};
```

Screen1View.cpp

```
#include <gui/screen1_screen/Screen1View.hpp>

Screen1View::Screen1View()
{
    tickCounter = 0;
}

void Screen1View::handleTickEvent()
{
    tickCounter++;

    // Insert each second tick
    if (tickCounter % 2 == 0)
    {
      // Insert data point
      dynamicGraph.addDataPoint(/* Your data point here, either float or integer */);
    }
}
```

TouchGFX Designer Examples

To further explore the DynamicGraph, try creating a new application within TouchGFX Designer with one of the following UI templates:



DynamicGraph Example UI template in TouchGFX Designer

API Reference

FURTHER READING

- **API reference for the AbstractDataGraph class**
- **API reference for the GraphScroll class**
- **API reference for the GraphWrapAndClear class**
- **API reference for the GraphWrapAndOverwrite class**

Version: 4.16

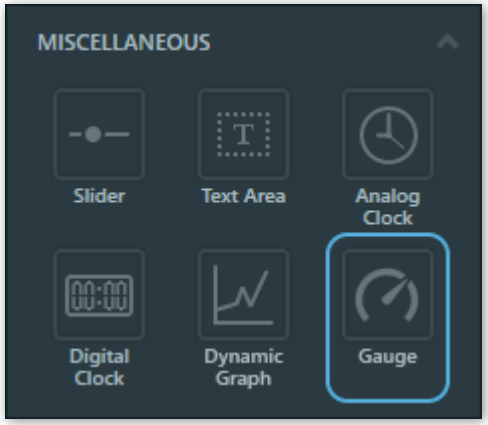
Gauge

A Gauge is a widget capable of drawing a needle and arc indicating a given value within a specified range.

Gauge running in the simulator

Widget Group

The gauge can be found in the Miscellaneous widget group in TouchGFX Designer.



Gauge in TouchGFX Designer

Properties

The properties for a Gauge in TouchGFX Designer.

Property Group	Property Descriptions
Name	*Name* of the widget. *Name is the unique identifier used in TouchGFX Designer and code.*
Location	*X* and *Y* specify the top left corner of the widget relative to its parent. *W* and *H* specify the width and height of the widget. *Lock* specifies if the widget should be locked in its current X, Y, W and H. *Locking the widget also disables interacting with the widget through the screen.* *Visible* specifies the visibility of the widget. *Making the widget invisible also disables interacting with the widget through the screen.*
Style	*Style* specifies a predefined setup of the widget, that sets select properties to predefined values. *These styles contain images that are free to use.*
Background	*Background Image* specifies the image to be used as background. *Background offset* specifies the x- and y-axis offset of the selected background image.
Gauge	*Rotation Center* specifies the x- and y-axis point in the widget at which the needle and arc should rotate. *Start and End Angle* specify the start and end angle of the needle and arc. *Value Range* specifies the minimum and maximum integer values of the indicator. *Initial Value* specifies the initial value of the gauge.
Animation	*Animate Movement* specifies if animation of the needle and arc are enabled. *Easing* and *Easing Option* specify the easing equation used.
Needle	*Needle Image* specifies the image to be used as background. *Needle Rotation Center* specifies the position in the needle image around which it will rotate. *Moving Rendering Algorithm* specifies the algorithm used to draw the needle image while moving to new value. *Steady Rendering Algorithm* specifies the algorithm used to draw the needle image while stationary.

Property Group	Property Descriptions
Arc	*Use Arc* specifies whether or not to use an arc. *Image/Color* specifies either a color or an image to use as fill for the arc. *Set Arc position* specifies whether or not to override the default arc size and position. *By default arc is placed in 0, 0 and has the same size as the gauge itself. Overriding the default setting is useful when you want to use an image as painter for the arc, but the arc, and thus the image, is smaller than the size of the gauge. So instead of having a large image of the size of the gauge with a lot of transparent lines, the the image can be cut to the wanted size and place the arc at a specific position. The arc will still follow the same rotation center.* *Arc Position X and Y* specifies the x- and y-axis position of the arc. *Arc Width and Height* specifies the size of the arc. *Radius* specifies the radius of the arc. *Line Width* specifies the line width of the arc. *If the value is 0, the arc is as large as the radius.* *Cap Style* specifies line ending style of the arc. *If the line width is set to 0, the capstyle selected will have no effect.* *Arc on top of Needle* specifies whether or not the arc is on top of the needle.
Appearance	*Alpha* specifies the transparency of the widget. *The alpha value ranges between 0 and 255 for the widget. 0 is fully transparent and 255 is solid*.
Mixins	*Draggable* specifies if the widget is draggable at runtime. *ClickListener* specifies if the widget emits a callback when clicked. *FadeAnimator* specifies if the widget can animate changes to its *Alpha* value. *MoveAnimator* specifies if the widget can animate changes to *X* and *Y* values.

By default arc is placed in 0, 0 and has the same size as the gauge itself. Overriding the default setting is useful when you want to use an image as painter for the arc, but the arc, and thus the image, is smaller than the size of the gauge. So instead of having a large image of the size of the gauge with a lot of transparent lines, the the image can be cut to the wanted size and place the arc at a specific position. The arc will still follow the same rotation center.

Interactions

The actions and triggers supported by a Gauge in TouchGFX Designer.

Actions

Widget specific actions	Description
Set value	Set the value of the Gauge.
Update value	Update the value of the Gauge with a duration.

Standard widget actions	Description
Move widget	Move a widget to a new position over time.
Fade widget	Modify alpha value of widget over time.
Hide widget	Hides a widget (sets visibility to false).
Show widget	Make a hidden widget visible (sets visibility to true).

Triggers

Trigger	Description
Gauge value set	Will be triggered for both instant updates and intermediate steps during an update animation. Will only trigger when the new value differs from the current one.
Gauge value updated	Will be triggered when an update animation is completed. If duration of the update is 0 the update will happen instantly but will still trigger this event.

Performance

A Gauge consists of an Image, Circle and a TextureMapper.

The Circle and TextureMapper components are MCU resource intensive components. Therefore, the Gauge is considered a demanding widget on most platforms.

For more details on drawing performance, read the General UI Component Performance section.

Examples

Generated Code

In the generated code for the View base class we can see how the Designer sets up a Gauge.

MainViewBase.cpp

```
#include <gui_generated/main_screen/mainViewBase.hpp>
#include "BitmapDatabase.hpp"

mainViewBase::mainViewBase()
{
    gauge.setBackground(touchgfx::Bitmap(BITMAP_BLUE_GAUGES_ORIGINAL_GAUGE_BACKGROUND_STYL
    gauge.setPosition(115, 11, 251, 251);
    gauge.setCenter(125, 125);
    gauge.setStartEndAngle(-85, 85);
    gauge.setRange(0, 100);
    gauge.setValue(0);
    gauge.setEasingEquation(touchgfx::EasingEquations::elasticEaseOut);
    gauge.setNeedle(BITMAP_BLUE_NEEDLES_ORIGINAL_GAUGE_NEEDLE_STYLE_01_ID, 11, 55);
    gauge.setMovingNeedleRenderingAlgorithm(touchgfx::TextureMapper::BILINEAR_INTERPOLATIC
    gauge.setSteadyNeedleRenderingAlgorithm(touchgfx::TextureMapper::BILINEAR_INTERPOLATIC
    gauge.setArcVisible();
    gaugePainter.setBitmap(touchgfx::Bitmap(BITMAP_BLUE_GAUGES_ORIGINAL_GAUGE_FILL_STYLE_0
    gauge.getArc().setPainter(gaugePainter);
    gauge.getArc().setRadius(94);
    gauge.getArc().setLineWidth(14);
    gauge.getArc().setCapPrecision(180);
    gauge.setArcPosition(28, 30, 196, 88);

    add(gauge);
}
```

TIP

You can use these functions and the others available in the Gauge class in user code. Remember to force a redraw by calling `gauge.invalidate()` if you change the appearance of the widget.

User Code

To set the value of the Gauge, `setValue(int value)` and `updateValue(int value, uint16_t duration)` can be used.

`setValue(int value)` will immediately move the needle and arc to the new value with no animation.

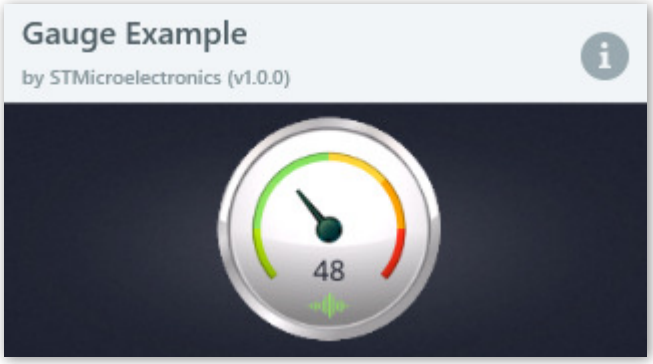
`updateValue(int value, uint16_t duration)` animates needle and arc to new value over the defined duration in ticks. If duration is equal to 0, the update will be done immediately. The animation used will be the one defined on the gauge using `setEasingEquation(EasingEquation easingEquation)`

MainView.cpp

```
#include <gui/main_screen/MainView.hpp>

MainView::MainView()
{
    tickCounter = 0;
}

void MainView::handleTickEvent()
{
    tickCounter++;

    // Change value every 120 tick.
    if (tickCounter % 120 == 0)
    {
      // Insert data point
      gauge.updateValue(/* new integer value */, 30); // animates needle and arc to new v
    }
}
```

TouchGFX Designer Examples

To further explore the Gauge, try creating a new application within TouchGFX Designer with one of the following UI templates:



Gauge Example UI template in TouchGFX Designer

API Reference

FURTHER READING

- **API reference for the gauge class**

Version: 4.16

General UI Component Performance

This section describes the performance of general TouchGFX rendering operations used by most UI components.

Image Drawing

Image drawing is one of the most essential drawing operations in TouchGFX, as almost all UI components to some extent rely on drawing one or more images. The ability of your system to draw images in a fast and effective way is therefore often very important. There are a lot of factors that influence the performance of image drawing. However, on almost all hardware setups, TouchGFX image drawing is considered a fast operation compared to other drawing operations.

Hardware support for data copy

TouchGFX stores the image data uncompressed in the selected image format (e.g. RGB565, L8, ARGB8888). The advantage of the uncompressed format is that it allows TouchGFX, in most cases, to use the image directly and transfer it unmodified to the framebuffer. If the MCU has a DMA, this can and should be used for the memory copy, as this speeds up the transfer and minimizes the MCU load.

One limitation to this approach is if the image format includes an alpha channel. Here a normal DMA transfer cannot be used since the MCU needs to perform pixel blending of the image data with framebuffer pixels. However, if you are using an STM32 with graphics acceleration like Chrom-ART / DMA2D, you can utilize the DMA for these types of images as well. Here the DMA is not only capable of copying data, but actually does a copy and blending operation in one go, thereby improving speed and lowering the MCU load considerably.

Image format

The image format has an impact on the image drawing performance as well, depending on the hardware support you have. A rule of thumb is, that the less data you have to transfer, the faster you can do it. So transfering an RGB565 image compared to a similar RGB888 will be faster in most cases, since an RGB565 image is two thirds the size of the equivalent RGB888 image.

Access to the Image data

The time needed to access the image data is very important, since this will be accessed each time the image is rendered. The image data can be stored in different hardware locations, with different access times, in a TouchGFX application.

Image data location	Description
External Flash	The advantage of external flash is its low cost and the size, which is often quite large, allowing you to have a lot of images in your application. However, access time varies a lot, but choosing QSPI or alternatives like it, will give you a high throughput, resulting in a significant boost to the image drawing performance.
External RAM	In some cases you might need to cache your images in External RAM. This is often the case when you are forced to use non-memory mapped flash (e.g. NAND, EMMC) which cannot be used directly for image rendering in TouchGFX. In this case the access to the external RAM is essential for the performance of image drawing in your application.
Internal Flash	In some cases you can store some or all of your images in internal flash, even though the storage space here is very limited. Access is very fast, so if you have some images that are essential for an animation and performance is an issue (e.g. if it is used by a TextureMapper) it might be worth trying to store it in internal flash if possible.
Internal RAM	In very rare cases, you will render images from the internal RAM. The storage space is very limited but the access time is very fast, so images stored here (using TouchGFX Image Caching) will be rendered very fast.

Access to the framebuffer

Rendering an image will always end up in an update to the framebuffer. If the image includes an alpha channel, you will not only write, but also read pixel data in the framebuffer to perform the actual blending. Therefore, the read/write access time to the RAM you are using for storing the framebuffer is key to have a good image drawing performance.

Image resolution

Since the data that needs to be transfered is proportional to the resolution of the image, the image resolution naturally has an effect on the image drawing operation.

Transparency

The opacity of an image has an effect on the rendering time for an image. An image with alpha will have a longer rendering time than an image without due to the fact that it will have to be blended

with the framebuffer. Therefore, a blending operation has to read from the framebuffer, whereas a solid image can simply overwrite data in the framebuffer. This is the case even if you have hardware acceleration. The ratio between rendering solid and semi-transparent images may, however, vary from one setup to another.

MCU Drawing

Some widgets rely on direct framebuffer manipulation. This approach performs one or more calculations for each pixel in the invalidated area, then updates the pixel in the framebuffer. This is a rather slow operation, especially if the calculation for each pixel is complex.

The MCU processing power is essential if your MCU drawing is performing a lot of calcuations. Access to the framebuffer (access to either internal or external RAM) will also have an impact since writing (and possibly reading) the framebuffer data is done per pixel in the invalidated area.

Canvas Widgets

Canvas widgets are a special type of TouchGFX widget used for drawing anti-aliased geometric shapes. They are typically quite complex and thus potentially fairly slow to render.

The rendering time is linear to the size of the invalidated part of the geometric shape.

Canvas widgets requires a memory block to store intermediate calcuation results. The size and performance impact of this is described in the canvas widgets section.

TIP

Most of the standard TouchGFX canvas widgets, like Circle, have update methods that will only invalidate the changed part of the widget. So if you are updating a Circle for example, use `circle::updateArc(...)`, which will not invalidate the entire circle but only the changed part. Be sure to use these kind of operations for optimal performance.

Texts

Text rendering depends on image drawing, as all the used characters are transformed into images as described in the text section. The images are in A4 format which is basically a 4 bit alpha value for each of the pixels in the image. If you apply a color to this pattern, you will have an anti-aliased image of a character.

Since text rendering is a set of image drawing operations, one for each character, the performance characteristics for image drawing applies to text rendering as well, including performance improvements using hardware acceleration like Chrom-ART / DMA2D.

Version: 4.16

Achieving Better Performance with CacheableContainer

In this section you will see how to achieve better performance in some animation scenarios by using RAM to save some reusable drawings.

When moving widgets in your application (like Image or TextArea), either through dragging or animation, TouchGFX needs to redraw these widgets in their new positions in every frame, while also in most cases redraw part of the background that was previously covered by these widgets.

When these widgets are computationally complex such as the TextureMapper widget, Shapes, and also large transparent Images it is hard for the MCU to render effeciently, as these are rendered without hardware acceleration. This results in a screen redraw that takes many milliseconds and impacts the performance of the application.

In this we will now see how to use the CacheableContainer to speed up animations that involve computationally complex elements by avoiding costly redrawing. While measurements in this article were performed using an STM32F429Discovery board, the CacheableContainer technique applies generally to other hardware platforms. Some available RAM is required for creation of a bitmap cache.

FURTHER READING

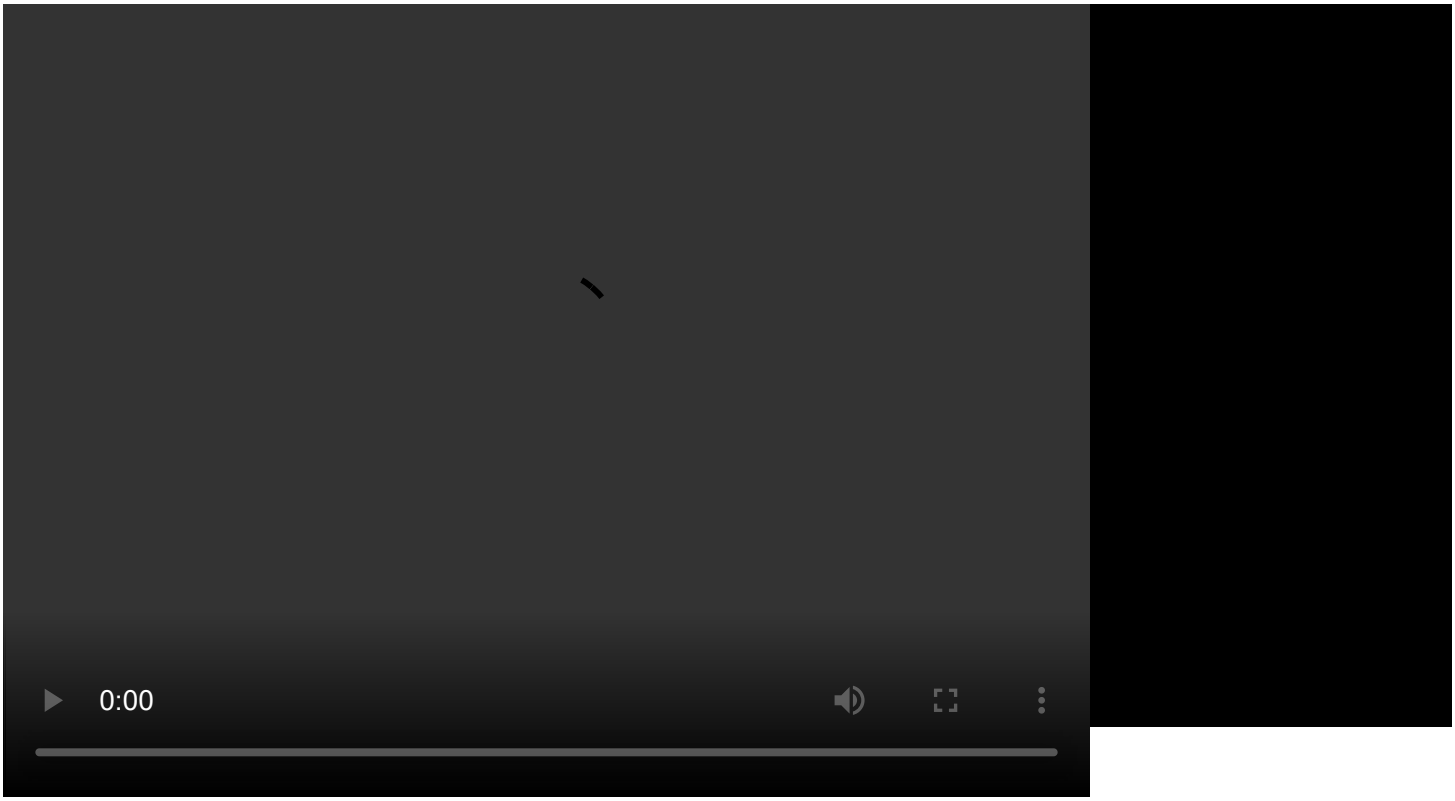
Read also about **Dynamic Bitmaps**.

Performance Impact

Due to the performance implications of moving computationally expensive widgets with the MCU, an animation that evolves in many small steps will appear slow and sluggish due to a high render time for each frame. Programming the animation to complete faster (in time) will cause individual steps to be large, and the animation will not appear smooth to the user.

The following is an example running on an STM32F429-DISCO board (240x320), where a fullscreen Container is moved up vertically, while a similar Container is moved in from the bottom.

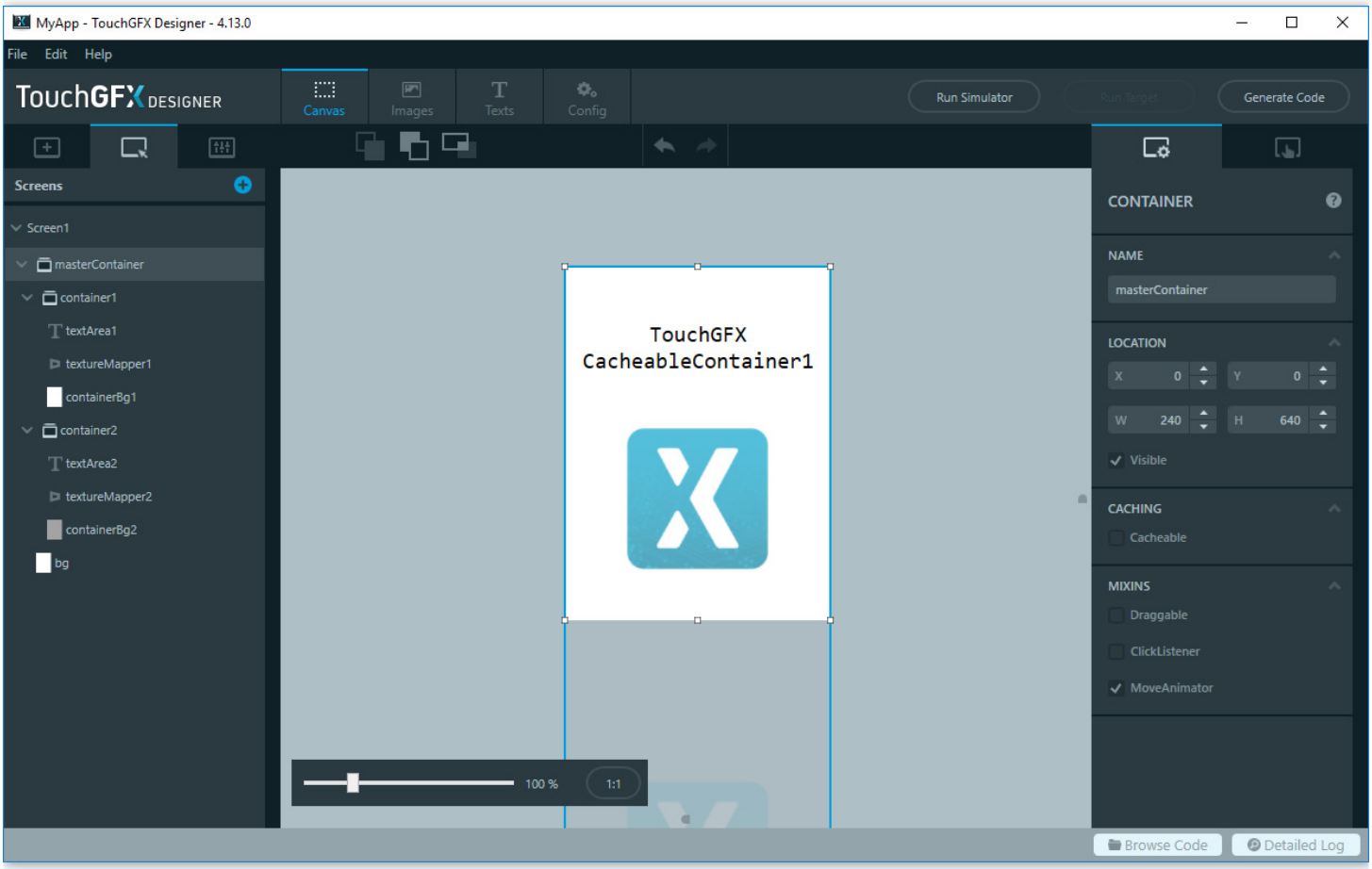
In the video below, the ToggleButton switches between CacheableContainer being enabled and disabled. The performance difference is clearly visible.



The two Containers that are moved each consist of a background Box, a TextArea, and a TextureMapper. The TextureMapper is configured to use the bilinear rendering algorithm and a global alpha of 174, making it very expensive to draw. The rendering time for the whole screen is around 100 ms on the STM32F429-DISCO board.

Test Application

In order to move the two elements relative to each other, they are put in a parent Container named `masterContainer` which is given twice the height of either child Container, giving it a size of `240 x 640 (2*320)`. By declaring the container as a move animator in TouchGFX Designer, it will be able to receive application ticks and animate over time during which performance can be measured.



CacheableContainer test application overview

The upper container named `container1` is placed at position x=0, y=0. The lower container named `container2` is placed at position x=0, y=320 directly below container1 in the parent `masterContainer`.

Since `container1` and `container2` are placed in the `masterContainer`, the two elements will move together when we move the `masterContainer`. For example, if we move the `masterContainer` to position x=0, y=-320, `container1` will be invisible, but `container2` will be fully visible. The animation between these two states can be created using an interaction in TouchGFX Designer.

The code below will move the `masterContainer` up if it is down, and down if it is already up. For simplicity, the code is inserted into the `handleClickEvent` eventhandler of the view, and is therefore executed whenever the user touches anywhere on the screen (below the ToggleButton):

Screen1View.cpp

```
void Screen1View::handleClickEvent(const ClickEvent& evt)
{
    //Forward event to base View (for the ToggleButton to work)
    View::handleClickEvent(evt);
    //If touch is released and y > 50 (below the ToggleButton), move masterContainer
    if (evt.getType() == ClickEvent::RELEASED && evt.getY() > 50)
    {
        const int endPosition = masterContainer.getY() >= 0 ? -320 : 0;
        masterContainer.startMoveAnimation(masterContainer.getX(), endPosition,
```

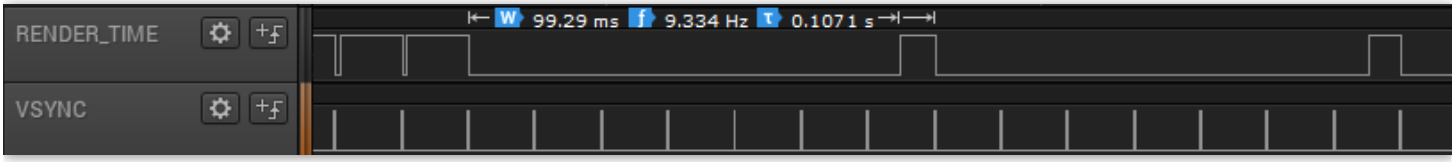
```
                                20 /* ticks */,
                                EasingEquations::cubicEaseInOut,
                                EasingEquations::cubicEaseInOut);
    }
}
```

Performance of Redrawing Complex Containers

As mentioned, the render time for one frame is around 100 ms when the MCU has to redraw the expensive TextureMapper at each small step of the animation. This gives us 10 frames per second (fps). The whole animation is 20 frames and will therefore take around two seconds.

On the STM32F429-DISCO evaluation kit, the rendering time is available as a digital signal on GPIO G14. The VSYNC signal is available on G13. The GPIO configuration is set up in the `GPIO.cpp` file.

The following image is a measurement of VSYNC and RENDER_TIME for the application when moving the `masterContainer` upwards:



Saleae Logic Software vsync and render time measurement

The rendering time is the first signal (active low). We can see that the rendering time for the first frame in the move animation is 99.29 ms.

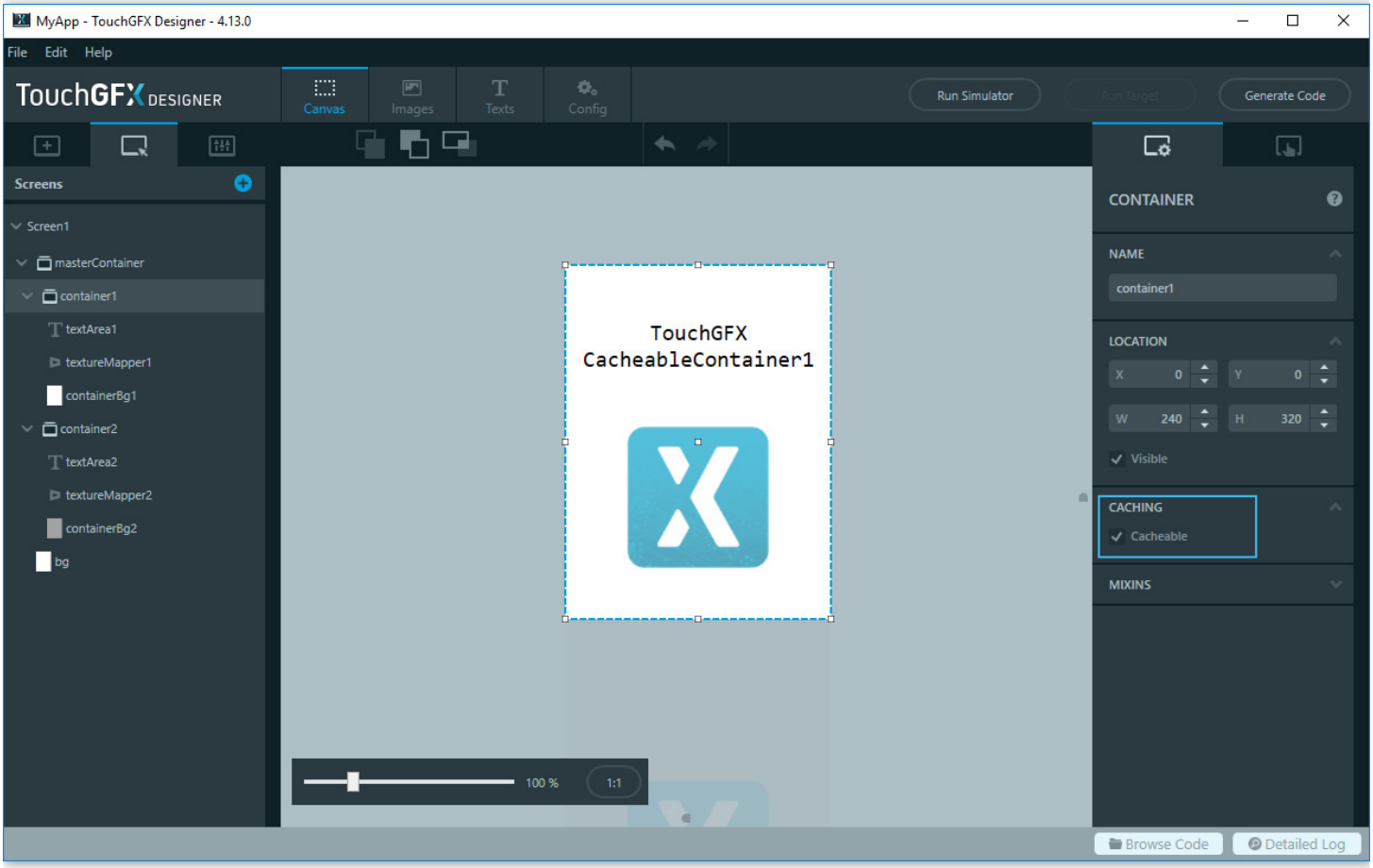
The lower signal is the VSYNC, which transitions high to low on every frame when pixels are clocked out to the display. We can see on the measurement above that drawing a single frame covers the time for 7 frames on the display. On the 8th VSYNC signal the rendering of the next frame starts. During the rendering, the display is repeatedly showing the previously drawn frame (in the other framebuffer).

Improving Performance Through Caching

We can improve the performance of the above move animation by caching the rendering of the container to memory. After doing that we can simply move the pixels located in that memory (using DMA) to the framebuffer, rather than redrawing a complex widget using the MCU. Even if an application could achieve 60 frames per second using the MCU alone it would be busy (perhaps with 100% MCU load) making the same calculations repeatedly rather than doing something more important.

This "in-memory-image" of the Container can now be shown on the screen at different places, instead of re-rendering the Container.

The first thing to do is to enable caching through TouchGFX Designer by checking the *Cacheable* property on the two Containers `container1` and `container2`:



CacheableContainer option on Container widget

The next step is to create two dynamic bitmaps in RAM that the Containers can be cached into.

Decide on an address in RAM where the bitmap cache should be located. In this particular example, we placed it in SDRAM (starts at address 0xd0000000 on an STM32F429) just after the framebuffers.

For the Windows simulator, the cache is allocated in a global variable:

Screen1View.hpp

```
#ifdef SIMULATOR
    uint32_t sdramBuffer[8*1024*1024/4];
    uint16_t* sdram = (uint16_t*)sdramBuffer;
#else
    uint16_t* sdram = (uint16_t*)(0xd0000000 + 320*240*2*2);
#endif
```

Initialize the bitmap cache and create two dynamic bitmaps for caching:

Screen1View.cpp

```
//Create bitmap cache and two dynamic bitmap for caching, each bitmap is 150Kb
Bitmap::setCache(sdram, 320*1024, 2); //320Kb cache
dynamicBitmap1 = Bitmap::dynamicBitmapCreate(240, 320, Bitmap::RGB565);
dynamicBitmap2 = Bitmap::dynamicBitmapCreate(240, 320, Bitmap::RGB565);
```

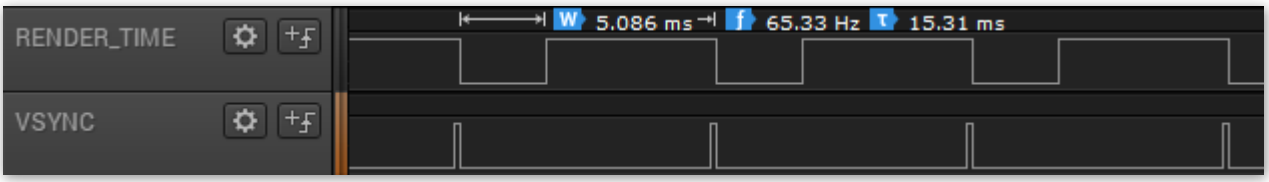
Assign the dynamic bitmaps to the Containers and set them in caching mode:

Screen1View.cpp

```
//Assign the bitmaps to the CacheableContainers
container1.setCacheBitmap(dynamicBitmap1);
container2.setCacheBitmap(dynamicBitmap2);

//Enable caching
container1.enableCachedMode(true);
container2.enableCachedMode(true);

//Finally update the cached bitmaps
container1.updateCache();
container2.updateCache();
```

Calls to `Container::updateCache()` will render the two Containers into their respective bitmaps. Call this method whenever an updated state of the containers is needed. This must be handled in application code by the developer.

With caching enabled for `container1` and `container2`, performance measurements now show a factor 20 improvement in render time from ~99ms to ~5ms meaning we can easily render in 60 frames per second completing the entire animation within 20 frames.



Saleae Logic Software vsync and render time measurement

Conclusion

Using CacheableContainer with DynamicBitmap when animating (frequent moves) can improve the render time dramatically when the subject is computationally complex and does not change between animation steps. In the event that the cache must update (e.g. a watch face when the time is updated) the contents of the cache can be recomputed at certain points during the animation controlled by the application.

FURTHER READING

- **Dynamic Bitmaps**
- **Dynamic Bitmaps: Load images at runtime**

Version: 4.16

Loading Images at Runtime

This section describes how dynamic bitmaps can be used to create applications where some of the graphic content is read from files or other input at runtime. The dynamic bitmaps can be used to show e.g. image files from an SD-card.

> **NOTE**
>
> Read first about **Dynamic Bitmaps**.

Recall that standard bitmaps are compiled into the application and therefore must be available at compile time. The Dynamic Bitmap feature allows you to read images from files at runtime, or even download images through an internet connection.

Loading BMP file Example

Here we will see how to use a BMP loader to load pixels from a Windows BMP file. The code for the loader is later in the article.

Insert first an Image widget in the view. This widget will show the BMP:

```
class TemplateView : public View
{
private:
    Image image;
}
```

Load the image date in setupScreen:

```
void TemplateView::setupScreen()
{
    FILE* f = fopen("image.jpg", "rb");
    uint16_t width, height;

    //Get the image dimensions from the BMP file
    BMPFileLoader::getBMP24Dimensions(f, width, height);
    BitmapId bmpId;

    //Create (16bit) dynamic bitmap of same dimension
    bmpId = Bitmap::dynamicBitmapCreate(width, height, Bitmap::RGB565);

    //Load pixels from BMP file to dynamic bitmap
```

```
    BMPFileLoader::readBMP24File(Bitmap(bmpId), f);

    //Make Image show the loaded bitmap
    image.setBitmap(Bitmap(bmpId));

    //Position image and add to View
    image.setXY(20, 20);
    add(image);
    ...
}
```

The BMP loader

Here is the code for a simple BMP file loader. It only supports 24bpp BMP files. You may have to adjust the file system calls to match your system.

BMPFileLoader.hpp

```
#include <touchgfx/hal/Types.hpp>
#include <touchgfx/Bitmap.hpp>

using namespace touchgfx;

class BMPFileLoader
{
public:
    typedef void* FileHdl;

    static void getBMP24Dimensions(FileHdl fileHandle, uint16_t& width, uint16_t& height);
    static void readBMP24File(Bitmap bitmap, FileHdl fileHandle);
private:
    static int readFile(FileHdl hdl, uint8_t* const buffer, uint32_t length);
    static void seekFile(FileHdl hdl, uint32_t offset);
};
```

BMPFileLoader.cpp

```
#include <gui/common/BMPFileLoader.hpp>
#include <touchgfx/Color.hpp>

int BMPFileLoader::readFile(FileHdl hdl, uint8_t* const buffer, uint32_t length)
{
    uint32_t r = fread(buffer, 1, length, (FILE*)hdl);
    if (r != length)
    {
        return 1;
    }
    return 0;
}
```

```
void BMPFileLoader::seekFile(FileHdl hdl, uint32_t offset)
{
    fseek((FILE*)hdl, offset, SEEK_SET);
}

void BMPFileLoader::getBMP24Dimensions(FileHdl fileHandle, uint16_t& width, uint16_t& heig
{
    uint8_t data[50];
    seekFile(fileHandle, 0);
    readFile(fileHandle, data, 26); //read first part of header.

    width = data[18] | (data[19] << 8) | (data[20] << 16) | (data[21] << 24);
    height = data[22] | (data[23] << 8) | (data[24] << 16) | (data[25] << 24);
}

void BMPFileLoader::readBMP24File(Bitmap bitmap, FileHdl fileHandle)
{
    uint8_t data[50];
    seekFile(fileHandle, 0);
    readFile(fileHandle, data, 26); //read first part of header.

    const uint32_t offset = data[10] | (data[11] << 8) | (data[12] << 16) | (data[12] << 2
    const uint32_t width = data[18] | (data[19] << 8) | (data[20] << 16) | (data[21] << 24
    const uint32_t height = data[22] | (data[23] << 8) | (data[24] << 16) | (data[25] << 2

    readFile(fileHandle, data, offset - 26); //read rest of header.

    //get dynamic bitmap boundaries
    const uint32_t buffer_width = bitmap.getWidth();
    const uint32_t buffer_height = bitmap.getHeight();

    const uint32_t rowpadding = (4 - ((width * 3) % 4)) % 4;

    const Bitmap::BitmapFormat format = bitmap.getFormat();
    uint8_t* const  buffer8  = Bitmap::dynamicBitmapGetAddress(bitmap.getId());
    uint16_t* const buffer16 = (uint16_t*)buffer8;

    for (uint32_t y = 0; y < height; y++)
    {
        for (uint32_t x = 0; x < width; x++)
        {
            if (x % 10 == 0) //read data every 10 pixels = 30 bytes
            {
                if (x + 10 <= width) //read 10
                {
                    readFile(fileHandle, data, 10 * 3); //10 pixels
                }
                else
                {
                    readFile(fileHandle, data, (width - x) * 3 + rowpadding); //rest of li
                }
            }
```

```
            //insert pixel, if within dynamic bitmap boundaries
            if (x < buffer_width && ((height - y - 1) < buffer_height))
            {
                switch (format)
                {
                case Bitmap::RGB565:
                    buffer16[x + (height - y - 1) * buffer_width] =
                        touchgfx::Color::getColorFrom24BitRGB(data[(x % 10) * 3 + 2], data
                    break;
                case Bitmap::RGB888:
                    {
                        //24 bit framebuffer
                        const uint32_t inx = 3 * (x + (height - y - 1) * buffer_width);
                        buffer8[inx + 0] = data[(x % 10) * 3 + 0];
                        buffer8[inx + 1] = data[(x % 10) * 3 + 1];
                        buffer8[inx + 2] = data[(x % 10) * 3 + 2];
                        break;
                    }
                case Bitmap::ARGB8888:
                    {
                        //24 bit framebuffer
                        const uint32_t inx = 4 * (x + (height - y - 1) * buffer_width);
                        buffer8[inx + 0] = data[(x % 10) * 3 + 0];
                        buffer8[inx + 1] = data[(x % 10) * 3 + 1];
                        buffer8[inx + 2] = data[(x % 10) * 3 + 2];
                        buffer8[inx + 3] = 255; //solid
                        break;
                    }
                default:
                    assert(!"Unsupported bitmap format in BMPFileLoader!");
                }
            }
        }
    }
}
```

This code is for illustrative purposes. A more optimal reader for RGB888 can read directly from the file to the dynamic bitmap memory (remember to skip the row padding). The reader above reads 10 pixels from the BMP file to a temporary buffer. The pixels are then copied to the bitmap while converting to the correct format.

Configure memory for dynamic bitmaps

Before you can create and use dynamic bitmaps you must configure TouchGFX. It is a prerequisite to provide a buffer and the maximum number of dynamic bitmaps (also for the simulator).

Here is an example for STM32F7xx where we allocate a buffer in external RAM: We wish to load and show a 24-bit bitmap of size 320x240. The memory requirement is thus 320x240x3 = 230400. We also

need a little space for bookkeeping, so we allocate 232000 bytes for the buffer.

```
static uint32_t bmpCache = (uint32_t)(0xC00C0000); // SDRAM
void touchgfx_init()
{
  HAL& hal = touchgfx_generic_init<STM32F7HAL>(dma, display, tc, 480, 272, (uint16_t*)bmpC
  ...
}
```

The final argument is the maximum number of dynamic bitmaps, so adjust this according to your needs.

> **NOTE**
>
> **Note that in case of insufficient memory, the BitmapId returned by dynamicBitmapCreate will be BITMAP_INVALID.**

Loading JPEG files

A JPEG File Loader example can be found here which shows how to use LibJPEG to use JPEG-files. It uses a JPEGLoader class similar to the above BMPFileLoader.

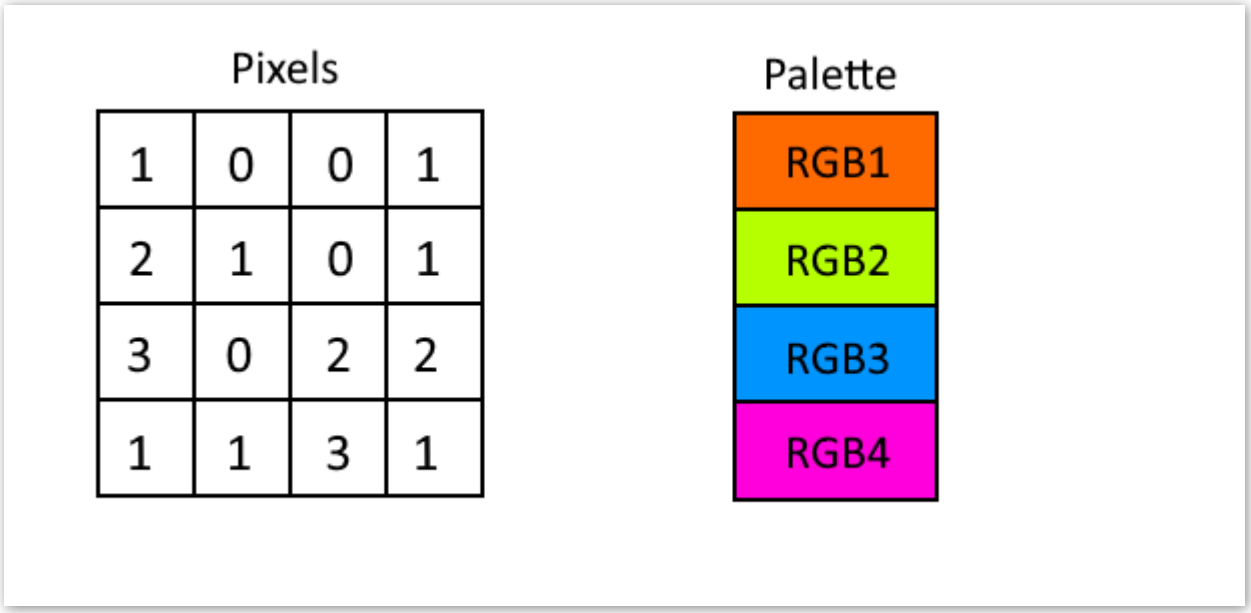
Version: 4.16

Using the L8 Image Format to Reduce Memory Consumption

Images in the L8 format take up less flash storage and are faster to draw than e.g. ARGB8888.

An image in L8 format consists of a color palette and a pixel array: The color palette lists up to 256 different colors specified in either 16-bit format RGB565, 24-bit format RGB888, or 32-bit format ARGB8888. The pixel array consists of one byte for each pixel. This byte is an index into the color palette (list of colors), pointing out the color for the pixel. The TouchGFX framework draws an L8 image by reading the pixels one-by-one, looking up the colors in the palette and writing these to the framebuffer. This happens automatically and is accelerated by the STM32 Chrom-ART hardware accelerator, if available on the hardware.

8-bit per pixel means that one L8 image can use 256 different colors. Another L8 image can use 256 other colors, since the two images each have their own palette.



An L8 image with 4 x 4 pixels and a palette with 4 colors

Pixels are one byte (8-bit) each. The size of the pixels is therefore width x height bytes. The palette colors can be 16-bit, 24-bit, or 32 bit colors. Each color definition will therefore take up 2, 3, or 4 bytes.

Solid images should be stored in L8_RGB565 if the framebuffer is 16-bit (RGB565 format). If the framebuffer is 24-bit (RGB888) the L8 images must be stored in L8_RGB888 format. If the image is transparent the 32-bit format (ARGB8888) must be used:

Format	Framebuffer format	Supports transparent pixels

Format	Framebuffer format	Supports transparent pixels
L8_RGB565	16-bit RGB565	No
L8_RGB888	24-bit RGB888	No
L8_ARGB8888	Both	Yes

FURTHER READING

- Read more about palette image formats here: **https://en.wikipedia.org/wiki/Indexed_color**

Example L8 Image

Here is a typical logo image. This image only uses 16 different colors:

200 x 200 pixels L8 image with 16 different 24-bit colors.

The size in flash of this image is significant lower than the original image in the standard 24-bit format (RGB888). The table below lists the flash usage for this concrete image for the three different palette formats and for the non L8 format

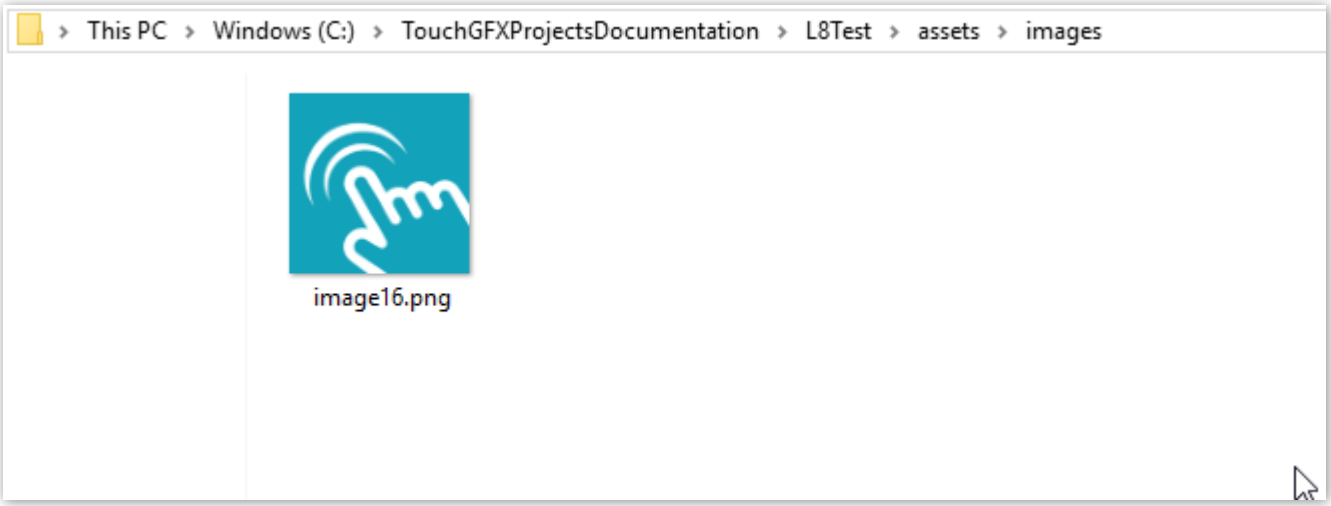
Format	Size of pixels (bytes)	Size of palette (bytes)	Total size (bytes)	Reductio
RGB888	120,000	0	120,000	-
L8_RGB565	40,000	32	40,032	66.6%
L8_RGB888	40,000	48	40,048	66.6%
L8_ARGB8888	40,000	64	40,064	66.6%

We see that the size reduction is very large, and that the size of the palette is insignificant on a medium sized image.

Using L8 Images in TouchGFX Designer

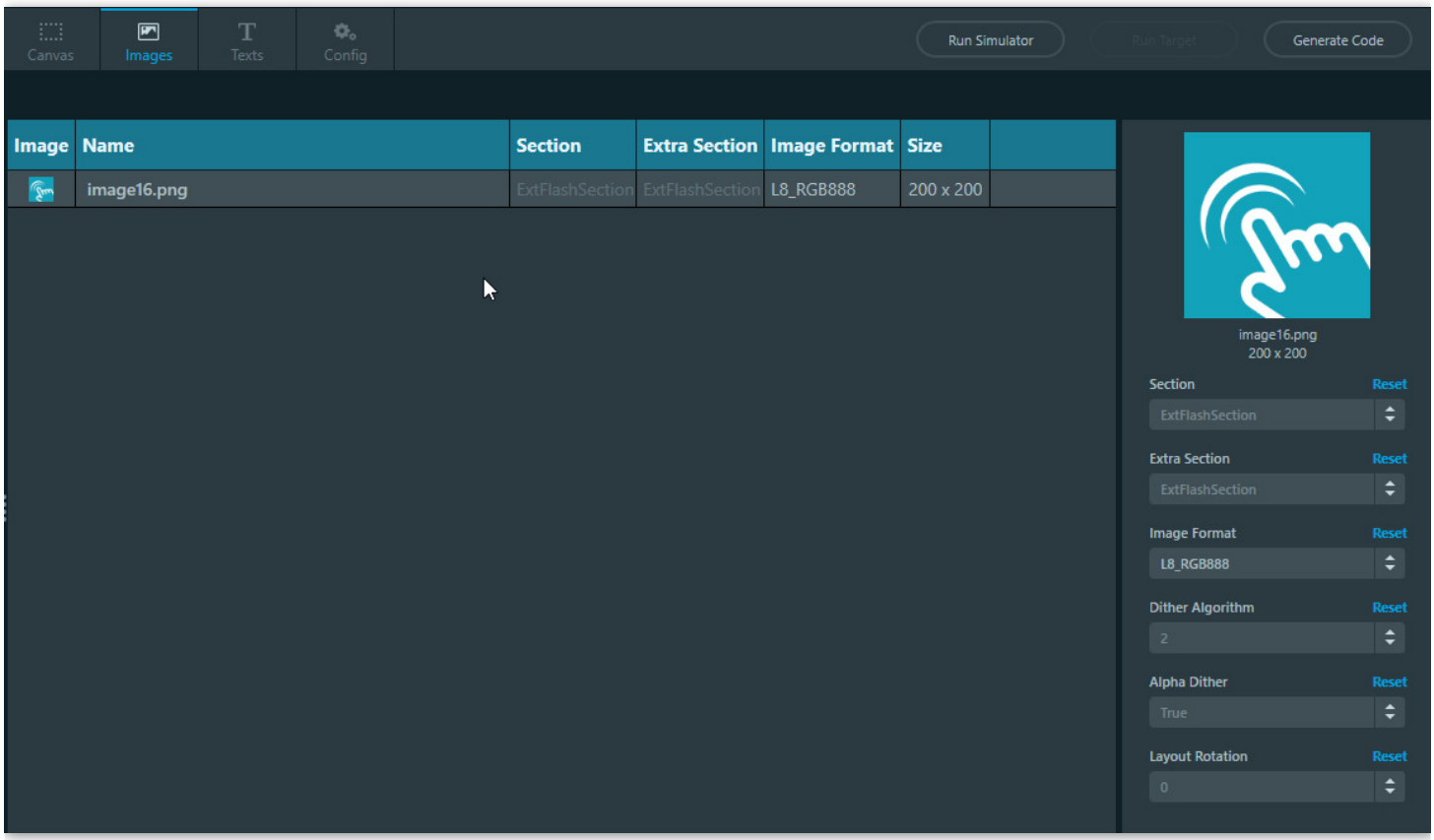
It is very easy to use the L8 image format in TouchGFX. The only thing to do is to configure the image converter to convert the image from PNG to L8 format. Here we will go through the whole process:

Start a new project in the TouchGFX Designer. Copy your image to the assets/images folder in the new project:



Images folder of TouchGFX project

Now go to the TouchGFX Designer and click the Images tab in the top bar and select the image:



Logo in Images view of TouchGFX Designer

On the right side on the window, select image format L8_RGB888 (this example is running 24 bit colors).

An image Widget can now be inserted on the canvas (here we inserted a Box in the background):

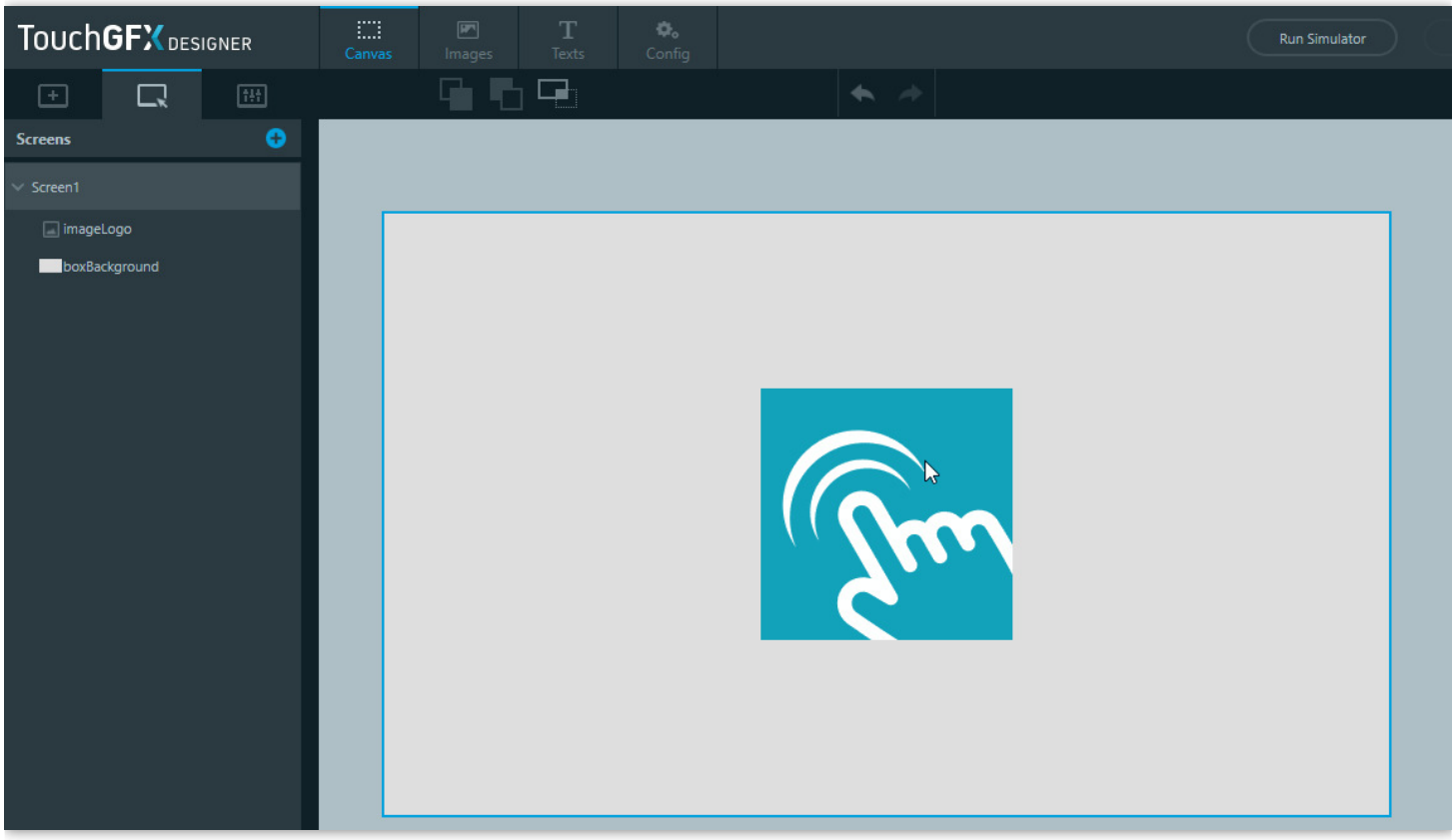


Image widget on Canvas in TouchGFX Designer

Nothing needs to be changed in the UI code. The Image Converter converts the PNG file and generates an image in L8 format because of the configuration we did in the Images tab.

Transparent Images

As mentioned above it is also possible to use L8 format for images with transparency.

170 x 60 pixels button image in 32 bit format ARGB8888

The above image uses 108 colors (many shades of blue). This image can use the format L8_ARGB8888. The size will be significantly lower:

Format	Size of pixels (bytes)	Size of palette (bytes)	Total size (bytes)	Reductio
ARGB888	40,800	0	40,800	-

Format	Size of pixels (bytes)	Size of palette (bytes)	Total size (bytes)	Reductio
L8_ARGB8888	10,200	432	10,632	73.9%

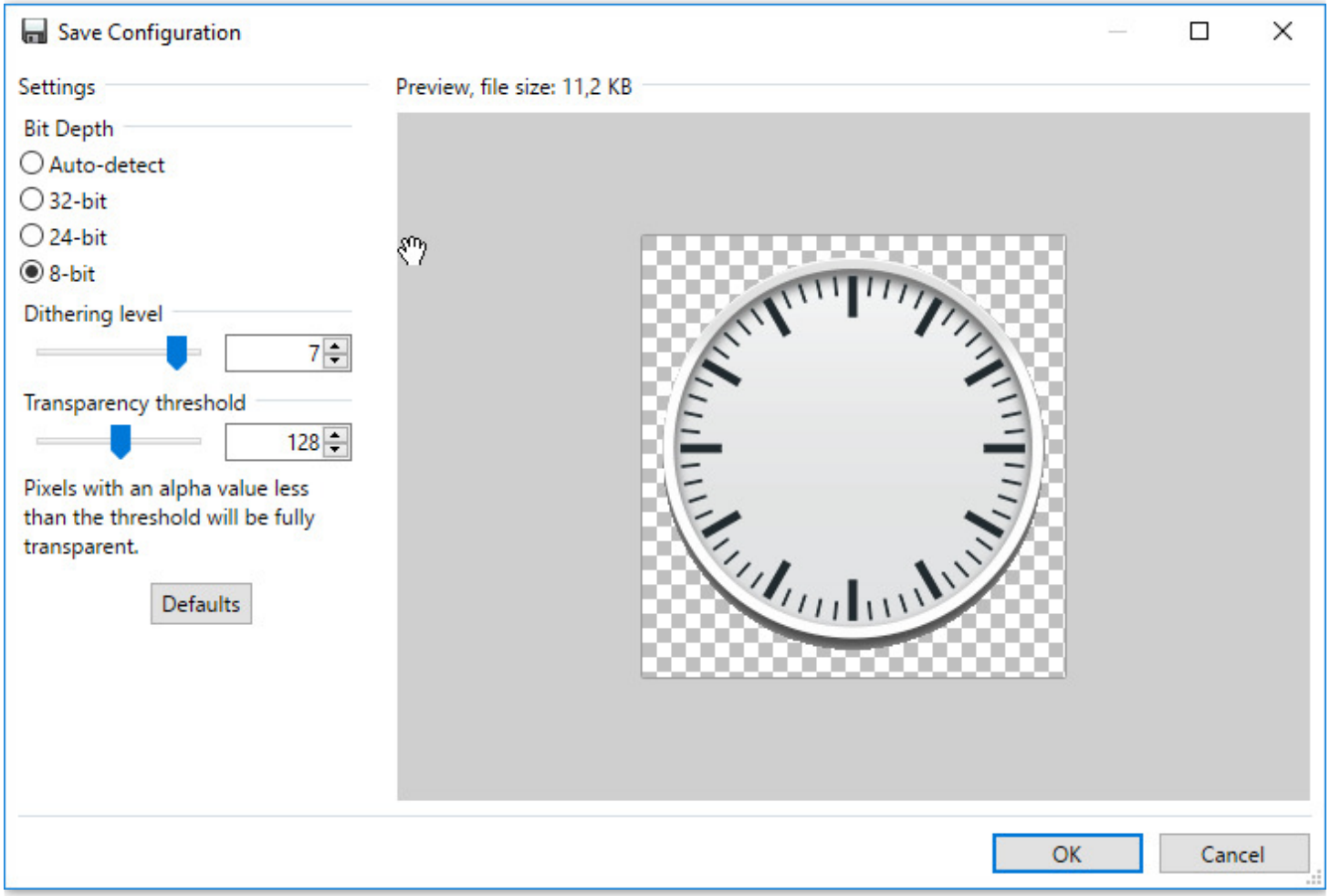
Converting an image to 256 colors

Many images use more than 256 colors. This is common for images that are photo-realistic or images with gradients. These images cannot directly be converted to L8 image format in the TouchGFX Designer, because they contain to many colors.

In many cases though, it is possible to reduce the number of colors used in a specific image. Ideally, a graphics artist will convert/supply the images in 256 colors, however image manipulation tools can also perform the conversion without loosing too much of the image quality.

Paint.NET

The simplest way is to use Paint.NET. Open the original image and use Save As to save the image in another file. In the Save Configuration dialog, select 8-bit, as pixel depth:



Paint.NET saving image in 8 bit format

Now use the new PNG in your project. Remember to select the L8_ARGB8888 format in the Images tab in the TouchGFX Designer. Shadows are not handled very well, but icons with transparent edges looks good in many cases. It is possible to adjust the "Transparency threshold" value and in some cases improve the result.

ImageMagick

Another suitable tool, that sometimes results in better L8 images, is ImageMagick (download from www.imagemagick.org). This tool can convert images from the command line. This makes it suitable for use in scripts. To convert the clock_bg.png to an image using at most 256 colors, use the following command:

```
magick convert clock_bg.png -colors 256 clock_bg_l8_256.png
```

ImageMagick can also tell you how many colors are used in an image. Use this command:

```
magick identify -format %k Blue_Buttons_Round_Edge_small.png
```

Comparison

The three images (original, L8 using Paint.NET, L8 using ImageMagick) are seen below for comparison:

Clock image comparison, left to right: original, Paint.NET, ImageMagick

The middle clock lost the details in the border shadow. In both cases the central part of the clock background looks useable.

Manual Configuration

It is also possible to select image formats without using the TouchGFX Designer. The image formats are specified in file application.config located in the project root:

application.config

```
{
  "image_configuration": {
    images": {
      "Blue_Buttons_Round_Edge_small.png": {
        "format": "L8_ARGB8888"
      }
    },
    "dither_algorithm": "2",
    "alpha_dither": "yes",
    "layout_rotation": "0",
    "opaque_image_format": "RGB888",
    "nonopaque_image_format": "ARGB8888",
    "section": "ExtFlashSection",
    "extra_section": "ExtFlashSection"
  }
}
```

The "images" section under "image_configuration" specifies the format for individual images. If an image is not mentioned here, the image will be generated in the default format (opaque_image_format or nonopaque_image_format).

We recommend using the TouchGFX Designer for image configuration when possible.

Version: 4.16

Creating Dynamic L8 Images

This section explains the use of dynamic L8 images and especially how to create the palette.

Read in general about Dynamic Bitmaps here and about the L8 image format here.

Dynamic L8 Images

Dynamic L8 bitmaps are created like any other dynamic bitmap, except that we also have to specify the type of palette to create for the bitmap.

TouchGFX supports 3 types of palettes:

Palette	Description
CLUT_FORMAT_L8_ARGB8888	32-bit, 8 bits for each of red, green, blue and per pixel alpha channel
CLUT_FORMAT_L8_RGB888	24-bit, 8 bits for each of red, green and blue, no per pixel alpha channel
CLUT_FORMAT_L8_RGB565	16-bit, 5 bits for red, 6 bits for green, 5 bits for blue, no per pixel alpha channel

Creating a Dynamic L8 image with 24-bit palette

Here we create a 100x100 pixels L8 bitmap with a 24-bit palette:

Screen1View.cpp

```
BitmapId dynamicBitmap1 = Bitmap::dynamicBitmapCreate(100, 100, Bitmap::L8, Bitmap::CLUT_F
```

This call allocated a 100x100 L8 bitmap and a 24-bit palette in the bitmap cache. The palette always holds 256 colors for dynamic bitmaps.

Accessing the palette

The palette is located 4 bytes after the pixels (aligned on 32-bit). The bytes contains information about the palette type and length of the palette.

We can get a pointer to the (so far uninitialized) palette like this:

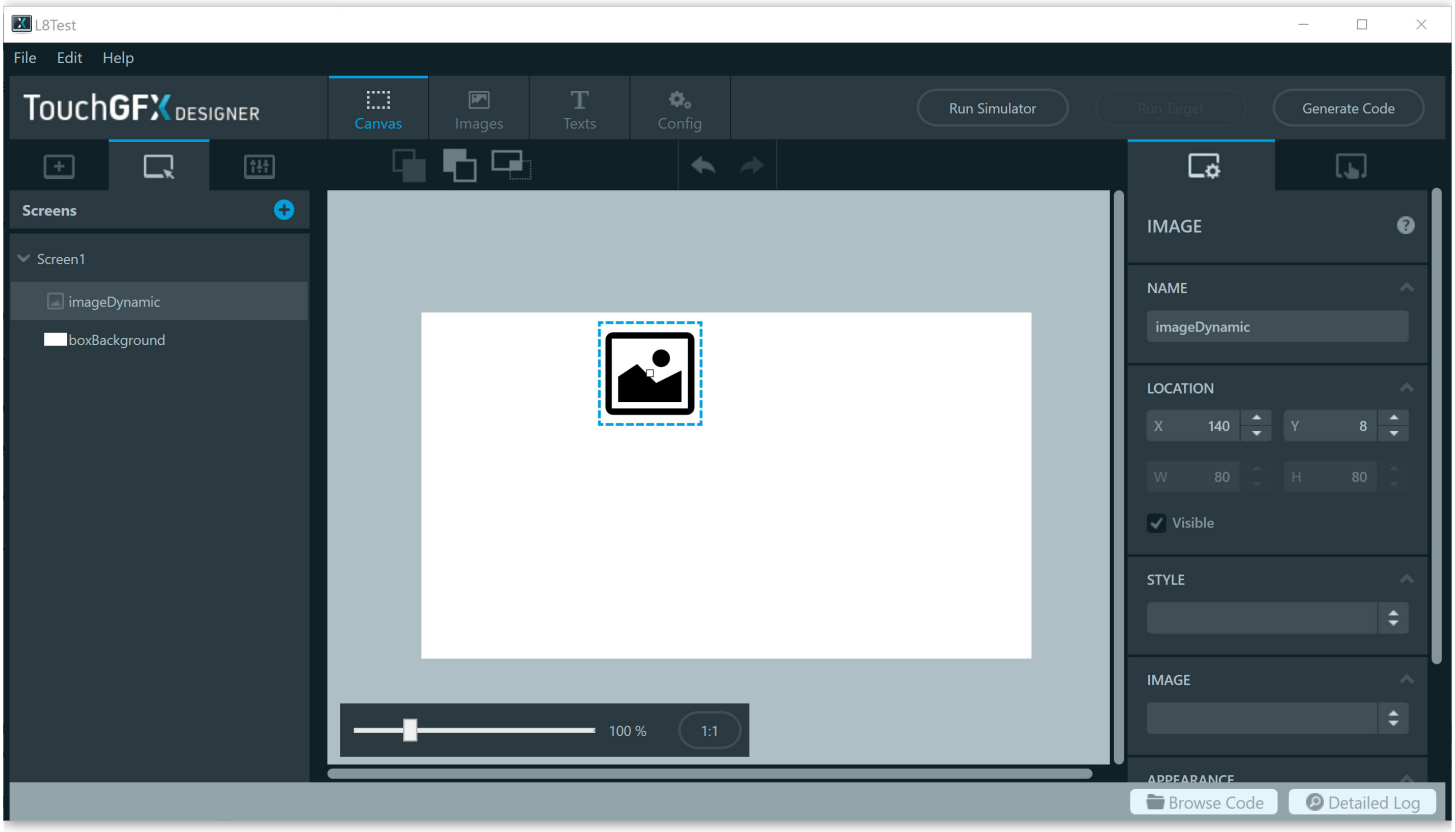
Screen1View.cpp

```
//Get a pointer to the bitmap data (pixels and palette)
uint8_t* data = Bitmap::dynamicBitmapGetAddress(dynamicBitmap1);

//1 byte pr pixel, round up to 32-bit
uint32_t byteSize = 100*100;
byteSize = ((byteSize + 3) & ~3);

//Palette starts four bytes after the pixels
uint8_t* pal = (data + byteSize + 4);
```

Dynamic L8 Bitmap Example

We will now go through an example of creating a dynamic L8 bitmap and manipulating the palette. Manipulating the palette if not a typical thing to do for a general application. The example serves the purpose of illustrating how to access and generate a palette.

First we create a Screen in TouchGFXDesigner and insert a white Box in the background and an Image at e.g. x=140, y=8:



Creating a Screen

Now generate the code and open the Screen1View.cpp file. We must insert code in *setupScreen* to initialize the bitmap cache and create a dynamic bitmap.

We create a bitmap of 200*256 pixels. Remember each pixel in an L8 bitmap is one byte. We color each row of the image with a different color. First row has color 0, last row has color 255.

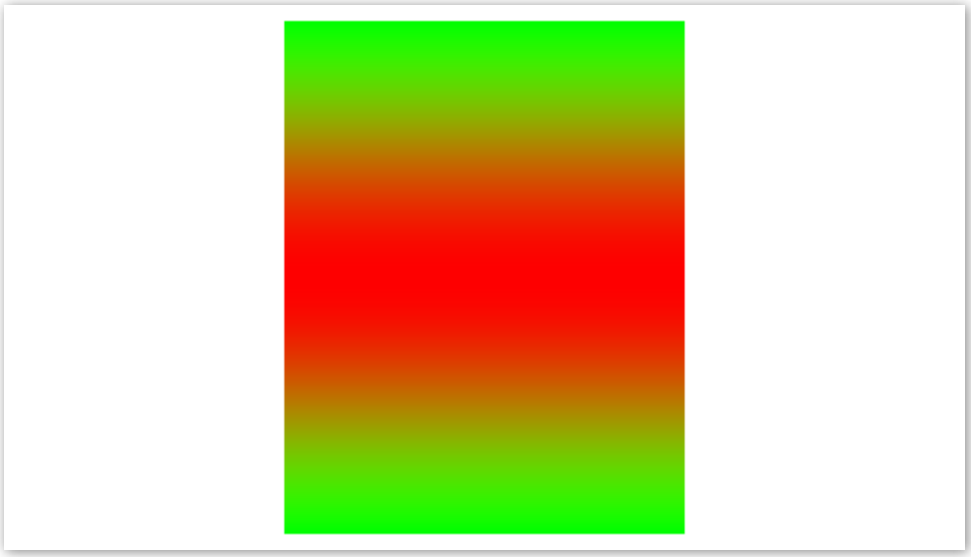
Then we initialize the colors in the palette. We calculate the start address of the palette and set the RGB values of the 256 colors. Here we create colors that go from green to red and back to green again.

Screen1View.cpp

```
#ifdef SIMULATOR
uint32_t cacheBuffer[320*1024/4]; //simulate PSRAM
uint16_t* psram = (uint16_t*)cacheBuffer;
#else
uint16_t* psram = (uint16_t*)(0xd0000000 + 480*272*2*2); //Address after two 16bit framebu
#endif

Screen1View::Screen1View()
{

}

void Screen1View::setupScreen()
{
    Screen1ViewBase::setupScreen();

    //Create one dynamic bitmap
    Bitmap::setCache(psram, 320*1024, 1); //320Kb cache
    BitmapId dynamicBitmap1 = Bitmap::dynamicBitmapCreate(200, 256, Bitmap::L8, Bitmap::CL
    imageDynamic.setBitmap(Bitmap(dynamicBitmap1));

    if (dynamicBitmap1 == BITMAP_INVALID)
    {
        touchgfx_printf("Unable to create dynamic bitmap\n");
    }
    else
    {
        uint8_t* data = Bitmap::dynamicBitmapGetAddress(dynamicBitmap1);

        uint8_t* pixel = data;
        //make colored rows
        for (int y = 0; y<256; y++)
        {
            for (int x = 0; x<200; x++)
            {
                *pixel++ = y;
            }
        }
```

```
        uint32_t byteSize = 200*256;
        byteSize = ((byteSize + 3) & ~3);

        //Palette starts four bytes after the pixels
        uint8_t* pal = (data + byteSize + 4);

        //Make palette with 256 colors from green to red to green
        for (int i = 0; i<256; i++)
        {
            //BGR
            pal[i*3 + 0] = 0x00;
            pal[i*3 + 1] = 127*(1+cosf(i*6.28f/255));
            pal[i*3 + 2] = 255*(sinf(i*3.14f/255));
        }
    }
}
```

This gives us a Screen that looks like this:



Showing L8 image

Manipulating the Palette

Since we have access to the palette used for the dynamic L8 bitmap, we can easily manipulate it.

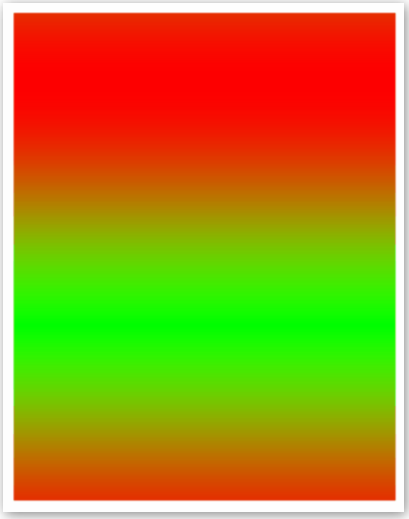
Here we cycle the colors one index down and force a redraw of the image in every frame:

Screen1View.cpp

```
...
void Screen1View::handleTickEvent()
{
    //get palette address
    uint8_t* data = Bitmap::dynamicBitmapGetAddress(imageDynamic.getBitmap());
```

```
    uint32_t byteSize = 200*256;
    byteSize = ((byteSize + 3) & ~3);
    uint8_t* pal = (data + byteSize + 4);

    //Cycle palette, copy color 0
    int8_t blue = pal[0], green = pal[1], red = pal[2];

    //Move palette down, 1->0, 2->1, ...
    for (int i = 0; i<255*3; i+=3)
    {
        pal[i] = pal[i+3];
        pal[i+1] = pal[i+4];
        pal[i+2] = pal[i+5];
    }

    //Insert color 0 as color 255
    pal[255*3+0] = blue;
    pal[255*3+1] = green;
    pal[255*3+2] = red;

    //Force redraw by invalidating
    imageDynamic.invalidate();
}
```

This will move colors in the dynamic bitmap "upwards":



Cycling the L8 palette

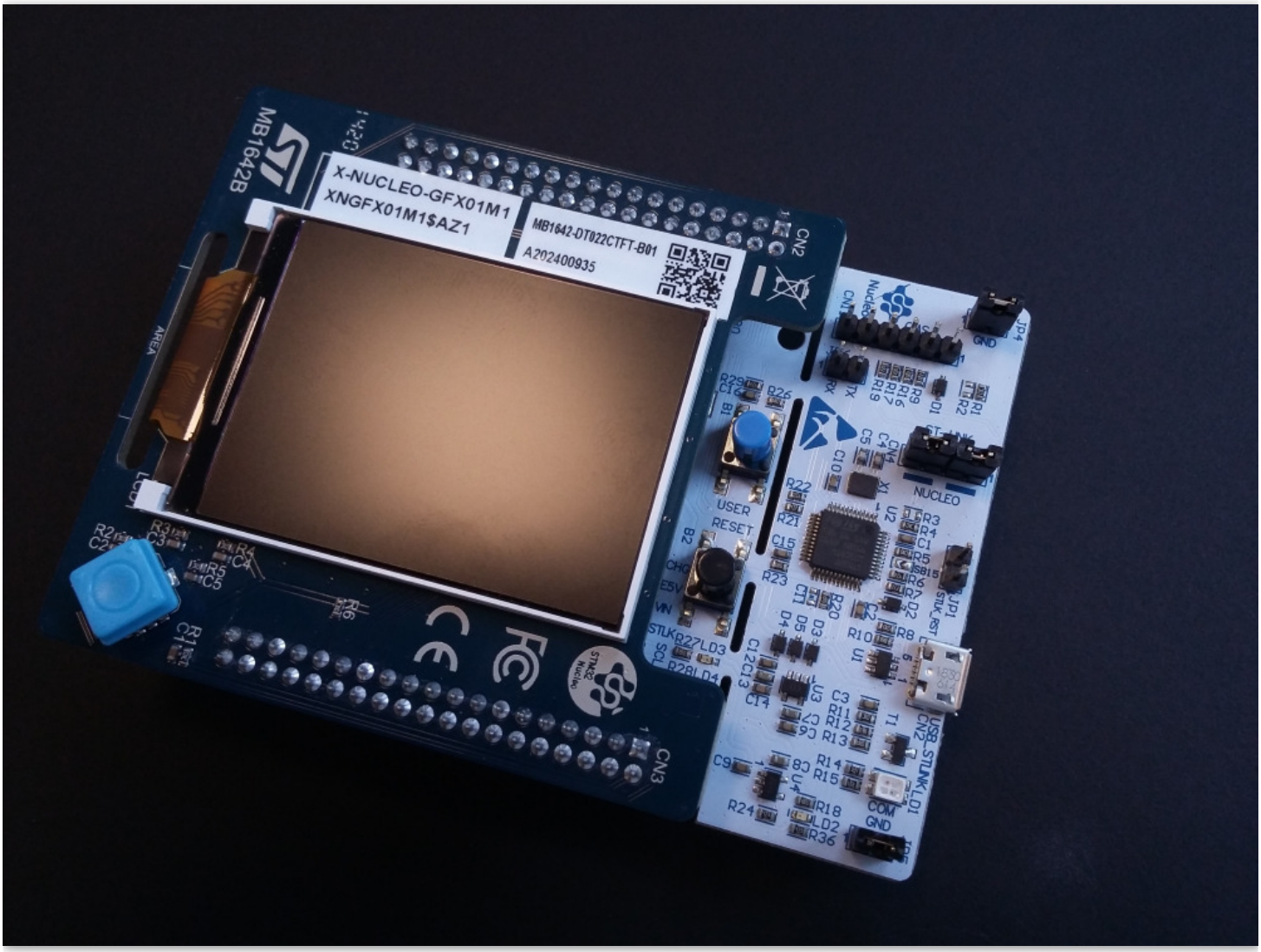
Version: 4.16

TouchGFX on Low Cost Hardware

This section discusses how to use TouchGFX on low cost hardware with limited amount of RAM and flash, no acceleration and slow connection to external flash and display.

We will try to give some advice on writing the best applications of the given hardware.

Throughout this section we will use the application template for the STM32G071 nucleo board with the X-Nucleo-GFX01M1 expansion board as example hardware.



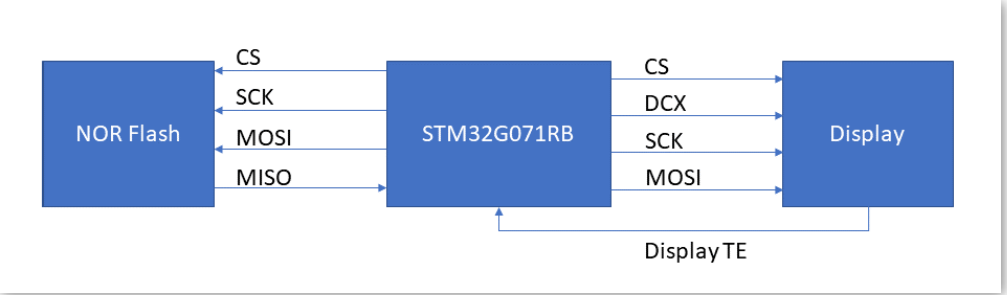
Nucleo-G071RB with X-Nucleo-GFX01M1 expansion board

Hardware Overview

The hardware setup in this kit consist of the STM32G071 MCU, a SPI NOR flash, a SPI display, and joystick button.

Component	
MCU	STM32G071RB
MCU RAM	32 Kb
MCU Flash	128 Kb
Display	Displaytech DT022CTFT
Display resolution	240 x 320
Display controller	ILI9341V
Display connection	SPI
Display connection speed	32 MHz
NOR Flash	Macronix MX25L6433F
NOR Flash size	64 Mbit
NOR Flash connection speed	32 MHz

The display is connected to the SPI1 peripheral and the flash is connected to the SPI2 peripheral. This allows the MCU to read data from the flash while transmitting data to the display.



Nucleo-G071RB with X-Nucleo-GFX01M1 architecture

GPIO Allocation

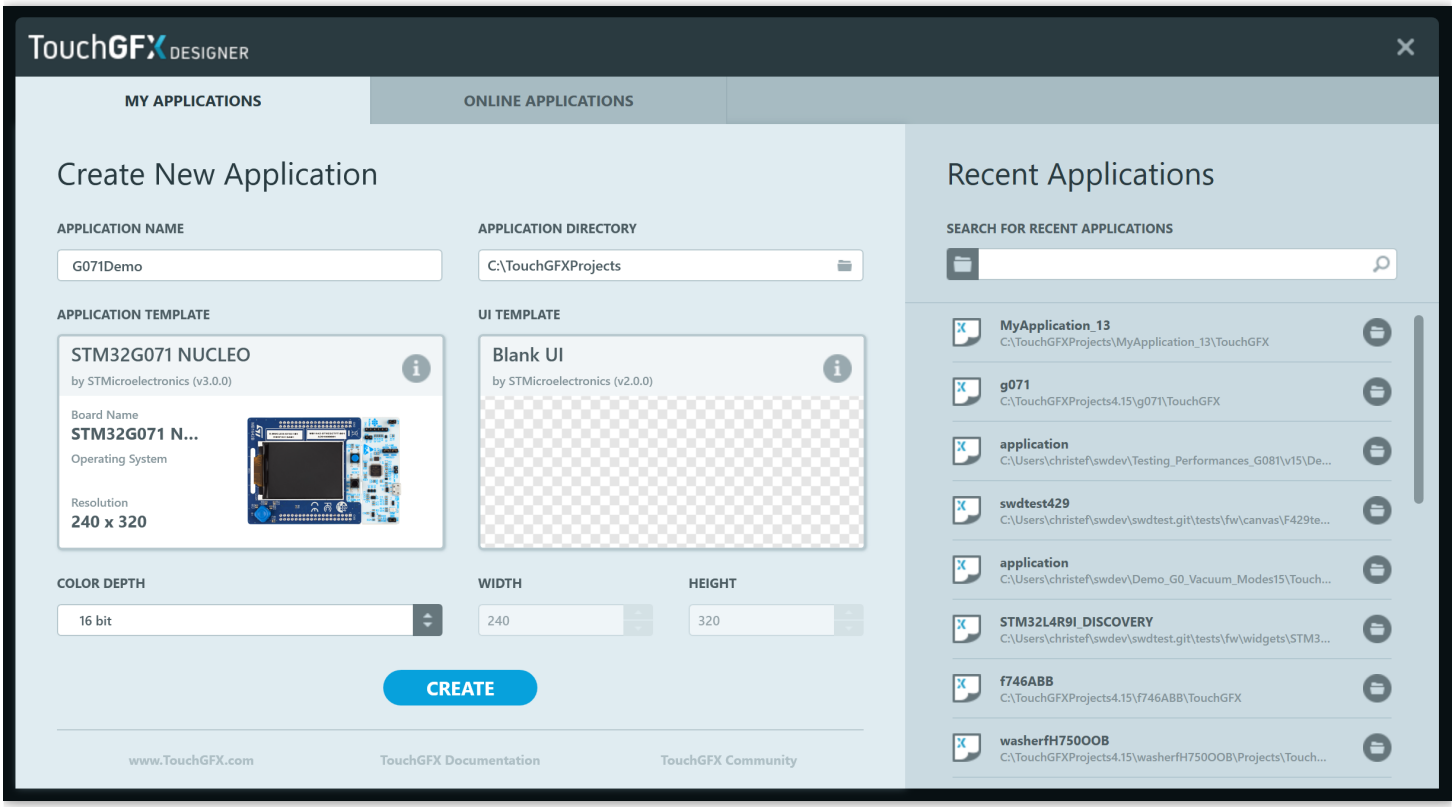
Signal	GPIO Pin
Display CS	PB5

Signal	GPIO Pin
Display DCX	PB3
Display SCK	PA5
Display MOSI	PA7
Display TE	PA0
Flash CS	PB9
Flash SCK	PB13
Flash MOSI	PC3
Flash MISO	PC2

The table above lists the GPIO allocation for the signals to the flash and display. These signal can be monitored with a oscilloscope or logic analyzer. This is very usefull during debugging of e.g. performance problems.

Starting a Project

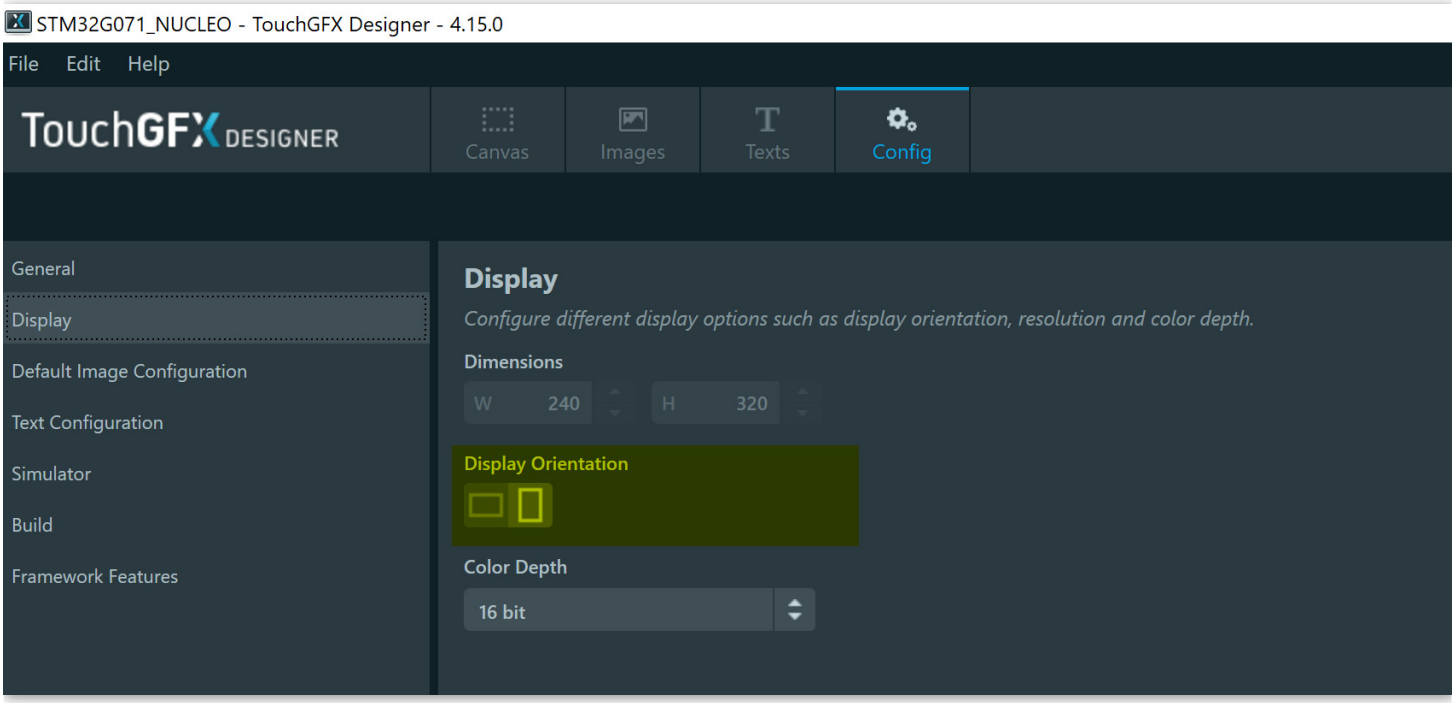
It is easy to start a project for the STM32G071RB Nucleo evaluation kit in the TouchGFX Designer. Click "Change" on the Application Template and select the STM32G071 Nucleo. This template is developed specifically for the Nucleo-G071RB kit with the X-Nucleo-GFX01M1 display shield.



New project for Nucleo-G071RB

The application template supports the NOR flash, the display, and the buttons. The display can be used both in portrait and horisontal mode.

The display orientation can be change in the TouchGFX Designer in the *Config -> Display* section:



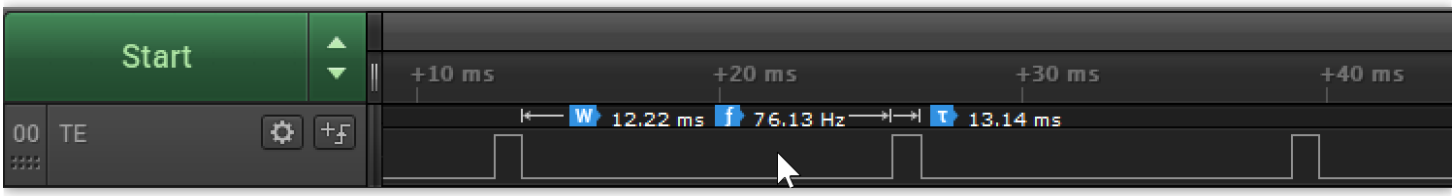
Selecting portrait or horizontal display orientation

The display on the X-Nucleo-GFX01M1 shield is natively portrait orientated (higher than wide), but it can easily be used with horizontal orientation.

Display Updates

As mentioned above the display resolution is 240 x 320 pixels. A total of 76.800 pixels or 153.600 bytes. The SPI connection between the MCU and the display is running at 32 MHz. This allows us to transfer 4 MBytes/s or 2M pixels/s.

The refresh rate of the display is 76.1 Hz which gives us a frame time of 13.14 ms.



Tearing effect signal from the display

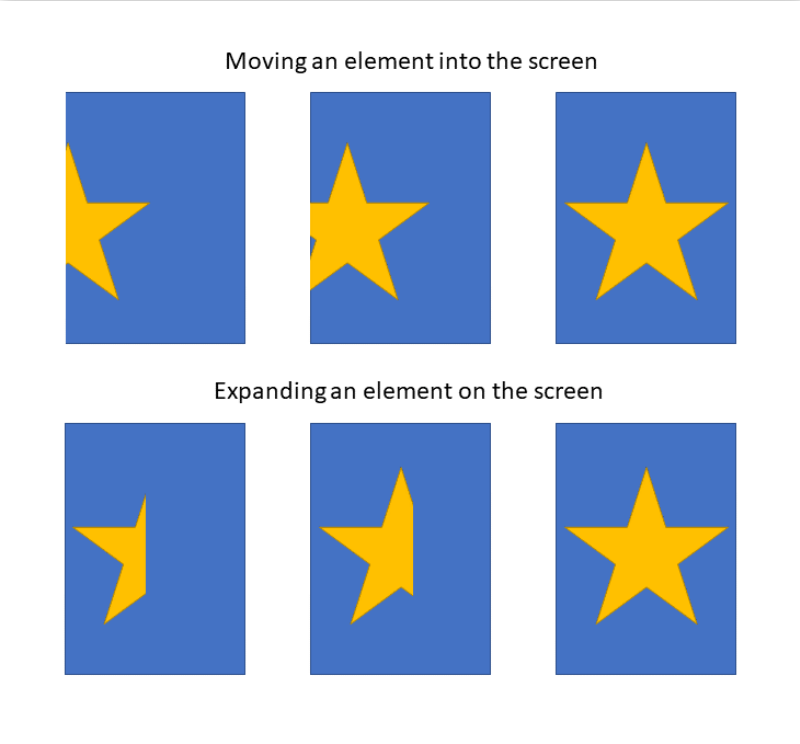
This means that we have at most 13 ms to send data for the next frame. In that time we can send 2.000.000 pixels/s / 76 fps = 26.280 pixels / frame or 34% of a full screen.
In practice we cannot sustain that transfer speed on the SPI bus because of the protocol overhead so we cannot expect to send more than approximately 30% of a full frame.

If the application updates more than that amount of pixels the hardware cannot complete the transmission within the frame time. The result is that the display will start showing the updated frame before it is completely updated. The user will then in some cases see a mix of the old frame and the new frame.

For some animations this is not noticeable to the user, for others the result will be unacceptable.

We therefore recommend to keep the level of updates below the 30% limit. E.g. by incrementally updating the frame step-by-step.

Because of this, it is generally better to expand an item on the screen, than moving the item.



Tearing effect signal from the display

When the star is moved to the right, all the pixels covered by the star must be updated. When the star is expanded only the new pixels must be updated. The pixels updated in previous frame remain unchanged.

Drawing Speed

The transmission to the display is running at maximum 32 MHz.

The serial flash can run at the same speed as the display transmission. This means that the serial flash is fast enough to feed bits to the display at maximum speed.

This is only achieved if the pixel format of an image in the flash is RGB565. In this case is two bytes read from the flash equal to 1 pixel, which is also two bytes on the display.
If the pixel format in the flash is ARGB8888, we need to read double the amount of data from the flash to produce a pixel on the display, and the serial flash will not be able to keep up with the display.

When this happens we are not sending data to the display continously and it will not be possible to update all 30% of the display in a frame. One possibility is to move the image to internal flash, another of course to change the pixel format.

Other widgets are not bound by the speed of the flash. E.g. the Box Widget, which draws a colored rectangle. This widget is of course very fast and much faster than the display transmission. Other widgets like Line and Circle uses much more CPU resources. These Widget are not able to produce pixels in the speed they can be transmitted to the display. Using these Widgets an application cannot expect to be able to update 30% of the display in every frame.

Find about pixel rendering complexity here

TouchGFX Limitations with Serial Flash

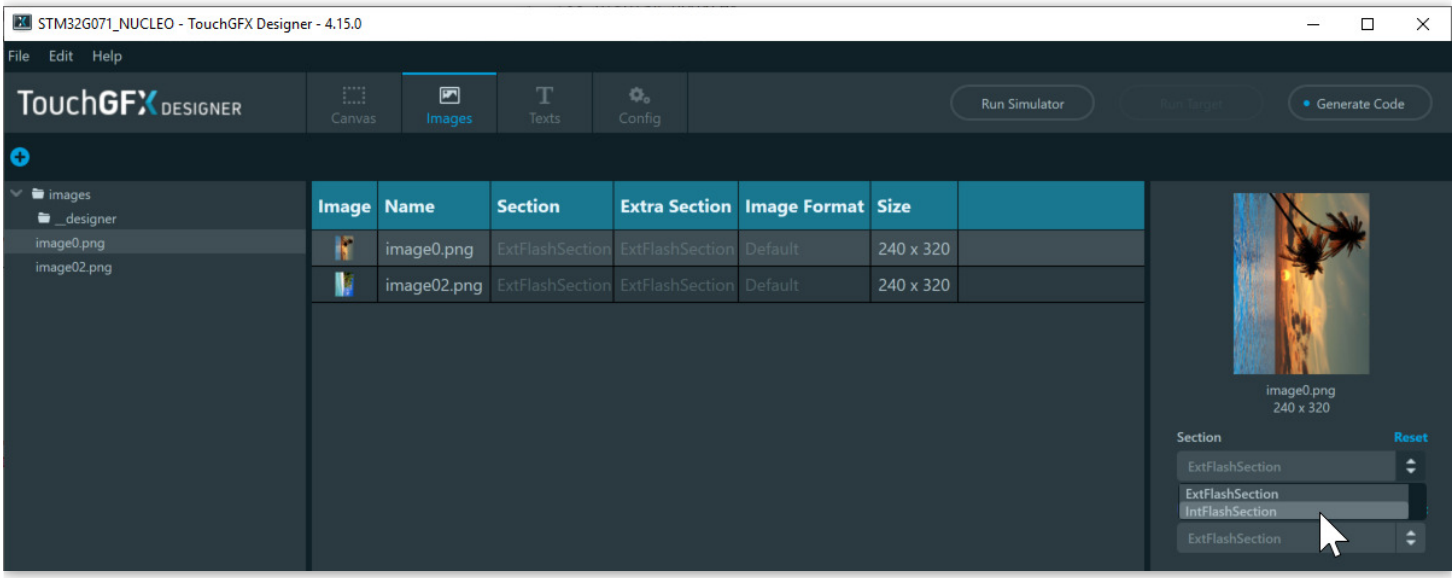
TouchGFX on STM32G0 with serial flash has a few limitations that the application programmer must be aware of.

Texture Mapper

The texture mapper widgets (TextureMapper, AnimationTextureMapper, ScalableImage) can not draw an image that is stored in the external SPI flash. The reason is that it is not possible to get an acceptable performance of e.g. image rotation with an image stored in a serial flash.

To use an image with a texture mapper you must store the image in internal flash or RAM. An image is easily stored in internal flash by modifying the image configuration in TouchGFX Designer.

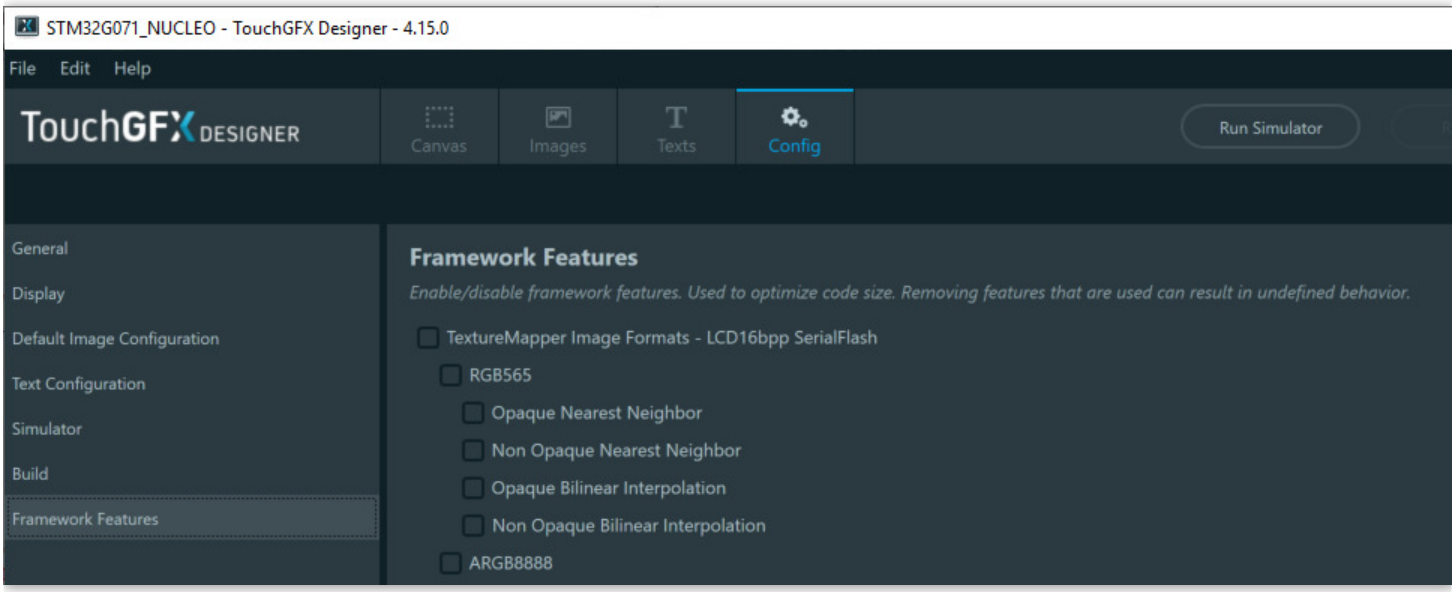
Go to the Images tab and select the image. In the right side of the window, change the "Section" attribute to "IntFlashSection".



Placing an image in internal flash

The texture mapper code is too large to include in all projects. It is therefore disabled pr. default for STM32G0 projects. This means that you must enable the texture mapper before you can use it in your STM32G0 projects.

Go to the "Config" tab, select "Framework Features", and click the relevant texture mapper or a group of texture mappers.



Enabling a texture mapper for a specific image format

It is also possible to temporarily move an image to RAM using the Bitmap Cache

Bitmap Painter

Line, Circle, and DynamicGraph widgets can be colored with an image. This is not possible with images located in the SPI flash. Images used with these widgets must be placed in internal flash or RAM.

L8 Palette

Images in L8 format can be used on hardware with a serial flash. The limitation is that the palette data must be in the internal flash (also for performance reasons).

The palette can be moved to internal flash by changing the "Extra Section" to "IntFlashSection" in the TouchGFX Designer.

Drivers

The application template is created using the TouchGFX Generator. Read more about the Generator here. The TouchGFX Generator generates a HAL layer that links the TouchGFX framework with a set of low-level drivers (already implemented in this application template). The low-level drivers for this application template are located in the `Core/Src` folder in your project.

The drivers are in 3 files:

Driver	File
Display	Core/Src/MB1642BDisplayDriver.c

Driver	File
Flash	Core/Src/MB1642BDataReader.c
Buttons	Core/Src/MB1642BButtonController.cpp

Display

The display uses a fairly standard SPI protocol. A number of registers is the diplay can be written to configure the display. The chip select is asserted when data is transmitted to the display. An extra GPIO, DCX, is used to differentiate command bytes from data bytes.

The driver uses a DMA channel to send display pixel data. This allows the transmission to run while the MCU is calculating pixels. An DMA complete interrupt is used to free the memory transmitted for reuse in a future drawing and to restart the transmission if new data is already available.

Configuration data is not send with DMA, because the CS and CDX pins must be toggle between and in the small configuration packages.

The driver uses the SPI in 8 bit mode when sending configuration data, but changes to 16-bit mode when transmitting the pixel data. The reason for this is that the MCU memory is read in little endian mode. A pixel in RGB565 format is stored in RAM with the low byte (G and B) first and the high byte (R and G) second. This order is preserved when the 8-bit SPI is reading the memory for transmission. When the SPI is in 16-bit mode, the data is read as 16-bit RGB565 from memory and transmitted in correct order for the display.

A driver not using 16-bit DMA must swap the bytes in a pixel before transmitting.

Initialisation

The display initialisation is found in the function `MB1642BDisplayDriver_DisplayInit(void)`

The driver sends 6 commands to the display which follows the recommended power on sequence:

1. Exit Sleep Mode (11h)
2. Enter Normal Mode (13h)
3. Set Memory Access Control (36h) with MX and BGR bits set
4. Set Pixel Format (3Ah) with format 16 bits
5. Tearing Effect Line On (35h)
6. Set Tear Scanline (44h) with line = 0

The driver sleeps for 100 ms between these command.

Tearing Effect

The Tearing Effect (TE) signal from the display is very important. It allows the application to synchronize correctly with the display. This helps the application to avoid tearing on the display. The display asserts the signal whenever it starts an update cycle. The MCU uses this signal to also sending data to the display.

The TE signal is connected to the external interrupt input of the MCU. CubeMx generates and configures an interrupt on this pin.

The callback in the driver signal TouchGFX to start drawing:

MB1642BDisplayDriver.c

```
void HAL_GPIO_EXTI_Rising_Callback(uint16_t GPIO_Pin)
{
  ...
  touchgfxSignalVSync();
}
```

External flash

The SPI NOR flash on the display shield is a standard SPI flash. The driver is simpler than the display driver. No special initialisation is required to read data from the flash.

The driver can read data using polled SPI (busy waiting for each byte) or DMA. The time to start a DMA reception is similar to the time it takes to read 20 bytes in polled mode, so it is often slower for short reads. On the other hand, the DMA continues to run during interrupt and can run in the background when the MCU is busy rendering pixels. For this reason both methods are used.

The flash driver is using another DMA channel than the display driver, so both reception of data and transmission of pixels can run concurrently.

Linker Script

The linker controls where the various data in the application is located. This is specified in the linker script. Here is the first part of the linker script for the gcc compiler:

```
MEMORY
{
  RAM        (xrw)    : ORIGIN = 0x20000000,   LENGTH = 36K
  FLASH      (rx)     : ORIGIN = 0x8000000,    LENGTH = 128K
  SPI_FLASH  (r)      : ORIGIN = 0x90000000,   LENGTH = 8M
}
```

It declares the NOR flash as starting from the address 0x90000000. This value is arbitrarily chosen but required by the flash loader.

This next section puts the image (ExtFlashSection) and font (FontFlashSection) data in the SPI flash.

```
ExtFlashSection :
{
  *(ExtFlashSection ExtFlashSection.*)
  *(.gnu.linkonce.r.*)
  . = ALIGN(0x4);
} >SPI_FLASH

FontFlashSection :
{
  *(FontFlashSection FontFlashSection.*)
  *(.gnu.linkonce.r.*)
  . = ALIGN(0x4);
} >SPI_FLASH
```

Other data can be put into the SPI flash by adding similar sections to the linker script.

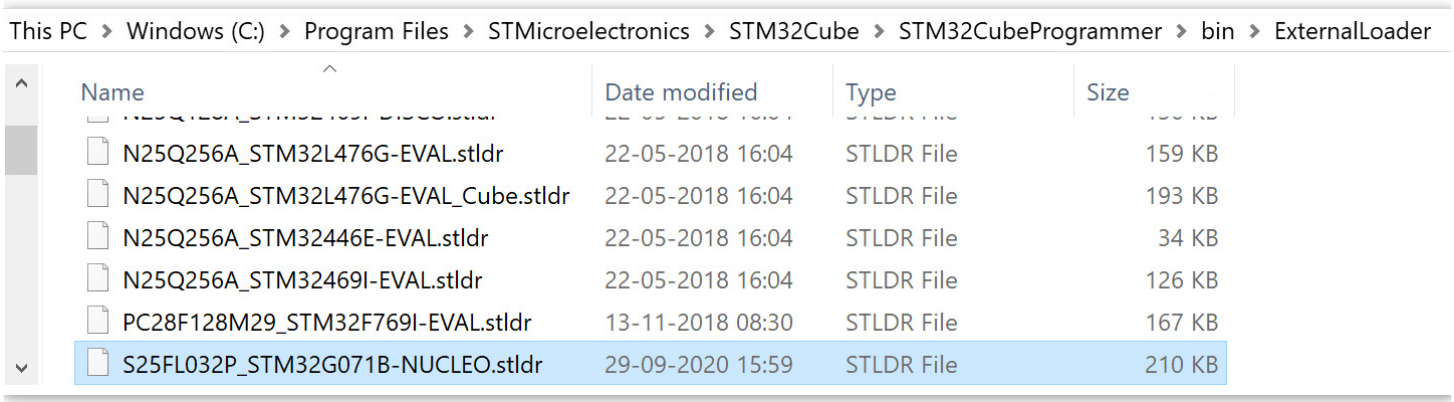
Flash Loader

The G071 application template contains a flash loader for STM32CubeProgrammer. This flash loader can write data to the SPI NOR flash.

The flash loader is found in the file `gcc/S25FL032P_STM32G071B-NUCLEO.stldr`

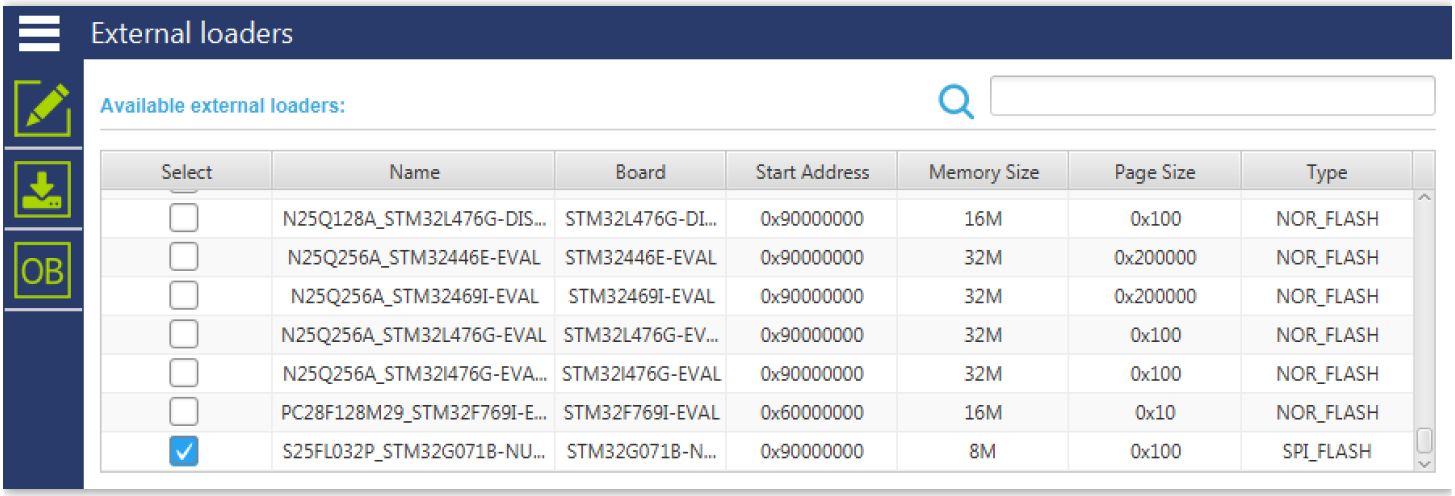
A STM32CubeIde project can be flashed directly from the IDE, but an IAR or Keil application must be flashed from STM32CubeProgrammer.

The flashloader is not available in STM32CubeProgrammer initially, so it must be installed by copying the stldr to the installation:



Copy flash loader to STM32CubeProgrammer installation

Now the flashloader can be selected in STM32CubeProgrammer as normal:



Copy flash loader to STM32CubeProgrammer installation

TIP

The flash loader is only working with the specific GPIO configuration that is used on the Nucleo-G071RB board.
If a different GPIO configuration for the serial flash is used on custom hardware, the flash loader must be modified similarly.

Buttons

The button driver is very simple. It samples the state of the 5 GPIOs used for the joystick on MB1642B and the blue user button on the Nucleo board.

This button driver is installed as BottonController in TouchGFX. This means that the button presses are available directly in the TouchGFX Designer to use in interactions. They can also be used in user code like this:

```
void Screen1View::handleKeyEvent(uint8_t key)
{
    if (key == '6')
    {
        application().gotoScreen2Screen();
    }
}
```

The key codes used are:

Key	Code
Left	'4'

Key	Code
Right	'6'
Up	'8'
Down	'2'
Center	'5'
Blue User Button	'0'

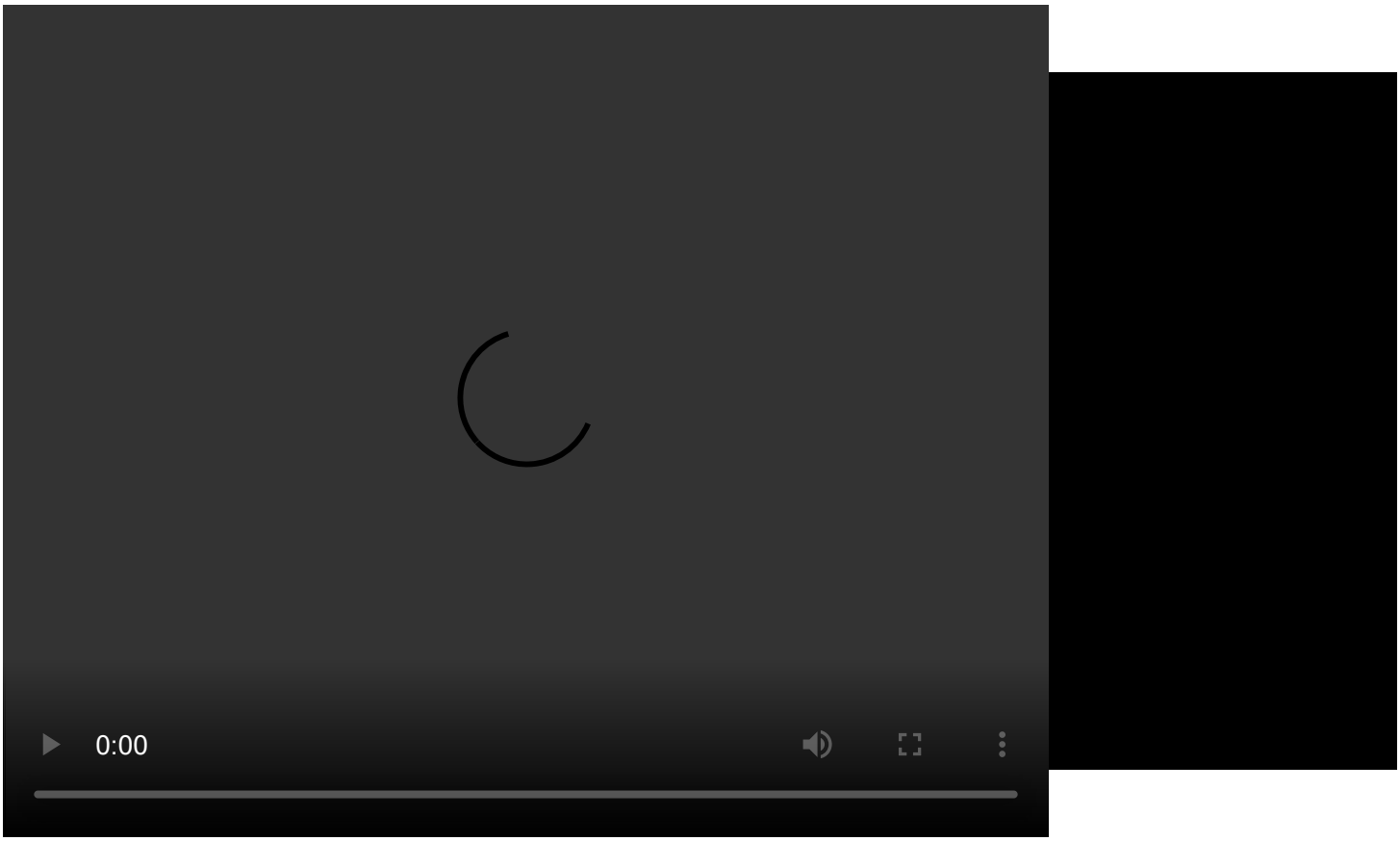
These keys are also available in the Simulator bye using the keyboard numpad.

Version: 4.16

Lowering Memory Usage with Partial Framebuffer

This section explains, by exemplifying with a clock application, how to configure and use Partial Frame Buffers, to lower memory requirements at the expense of some performance.

A video of the application running on the STM32L4R9Discovery evaluation kit can be seen below



Full-size Frame Buffer Memory

Normally, your frame buffer is a big memory array with enough memory to hold all the pixels available on your display. If you are running on a 24-bit display with a resolution of 480 x 272, a full-size frame buffer holds 480 x 272 x 3 bytes = 391.680 bytes.

Some applications may have 2- ("Double buffering") or even 3 frame buffers. The total memory requirement in these cases would then be 783.360 and 1.175.040 bytes.

TouchGFX writes pixel values to the frame buffer when drawing any part of the UI, after all drawing operations have completed, the frame buffer is transferred to the display. Typically, the whole frame

buffer is transferred to the display even if only a part of the UI is updated. Generally, the framebuffer can be updated in many small blocks before itis transferred.

Update1, Update 2, Update 3, ..., Update N, Transfer to display

In some cases, particularly in low cost solutions with no external RAM, frame buffers are required to be small enough to allow the rest of the application to fit in the internal RAM together with the framebuffer. This is where partial frame buffers are useful.

Partial Frame Buffer Memory

Partial frame buffers allows a TouchGFX application to run on top of a few, less than full-size frame buffers. The number and size of the frame buffers are configurable. This technique can lower the memory requirements of an application by a substantial amount, but comes with some limitations:

- Partial frame buffers will only work on displays that have built-in memory. These are typically DSI displays or displays with a parallel bus connection (DBI type A/B, 8080/6800) or SPI-bus connection.
- Potential tearing for complex applications.

Rather than using a frame buffer representing every pixel on the display, partial frame buffers typically cover a smaller part. In the clock example used in this article, three frame buffers of 11.700 bytes each are used. This results in a memory footprint for frame buffers of 35.100 bytes.

Whenever the application needs to update a part of the UI, TouchGFX will select one of the configured, partial frame buffers, complete its drawing operation in the partial framebuffer, and transfer that part to the display. This is repeated for all areas of the UI that need to be rendered - This changes the formula for updating and transferring data to:

Update1, Transfer1, Update2, Transfer2, Update3, Transfer3, ..., UpdateN, TransferN

In some cases the transfer of one partial frame buffer can run while the update of the next buffer is running.

Display Tearing

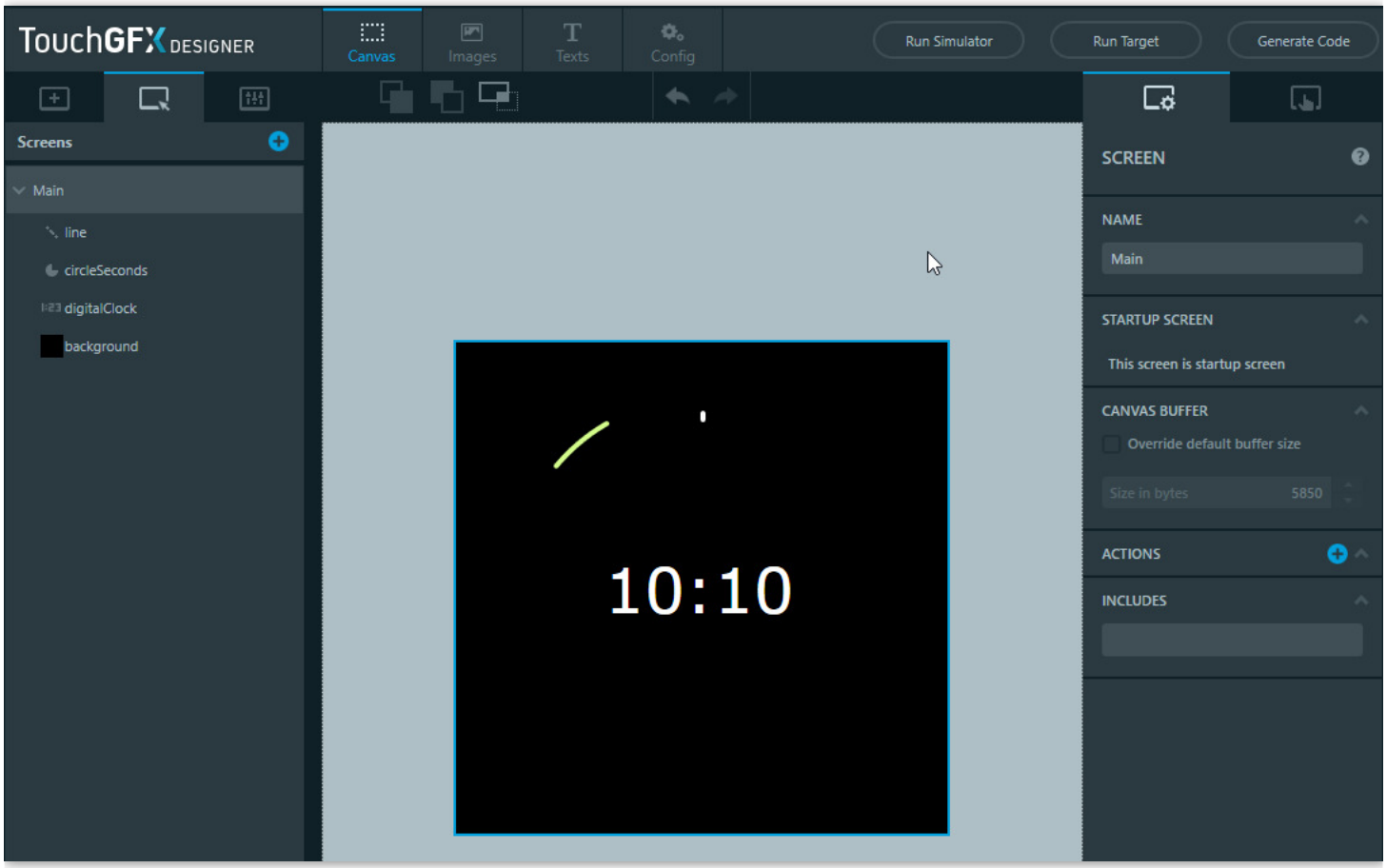
Contrary to using full-size frame buffers, TouchGFX will transfer parts of the UI as soon as they are updated, when using partial frame buffers. The display will show the received updates on its glass after at most 16 ms (for 60 fps displays) because the display needs to be refreshed regularly. Because of this, the first updates to the display can potentially be visible to the user before all updates have been transferred.

If the total sequence of draw operations and transfers take a long time to complete (> 16 ms) it is highly possible that the user will see a combination of the previous frame and some of the new updates. This is called display tearing and is not desirable. For this reason, partial frame buffers are not suitable for applications that make use of complex animations that take a long time to render.

Display Update Example

Before we get into how to configure partial frame buffers in your application let's have a look at a concrete example showing a digital clock with a moving circle arc representing seconds. The green circle arc is moving 6 degrees each second and does a full rotation in a minute. The UI is built from four Widgets as seen in the image below:

- Line
- Circle
- Digital Clock
- Box



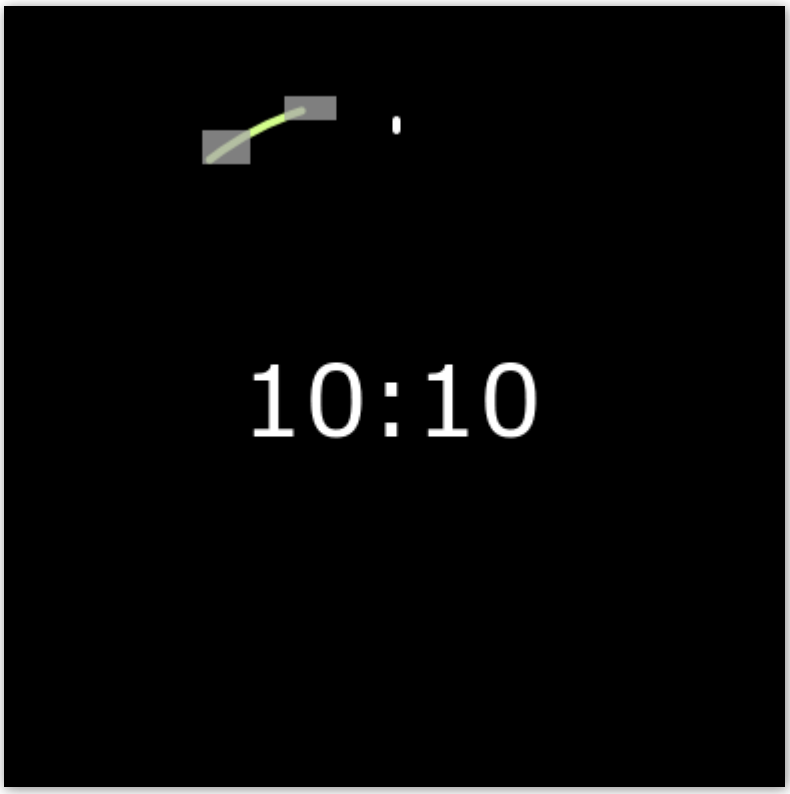
Here is the code that updates the digital clock and circle arc:

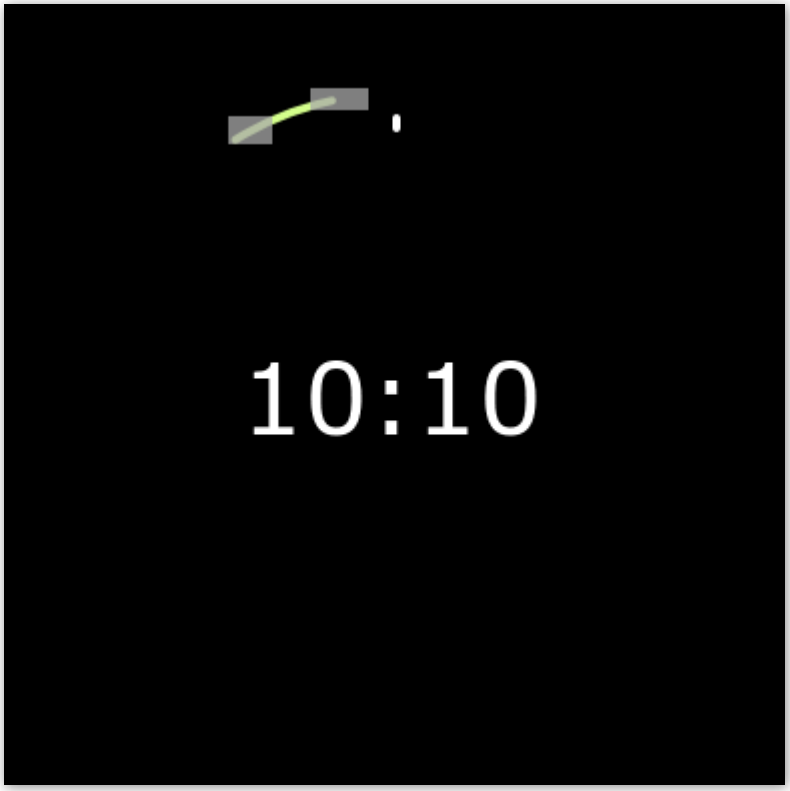
MainView.cpp

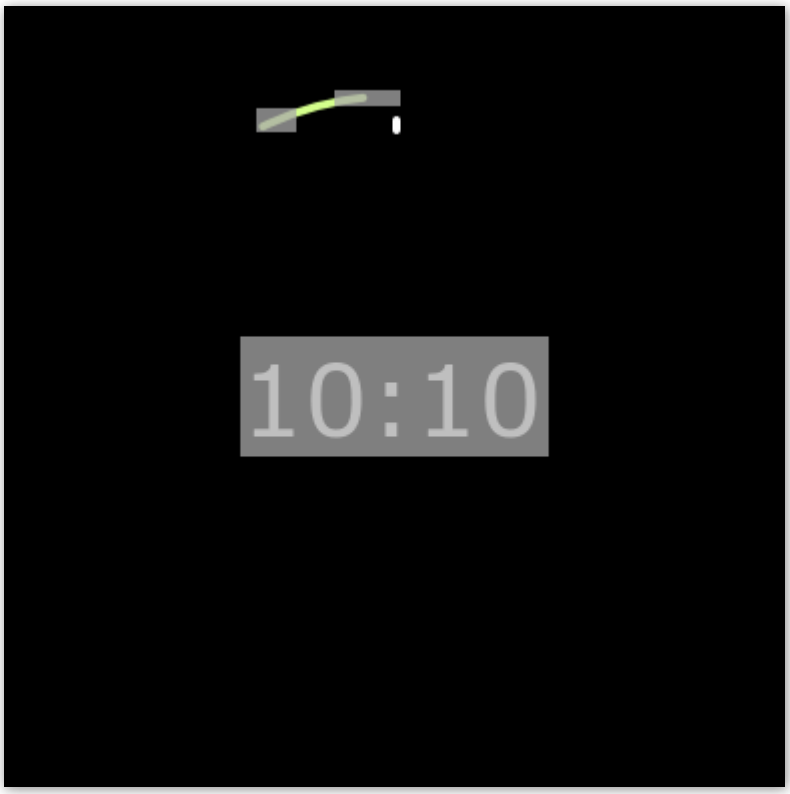
```
void MainView::handleTickEvent()
```

```
{
  ticks++;
  if (ticks == 10)
  {
      ticks = 0;
      secs += 1;
      if (secs == 60)  //increment minutes
      {
        secs = 0;
        min += 1;
        if (min == 60) //increment hours
        {
          min = 0;
          hour += 1;
          if (hour == 24)
          {
            hour = 0;
          }
        }
        //Only update digital clock when minutes or hours change
        digitalClock.setTime24Hour(hour, min, secs);
      }
    //Always update seconds
    circleSeconds.updateArc(secs*6 - 20, secs*6);
  }
}
```

The following images shows the areas that are updated in the first few seconds when the circle arc approaches the top and digital clock is updated (the grey rectangles). In the first two frames, only the seconds are changing (58 and 59 seconds). In the thirs the seconds reaches 60 and the hour and minutes text is updated:


10:10


10:10



The rectangles updated in the third image above are 154 x 60 pixels, 20 x 12 pixels, and 33 x 8 pixels. When using standard frame buffers these three rectangles would be drawn into the full frame buffer (overwriting the previous pixels), which would afterwards be transferred to the display. When using partial frame buffers, these three rectangles would be drawn into their own little frame buffers which would then immediately be transferred to the display and shown.

Configuring Partial Frame Buffers

There are two steps to configuring TouchGFX for partial frame buffers: Creating a frame buffer allocator object with a memory buffer, and configuring the TouchGFX HAL class to use it. Later we also need to write code to transmit the buffers to the display. The first two steps are typically done in the BoardConfiguration.cpp file.

Creating a frame buffer allocator as a global variable:

BoardConfiguration.cpp

```
//2 or more blocks of 10*390 pixels, one pixel is 3 bytes
ManyBlockAllocator<10*390*3, 2, 3> frameBufferAllocator;
```

This frame buffer allocator allocates 2 blocks of each 10 x 390 x 3 bytes = 11.700 bytes.

Configure HAL to use it:

BoardConfiguration.cpp

```
void touchgfx_init()
{
    HAL& hal = touchgfx_generic_init(dma, display, tc, GUI_DISPLAY_WIDTH,
                                     GUI_DISPLAY_HEIGHT, 0, 0, 0);
    hal.setFrameBufferStartAddress((uint16_t*)0, GUI_DISPLAY_BPP, false, false);
    hal.setFrameBufferAllocator(&frameBufferAllocator);
    hal.setFrameRefreshStrategy(HAL::REFRESH_STRATEGY_PARTIAL_FRAMEBUFFER);
   ...
```

With this configuration TouchGFX will allocate small frame buffers and draw the UI in them. What is left now, is to transfer the small frame buffers to the display.

Lets first see the position and size of the two frame buffers allocated to draw the small circle updates (second image above):

Rectangle	x	y	width	height	Pixels
Rectangle 1	112	56	22	14	308 pixels = 924 bytes
Rectangle 2	153	42	29	11	319 pixels = 957 bytes

Both these rectangles are so small, they can fit into the blocks allocated by the frame buffer allocator.

In the third image above, we have 3 updated rectangles: The small updates to the circle, and the larger rectangle covering the text:

Rectangle	x	y	width	height	Pixels
Rectangle 1	126	51	20	12	240 pixels = 720 bytes
Rectangle 2	165	42	33	8	264 pixels = 792 bytes
Rectangle 3	118	165	154	60	9.240 pixels = 27.720 bytes

Again, the rectangle 1 and 2 are so small, they can fit into the blocks allocated by the frame buffer allocator, but frame buffer 3 is too large. This rectangle is to large and will be split into multiple rectangles that each can fit into the frame buffers (11.700 bytes).

Here we are updating 3 rectangles, but the allocator only has 2 blocks. In that situation, TouchGFX will wait for the first blocks to transferred and then reuse the blocks.

Transferring Frame Buffers to the Screen

TouchGFX will allocate a frame buffer from the FrameBufferAllocator, when a rectangle needs to be redrawn. After drawing to the buffer TouchGFX will call this method:

```
void HAL::flushFrameBuffer(const Rect& rect);
```

This function can be overridden in a HAL subclass to transfer the frame buffer to the screen. This special implementation is required for partial framebuffers to work. The following sections will illustrate how to configure this for the STM32G081 and STM32G071 evaluation kits with a SPI displays, and the STM32L4R9Discovery evaluation kit which has a DSI display.

Transferring Frame Buffers to the STM32G081 SPI Display

The STM32G081 evaluation kit has a SPI display. The basic principle is to start a DMA transfer to the display as soon as a block is drawn or when a transfer is completed if a new block is ready to be transferred.

First, when a rectangle is drawn, we start a transfer if none is already in progress:

STM32G0HAL.cpp

```
void STM32G0HAL::flushFrameBuffer(const touchgfx::Rect& rect)
{
    HAL::flushFrameBuffer(rect);
    frameBufferAllocator->markBlockReadyForTransfer();
    //start transfer if not running already!
```

```
    if (!LCDManager_IsTransmittingData())
    {
        touchgfx::Rect r;
        const uint8_t* pixels = frameBufferAllocator->getBlockForTransfer(r);
        LCDManager_SendFrameBufferBlockWithPosition((uint8_t*)pixels, r.x, r.y, r.width, r
    }
}
```

The function *LCDManager_SendFrameBufferBlockWithPosition* starts a SPI transfer to the display using DMA. This function is highly dependent on the display and the GPIO configuration. It must be developed by the application programmer. The STM32G0 CubeFW HAL function *HAL_SPI_Transmit_DMA* is used to start the DMA.

The SPI transfer complete interrupt handler calls a function when the transfer is complete:

STM32G0HAL.cpp

```
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi)
{
    UNUSED(hspi);
    LCD_CS_HIGH();
    isTransmittingData = 0;

    //Change to SPI datasize to 8 bit from 16 bit
    heval_Spi.Instance->CR2 &= ~(SPI_DATASIZE_16BIT - SPI_DATASIZE_8BIT);

    //signal transfer complete
    LCDManager_TransferComplete();
}
```

The LCDManager_TransferComplete functions starts a new transfer. An important piece here is to call *freeBlockAfterTransfer*. This will allow TouchGFX to reuse the just transmitted block for a new drawing.

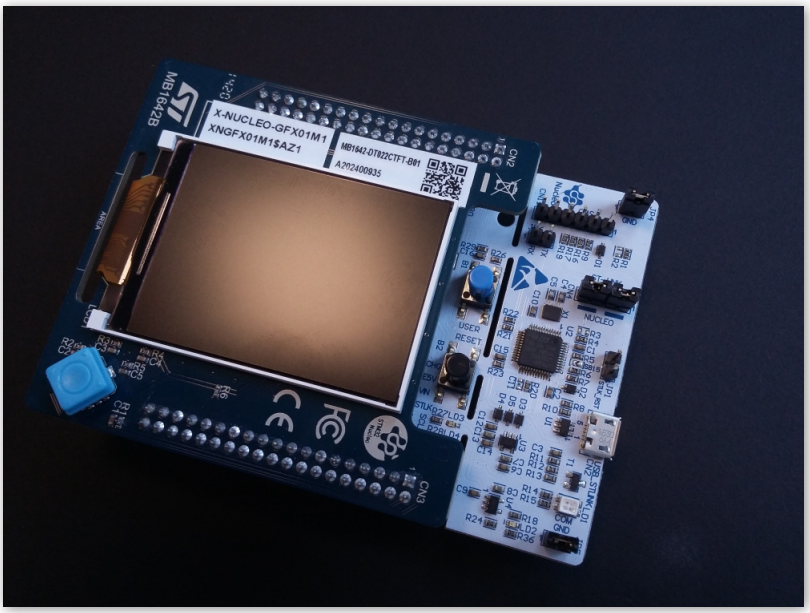
STM32G0HAL.cpp

```
void LCDManager_TransferComplete()
{
    touchgfx::startNewTransfer();
}

void startNewTransfer()
{
    FrameBufferAllocator* fba = HAL::getInstance()->getFrameBufferAllocator();
    fba->freeBlockAfterTransfer();
    blockIsTransferred = true;

    if (fba->hasBlockReadyForTransfer())
    {
```

```
        touchgfx::Rect r;
        const uint8_t* pixels = fba->getBlockForTransfer(r);
        LCDManager_SendFrameBufferBlockWithPosition((uint8_t*)pixels, r.x, r.y, r.width, r
    }
}
```

Transferring Frame Buffers to the X-NUCLEO-GFX01M1 SPI Display

In this section we will discuss the application template for the STM32G071 nucleo board with the X-Nucleo-GFX01M1 expansion board. This expansion board, MB1642B, contains a 2.2" 240x320 SPI display and a 64-Mbit SPI NOR flash.



Nucleo-G071RB with X-Nucleo-GFX01M1 expansion board

In this application template we use a C++ class from the framework to help managing the partial framebuffer blocks. This makes the code in the application template a little shorter.

The application template is build with the TouchGFX Generator. Read more about that here

The most important part is the flushFrameBuffer function:

TouchGFXGeneratedHAL.cpp

```
void TouchGFXGeneratedHAL::flushFrameBuffer(const touchgfx::Rect& rect)
{
    HAL::flushFrameBuffer(rect);
    // Try transmitting a block
    PartialFrameBufferManager::tryTransmitBlock();
}
```

Here we just call the *PartialFrameBufferManager* framework class to get the block transmitted.

In the TouchGFXGeneratedHAL::endFrame function we call *PartialFrameBufferManager* to get any remaining framebuffer blocks transmitted also:

TouchGFXGeneratedHAL.cpp

```
void TouchGFXGeneratedHAL::endFrame()
{
    // We must guard the next frame until we're done transferring all blocks over our disp
    // through either a semaphore if user is running an OS or a simple variable if not
    PartialFrameBufferManager::transmitRemainingBlocks();

    HAL::endFrame();
    touchgfx::OSWrappers::signalRenderingDone();
}
```

The *PartialFrameBufferManager* uses three functions to interact with the display driver code. These must be implemented in the Application Template:

TouchGFXGeneratedHAL.cpp

```
/**
 * Check if a Frame Buffer Block is beeing transmitted.
 */
__weak int transmitActive()
{
    return touchgfxDisplayDriverTransmitActive();
}

/**
 * Check if a Frame Buffer Block ending at bottom may be sent.
 */
__weak int shouldTransferBlock(uint16_t bottom)
{
    return touchgfxDisplayDriverShouldTransferBlock(bottom);
}

/**
 * Transmit a Frame Buffer Block.
 */
__weak void transmitBlock(const uint8_t* pixels, uint16_t x, uint16_t y, uint16_t w, uint1
{
    touchgfxDisplayDriverTransmitBlock(pixels, x, y, w, h);
}
```

The code above just forwards the calls to C functions in the MB1642B driver code.

MB1642BDisplayDriver.c

```
int touchgfxDisplayDriverTransmitActive(void)
{
  return IsTransmittingBlock_;
}

void touchgfxDisplayDriverTransmitBlock(const uint8_t* pixels, uint16_t x, uint16_t y, uir
{
  Display_Bitmap((uint16_t*)pixels, x, y, w, h);
}
```

The implementation of this driver code depends highly on the display used. For the MB1642B module this involves two GPIO to control SPI chip select and data/command mode.

The transmission state is implemented using a volatile uint8*t variable *IsTransmittingBlock*. This variable is set to 1 when a transmission is started and set to zero in the DMA callback:

MB1642BDisplayDriver.c

```
void MB1642BDisplayDriver_DMACallback(void)
{
  /* Transfer Complete Interrupt management ***********************************/
  if ((0U != (DMA1->ISR & (DMA_FLAG_TC1))) && (0U != (hdma_spi1_tx.Instance->CCR & DMA_IT_
  {
    /* Disable the transfer complete and error interrupt */
    __HAL_DMA_DISABLE_IT(&hdma_spi1_tx, DMA_IT_TE | DMA_IT_TC);

    /* Clear the transfer complete flag */
    __HAL_DMA_CLEAR_FLAG(&hdma_spi1_tx, DMA_FLAG_TC1);

    IsTransmittingBlock_ = 0;

    ...

    // Signal Transfer Complete to TouchGFX
    DisplayDriver_TransferCompleteCallback();
```

As we see above, the DMA callback also calls the transfer complete callback. This function is implemented in the generated HAL:

TouchGFXGeneratedHAL.cpp

```
extern "C"
void DisplayDriver_TransferCompleteCallback()
{
    // After completed transmission start new transfer if blocks are ready.
    PartialFrameBufferManager::tryTransmitBlockFromIRQ();
```

```
}
```

The call to the *PartialFrameBufferManager* here makes it start a new transfer if possible.

Transferring Frame Buffers on DSI Display

The STM32L4R9Discovery evaluation kit uses a DSI display. The normal HAL class is called STM32HAL_DSI (located in STM32HAL_DSI.cpp).

We override the HAL::flushFrameBuffer method to notify the FrameBufferAllocator that a block has been drawn:

STM32HAL_DSI.hpp

```
void STM32HAL_DSI::flushFrameBuffer(const Rect& rect)
{
    frameBufferAllocator->markBlockReadyForTransfer();
    HAL::flushFrameBuffer(rect); //call normal implementation
}
```

The FrameBufferAllocator subclass ManyBlockAllocator will call the global function FrameBufferAllocatorSignalBlockDrawn() when a block is ready for transfer. This method must be implemented in the BSP layer:

BoardConfiguration.cpp

```
void FrameBufferAllocatorSignalBlockDrawn()
  {
    if (!dsiIsTransferring)
    {
      sendBlock();
    }
  }
```

This function is calling the sendBlock function, unless a transfer is already ongoing on the DSI. For the first block drawn by TouchGFX, this will never be the case, so a transfer is started. If another block drawing is completed while the DSI transfer is still running, the block will be kept in the "ready to transfer state", and drawing will continue in another free block (if available).

When a DSI transfer is completed, we must first free the transferred block, so it can be reused for another rectangle, and then check to see if the next block is ready for transfer. This is all done in the ER interrupt:

BoardConfiguration.cpp

```
__irq void DSI_IRQHandler(void) {
  if (__HAL_DSI_GET_FLAG(&hdsi, DSI_IT_ER))
  {
    // End-of-refresh interrupt. Meaning last DSI transfer is complete
    __HAL_DSI_CLEAR_FLAG(&hdsi, DSI_IT_ER);
    if (dsiIsTransferring)
    {
      HAL::getInstance()->getFrameBufferAllocator()->freeBlockAfterTransfer();
      dsiIsTransferring = 0;
    }
    sendBlock(); //transfer next block if availble
  }
```

The function sendBlock is more complicated. Here we configure the LTDC and DSI peripherals to transfer the framebuffer. We also configure the display to put the transferred data into the correct place in the display memory. This part of the code is dependent on the specific display. Check the display datasheet for the command specifications.

BoardConfiguration.cpp

```
static void sendBlock()
{
    FrameBufferAllocator* fbAllocator = HAL::getInstance()->getFrameBufferAllocator();

    //Is a block ready for transfer?
    if (fbAllocator->hasBlockReadyForTransfer())
    {
        Rect transfer_rect;
        const uint8_t* src = fbAllocator->getBlockForTransfer(transfer_rect);
        dsiIsTransferring = 1;

        //1. Setup LTDC and layer address and dimension
        //2. Configure display active area
        //3. Start DSI

        __HAL_DSI_WRAPPER_DISABLE(&hdsi);

        //1: Setup LTDC
        LTDC_Layer1->CFBAR = (uint32_t)src;

        const uint32_t width = transfer_rect.width;
        const uint32_t height = transfer_rect.height;

        LTDC->AWCR = ((width + 1) << 16) | (height + 1);
        LTDC->TWCR = ((width + 1 + 1) << 16) | (height + 1 + 1);

        const uint16_t layer_x0 = 2 + 0;
        const uint16_t layer_x1 = 2 + width - 1;
        LTDC_Layer1->WHPCR = (layer_x1 << 16) | layer_x0;
```

```
        const uint16_t layer_y0 = 2 + 0;
        const uint16_t layer_y1 = 2 + height - 1;
        LTDC_Layer1->WVPCR = (layer_y1 << 16) | layer_y0;

        LTDC_Layer1->CFBLR = ((width * 3) << 16) | (width * 3 + 3);
        LTDC_Layer1->CFBLNR = height;

        LTDC->SRCR = (uint32_t)LTDC_SRCR_IMR;

        //2: Configure display
        const int16_t x = transfer_rect.x + 4;
        const int16_t x2 = transfer_rect.x + 4 + width - 1;
        uint8_t InitParam1[4] = { (uint8_t)(x >> 8), (uint8_t)(x & 0xFF), (uint8_t)(x2 >>
        HAL_DSI_LongWrite(&hdsi, 0, DSI_DCS_LONG_PKT_WRITE, 4, DSI_SET_COLUMN_ADDRESS, In

        const int16_t y = transfer_rect.y;
        const int16_t y2 = transfer_rect.y + height - 1;
        uint8_t InitParam2[4] = { (uint8_t)(y >> 8), (uint8_t)(y & 0xFF), (uint8_t)(y2 >>
        HAL_DSI_LongWrite(&hdsi, 0, DSI_DCS_LONG_PKT_WRITE, 4, DSI_SET_PAGE_ADDRESS, InitP

        //3: Start DSI transfer
        __HAL_DSI_WRAPPER_ENABLE(&hdsi);
        HAL_DSI_Refresh(&hdsi);
    }
}
```

Transferring Frame Buffers on SPI Display

The STM32G081 evaluation kit has a SPI display. The principle for transferring the rectangles to the display is the same as for the DSI, but some details are different.

First, when a rectangle is drawn, we start a transfer if none is already in progress:

STM32G0HAL.cpp

```
void STM32G0HAL::flushFrameBuffer(const touchgfx::Rect& rect)
{
    HAL::flushFrameBuffer(rect);
    frameBufferAllocator->markBlockReadyForTransfer();
    //start transfer if not running already!
    if (!LCDManager_IsTransmittingData())
    {
        touchgfx::Rect r;
        const uint8_t* pixels = frameBufferAllocator->getBlockForTransfer(r);
        LCDManager_SendFrameBufferBlockWithPosition((uint8_t*)pixels, r.x, r.y, r.width, r
    }
}
```

The function *LCDManager_SendFrameBufferBlockWithPosition* starts a SPI transfer to the display using DMA.

The SPI transfer complete handler calls a function when the transfer is complete:

STM32G0HAL.cpp

```
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi)
{
    UNUSED(hspi);
    LCD_CS_HIGH();
    isTransmittingData = 0;

    //Change to SPI datasize to 8 bit from 16 bit
    heval_Spi.Instance->CR2 &= ~(SPI_DATASIZE_16BIT - SPI_DATASIZE_8BIT);

    //signal transfer complete
    LCDManager_TransferComplete();
}
```

The LCDManager_TransferComplete functions starts a new transfer:

STM32G0HAL.cpp

```
void LCDManager_TransferComplete()
{
    touchgfx::startNewTransfer();
}

void startNewTransfer()
{
    FrameBufferAllocator* fba = HAL::getInstance()->getFrameBufferAllocator();
    fba->freeBlockAfterTransfer();
    blockIsTransferred = true;

    if (fba->hasBlockReadyForTransfer())
    {
        touchgfx::Rect r;
        const uint8_t* pixels = fba->getBlockForTransfer(r);
        LCDManager_SendFrameBufferBlockWithPosition((uint8_t*)pixels, r.x, r.y, r.width, r
    }
}
```

Conclusion

In this article we saw how the partial frame buffer strategy can help lowering the memory requirements for platforms that have displays with integrated frame buffer memory.

The method for configuring and setting up partial framebuffers is the same across all platforms, but the method of sending the content of the blocks to the display varies. We saw how, for an LTDC/DSI based platform (STM32L4R9-DISCO) we were able to reconfigure the LTDC Layer to fit the next block ready for transfer on DSI, while on a platform with no LCD controller (STM32G081) we were able to send the blocks to the display using SPI.

Version: 4.16

Using Non-Memory Mapped Flash for Storing Images

In this section we will discuss how to link all your images to a binary file that you can put in a non-memory mapped flash and how to use that file at runtime together with the bitmap cache. TouchGFX cannot draw bitmaps that are stored in non-memory mapped flash, but by caching the bitmaps in RAM we can make the bitmaps useable in the application.

See the article Caching Bitmaps for a general discussion on the bitmap cache.

In this article we assume that you have setup a bitmap cache, and that you want to store your bitmaps in non-memory mapped flash. This can be e.g. USB storage, NAND flash etc.

The goal is to link the images to a specific address, copy the images to a file, and help TouchGFX to copy from the file to the cache.

Copying bitmap data from flash to cache

Recall that when you cache a bitmap, TouchGFX copies the pixels from the original location to the bitmap cache.

This copying is done by calling this method from the HAL class:

HAL.hpp

```
bool HAL::blockCopy(void* RESTRICT dest, const void* RESTRICT src, uint32_t numBytes);
```

If your bitmaps are stored in normal addressable flash (like internal flash or memory mapped external flash), then the normal blockCopy function provided in the TouchGFX library is fine, and you do not need to do anything.

On the other hand, if your bitmaps is stored in storage that is not addressable, e.g. a filesystem, then the normal implementation is not sufficient and you need to provide an updated version that is able to read from your specific flash storage.

But first we need to make sure that our bitmaps is linked to a fixed address.

The BitmapDatabase table

All bitmaps in TouchGFX is generated to .cpp files in the folder generated/images/src. Here the bitmaps are represented as byte arrays.

These arrays are compiled by the C++ compiler as any other source code file and are linked into the application.

Here is a screenshot of a simple application with a Button and a TextureMapper showing a rotating logo:

Button and TextureMapper

This application uses 3 images: Button_Pressed, Button_Released, and Logo.

These 3 bitmaps are converted to .cpp files and linked in as part of the application. The images are referenced in a table called the bitmap_database. This table is located in the file BitmapDatabase.cpp. Here is the table from the above example (some details removed):

BitmapDatabase.cpp

```
extern const unsigned char _blue_buttons_round_edge_small[];
extern const unsigned char _blue_buttons_round_edge_small_pressed[];
extern const unsigned char _blue_logo_touchgfx_logo[];

const touchgfx::Bitmap::BitmapData bitmap_database[] =
{
    { _blue_buttons_round_edge_small, ... },
    { _blue_buttons_round_edge_small_pressed, ... },
    { _blue_logo_touchgfx_logo, ... }
};
```

The arrays declared first are the arrays containing the pixels of the individual bitmaps. These arrays are defined in other .cpp files. The bitmap_database array is holding the addresses of these arrays.

TouchGFX uses this array to find the address of the pixels of a bitmap.

When the programmer requests a bitmap to be cached, TouchGFX finds the address of the bitmap in flash (in the bitmap_database array) and copies data from here.

Linker script modifications

The linker selects an address for the bitmaps. This selection is based on the section the bitmaps are placed in. All bitmaps in TouchGFX is by default put into the **ExtFlashSection**.

The standard linker scripts (here for GCC) puts this section into flash together with other read-only data.

In this example we will put the image data in the ExtFlashSection at address **0x24000000**. You can select any address that is otherwise unused (not part of the code or data address space).

First we define a new memory area (USB-flash at address 0x24000000), in addition to the normal internal FLASH and RAM areas:

STM32F746.ld

```
MEMORY
{
  RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 320K
  FLASH (rx)     : ORIGIN = 0x08000000, LENGTH = 1024K
  USB(r)         : ORIGIN = 0x24000000, LENGTH = 1M
}
```

Then we instruct the linker to put the ExtFlashSection in the USB area:

STM32F746.ld

```
ExtFlashSection :
{
   *(ExtFlashSection ExtFlashSection.*)
} >USB
```

After linking we can check the addresses of the bitmaps by inspecting the map file (application.map). Here is the relevant part:

application.map

```
ExtFlashSection
```

```
                0x24000000    0x23ec0
 *(ExtFlashSection ExtFlashSection.*)
 ExtFlashSection
                0x24000000    0x10000 TouchGFX/build/.../Blue_Logo_touchgfx_logo.o
                0x24000000                _blue_logo_touchgfx_logo
 ExtFlashSection
                0x24010000     0x9f60 TouchGFX/build/.../Blue_Buttons_Round_Edge_small.o
                0x24010000                _blue_buttons_round_edge_small
 ExtFlashSection
                0x24019f60     0x9f60 TouchGFX/build/.../Blue_Buttons_Round_Edge_small_pre
                0x24019f60                _blue_buttons_round_edge_small_pressed
```

We can see here that the total size of the images is 0x23ec0 = 147.136 bytes. The 3 arrays holding the bitmaps are located sequentially from address 0x24000000.

Let's now assume the you want the bitmap data to go to a SD-card. We can extract the binary data for the bitmaps from the .elf file with a simple objcopy command:

```
$ arm-none-eabi-objcopy.exe --dump-section ExtFlashSection=images.bin TouchGFX/build/bin/t
$ ls -l images.bin
-rw-r--r-- 1 christef Administrators 147136 Feb 20 11:56 images.bin
```

This gives us a file (images.bin) containing the image byte arrays only. This file can be copied to an USB flash, an SD-card, or even programmed to a flash chip.

The idea is now that when TouchGFX asks for data above address 0x24000000 we take the data from the images.bin file on the SD-card.

Modifying the BlockCopy function

Recall that when you cache a bitmap to RAM TouchGFX calls HAL::BlockCopy to get the data.

To get this linked to the data on your SD-card we can implement a new BlockCopy in your specific HAL class. Here we assume the class is called TouchGFXHAL (as generated by the TouchGFX Generator):

TouchGFXHal.hpp

```
class TouchGFXHAL : public TouchGFXGeneratedHAL
{
public:
    ...
    virtual bool blockCopy(void* RESTRICT dest, const void* RESTRICT src, uint32_t numByte
}
```

TouchGFXHal.cpp

```
// This function is called whenever a bitmap is cached. Must copy a number of bytes from
// to the cache.
bool TouchGFXHAL::blockCopy(void* RESTRICT dest, const void* RESTRICT src, uint32_t numByt
{
  // If requested data is located within the memory block we have assigned for ExtFlashSe
  // provide an implementation for data copying.
  if (src >= 0x24000000 && src < 0x24100000)
  {
    void* dataOffset = src - 0x24000000;
    // In this example we assume graphics is placed in SD card, and that we have an appro
    // for copying data from there.
    sdcard_read(dest, dataOffset, numBytes);
    return true;
  }
  else
  {
    // For all other addresses, just use the default implementation.
    // This is important, as blockCopy is also used for other things in the core framewor
    return HAL::blockCopy(dest, src, numBytes);
  }
}
```

Now you can start caching bitmaps from the SD-card.

If TouchGFX tries to draw a bitmap that is not cached it will try to read the pixels from the address found in the `bitmap_database` table. This address will be in the range 0x24000000 - 0x24100000 and the read will result in an exception.

Linking images to RAM

If your available RAM is big enough to hold all the images (in the above example that is more than 147.136 bytes) then you do not need to use the bitmap cache to copy the images.

The strategy is as follows:

- Select a fixed address and range in RAM for the images
- Remove that range from the RAM area in the linker script
- Create a new area IMAGES with the selected address and size
- Place the ExtFlashSection in IMAGES area
- Link the application and check the .map file
- Create the images.bin file from the application.elf

- Before TouchGFX is started, copy the whole images.bin file from the SD-card to the selected address in RAM

This solution is simple, but has some drawbacks:

- The available RAM must be big enough to hold all the images
- Start up time will be larger because of the copying from the SD-card (megabytes can take seconds)

Version: 4.16

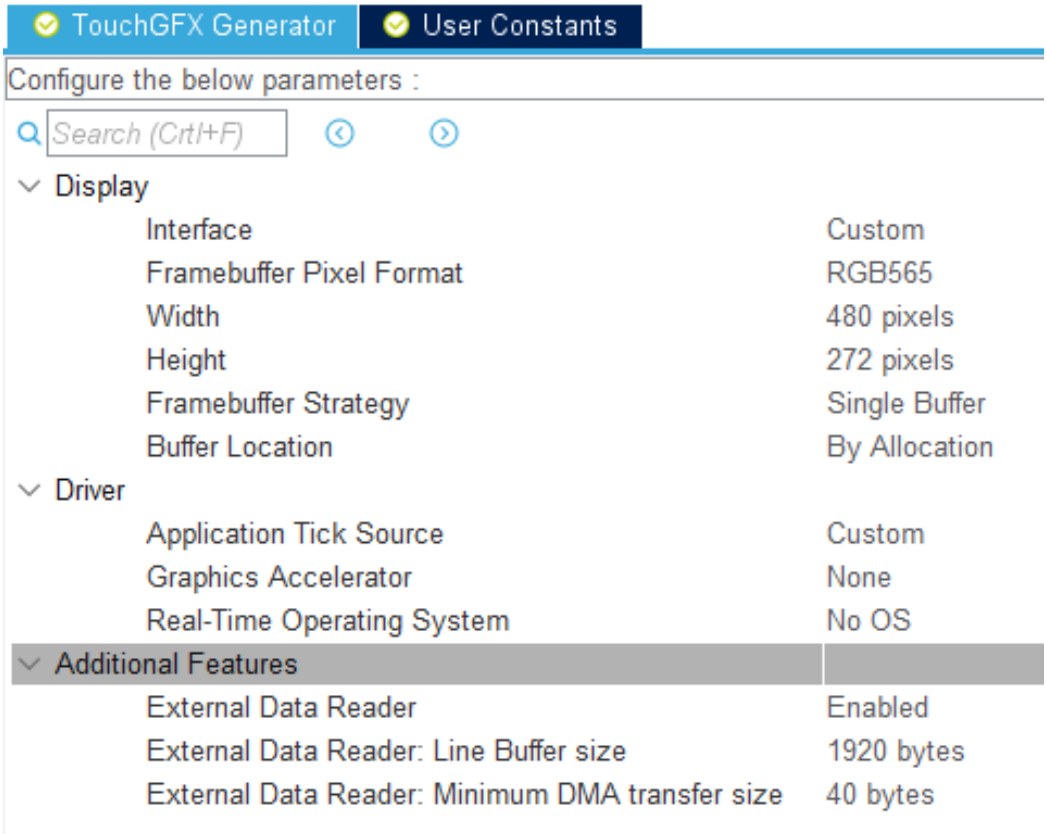
Using Serial Flash for images and fonts

This section discusses how to use a serial flash (or other unmapped storage) to store images and fonts. The technique described here is especially usefull on STM32G0 and other deviced with very little RAM.

See the article Lowering Memory Usage with Partial Framebuffer for a introduction to partial framebuffers which are often used together with a serial flash.
See also the article Using Non-Memory Mapped Flash for storing images for an introduction to caching bitmaps from unmapped flash to RAM.

Configuration

To use a serial flash with your TouchGFX application you must change the TouchGFX Generator configuration to enable the "External Data Reader" in "Additional Features".



Additional Features: Data Reader

With this feature enabled the TouchGFX Generator changes the configuration to use the `LCD16bppSerialFlash` LCD class. It also generates a subclass of the `touchgfx::FlashDataReader`:

TouchGFXConfiguration.cpp

```
static TouchGFXDataReader dataReader;
static LCD16bppSerialFlash display(dataReader);
static ApplicationFontProvider fontProvider;
...
void touchgfx_init()
{
    ...
    hal.setDataReader(&dataReader);
    fontProvider.setFlashReader(&dataReader);
    ...
}
```

This code creates an instance of the `TouchGFXDataReader` class and passes that instance to the display object, to the HAL object, and to the ApplicationFontProvider object. These three objects will use the dataReader object to access data in the serial flash. The data can be both images and font data.

The system programmer must finish the implementation of the `TouchGFXDataReader` class to actually read data from a flash.

Implementation

The *TouchGFXDataReader* class implements the *touchgfx::FlashDataReader* interface. This interface has the following 4 methods that needs to be implemented on a specific hardware.

include/touchgfx/hal/FlashDataReader.hpp

```
    bool addressIsAddressable(const void* address)
    void copyData(const void* src, void* dst, uint32_t bytes)
    void startFlashLineRead(const void* src, uint32_t bytes)
    const uint8_t* waitFlashReadComplete()
```

The `addressIsAddressable` method is used by the `LCD16bppSerialFlash` class to decide if some data can be directly read (i.e. is located in internal RAM or internal flash) or if it should be read through the dataReader object.

The `copyData` *` method is used to copy data synchronously from the flash to RAM. This function is typically used when the data is not further processed. E.g. when copying a solid image to a framebuffer.

The `startFlashLineRead` method is used when multiple lines of data are required from the flash. The `startFlashLineRead` method initiates a read of data. The method can initiate an asynchronous read

and should return immediately after starting the read. The *waitFlashReadComplete* method should wait for the read to finish, and return a pointer to a buffer holding the data.

The `LCD16bppSerialFlash` can issue one flash read while processing the previously read data (in some situations). This means that at least two buffers are required in the dataReader to gain full concurrency.

The TouchGFX Generator generates the `FlashDataReader` in two classes: `TouchGFXGeneratedDataReader` and `TouchGFXDataReader`. The `TouchGFXGeneratedDataReader` is the superclass of the two and contains a default implementation. If that implementation is not suitable, the application programmer can change the implementation of the virtual functions in the `TouchGFXDataReader` class.

The `TouchGFXGeneratedDataReader` implementation calls C-functions to do the work. These application are implemented by the system programmer.

TouchGFX/target/generated/TouchGFXGeneratedDataReader.cpp

```
extern "C" __weak void DataReader_WaitForReceiveDone();
extern "C" __weak void DataReader_ReadData(uint32_t address24, uint8_t* buffer, uint32_t l
extern "C" __weak void DataReader_StartDMAReadData(uint32_t address24, uint8_t* buffer, u

void TouchGFXGeneratedDataReader::startFlashLineRead(const void* src, uint32_t bytes)
{
    /* Start transfer using DMA */
    DataReader_StartDMAReadData((uint32_t)src, (readToBuffer1 ? buffer1 : buffer2), bytes)
}
```

The implementation is found in the `MB1642BDataReader.c` file:

Core/Src/MB1642BDataReader.c

```
void DataReader_StartDMAReadData(uint32_t address24, uint8_t* buffer, uint32_t length)
{
    readDataDMA(address24, buffer, length);
}

void readDataDMA(uint32_t address24, uint8_t* buffer, uint32_t length)
{
    // Pull Flash CS pin low
    isReceivingData = 1;
    FLASH_CS_GPIO_Port->BRR = FLASH_CS_Pin;

    *((__IO uint8_t*)&hspi2.Instance->DR) = CMD_READ;

    ...
}
```

This implementation is specific to the protocol used by the flash and the GPIO used for SPI and CS. All three C functions must be implemented for the `TouchGFXGeneratedDataReader` class to work.

Images

As mentioned in the introduction the `LCD16bppSerialFlash` class can read image pixels through the dataReader object. For this to work we must change the lnker script to put images in an address range outside the internal flash range.

On the STM32G071 we have selected the address 0x90000000 as start address for the serial flash:

gcc/STM32G071RBTX_FLASH.ld

```
MEMORY
{
  RAM        (xrw)    : ORIGIN = 0x20000000,   LENGTH = 36K
  FLASH      (rx)     : ORIGIN = 0x8000000,    LENGTH = 128K
  SPI_FLASH  (r)      : ORIGIN = 0x90000000,   LENGTH = 8M
}

/* Sections */
SECTIONS
{
  ...

  ExtFlashSection :
  {
    *(ExtFlashSection ExtFlashSection.*)
    *(.gnu.linkonce.r.*)
    . = ALIGN(0x4);
  } >SPI_FLASH

  FontFlashSection :
  {
    *(FontFlashSection FontFlashSection.*)
    *(.gnu.linkonce.r.*)
    . = ALIGN(0x4);
  } >SPI_FLASH
}
```

This puts the `ExtFlashSection` and `FontFlashSection` into the 0x90000000 address range.

The remaining task is to write the data to the external flash using a flasher tool.

A short description on writing flash loaders for STM32CubeProgrammer can be found in section 2.3.3 in this document:

Font data

The above linker script puts the font pixel data and the font character metadata (with and height) into the external flash (both types of data are in the FontFlashSection). This data is also read through the dataReader object when drawing characters on the Screen

If you are not using the "Unmapped Storage Format" for your you must change the linker script and the file `include/touchgfx/hal/Config.hpp` to move the font character metadata to internal flash.

See the article about Fonts in unmapped storage for more information on the font formats.

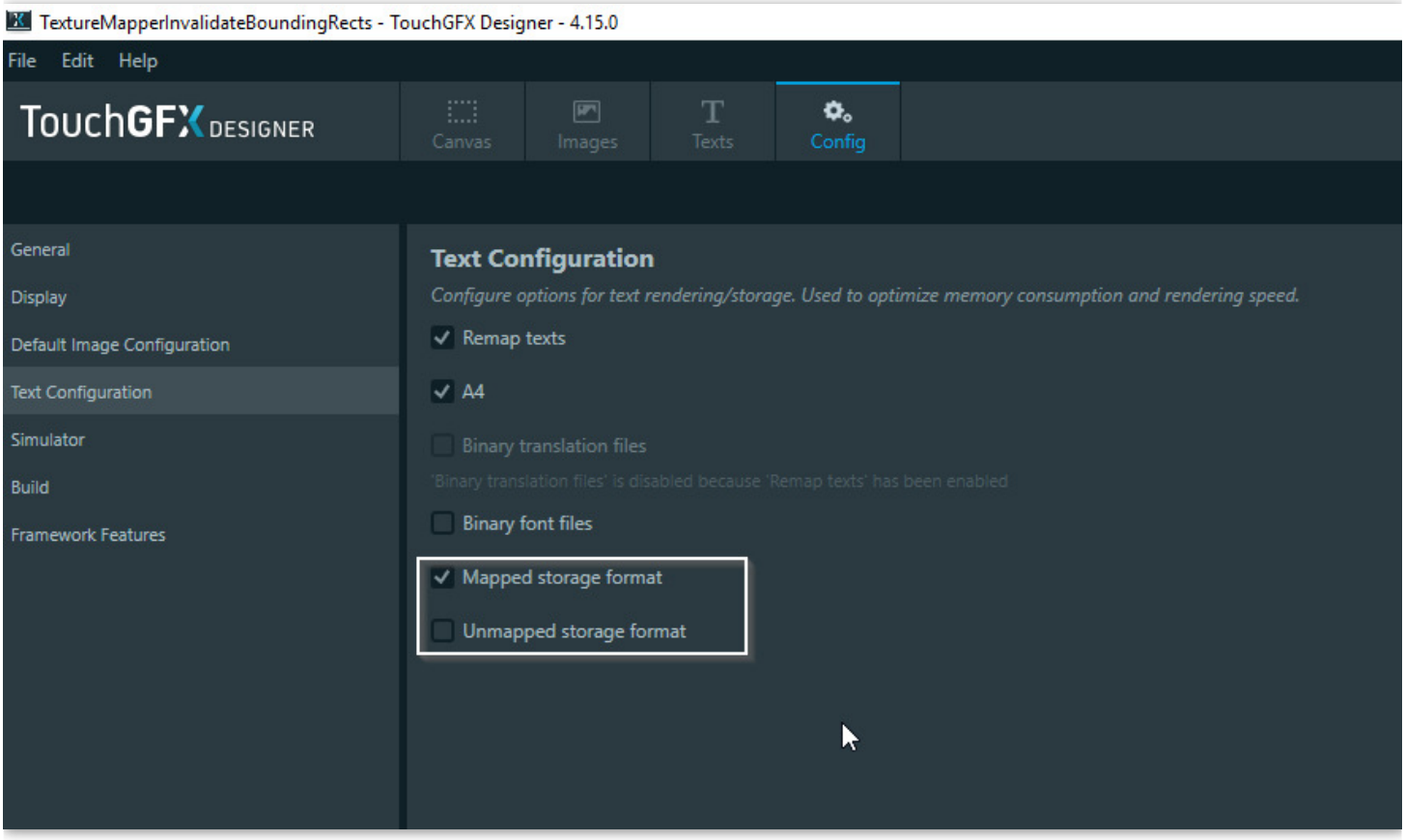
Version: 4.16

Using Non-Memory Mapped Flash for Font Data

In this section we will discuss how to use a new font-layout that will allow you to put almost all font data into unmapped external flash. The effect is that you can have many thousand letters in an application using only 50 kb of flash.

Font Layouts

TouchGFX supports two different font layouts for the fonts compiled into your application. The layout used is selected in TouchGFX Designer in the configurations tab:



Configurating font layout

The `Mapped storage format` is the default font layout and should be used on systems where fonts are stored in memory mapped flash (internal or e.g. external QSPI flash).

The `Unmapped storage format` is the new font layout. It allows most of the font data to be stored in unmapped flash. This will typically be a SPI-flash, but can be any type of storage.

Mapped Storage Format

The mapped storage format keeps the font data in two tables.

The first table is an array of *touchgfx::GlyphNode*. These contain the properties of the individual characters: height, width, unicode, and similar.

generated/fonts/src/Table_verdana_20_4bpp.cpp

```
FONT_TABLE_LOCATION_FLASH_PRAGMA
KEEP extern const touchgfx::GlyphNode glyphs_verdana_20_4bpp[] FONT_TABLE_LOCATION_FLASH_A
{
    {     0, 0x0020,   0,   0,   0,   0,   7,   0,   0, 0x00 },
    {     0, 0x002C,   5,   7,   3,   1,   7,   0,   2, 0x00 },
    {    21, 0x0030,  11,  14,  14,   1,  13,   0,   0, 0x00 },
    {   105, 0x0032,  11,  14,  14,   1,  13,   0,   0, 0x00 },
    {   189, 0x0033,  11,  14,  14,   1,  13,   0,   0, 0x00 },
    {   273, 0x0034,  12,  14,  14,   0,  13,   0,   0, 0x00 },
    ...
}
```

The second table (split in multiple files for large fonts) contains the pixel-patterns for the characters.

generated/fonts/src/Font_verdana_20_4bpp_0.cpp

```
FONT_GLYPH_LOCATION_FLASH_PRAGMA
KEEP extern const uint8_t unicodes_verdana_20_4bpp_0[] FONT_GLYPH_LOCATION_FLASH_ATTRIBUTE
{
    // Unicode: [0x0020]
    // (Has no glyph data)
    // Unicode: [0x002C]
    0x00, 0x87, 0x04, 0x20, 0xFF, 0x03, 0x60, 0xBF, 0x00, 0xA0, 0x5F, 0x00, 0xE0, 0x0D, 0x
    0x07, 0x00, 0xF6, 0x01, 0x00,
    // Unicode: [0x0030]
    0x00, 0xA3, 0xFE, 0x9D, 0x01, 0x00, 0x40, 0xFF, 0x9B, 0xFC, 0x1D, 0x00, 0xD0, 0x4F, 0x
    0x9F, 0x00, 0xF3, 0x0B, 0x00, 0x10, 0xEE, 0x00, 0xF7, 0x07, 0x00, 0x00, 0xFB, 0x03, 0x
    ...
}
```

The GlyphNodes will be used by the TouchGFX engine during text layout. The pixels will be read by the DMA2D or software routines during drawing.

On platforms using the normal LCD classes, e.g. LCD16Bpp or LCD24Bpp, these tables must be stored in internal flash or memory mapped external flash.

On platforms using a unmapped external flash, the LCD16BppSerialFlash can read the pixels-patterns from unmapped serial flash, but the GlyphNodes must be in internal flash.

Unmapped Storage Format

The unmapped storage format splits the font data in three tables. The two tables from the mapped storage layout is reused, but a third table is added:

generated/fonts/src/Table_verdana_20_4bpp.cpp

```
FONT_SEARCHTABLE_LOCATION_FLASH_PRAGMA
KEEP extern const uint16_t unicodelist_verdana_20_4bpp[] FONT_SEARCHTABLE_LOCATION_FLASH_A
{
    0x0020,
    0x002E,
    0x003F,
    0x004E,
    0x0054,
    ....
}
```

This third table just contains the unicodes present in the font.

When this font layout is used, the third table must be present in internal flash, but the other two tables can be moved to external flash. This is a considerable saving, as the third table uses two bytes for each character, whereas the GlyphNode table uses 14 bytes. This reduces the storage requirement in internal flash.

When the font data is placed in unmapped flash, the mcu cannot access it directly. We therefore have to provide a flash reader object to the font subsystem.
The code for this is automatically generated by TouchGFXGenerator:

TouchGFXConfiguration.cpp

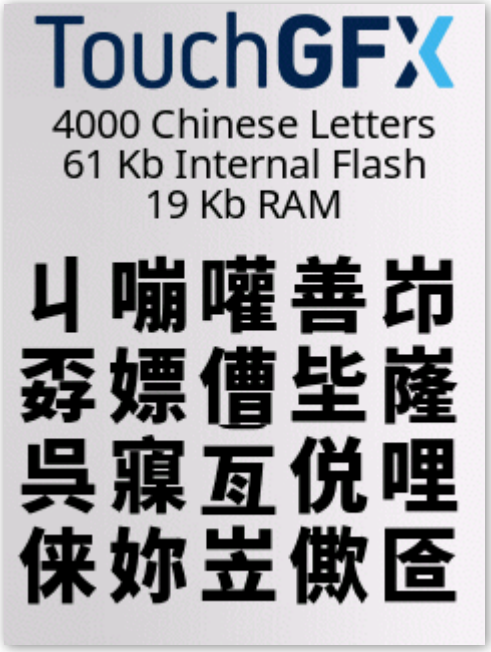
```
static TouchGFXDataReader dataReader;
static LCD16bppSerialFlash display(dataReader);
static ApplicationFontProvider fontProvider;
static Texts texts;
static TouchGFXHAL hal(dma, display, tc, 240, 320);
void touchgfx_init()
{
  Bitmap::registerBitmapDatabase(BitmapDatabase::getInstance(), BitmapDatabase::getInstanc
  TypedText::registerTexts(&texts);
  Texts::setLanguage(0);
  hal.setDataReader(&dataReader);
  fontProvider.setFlashReader(&dataReader);
...
```

If you are not using the generator, you must do this manually.

Remember to implement the functions in *TouchGFXDataReader* so data is actually read from your flash.

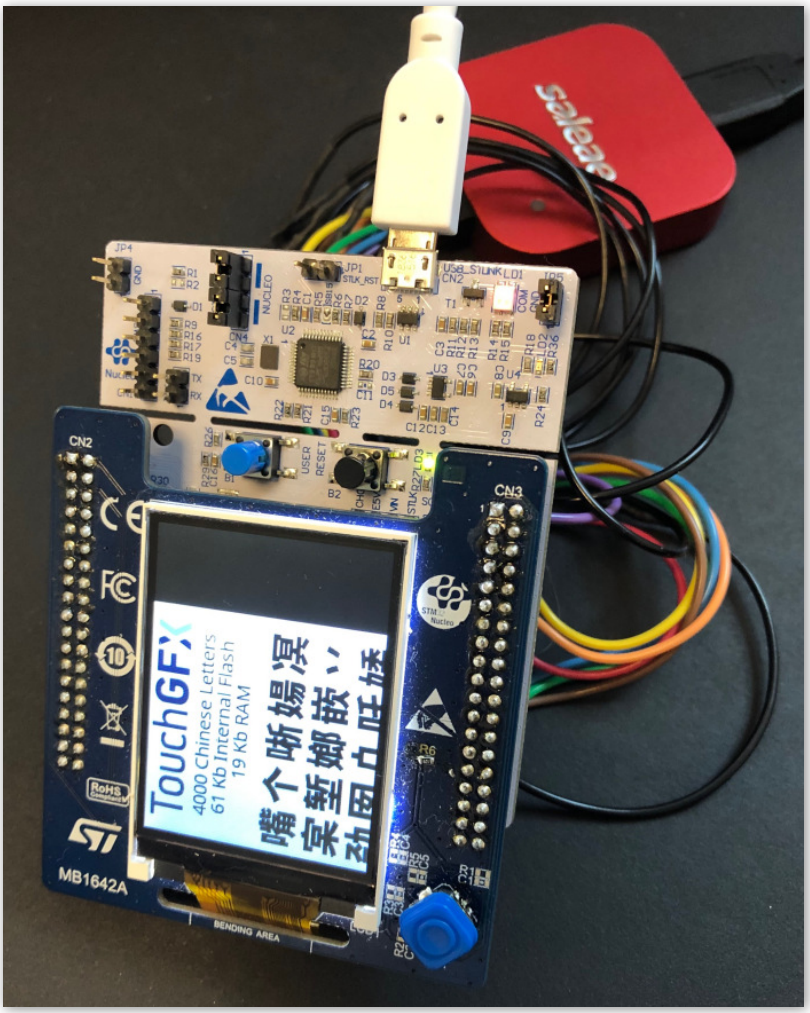
Example

Below is a screenshot of an application using the new font layout:



Example application with 4000 Chinese characters

This application runs on a STM32G071 Nucleo board with a MB1642A display module:



Example application running on STM32G071Nucleo

In this application we have 4000 Chinese characters in size 20, 4 bits pr pixel. The application and data takes up 61Kb of 128 Kb available on the STM32G071. The font data is distributed as follows (excluding minor objects):

Table	Location	Size
GlyphNodes	External SPI flash	57.372 bytes
Pixel patterns	External SPI flash	3.116.296 bytes
Unicode list	Internal flash	8.000 bytes

Linker Script Modifications

To use the unmapped font layout correctly, you must update your linker script to place the tables correctly.

STM32F746.ld

```
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
```

```
define symbol __ICFEDIT_region_ROM_end__   = 0x0801FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20008FFF;
define symbol __ICFEDIT_region_SERIAL_FLASH_start__ = 0x90000000;
define symbol __ICFEDIT_region_SERIAL_FLASH_end__   = 0x91000000;

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };

place in SERIAL_FLASH_region {section ExtFlashSection, section FontFlashSection };
```

In this linker script we put both the ExtFlashSection (images and font pixels) and FontFlashSection (the GlyphNodes) in the external flash. Any other read-only data is in the internal flash (ROM_region).

Version: 4.16

Changing the Pixel Format of an Application

This article shows how to change the pixel format of an application after a project has been created. Concretely, this article exemplifies modifying a 24-bit RGB888 application to be 16-bit RGB565 through the TouchGFX Generator and also shows the impact on the TouchGFX project configuration. Reasons to change the pixel format could be the following:

1. Modified display requirements
2. Modified application requirements
3. Mistake during initial project creation

> **FURTHER READING**
>
> Please read the article on **Color Formats**.

Generally, the following changes could be required to change the pixel format of your application.

1. **Board Bring Up:** Change the pixel format of the LTDC.
2. **HAL Development:** Modify TouchGFX Generator settings to reflect LTDC settings or location of framebuffer(s) in memory.
3. **TouchGFX Designer**: Ensure that the designer is using this new converted bit depth and correct format for image assets.

Starting from the designer we can inspect the current configuration for *Display* and *Image* and return to these screens later to validate.

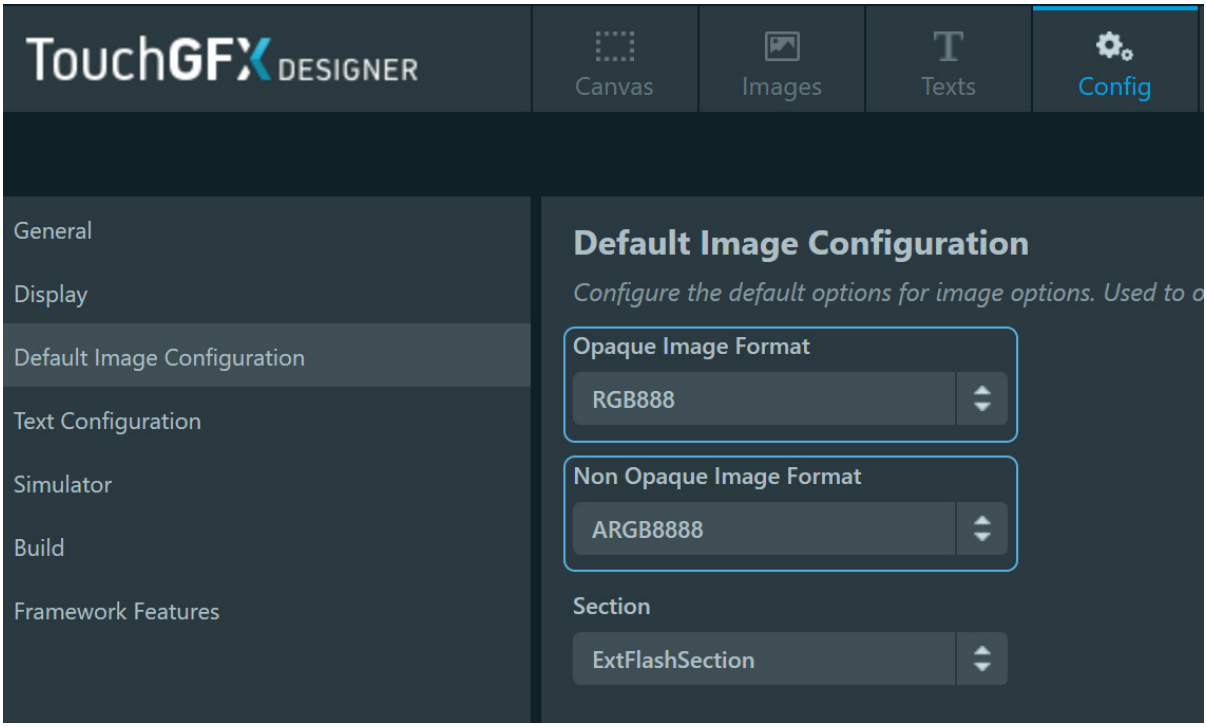
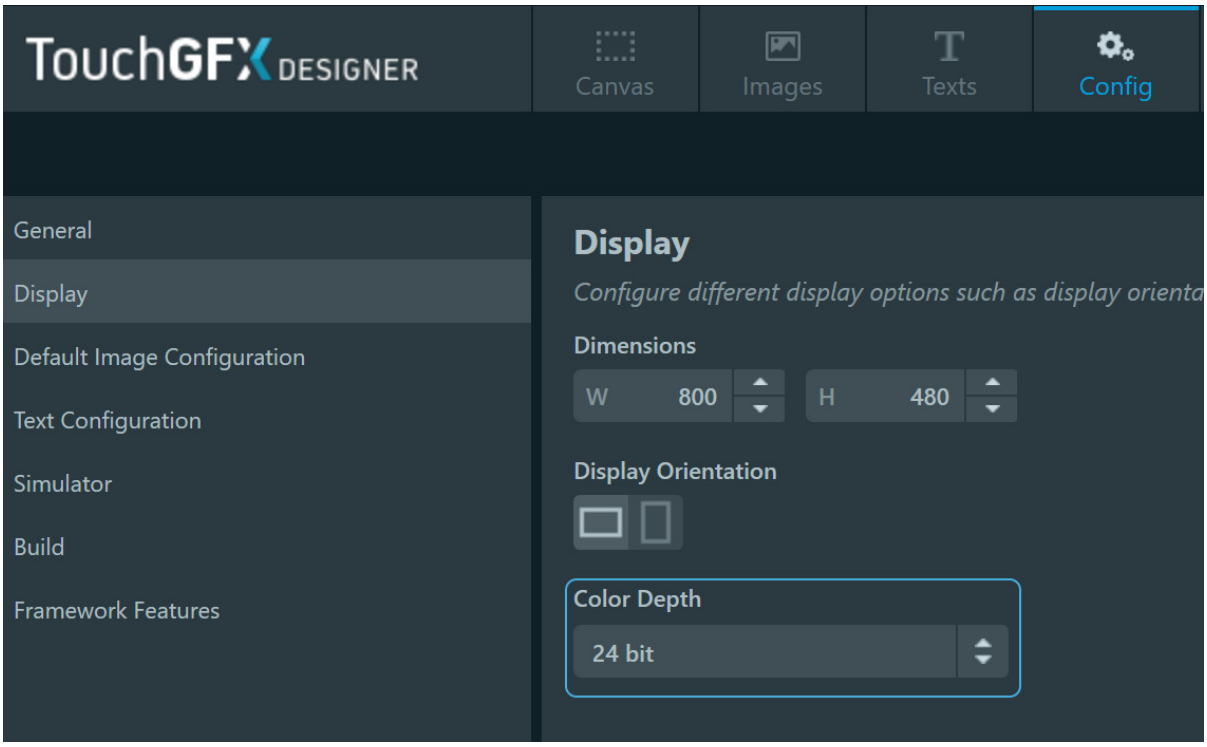


Image configuration



Display configuration.

Board Bring Up

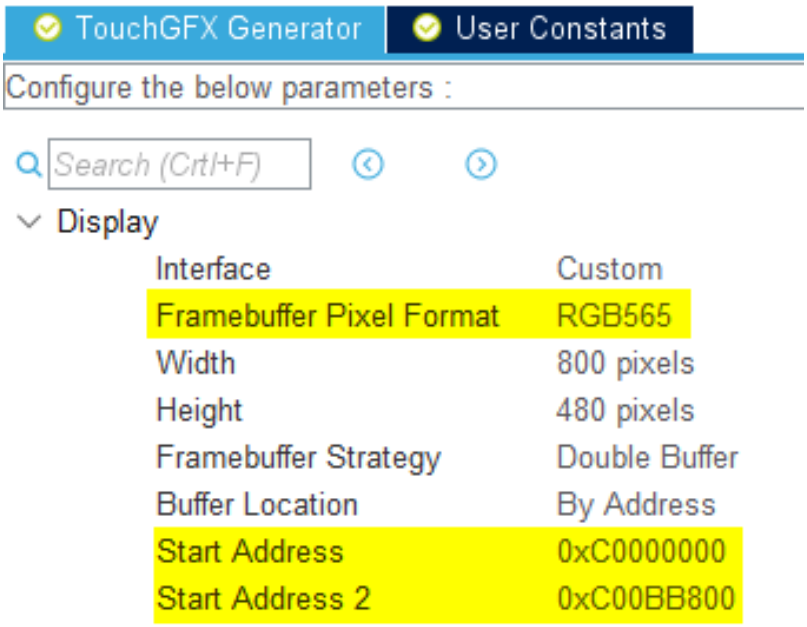
If certain settings in the CubeMX project can impact the desired pixel format of the application, as is the case for LTDC, developers are required to make appropriate changes here such that TouchGFX renders in the format expected by the LTDC.

FURTHER READING

Please read the article on **LTDC** for details on valid pixel format settings for TouchGFX applications.

HAL Development

If the bit-depth of the applications changes along with the pixel format of the framebuffer (e.g. changing ARGB2222 to BRGA2222 you would still have an 8-bit application), developers *may* be required to modify the memory locations of the framebuffers. In the case below:



Modify pixel format

Generating code using this configuration will generate a `TouchGFXConfiguration` that uses the 16-bit TouchGFX `LCD` class.

TouchGFXConfiguration.cpp

```
static LCD16bpp display;
```

If the project is open in TouchGFX Designer, developers will be prompted to update to reflect the newly generated configuration changes.

```
{
  "image_configuration": {
    "images": {},
    "dither_algorithm": "2",
    "alpha_dither": "yes",
    "layout_rotation": "0",
    "opaque_image_format": "RGB565",
    "nonopaque_image_format": "ARGB8888",
    "section": "ExtFlashSection",
```

```
    "extra_section": "ExtFlashSection"
  },
```

The table below lists the impact on opaque and non-opaque image formats given a pixel format:

Framebuffer pixel format	Opaque format	Non-Opaque format
Gray2	Gray2	Gray2
Gray4	Gray4	Gray4
ABGR2222	ABGR2222	ABGR2222
ARGB2222	ARGB2222	ARGB2222
BGRA2222	BGRA2222	BGRA2222
RGBA2222	RGBA2222	RGBA2222
RGB565	RGB565	ARGB8888
RGB888	RGB888	ARGB8888

TouchGFX Designer

Once the TouchGFX project has been updated based on the new TouchGFX Generator settings defined in CubeMX, developers will find that the TouchGFX Designer configuration has changed to match.

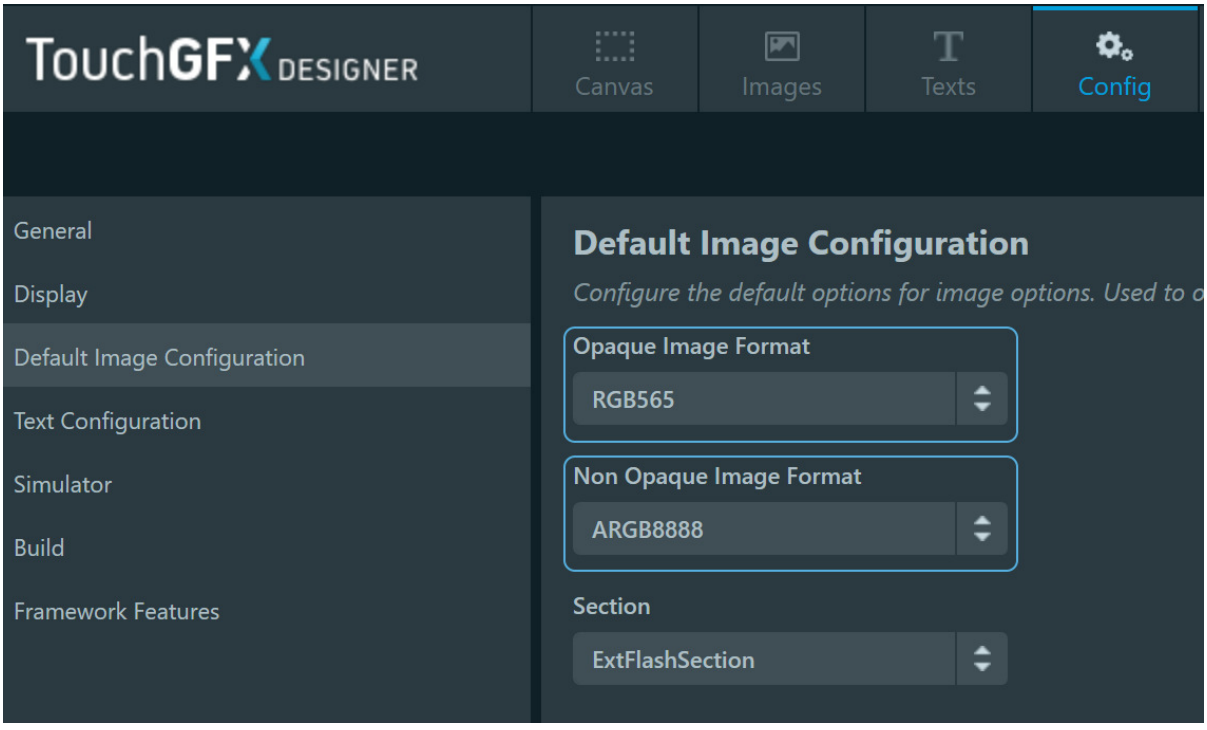
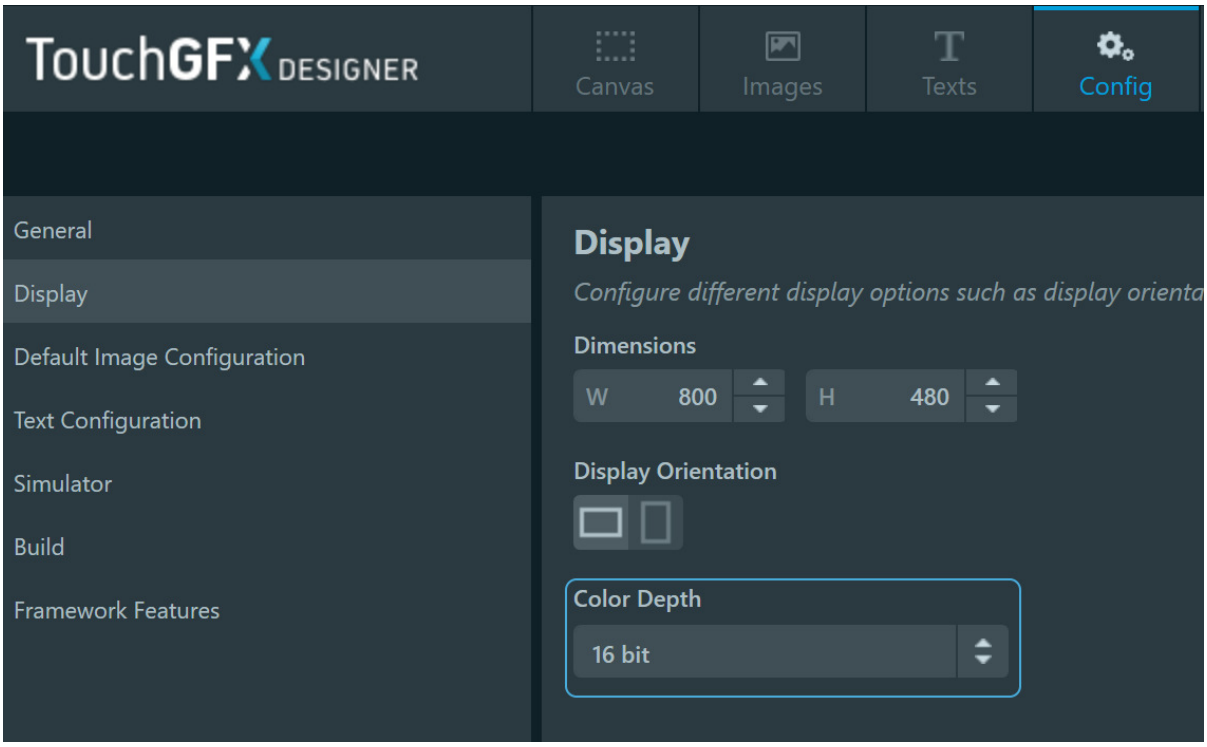


Image configuration



Display configuration.

After generating code from within TouchGFX Designer the code for image assets now reflect the updated pixel format:

TouchGFX\generated\images\src\my_image.png

```
LOCATION_PRAGMA("ExtFlashSection")
KEEP extern const unsigned char my_image[] LOCATION_ATTRIBUTE("ExtFlashSection") = // 320x
```

Conclusion

Modifying the current pixel format of an application can be done by simply using TouchGFX Generator. For MCUs with an LTDC, the layer running TouchGFX must match the Framebuffer pixel format defined in TouchGFX Generator (Limited to RGB565 and RGB888 for LTDC) settings to ensure compliance with the format rendered by TouchGFX Core.

Once code has been generated from CubeMX, the TouchGFX project will be updated and upon subsequent code generation in TouchGFX Designer both image assets and framebuffer driver will have the specified formats.

Version: 4.16

Creating an Application Template

Application Templates (ATs) are `.tpa` files that define the *platform* on which a TouchGFX application runs. This approach is for developers who wish to be able to distribute easy-to-use ATs separately from the *UI* code that runs on top of them. This article describes how an existing TouchGFX project can be packaged into a redistributable AT using the built-in tool `tgfx.exe`. For the duration of this article examples are based on an application called "*MyApplication*".

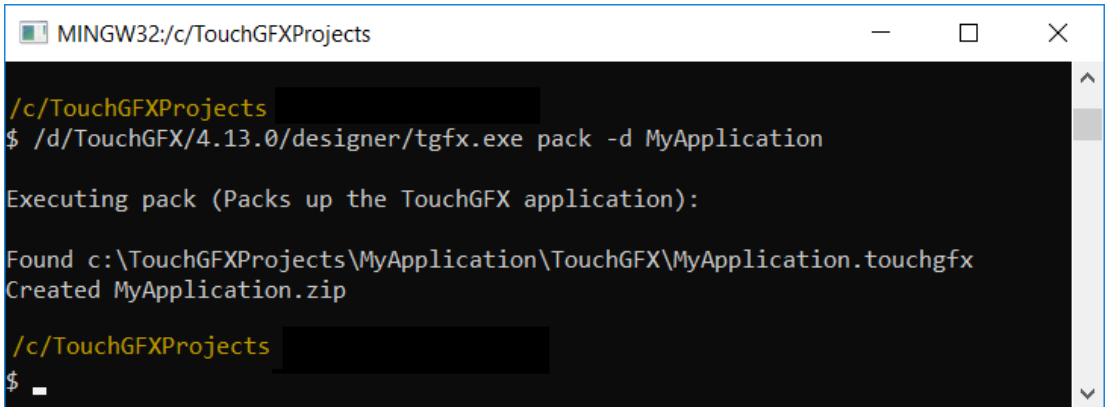
Once you have a fully functional TouchGFX project the following steps are required to create an AT.

- **Describe Application Template** Call `tgfx.exe` and edit json file (inherits from .touchgfx)
- **Create Application Template** Call `tgfx.exe` to finalize `.tpa`
- **Test & Verify** Import into designer, create- and verify application

Describe Application Template

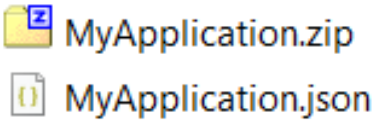
The `tgfx.exe` tool generates a configuration file (.json) that describes the internals of the AT. This information is read by TouchGFX Designer and presented to the user. Open a *TouchGFX Environment* console and execute the following command in the *parent* directory of the application:

```
$ /d/TouchGFX/4.13.0/designer/tgfx.exe pack -d MyApplication
```



Prepare files for .tpa

The following files are created in the directory where the command was run:



List of generated files

Before creating the final `.tpa` file, edit `MyApplication.json` to control how the AT is displayed to users in TouchGFX Designer. Users should edit the following sections:

- **Author** Use the fields in the *Author* section to specify name of author, contact email and a URL.
- **Data** Use the fields in the *Data* section to specify AT version, images, board name, vendor, description, and link to further information.

MyApplication.json

```
  ...
   "Author": [
     {
       "Name": "Chad Brody",
       "Contact": "chad.brody@mycompany.com",
       "URL": "http://mycompany.com/"
     }
   ],
 ...
 "Data": {
   "Version": {
     "Major": 1,
     "Minor": 0,
     "Build": 0
   },
   "Name": "MyApplication",
   "HumanFriendlyName": "MyApplication",
   "BoardName": "Custom STM32 Board",
   "Type": "TGAT",
   "Vendor": "MyCompany",
   "Description": "This is a working project on which to base your UI on top of.",
   "DocumentationLink": "",
   "Category": "",
   "Images": [
     "http://mysite.com/MyCustomBoard-front.png",
     "http://mysite.com/MyCustomBoard-back.png"
   ],
   ...
 }
}
```

TIP

Be sure to set the 'Type' attribute to **TGAT**. Otherwise the AT won't show up in TouchGFX Designer!

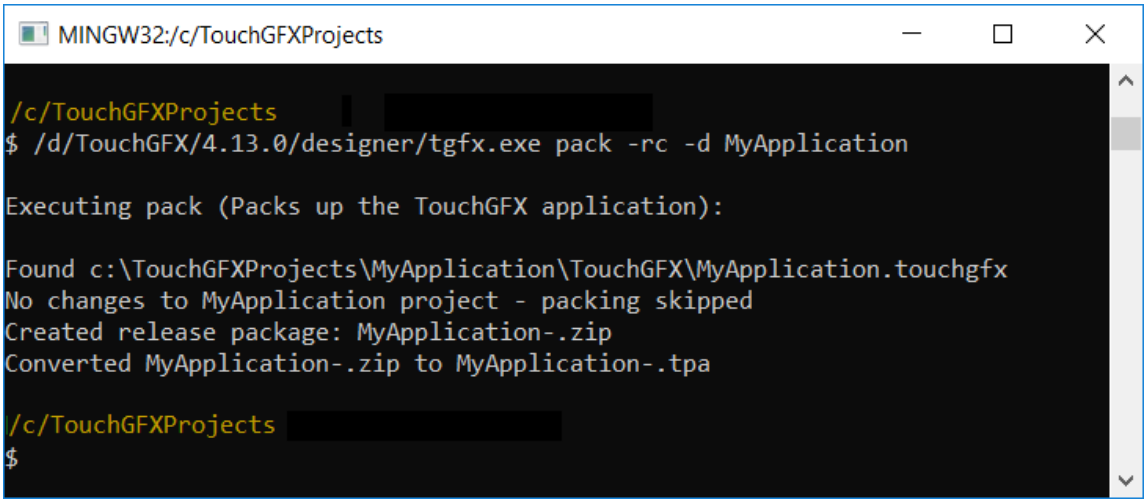
TIP

TouchGFX Designer is able to display up to three images (Image references must be URLs) from this list when displaying the extended information card for an AT. The optimal resolution for the images is 400x280 pixels.

Create Application Template

Execute the following command to create the final ´.tpa´ file and finalize the Application Template.

```
$ /d/TouchGFX/4.13.0/designer/tgfx.exe pack -rc -d MyApplication
```

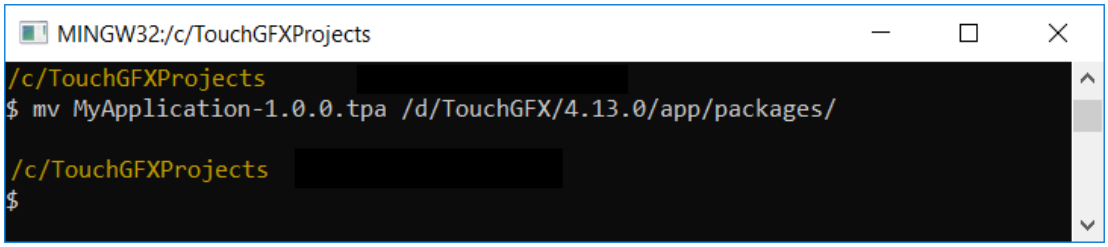


Create Application Template

Test & Verify

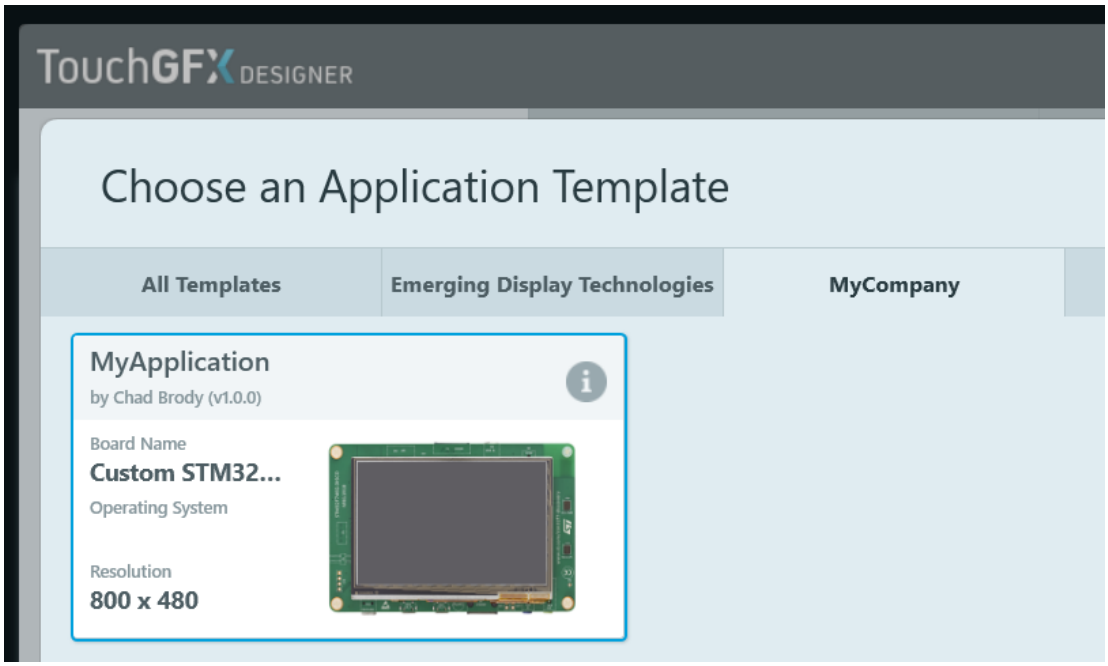
To verify that the `.tpa` file can be seen by TouchGFX Designer as an AT and used to create new applications, perform the following steps:

1. Rename the `.tpa` file to your requirements.
2. Copy or move the `.tpa` file to `C:\TouchGFX\4.13.0\app\packages`. This allows users to import ATs into TouchGFX Designer from a local folder.

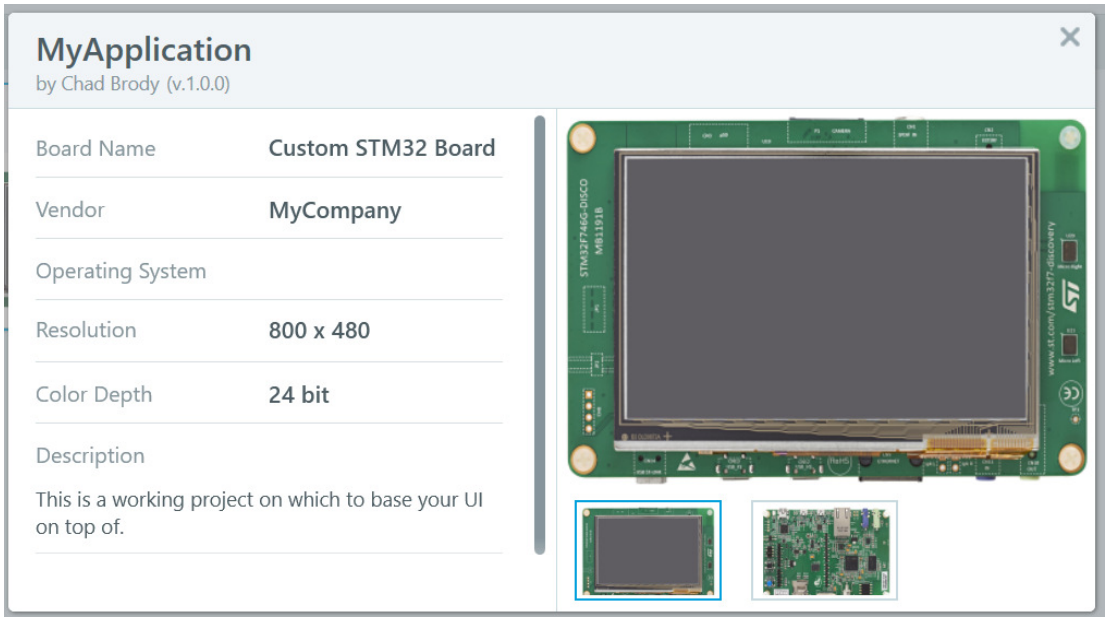


Install AT to local folder

3. Open the Designer and select the AT.



MyApplication-1.0.0.tpa displayed in Designer



Information about the AT

Final Notes

The following section contains tips about what to consider when developing code for/distributing ATs.

General Tips

Generally, before distributing the `.tpa` one should:

1. Ensure that all supplied IDE projects work as expected.
2. Delete *build-* and *generated* folders to reduce the size of the AT.
3. Ensure that custom commands (*PostGenerate-*, etc.) defined in the TouchGFX project file `.touchgfx` work as expected.
4. Ensure that the AT can be read by TouchGFX Designer and used to create a new application.
5. There is no immediate way to specify an upgrade procedure between versions of ATs.

> **TIP**
>
> Be sure to re-pack the application template after modifying the content of either the TouchGFX Project or the `.json` file.

After distributing the `.tpa` one should instruct users to copy the `.tpa` file into `C:\TouchGFX\4.13.0\app\packages` and restart the designer, if open.

Version Control

Usually, developers will keep an entire development project (Board bringup, TouchGFX AL, TouchGFX UI) in the same repository which eliminates the need for distributable `.tpa` files. However, for team members to be able to start a new TouchGFX application, unified platform code is powerful when it comes to test and verification.

For those that do distribute `.tpa` files and/or use tools like `repo`, `git submodules` to modularize their codebase it is wise to let the version of the AT-component follow the version specified in the `.json` descriptor mentioned previously in this article. If using a modularized approach, the *platform* code could still be used to create a distributable `.tpa` file while also being used as a module in a main project struture.

```
  "Data": {
    "Version": {
      "Major": 3,
      "Minor": 0,
      "Build": 0
    },
```

```
$ git tag
v1.1.0
v2.0.0
v2.1.0
v3.0.0
```

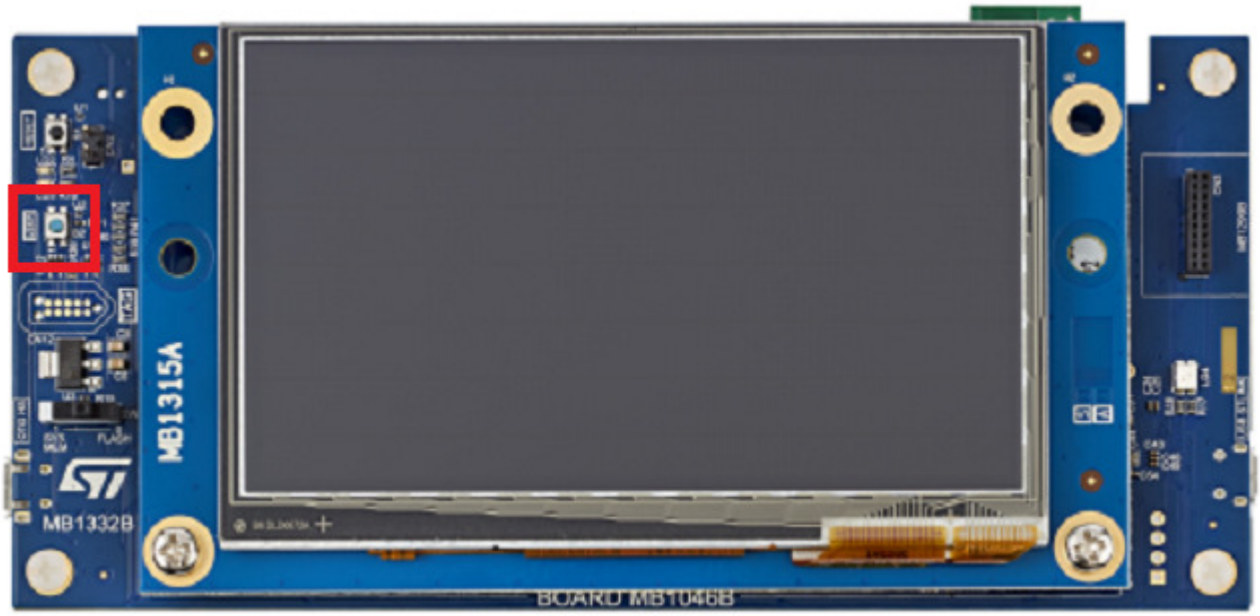
Version: 4.16

External Events as Triggers

This section describes how to use external events, such as physical buttons, as triggers in TouchGFX Designer.

Application requirements to respond to input from peripherals, such as physical buttons, requires that GPIO pins on the MCU are configured in accordance with board schematics.

In this example the H7B3I-DK is used to show how to make the UI react to the press of a physical button. This example uses polling, while EXTI could also be used instead.



The schematic can be downloaded here at: st.com

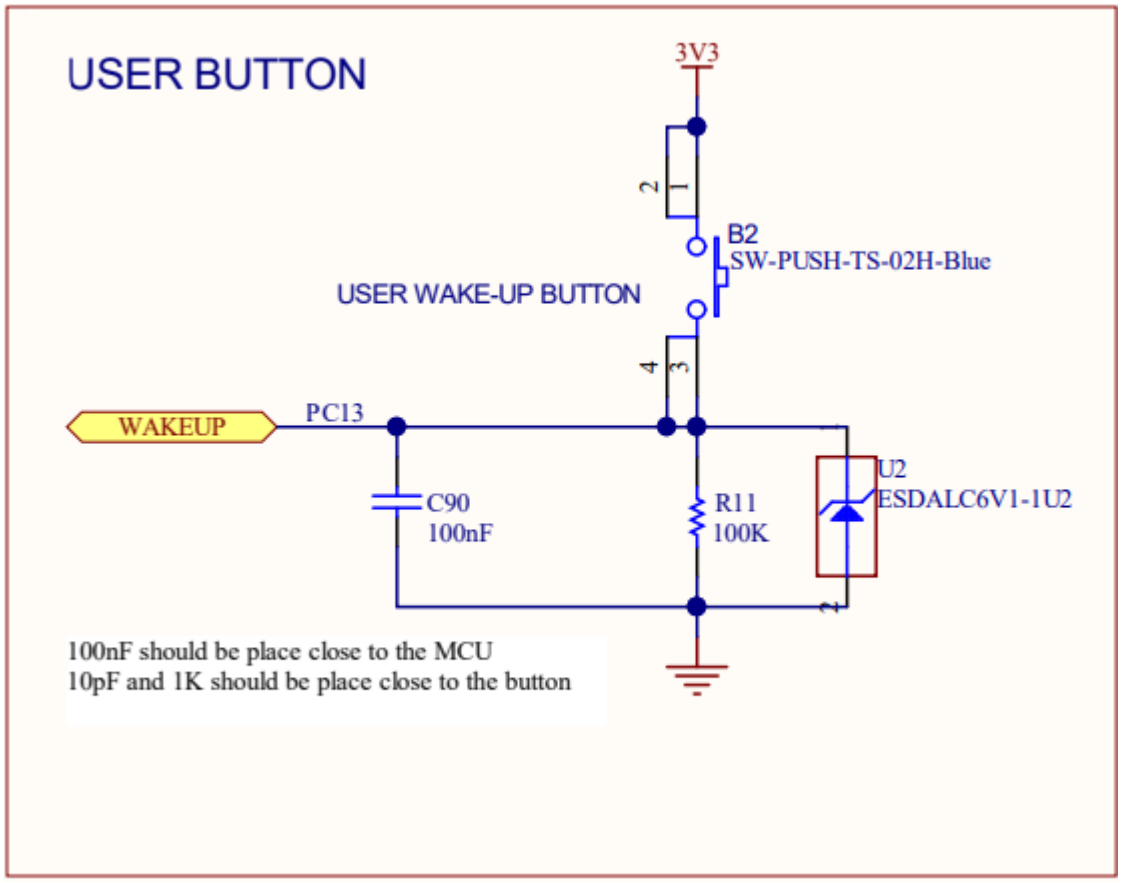
TIP

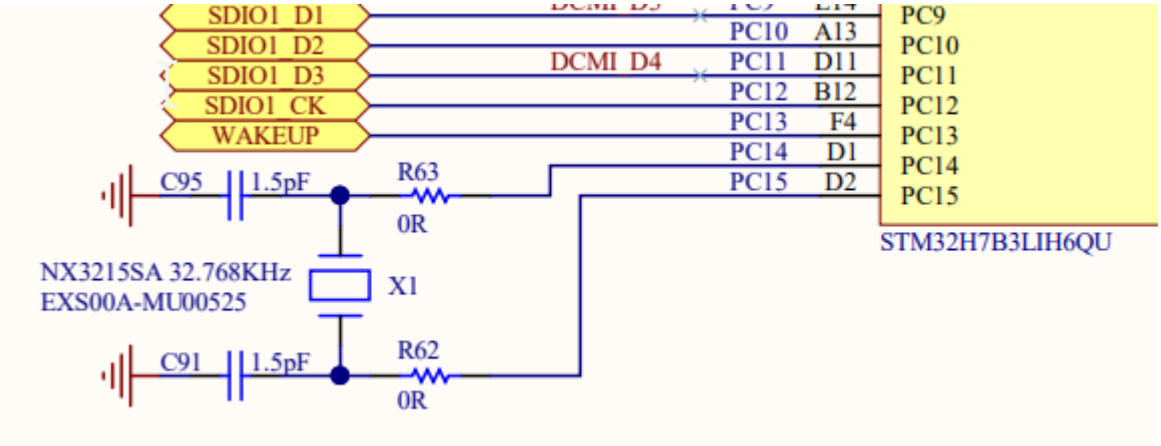
If a GPIO pin is readable it is usable as a trigger to an event in TouchGFX Designer.

Board Bringup

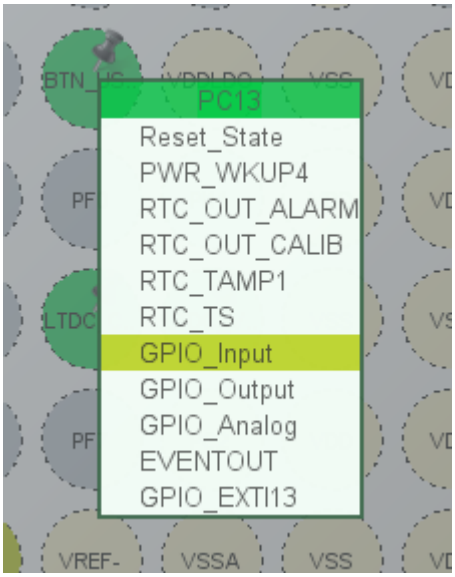
The following images depict a part of the schematics for the STM32H7B3I-DK board, and the examples that follow use CubeMX to configure the appropriate GPIO port and pin as an input to read the state of the button.

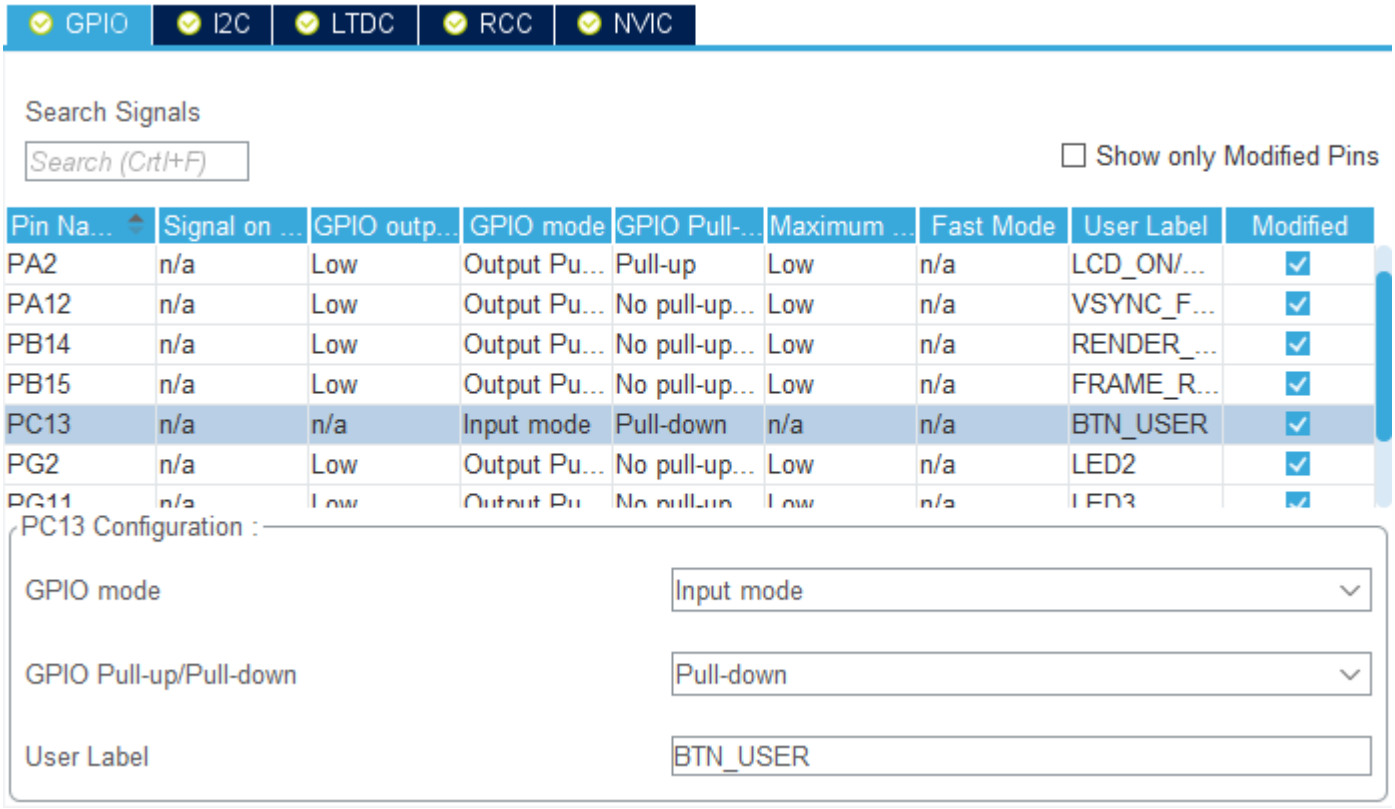
Start a new application based on the STM32H7B3I-DK application template. According to the User Button is attached to GPIO Port C Pin 13 (PC13).





In CubeMX PC13 can be configured as an input and configured as a pull-down in the **GPIO** section of the **System Core** category.





The following code has been generated by CubeMX based on the name given to it in the Pinout View.

```
#define MCU_ACTIVE_GPIO_Port GPIOI
#define VSYNC_FREQ_Pin GPIO_PIN_12
#define VSYNC_FREQ_GPIO_Port GPIOA
#define BTN_USER_Pin GPIO_PIN_13
#define BTN_USER_GPIO_Port GPIOC
#define LED2_Pin GPIO_PIN_2
#define LED2_GPIO_Port GPIOG
#define LCD_INT_Pin GPIO_PIN_2
```

```
static void MX_GPIO_Init(void)
  __HAL_RCC_GPIOJ_CLK_ENABLE();
```

```
  __HAL_RCC_GPIOI_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOD_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  ...
    /*Configure GPIO pin : BTN_USER_Pin */
  GPIO_InitStruct.Pin = BTN_USER_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  HAL_GPIO_Init(BTN_USER_GPIO_Port, &GPIO_InitStruct);
  }
```

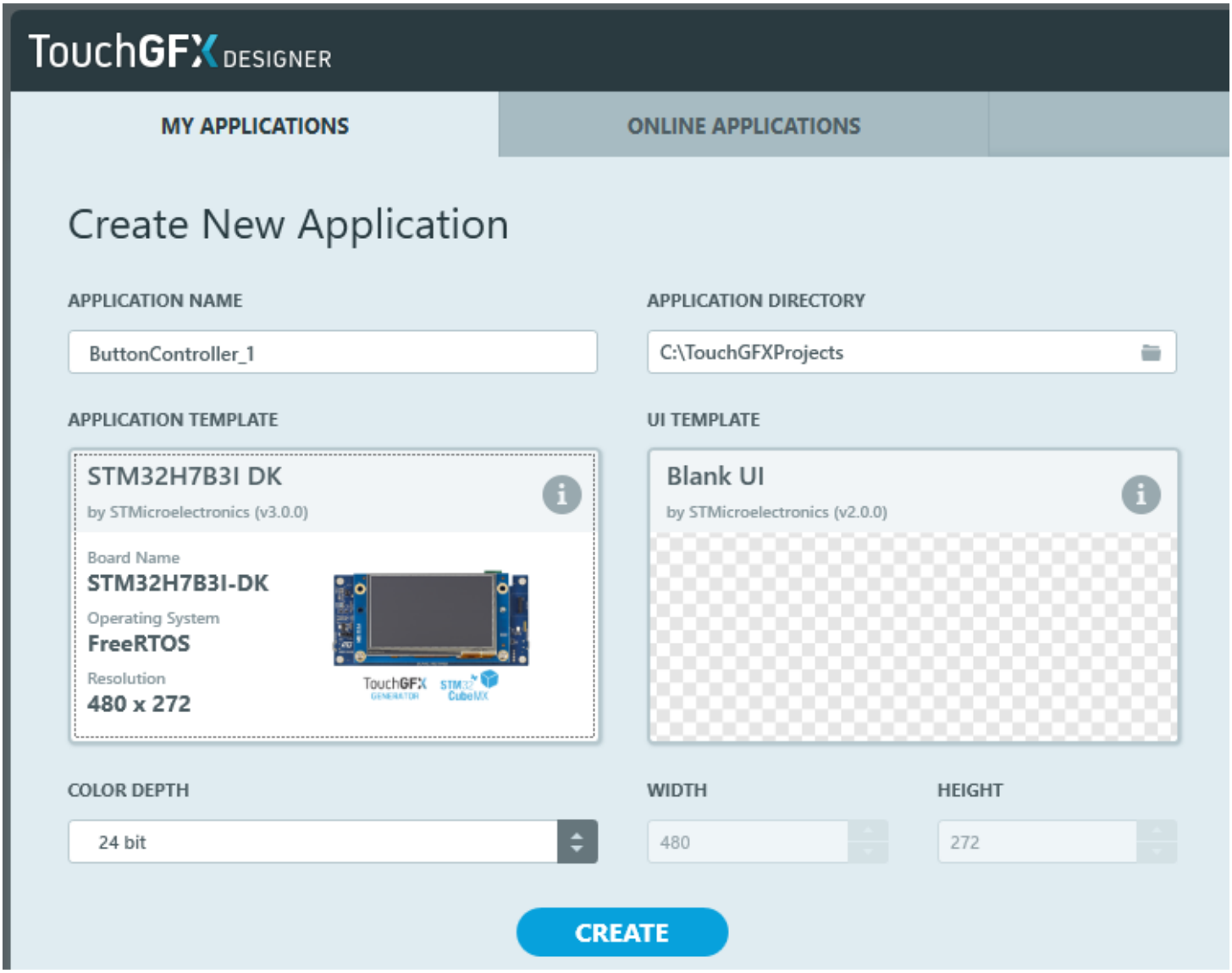
TouchGFX HAL Development

A part of the rendering cycle of TouchGFX engine is to check for possible input

Once the input state can be read (external event), TouchGFX HAL can now read this event as part of the rendering cycle through the ButtonController interface.

```
#include <platform/driver/button/ButtonController.hpp>
class H7B3ButtonController : public touchgfx::ButtonController
{
    virtual void init() {  }
    virtual bool sample(uint8_t& key)
    {

      if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) != GPIO_PIN_RESET)
      {
        key = 1;
        return true;
      }
      return false;
    }
private:

};

...
H7B3ButtonController bc;
void touchgfx_init()
{
  ...
  hal.initialize();
  hal.setButtonController(&bc);
}
```

TouchGFX Designer

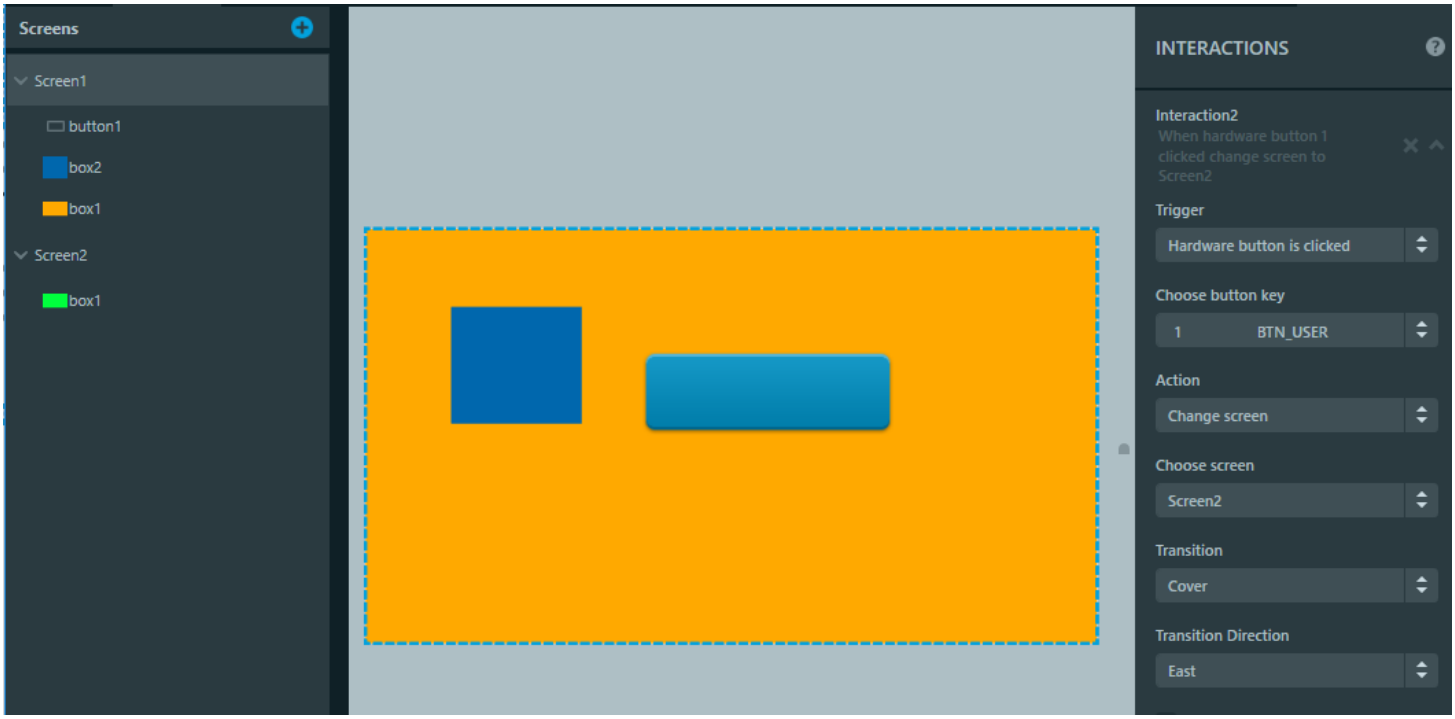


To use a value sampled by the `ButtonController` in interactions from the TouchGFX designer a name/value-mapping must be created in the `.touchgfx` project file.

```
"PhysicalButtons": [],
```

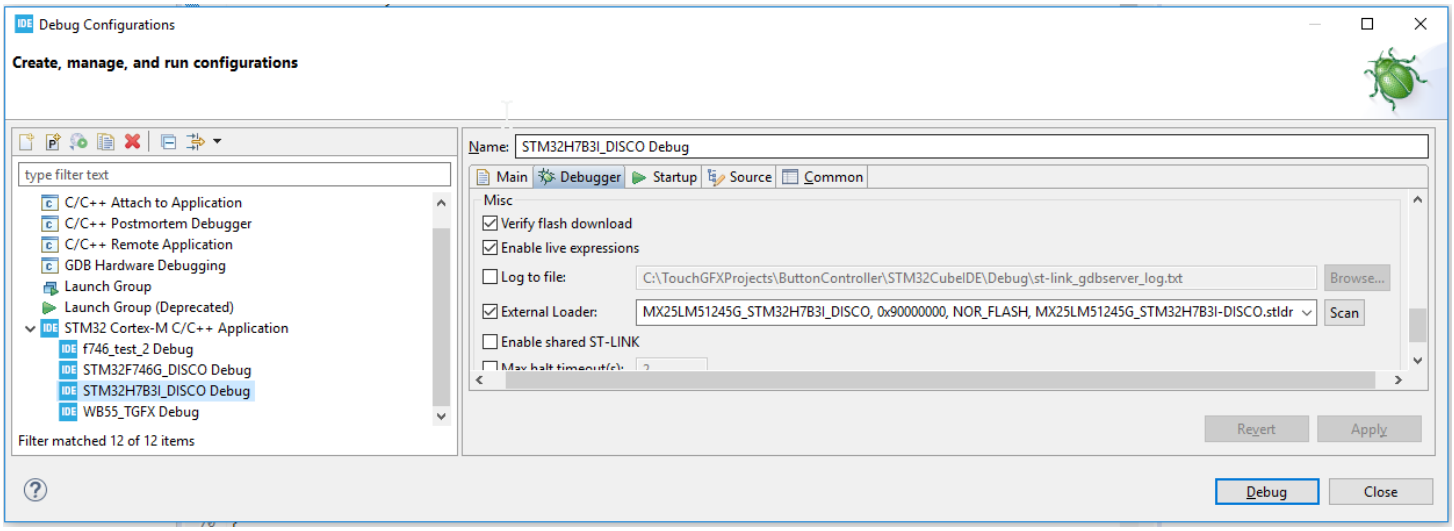
```
"PhysicalButtons": [
  {
    "Key": 1,
    "Name": "BTN_USER"
  }
],
```

"Hardware Button is clicked" is now available as a trigger when creating an interaction. Selecting the "Key"/"Name" pair defined in the .touchgfx file allows users to specify an action, such as "Change screen".



Running on target

After pressing "Generate code" in the designer, open the CubeIDE project, configure the debug configuration.



Links

https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-discovery-kits/stm32h7b3i-dk.html#

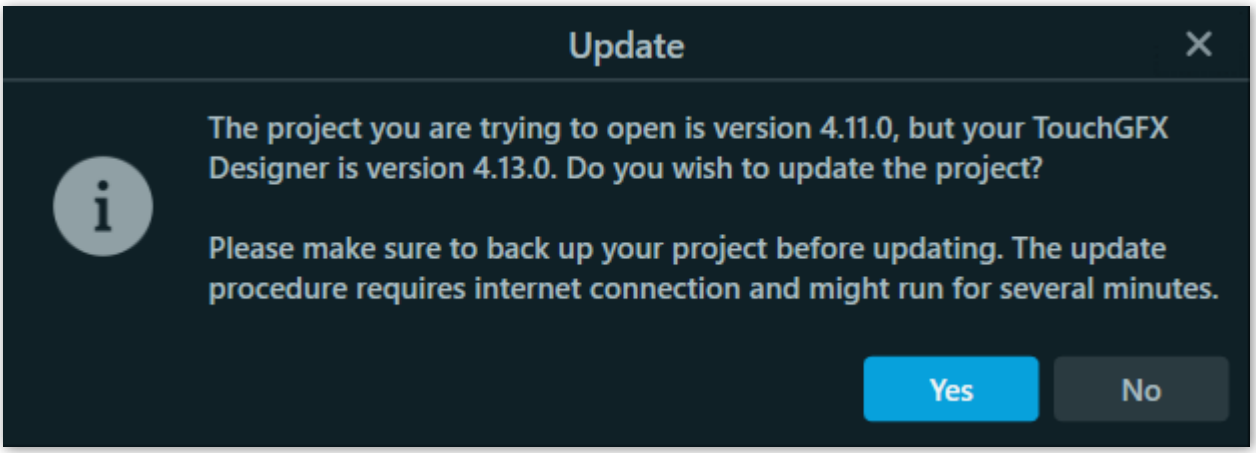
Version: 4.16

Updating to a new TouchGFX Version

When a TouchGFX Designer application is created, the .touchgfx project file created will have the same version as the TouchGFX version used to create the application. But this does not mean that you are only able to use that specific version of TouchGFX to further develop your application.

TouchGFX is backwards-compatible by design and in most cases it requires a very simple procedure to make an older versioned application work with a new version of TouchGFX.

Simply open your newly installed version of TouchGFX Designer and try to open your older application either through recent applications or the Open dialog. You will be greeted with the following popup:



Update popup

CAUTION

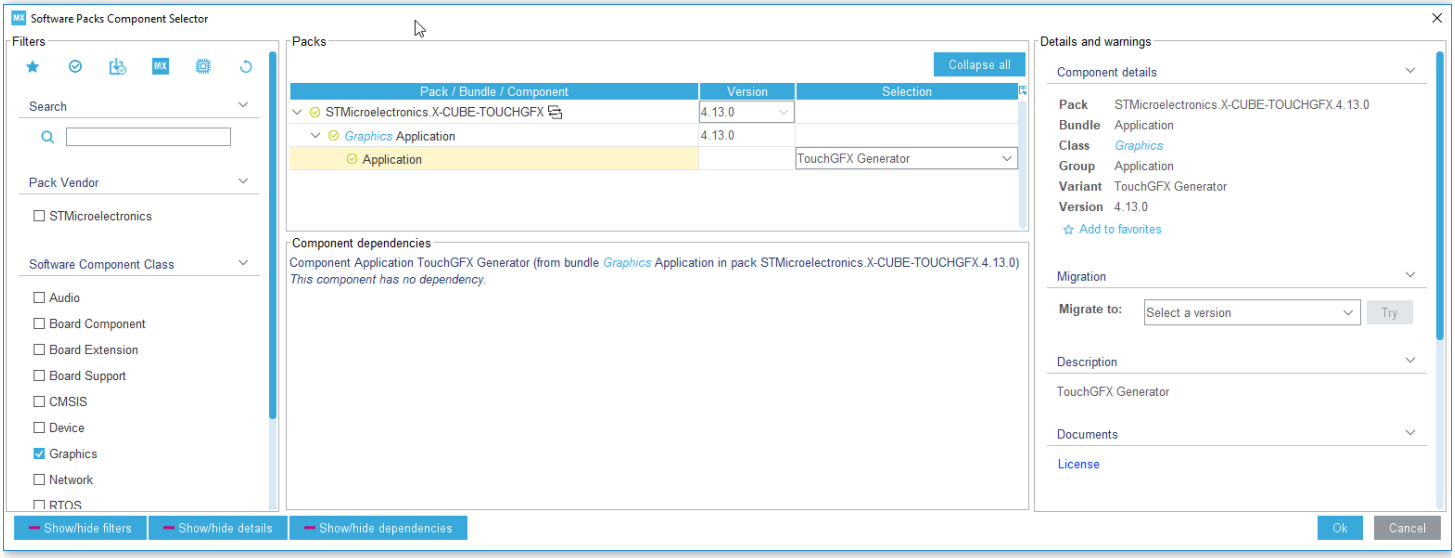
As stated in the popup, it is recommended to always have a backup of your application before running the updater.

Press 'Yes' and the updater will start. After it is finished, the application will open as normal in TouchGFX Designer and you are ready to use the new version.

In rare cases, you will have to do some manual changes to an application to make it fully updated to a new TouchGFX version. Consult the Known Issues section if you are having additional problems updating an application to a new version.

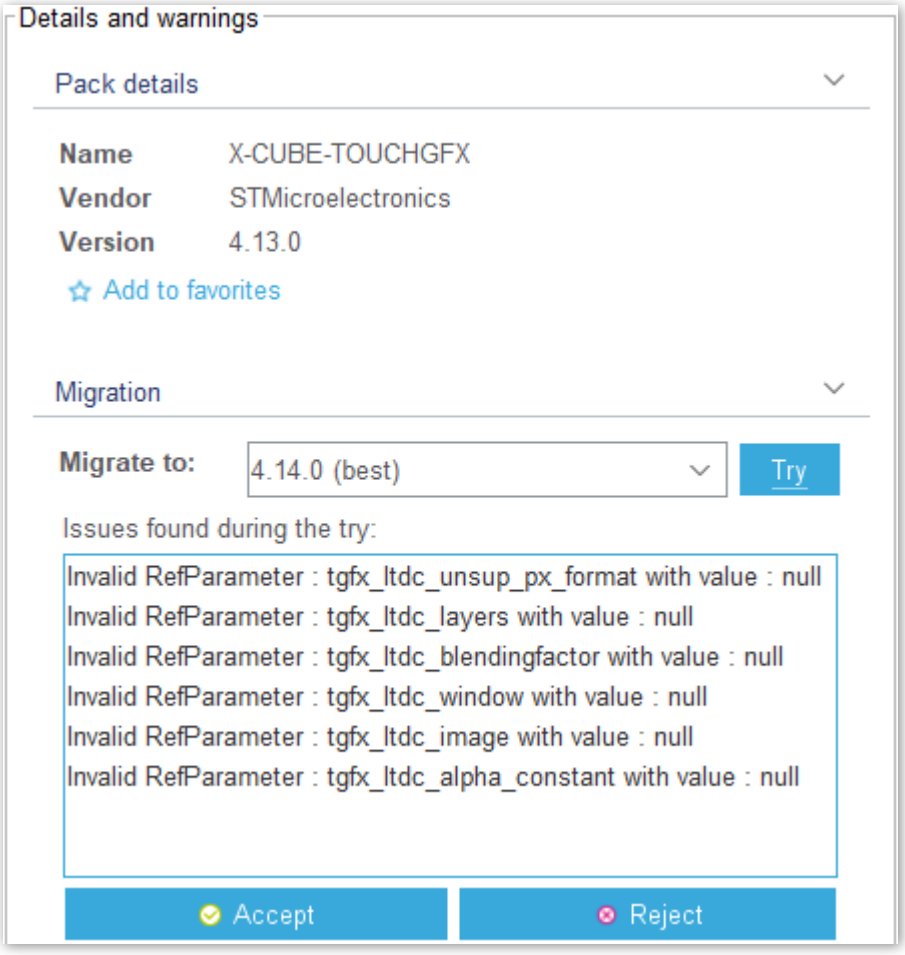
Update TouchGFX Generator

After installing X-Cube-TouchGFX-4.14.0 in CubeMX 6.0.0 or later by following Installing TouchGFX Generator in CubeMX open the project and navigate to `Software Packs` -> `Select Components` (or press `ALT + U`)



Software Pack Component Selector

In the details and warnings pane select the version to migrate to and press try. CubeMX will perform a trail migration and present the result, some refparameters are not able to be migrated to the new version and therefore will be presented as null values. Select accept to save the migration and then press OK to close the Software Pack Component Selector window.



Software Pack Component Selector : Migrate Parameters for TouchGFX Generator

Now TouchGFX Generator is migrated to the new version and the changes are saved in the .ioc file. Please validate the configuration of TouchGFX Generator before generating code with CubeMX. After generating code with CubeMX open TouchGFX Designer and validate the ui before generating code from TouchGFX Designer.

⚠ CAUTION

- **The migration is a two step process to update TouchGFX to the new version. It requires code generation from both CubeMX and TouchGFX Designer.**
- **C++ code provided by TouchGFX Generator is first written to disk after the Generate Code button in CubeMX has been pressed.**

Version: 4.16

Getting Help

There are numerous ways you can seek help on TouchGFX relative issues in case you get stuck or needs more information in certain areas. First of all you should look through this TouchGFX Documentation. This is the greatest source of knowledge to TouchGFX related issues and cover a lot of areas. If this does not answer your question you can seek for further information at following:

- **TouchGFX Community**
  Public forum site in ST Community dedicated for GUI/TouchGFX related topics. Here you will find a lot of questions and answers, some tutorial and videos covering all kind of development issues. And with an myST account (easy and free registration) you can ask specific technical questions and get answers.
- **Webinar and videos**
  - MOOC TouchGFX Webinar (Training videos)
  - Other TouchGFX Webinars
  - ST Youtube channel (TouchGFX playlist)
- **Online Support**
  Online support site for support requests via web form. Can be used when it was not possible to find any information in the community nor the TouchGFX documentation.
- **Your local ST support channel**
  Your local ST contact can either help you directly or get required back-office support
- **TouchGFX Implementer (ST partner)**
  Get assistance in any stage of your UI project from one of our dedicated and highly skilled TouchGFX Implementers (scroll down the webpage for a list of implementers). Covering display solutions ready to embed into your project, and services within graphical design, hardware development and production, and software development. Their innovative approach as well as extensive knowledge about TouchGFX and STM32 microcontrollers make them your ideal partner for your next embedded product. Find your implementer and go from idea to end-product fast and easy.
- **ST blog**
  Technical news on STM32 graphics and TouchGFX.
- **ST Graphic website**
  Website covering ST Graphics.

Version: 4.16

Known Issues

This article lists the issues that are known to be present in all TouchGFX versions, along with potential workarounds. Also, if there are any specific upgrade steps you need to perform to upgrade TouchGFX to a certain version, these will be mentioned. Note that if your current version is several releases old, you need to perform the upgrade steps for all the releases up to the new one.

Issues with CubeMX 6.1.0 and CubeProgrammer 2.6

As of version CubeMX 6.1.0 EWARM projects generated by CubeMX do not work with X-CUBE-TOUCHGFX because of a wrong setting for "C/C++ Compiler" / "Language" option which was changed from "Auto" to "C++" causing compilation errors. This issue will be fixed in CubeMX 6.1.1. In the mean time, changing the option back to "Auto", manually, will solve compilation issues but will be reverted upon code generation from CubeMX.

A bug in CubeProgrammer 2.6 related to how external loaders (`.stldr`) are referenced breaks Makefiles for all existing Application Templates (AT) and also prevents the "Run Target" feature in TouchGFX Designer from functioning correctly. This issue also extends to user projects based on current versions of the ATs. Application templates will receive an update to compensate for this bug and will work for both CubeProgrammer 2.5 and 2.6. If you've got a project based on an AT that does not work with CubeProgrammer 2.6, you can make the following modifications to add support. Users must execute `STM32CubeProgrammer_CLI.exe` from within the `bin` folder when making a reference to an external loader. Generally, speaking:

- `cd` into the `bin` folder of the STM32CubeProgrammer installation folder.
- Execute the command to program the connected target with a *relative* reference to the `.stldr` file.

```
@cd "$(st_stm32cube_programmer_bin_path)" && ./$(stm_stm32cube_programmer_exe) -c port=SWD
```

TouchGFX 4.15.0

MCU support

While Cortex-M33 is fully supported by TouchGFX, "Software Packs" (TouchGFX Generator, among others) cannot be enabled in the current verison of CubeMX (v6.0.1) for multi-context MCUs until support is added in CubeMX. Disabling "Trust Zone" for Cortex-M33 based MCUs, thus limiting the

MCU to a single context, will allow you to enable TouchGFX Generator. TrustZone should be enabled manually in User Code sections.

TouchGFX 4.14.0

ARMCLANG Support

While TouchGFX now provides an ARMCLANG (ARM compiler v6.x) library for *Cortex-M0*, *Cortex-M4f*, *Cortex-M7* and *Cortex-M33*, CubeMX is not able to generate projects that enable the ARMCLANG compiler (ARM Compiler v6.x). This requires users who wish to use the compiler in their projects to select the compiler manually from the project options in Keil uVision.

If configuring the FreeRTOS middleware from within CubeMX, any generated project using ARMCC (ARM compiler v5.x) will have FreeRTOS *portable* files that are not compatible with ARMCLANG; And these have to be replaced manually. Whenever "Generate code" is run from within CubeMX any manual changes will be overwritten and it would be wise to keep the project under version control (git, etc.) to undo these particular changes.

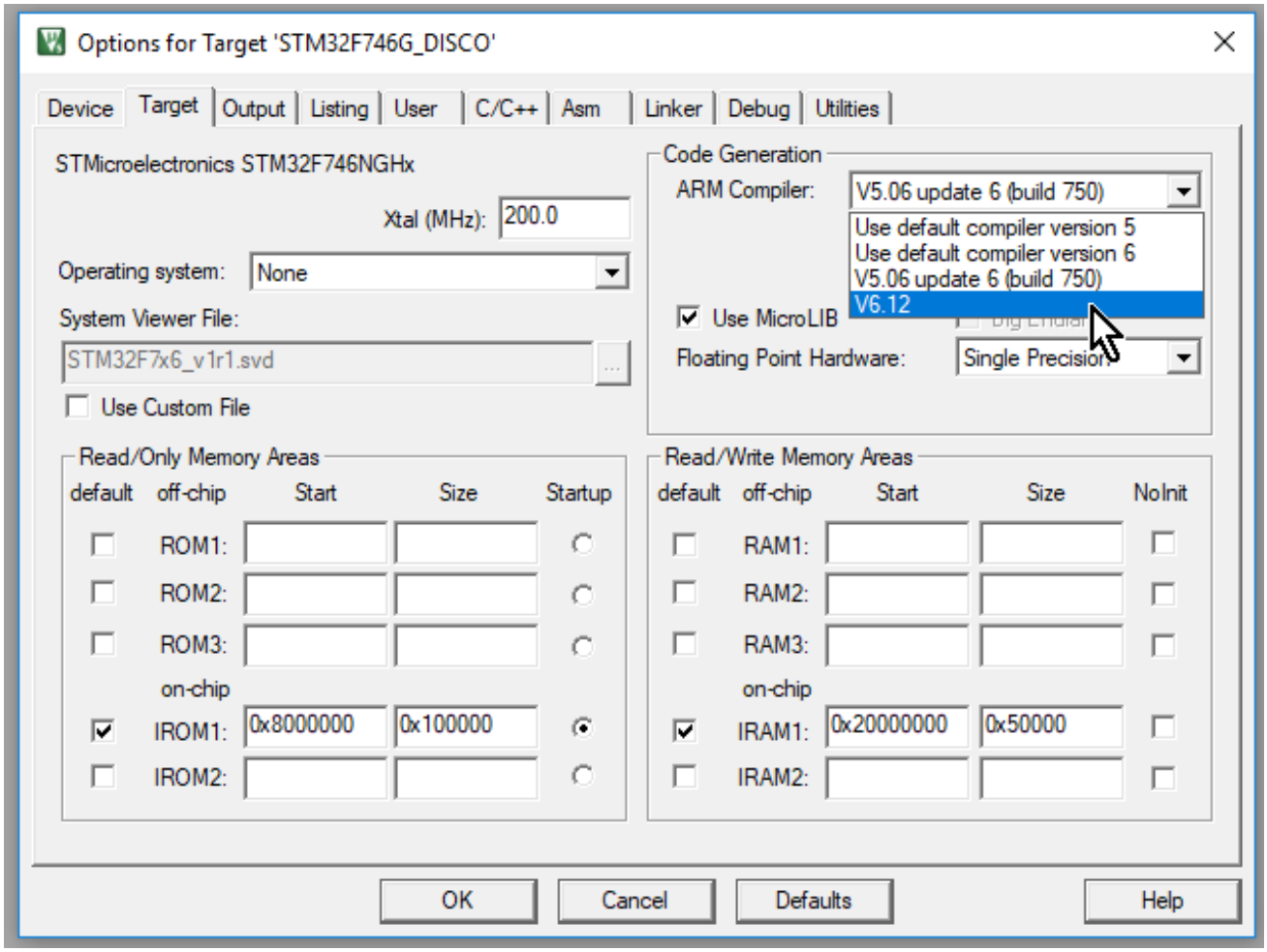
In summary. Since CubeMX can only generate ARM Compiler v5.x compiler projects, users have to modify the following every time code is generated from CubeMX unless they keep their project under version control.

1. Select ARM Compiler v6.x in project options.
2. Link with the ARMCLANG library instead of the ARMCC library (configured by CubeMX).
3. If configuring FreeRTOS from within CubeMX, then the FreeRTOS portable files should be taken from the `portable/GCC` folder rather than `portable/RVDS` (default for ARM Compiler v5.x) in order to be compatible with ARM Compiler v6.x.

Workflow

The following workflow describes how to use v6.x ARM Compiler from Keil uVision with CubeMX generated projects and a TouchGFX ARMCLANG library.

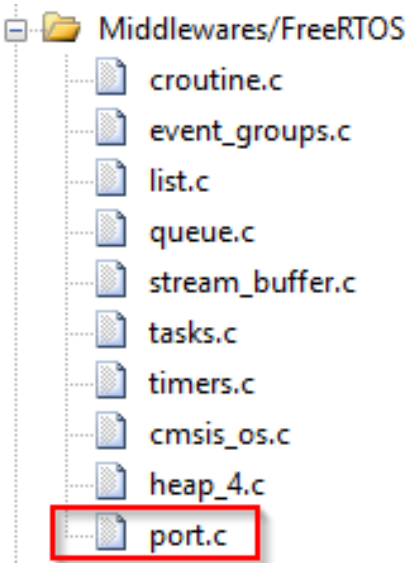
1. Select ARMCLANG (v. 6.x) in Keil uVision.



ARMCLANG Support

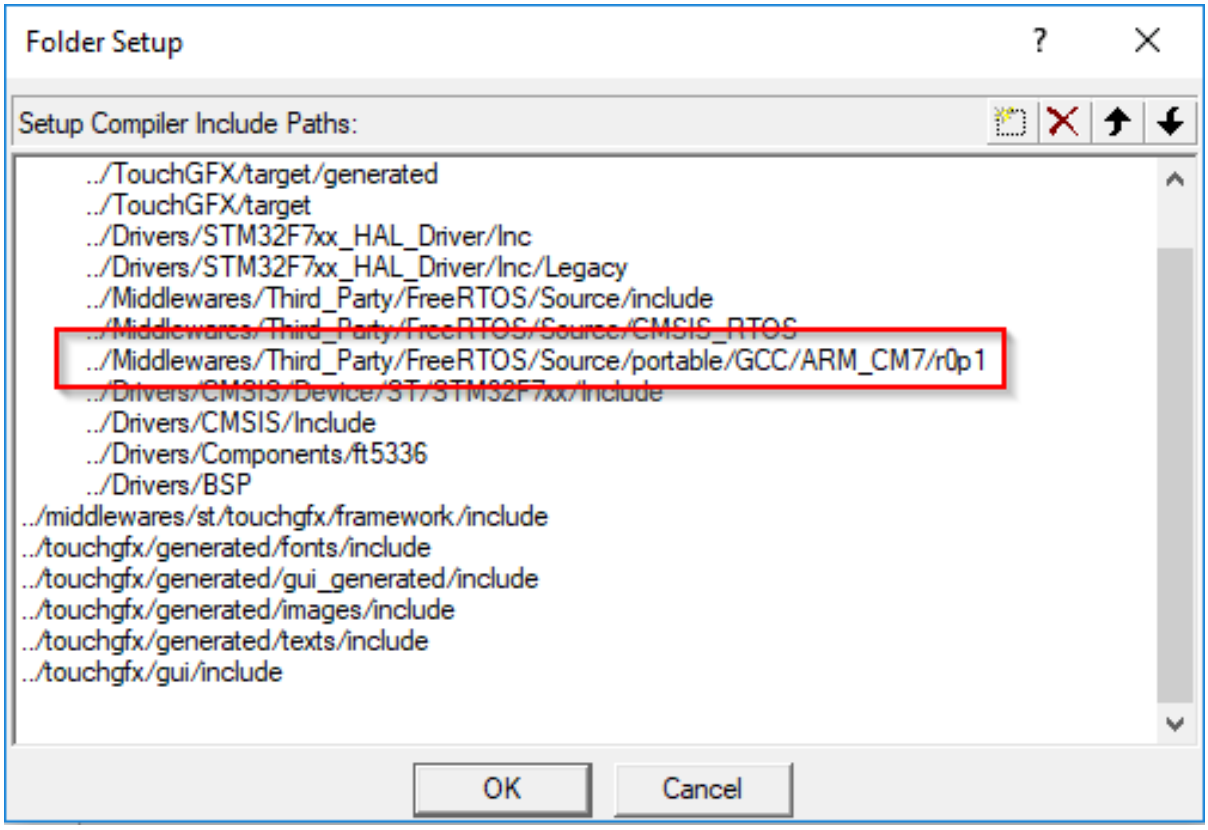
2. If you're configuring FreeRTOS from CubeMX, CubeMX will generate the wrong portable files and configure your project to use those. You have to manually replace these with the ones (from https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/6199b72fbf57a7c5b3d7b195a3bd1446779314cd/portable/GCC (`port.c` and `portmacro.h`) or download a FreeRTOS release and find the files in there.

   Replace `port.c` :



port.c

Change your include path settings to include `portmacro.h` from the `portable/GCC` folder (in this case for Cortex-M7):



Portable include path

3. TouchGFX designer Post-Generate step during "Generate Code" will automatically insert the correct library based on the compiler version you've chosen.

TouchGFX 4.13.0

Bugs

Font Converter

The FontConverter tool would generate glyph pixel data for unicodes that were a part of a rule in the font, regardless of that glyph being used in an actual text in the application. This led to several megabytes, potentially, of additional glyph pixel data. New FontConverter tools (windows and linux) that no longer generate pixel data for glyphs that aren't in use by the application can be found here:

- fontconvert_fix.zip

Extracting this file at the root of your 4.13.0 installation will update the fontconverter binaries inside

`touchgfx\framework\tools`

Additional Compiler Support

A library built with ARMCLANG compiler (v6.12) can be found here:

- touchgfx_core_clang.zip

Extracting this file at the root of your 4.13.0 installation will place the library `touchgfx_core_clang.lib` inside.

`touchgfx\lib\core\cortex_m7\Keil`

Backwards Compatibility

Deprecated Features

The following deprecated features have been removed. If you have referenced them in your code, you may need to rewrite parts of your application:

- Definition of deprecated `TRANSPARENT_COL`
- `Drawable::getType()`
- `HAL::blitSetTransparencyKey()`
- `HAL::registerTextCache()`
- `HAL::cacheTextString()`

TextureMapper is Disabled by Default

The TextureMapper is disabled by default to reduce the code space used by TouchGFX. TouchGFX designer will insert code to enable texture mapper in all new project.

If you are migrating an old project to TouchGFX 4.13 and you are updating to TouchGFX 4.13 this is handled by TouchGFX Designer.

If you are updating manually then you need to insert code to enable the TextureMapper. Otherwise any TextureMapper will not draw on the screen.

Read more here: Configuring TouchGFX Features.

HAL SDL1 incompatible

Two functions were moved from the TouchGFX library code to the `HALSDL2.cpp`. This makes no difference for applications that uses the `HALSDL2.cpp` as HAL for the Windows simulator.

If you have a old application (before TouchGFX 4.8.0) your simulator is maybe using HALSDL (not 2). This simulator HAL is not included in TouchGFX anymore. The HALSDL is missing the two functions that were previously included in the TouchGFX library. You need to add them manually to `HALSDL.cpp`:

HALSDL.cpp

```
void simulator_enable_stdio();
void simulator_enable_stdio()
{
    HWND consoleHwnd = GetConsoleWindow(); // Get handle of console window
    if (!consoleHwnd)                      // No console window yet?
    {
        HWND activeHwnd = GetActiveWindow(); // Remember which window is active

        AllocConsole();                   // Allocate a new console
        consoleHwnd = GetConsoleWindow(); // Get handle of console window

        FILE* dummy;
        freopen_s(&dummy, "CONIN$", "r", stdin); // Redirect stdin/stdout/stderr to the ne
        freopen_s(&dummy, "CONOUT$", "w", stdout);
        freopen_s(&dummy, "CONOUT$", "w", stderr);

        SwitchToThisWindow(activeHwnd, true); // Switch back to the originally active wind
    }
    if (consoleHwnd)
    {
        ShowWindow(consoleHwnd, SW_SHOW); // Show/hide it!
    }
}
void simulator_printf(const char* format, va_list pArg);
void simulator_printf(const char* format, va_list pArg)
{
    // Create a console window, if window is visible.
    simulator_enable_stdio();
    if (GetConsoleWindow()) // Only print if we have a console window
    {
        vprintf(format, pArg);
    }
}
```

TouchGFX 4.12.3

Backwards compatibility

Screen transitions

Earlier versions are redrawing the entire screen after a screen transition is completed. This additional redraw caused performance issues on some slow platforms. If you require this redraw for some reason, you need to invalidate the relevant part of the screen, e.g. by calling `container.invalidate()` in the `Screen::afterTransition()` virtual of the Screen transitioned to.

Binary Fonts

The format of the binary fonts has changed because kerning data is now also included in the binary fonts. Binary font files needs to be regenerated, old files will not work correctly. Depending on how your Makefiles are setup, this is normally done by regenerating all assets (e.g. `make -f simulator/gcc/Makefile clean assets`).

TouchGFX 4.11.0

Backwards compatibility

In `touchgfx/framework/include/touchgfx/lcd/LCD.hpp` we have removed an include of the file `touchgfx/hal/HAL.hpp` that was inserted by mistake in an earlier version. This may give you a compile error in a file where you have included `LCD.hpp` and also make use of the `HAL.hpp` content. The solution is to also include `touchgfx/framework/include/touchgfx/hal/HAL.hpp` in your file.

TouchGFX 4.10.0

Upgrading from 4.9.x

From version 4.10.0 TouchGFX runs exclusively on STM32 MCUs.

You need to do the following if updating an old project.

Add the highlighted code below at application startup to inform TouchGFX that you are running on STM32 hardware. A suitable location is the end of the `hw_init()` function in `BoardConfiguration.cpp`

BoardConfiguration.cpp

```
void hw_init() {
    ...
    //Enable CRC engine for STM32 Lock check
    __HAL_RCC_CRC_CLK_ENABLE();
}
```

Backwards compatibility

Unused file `\touchgfx\framework\include\touchgfx\canvas_widget_renderer\RGBA8.hpp` removed.

Project Updater Issue

IAR and Keil project updaters invoked from TouchGFX Designer do not respect custom file level optimization set in the respective IDE.

TextArea and ChromART (DMA2D)

Rendering of TextAreas with ChromART (when TextArea is the top most element / last to be drawn) cause a premature unlocking of the framebuffer and subsequently a premature completion/transfer of the current frame to the display. Once `endFrame()` is called by TouchGFX all drawing operations, including DMA operations, should already be completed. Due to a breach of contract by TextArea in how to appropriately protect the framebuffer, DMA operations are allowed to continue even after returning from `endFrame()`.

The contract for a widget is to either:

1. Lock the framebuffer (returns a pointer to framebuffer).
2. Draw something in the framebuffer.
3. Unlock the framebuffer.

Or to use DMA operations, in which case syncronization of the framebuffer is handled automatically.

TextArea, in 4.10.0, mixes the two procedures which can cause glitches if it is the top most element (last to be drawn) of the current screen. The bug can be fixed by manually guarding `endFrame()` with the following override of `endFrame()` (based on F4 HAL). It ensures that `endFrame()` does not return if ChromART operations are still being processed.

```
void STM32F4HAL::endFrame()
{
    if (dma.isDMARunning())
    {
        OSWrappers::tryTakeFrameBufferSemaphore();
    }
    HAL::endFrame();
}
```

TouchGFX 4.9.0

Upgrading from 4.8.0

With the introduction of Application Templates, which essentially separates board support packages from the core framework, we have removed a lot of low-level drivers and other files from the *touchgfx* folder in 4.9.0. These files are now provided by application templates instead. However, this means that you cannot upgrade to 4.9.0 by just replacing the touchgfx folder, since that would likely lead to some BSP files missing. Instead, the TouchGFX Designer is able to perform the upgrade automatically. The upgrade can be done in two different ways, and you will need to decide which one is most suitable for you.

> **⚠ CAUTION**
>
> **Please make sure you have a backup of your project before upgrading**

Method 1: retain original file structure

This method is done by simply opening your project in the new 4.9.0 Designer. TouchGFX Designer will prompt you whether you want to upgrade, and by clicking OK, TouchGFX Designer will apply the necessary changes. TouchGFX Designer will perform the following operations:

1. Unpack the new reduced touchgfx folder into your application, and modify your TouchGFX path to match this
2. Download and unpack all the files we have removed from the touchgfx folder, so that your project still compiles

This approach will leave pretty much everything as they used to be, so if the old file structure suits your project, this is the easiest choice. The main drawback is that you will not have the benefit of the image converter speedup (by working on image folders instead of individual files). But you can modify your makefile manually to use this approach though.

Method 2: import into new template

Using this method you can transition your project into the new template-based file organisation. To achieve this, you must first let the Designer upgrade your project using Method 1 above. Then, create a new project using the appropriate application template (simulator, or one matching your eval board). With this new project opened in the Designer, go to the top menu and click "Edit -> Import GUI". In the dialog box point out the .touchgfx file of your project. The Designer will then automatically import all the UI files, including assets, into the newly created project. If you have added additional code outside of the gui folder, you would need to manually copy those files over to the new project.

Method 3: Manual upgrade without Designer

If you are not using the Designer, you can perform the upgrade by:

1. Replacing the touchgfx folder used by your project with the one from the 4.9.0 installation.
2. Download this zip and unpack it into the touchgfx folder, restoring the removed files.

Version: 4.16

Changelog

Version 4.16.0

- Release date: December 15th, 2020

- New TouchGFX Designer Features:

  - New widget: Gauge.
  - Added new trigger "When Screen Transition Begins".
  - Renamed "When Screen Entered" Trigger to "When Screen Transition Ends".
  - Added new "Set Language" Action.
  - Adding one image via the Image Picker on a widget now selects it.
  - Added link to shortcuts documentation in 'Help->Keyboard Shortcuts'.
  - Added a better code editor for 'Execute C++ code' actions.
  - Added new Block Transition for 'go to screen' actions.
  - Improved usability/user experience of 'Add Widget' menu.
  - Added preliminary support for LCDNemaP.

- Bugfixes in TouchGFX Designer:

  - Fixed generated mainBase.cpp compilation failing on unix by adding: '#include <string.h>'.
  - Fixed image file validation reporting an image to be erroneous when file name contained underscores.
  - Fixed being able to create an application with a space in its name.
  - Fixed Dynamic Graph callback handlers being genereated twice in Custom Containers.
  - Fixed Dynamic Graph crashing the TouchGFX Designer in certain edge cases.
  - Fixed labels in Dynamic Graph not updating when switching between single use and resource text.
  - Fixed labels in Dynamic Graph not updating their position when changing the size og the Dynamic Graph.

- New TouchGFX Core Features:

  - New invalidation algorithm for improved performance.
  - Added new container Gauge.
  - BoxWithBorder is now a subclass of Box.

- LCD16 and LCD8 fillRect functions now write 32/16 bits at a time for improved performance.
- Added CanvasWidget::resetMaxRenderLines().
- Moved LCD2shiftVal(), LCD2getPixel() and LCD2setPixel() to class LCD2bpp.
- Moved LCD4getPixel() and LCD4setPixel() to class LCD4bpp.
- ScrollableContainer::setScrollbarsPermanentlyVisible() now takes a boolean argument to allow disabling permanently visibility.
- TextureMapper and ScalableImage are now each a subclass of Image.
- Added AnimatedImage::setEndBitmap().
- Added AbstractClock::getCurrent12Hours and AbstractClock::getCurrentAM().
- Unicode::itoa() and Unicode::utoa() can handle radix up to 36.
- Added AnalogClock::setAlpha() and getAlpha().
- ScrollableContainer::setScrollbarsPermanentlyVisible() now takes a boolean argument to allow disabling permanently visibility.
- Added TextArea::resizeHeightToCurrentTextWithRotation().
- Added Drawable::setWidthHeight() to set width and height in one call given by (width,height), Drawable, Bitmap or Rect.
- Added Drawable::setXY(Drawable&) to set top left corner of a Drawable at the same position as a nother Drawable.
- Added Drawable::setPosition(Drawable&) to place a Drawable at the same position as another Drawable.
- Added Color::getRGBFrom24BitHSV() and Color::getHSVFrom24BitRGB() to convert between (hue, saturation, value) and (red, green, blue).
- Added Color::getColorFrom24BitHSV() and Color::getHSVFromColor() to convert between (hue, saturation, value) and colortype.
- Added Color::getHSVFromHSL() and Color::getHSLFromHSV() to convert between value and luminance
- Added Color::getRGBFrom24BitHSL() and Color::getHSLFrom24BitRGB() to convert between (hue, saturation, luminance) and (red, green, blue).
- Added Color::getColorFrom24BitHSL() and Color::getHSLFromColor() to convert between (hue, saturation, luminance) and colortype.
- PainterBW now supports alpha.
- SnprintFloat(s) now handles NaN ("nan") and Inf ("inf").
- SnprintFloat(s) now defaults to 6 digits after the decimal point instead of 3. ANSI C says "If the precision is missing, 6 digits are given".
- Added Circle::setPixelCenter().
- Added updateValue(), setEasingEquation(), setValueSetAction() and setValueUpdatedAction() to progress indicators, to allow smooth transition from one value to another value.

  - Added SwipeContainer::getSelectedPage.
  - Added BlockTransition.
  - Added CacheableContainer::setSolidRect() and getCacheBitmap().

- Bugfixes in TouchGFX Core:

  - Corrected spelling of getGraphAreaPaddingRight().
  - In rare occations the TextureMapper would draw some scanlines twice.
  - AnimationTextureMapper, ZoomAnimationImage, MoveAnimation and FadeAnimation all work if steps=0 and will notify of animation ended on the last animation step.
  - Fixed bug in Keyboard when dragging away from pressed key, and releasing.
  - Slider::getIndicatorMin() would return indicator max instead of min.
  - ImageConvert would corrupt the heap (and most likely crash) on BMP images.
  - Fixed the border of images drawn in bilinear mode by the texture mapper.
  - Fixed BoxWithBorder with a very wide border when alpha<255.
  - Fixed ProgressIndicators' range and value to all be type 'int'.

- Deprecated TouchGFX Core Features:

  - All deprecated functions from 4.15 and earlier has been removed.
  - Removed ST1232TouchController.
  - ZoomAnimationImage::setDimension is deprecated. Use setWidthHeight.
  - Drawable::setXY and Drawable::setPosition are no longer virtual functions. Only setX, setY, setWidth and setHeight are virtual.
  - AbstractProgressIndicator::getRange methods with int16_t& parameters.

Version 4.15.0

- Release date: October 5th, 2020

- New TouchGFX Designer Features:

  - New widget: Dynamic Graph.
  - M0 platforms now have all texture mapper features disabled by default.
  - Added support for Wipe-transition.
  - Overhauled the Add Widget Menu (is now found by clicking a button in the top left of the canvas toolbar or pressing 'A' on the keyboard): added search functionality, resizability.
  - The whole bottom status bar can now be clicked to bring forward the log.
  - The status bar now turns red on error and green on success.

    - Moved zoom functionality to top right of toolbar.
    - Added button to center canvas in the viewport.
    - Added keyboard shortcuts for zoom in (Ctrl + '+'), zoom out (Ctrl + '-') and reset zoom (Ctrl + 0).
    - It is now possible to lock the position of a widget, which also disables selection on canvas (useful for background images, boxes).
    - A black Box is now always generated in base views for a better experience when inserting widgets on an empty canvas.
    - Many tooltips have received a visual overhaul and also display keyboard shortcuts.
    - It is now possible to select if fonts should be output in mapped or unmapped format.
- Bugfixes in TouchGFX Designer:
    - Designer would sometimes crash when importing a project with identical fonts, bitmaps.
    - Generated code in FrontendHeapBase.hpp would include multiple copies of the same transition header file.
    - It was possible to drag and drop widgets into a scroll-list or scroll-wheel in the treeview.
    - "Choose button key" for interactions were cleared when adding widgets.
    - Wrong default version of packages were sometimes chosen.
    - Packages would be downloaded even if they already existed on disk.
- New TouchGFX Core Features:
    - New font format that allows most font data to be stored in unmapped flash.
    - Improved partial framebuffer block transfer algorithm.
    - Added new prototypes to OSWrapper: isVSyncAvailable() and signalRenderingDone() for use on platforms that cannot be block in waitForVSync.
    - touchgfx::SingleBlockAllocator is removed, use touchgfx::ManyBlockAllocator<block_size, 1, bytes_per_pixel>
    - New method on HAL, enableDMAAcceleration(), to disable hardware accelerations.
    - Added TextureMapper::invalidateBoundingRects().
    - ImageConvert is using updated nlohmann-json 3.9.1. Generated images look the same.
    - Added single stepping in simulator. Pressing F9 will start/stop execution. Pressing F10 will execute one tick. This can also be controlled using HALSDL2::setSingleStepping(), HALSDL2::isSingleStepping() and HALSDL2::singleStep().
    - Added new Graph classes.
- Bugfixes in TouchGFX Core:
    - Armenian (and some Cyrillic) characters were written right-to-left.
    - Quick touch and release after swipe could result in an extra unwanted GestureEvent.

- Very large glyphs that required only partial redraw would not render correctly.

- Deprecated TouchGFX Core Features:

- Update procedure:

  - For this release additional steps might be needed. Please refer to the Known Issues article for details: https://support.touchgfx.com/docs/miscellaneous/known-issues

Version 4.14.0

- Release date: July 2nd, 2020

- New TouchGFX Designer Features

  - Updated all links to direct to new documentation website.
  - Added support for the SlideMenu widget.

- Bugfixes in TouchGFX Designer:

  - FrontendHeap.hpp model declared before app to prevent potential errors.
  - Fixed UI Template selector not comparing available color-depths correctly.
  - Fixed .touchgfx.part file version not being checked before loading.
  - Fixed code generation of included painter when selecting a specific format for an image, L8 images and all 8 bit LCDs supported.
  - Fixed application name validation when creating new application.
  - Fixed error when dragging container type widgets inside themselves via treeview.

- New TouchGFX Core Features:

  - HAL::lockDMAToPorch default value is set to false instead of true.
  - Font::getDataFormatA4() is now called Font::getByteAlignRow() as it may be set for 2bpp fonts and 1bpp fonts as well as 4bpp fonts.
  - GestureType is now called GestureEventType for consistency. GestureType has been deprecated and will be removed soon.
  - Added Version.hpp with macros for current version of TouchGFX.
  - ImageConvert supports image files starting with a digit.
  - ImageConvert output .cpp files with "image_" prefix.
  - ImageConvert built-in help improved.
  - ImageConvert can write an application.config template file.

- Added Unicode::strncmp_ignore_whitespace which ignores whitespace and not just spaces.
- FontConvert is using updated freetype 2.10.2. This results in slightly nicer and better aligned characters.
- ImageConvert is using updated libpng 1.6.37. Generated images look the same.
- Added setDurationSpeedup, getDurationSpeedup, setDurationSlowdown and getDurationSlowdown to ScrollableContainer. This allows better control of the number of animation steps to use on a swipe gesture.
- Extended SlideMenu widget with possibility of not needing a button.
- Using a colortype variable as a number will automatically cast it to uint32_t instead of uint16_t.
- Added ARMCLANG-6.x support in Keil project.
- Support for Cortex-M33.

- Bugfixes in TouchGFX Core:

  - BoxWithBorder would not set borderColor and borderSize in constructor.
  - Several fixes in Unicode::snprintf(): Removed limit of 64 characters for format string. Do not force sign character on %o, %x and %X. No zeroes prefixed on %05c. Correct handling of %c with 0 as value. Sign on %s (%+s, %0s) handled properly.
  - ScrollListWithCenterSelect could crash if size was changed.
  - ScrollList with snapping would not report the correct clicked item.
  - ScrollList without snapping, non-circular could report wrong item.
  - ScrollList repects padding when item is clicked.
  - CWR Painters with setColor(color,alpha) now only accepts color. Use setAlpha(alpha) to set the alpha.
  - Cached bitmaps was not 32bit aligned with an unenven number of dynamic bitmaps.
  - LED.hpp no longer includes lpc_types.h.

- Deprecated TouchGFX Core Features:

  - Deprecated functions are now marked deprecated so the compiler can issue a warning on these functions. Deprecated functions will be removed in the future.
  - Removed definition of Unicode::EMPTY.
  - Definition of PI moved from Math3D.hpp to Types.hpp.
  - The 'pi' defined in EasingEquation.hpp has been replaced by PI.
  - ImageConvert no longer supports -f, -o and -header.
  - Image::hasTransparentPixels removed.

- Update procedure:

- For this release additional steps might be needed. Please refer to the Known Issues article for details: https://support.touchgfx.com/docs/miscellaneous/known-issues/#project-updater-issue

Version 4.13.0

- Release date: December 12th, 2019
- New TouchGFX Designer Features:
  - Support for ".touchgfx.part" files. These can be used for external input to a project (e.g. CubeMX integration).
  - Select supported image formats for the TextureMapper by navigating to the "Framework Features" category in the "Config" tab.
  - It is now possible to override the Generate Assets, Post Generate, Compile Simulator, Run Simulator, Compile Target and Flash Target commands from within the Designer.
  - Improved zoom/scroll on canvas and auto scrolling is now enabled when dragging widgets to the outskirts of the canvas.
  - The Generate Code button shows if the current code is up to date, by displaying a blue dot if it is out of date.
  - The File, Edit and Help menus can now be opened with the shortcuts Alt + F, Alt + E, and Alt + H.
  - The Help menu includes a direct link to the TouchGFX Help Center.
  - Modify text configurations through the "Config" tab
  - The detailed log can be floated or docked within the Designer and can be opened with the shortcut Alt + L.
  - The Designer version is shown in the title bar of the window.
  - The Recent Projects list now displays the full path to a project instead of just the project name.
  - Performance improvements when moving widgets on the canvas.
  - Performance improvements when rendering rows in the image manager.
  - Performance improvements when reordering items in the widget tree.
- Bugfixes in TouchGFX Designer
  - Fixed a bug where having delay and button callback interactions could cause faulty generated code.
  - Fixed a bug where using the TouchGFX CLI to generate projects did not properly include used TouchGFX assets.
  - Fixed a bug where the canvas buffer for a screen would not be properly updated in some cases.
  - Fixed a visual bug where the error message displayed on the startup window would not disappear when retrying a download.

- Fixed a bug where the function name of a call virtual function interaction was not properly validated.
- Fixed a bug where progress indicator in some cases would not render correctly on the Designer canvas.
- Fixed a bug where creating a new project and not saving would cause the default typographies to disappear when reloading the project.
- Fixed a visual bug where the text manager would seemingly keep focus on wrong cells.
- Fixed a bug where the properties tab for a widget would not properly display errors.
- Fixed a bug where generating button click handlers would sometimes yield empty if/else statements.
- Improved search fields in startup window.
- Fixed a bug where copying a shape widget and editing a point in one of them would cause the change to happen for both.
- Fixed a bug where scrolling by using the mouse wheel while changing fonts from within the typography picker would close the popup.
- Fixed a bug where loading project containing a go to screen action would not load correctly.
- Fixed a bug where expanding/collapsing a node in the widget tree view would also select the node.
- Fixed a bug where some values were imported incorrectly when importing a UI into an application.
- Fixed a bug where navigating through folders in the image picker was faulty.
- Fixed a bug where the order of pages in a swipe container was presented wrongly.
- Fixed a bug where the rendering of the texture mapper on the Designer canvas was faulty.
- Fixed a bug where validation of a removed interaction source was faulty.
- Fixed a bug where renaming a folder with subfolders located under assets/images could crash the Designer.
- Fixed interactions on RadioButtons generating duplicate code.
- Fixed a bug where dragging the same image from file explorer to the Designer twice would result in faulty behavior.
- Fixed a bug where radio button interactions would sometimes generate duplicate code.
- Fixed a bug where overriding a canvas buffer could result in a newline missing in the generated code.
- Fixed a bug where the Designer would generate faulty code when using Turkish region format.
- Fixed a visual bug where having long text in custom action/trigger text boxes would cause unwanted shifts in the UI.
- Fixed a crash bug where a sequence of steps after deleting the last custom container in an application would cause the Designer to crash.

  - Fixed a bug where copying a custom container instance from a screen to a custom container definition did not work.
  - Fixed the Matching UI Templates filter not working as intended.
  - The Designer now supports application names that include periods.
  - Fixed a bug where changing a slider with a style from horizontal to vertical would result in the style not being correctly set.
  - Fixed a bug where changing the font of a typography would visually not display the correct font name some places in the UI.

- New TouchGFX Core Features:

  - TextureMapper performance improvement. Decreased rendering time in the range of -10% to -60% depending on the image format, rendering algorithm, hardware setup and image layout.
  - Texture Mappers are disabled by default, must be enabled before use. Read more about this feature here: https://support.touchgfx.com/docs/miscellaneous/known-issues/#texturemapper-is-disabled-by-default
  - Added simple string printing for debugging in all LCD types. See: https://support.touchgfx.com/docs/development/ui-development/working-with-touchgfx/debugging/#using-the-debugprinter
  - Font caching now supports GSUB tables as used in Hindi. See: https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/font-cache/#caching-gsub-tables
  - Updated arm gcc compiler to version 7.3.1 2018q2.
  - Updated gcc compiler to version 7.3.0.
  - Updated mingw environment with latest version of packages.

- Bugfixes in TouchGFX Core:

  - TextureMapper: blending on edges corrected/improved in Bilinear mode.
  - TextureMapper: minor image quality improvements in Nearest Neighbor mode.
  - Bugfix for text order in arabic text "12:34" which would previously render as "34:12" in RTL.

- Deprecated TouchGFX Core Features:

  - Removed definition of deprecated TRANSPARENT_COL.
  - Removed Drawable::getType().
  - Removed HAL::blitSetTransparencyKey().
  - Removed HAL::registerTextCache().
  - Removed HAL::cacheTextString().

- Update procedure:

- For this release additional steps might be needed. Please refer to the Known Issues article for details: https://support.touchgfx.com/docs/miscellaneous/known-issues/#project-updater-issue

Version 4.12.3

- Release date: September 25th, 2019
- New TouchGFX Core Features (since 4.12.0):
  - Binary Fonts: Binary fonts can be used as an alternative to compiling and linking font information in to your application. The main advantages of this approach is to get a smaller application binary and get a flexibility in providing different sets of fonts with your device.
  - Font Caching: Support for caching binary fonts, suitable for loading only the required characters from a file system, when a string is displayed.
  - Binary Translations: Support for binary translations, suitable for loading translations from a file system as opposed to linking them into the application. Read more about these feature here: https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/using-binary-translations/
  - Support for non-memory-mapped flash storage for 16bpp displays, allows storage of images and fonts in e.g. inexpensive SPI flashes.
  - Recognition of Unicode sequences for Arabic ligatures Allah, Akbar, Mohammad, Salam, Rasoul, Alayhe, Wasallam and Rial Sign.
- Bugfixes in TouchGFX Core:
  - TextureMapper (bilinear) would fail to draw L8_RGB888 and RGB888 bitmaps on 24bpp displays correctly.
  - Setting a text with a wildcard in a TextArea without wildcard support in combination with RTL could cause a crash.
  - If a CacheableContainer was smaller than the associated bitmap, the size of the container would not be correct.
  - Fixed SnapshotWidget on 8bpp LCDs.
  - Fixed rendering of some Arabic ligatures.
  - Fixed rendering of some Hindi ligatures.
  - Fixed bug when applying certain GSUB substitution rules.
  - Fixed bug that binary fonts contained extra rules.

Version 4.12.2

- Release date: August 22nd, 2019
- New TouchGFX Core Features:
  - WordWrapping wide text using TextArea::setWideTextAction() now wraps at normal space as well as Unicode characters 0x200B (Zero Width Space).
- Bugfixes in TouchGFX Core:
  - Binary fonts: The fontConverter tool was not writing kerning data into binary font files when the "binary_fonts" option was specified in the application configuration. This caused texts to appear incorrect when using binary fonts.

Version 4.12.1

- Release date: August 15th, 2019
- Updated "Third Party Components.pdf" to reflect updated components
  - libpng-1.6.36
  - zlib-1.2.11
  - freetype-2.9.1
- Bugfixes in TouchGFX Designer:
  - Fixed a bug where having a delay action together with a button clicked action would result in compilation errors.
  - Fixed a bug where Canvas Buffer for a Screen was not correctly updated when adding a Canvas Widget to a Custom Container Instance.
  - Fixed a bug where an error message in the Online Applications window would get stuck.
  - Fixed faulty rendering on the Canvas in the Designer when using the Alpha value of the different Progress Indicators.
  - Fixed a bug where creating a new project, closing the Designer without saving it, and reloading the project would cause the project to have no available typographies.
  - Updated error message when trying to import an already open UI into another project to be more clear.
  - Fixed a bug where the Text Manager could have multiple foci visually in a specific circumstance.
  - Fixed a bug where the Properties tab for a Widget would not display a red border correctly, when an error is present on the Widget.
  - Fixed a bug where using the Consolas font would render incorrectly on the Canvas in the Designer after reloading a project using that font.

- Bugfixes in TouchGFX Core:
  - TextureMapper bug if Display Rotation was in use.
  - Disregard kerning data for CachedFont.
  - CachedFont did not look in font cache for the fallback character.

Version 4.12.0

- Release date: 06-07-2019
- Important upgrade information:
  - Public version of drawGlyph has been removed. Use drawString instead.
  - Using bitmap format ARGB8888 for opaque images will no longer dither to 565 but keep full 888 colors. Using ARGB8888 for non-opaque images will still dither to 565 when the opaque format is RGB565.
  - Images converted to BW_RLE will no longer fall back to BW if the BW_RLE format causes the converted image to be larger. Instead a warning will be generated by the image converter. Use the Designer (or the new configuration file) to specify BW or BW_RLE for each individual image.
- New TouchGFX Designer Features:
  - A custom container can now be nested within another custom container. This enables composing custom components into larger custom components indefinitely.
  - A custom container supports defining custom triggers and custom actions, a screen supports defining custom actions. These triggers and actions support the flow of information from one component to another component. Using such triggers and actions in interactions within the Designer enables doing more real world application behaviour without leaving the Designer. Check out the documentation for further introduction.
  - A Container can now be generated as a CacheableContainer.
  - A new "Images" tab has been added for setting up individual image configurations (Image Format, Dither Algorithm, Layout Rotation, etc.).
  - Application settings and other new settings have been relocated to the "Config" tab.
- New TouchGFX Core Features:
  - Upgraded 3rd party libraries used by framework tools. This results in much nicer looking texts.
  - Improved kerning through larger kerning table.
  - Thai fonts are now rendered better with tighter line spacing and better rendering of Sara Am in some cases.

    - Preliminary support for Hindi (Devanagari). The following GSUB tables are applied: nukt (Nukta Forms), akhn (Akhands), rkrf (Rakar Forms), cjct (Conjunct Forms), vatu (Vattu Variants), rphf (Reph Forms), pref (Pre-Base Forms), half (Half Forms), blwf (Below-base Forms), abvf (Above-base Forms), pstf (Post-base Forms), and cfar (Conjunct Form After Ro). The following are NOT currently supported: abvs (Above-base Substitutions), blws (Below-base Substitutions), and psts (Post-base Substitutions). Also, not all GSUB tables types are supported.
    - Added a new Line::updateLengthAndAngle() API.
    - Added support for partial framebuffers rendering and updates.
    - Added simple string printing for debugging.
    - Allow changing the BitmapCache after initialization.
    - New macros for setting sections names for flash programming.
    - Added Circle::updateArc() to update arc start and arc end with minimal invalidation areas.
    - Updated CircleProgress to use higher precision calculations for updates.
    - Added CacheableContainer for offscreen widgets rendering.
    - Added support for L8 graphics assets with 16bit, 24bit and 32bit palettes.
    - Added support for L8 hardware acceleration via DMA2D.
    - Added new LCD32bpp framebuffer renderer.
- Bugfixes in TouchGFX Designer:
    - ProgressIndicator is updated automatically after call to CircleProgress::setStartEndAngle(), ImageProgress::setAnchorAtZero() and TextProgress::setNumberOfDecimals().
- Bugfixes in TouchGFX Core:
    - Fixed redraw of circleProgressIndicator when setting new value.
    - Removed additional screen redraw after a screen transition is complete. This additional redraw caused performance issues on some platforms. Invalidating the entire screen in Screen::afterTransition(), if required, is now the responsibility of the application developer.
- Update procedure:
    - For this release additional steps might be needed. Please refer to the Known Issues article for details: https://support.touchgfx.com/docs/miscellaneous/known-issues/#project-updater-issue

Version 4.11.0

- Release date: March 1st, 2019
- Important upgrade information:

    - If your application includes LCD.hpp and expects to have access to HAL, this will no longer work since LCD.hpp no longer includes HAL.hpp. Make sure to include HAL.hpp in this case. Older versions of sample applications Demo1 and Demo2 had this issue and have been updated.

- New TouchGFX Designer Features:

    - Added Bring Forward/Send Backwards support for widgets, via UI Buttons and keyboard shortcuts Ctrl + F, Ctrl + B.
    - Added support for copy and paste of Screens and CustomContainerDefinitons.
    - Added support for reordering CustomContainerDefinitions.
    - Switching between Screen and CustomContainerDefinitions now remembers the previously selected Screen and CustomContainerDefinition.
    - The last used typography is now used when creating new texts and widgets that use text.
    - Added new tree icon for CustomContainerInstances.
    - Disabled continuous code generation and compiling.
    - Improved readability of the output in Detailed Log window.
    - Widget Wildcard Characters added to the Texts tab, which adds default wildcard characters when using some widgets
    - Improved performance when loading a project.
    - Improved performance when generating a project.
    - Improved performance of validation engine.
    - Added support for 6 bit color displays (8bpp).
    - Added support for setting RadioButtonGroup for RadioButtons.
    - Added support for Display Rotation (Landscape/Portrait).
    - Added support for setting Landscape/Portrait simulator skins in the Designer.
    - Added support for the following widgets: AnalogClock, DigitalClock, TextureMapper, AnimatedTextureMapper & Shape.
    - The Designer now generates the Makefile and Visual Studio files used for running the Simulator.

- New TouchGFX Core Features:

    - Added support for 6 bit color displays (RGBA2222, BGRA2222, ARGB2222 and ABGR2222 framebuffer formats).
    - Added support for Thai.
    - Improved rendering of Arabic text.
    - Added handling of negative line spacing.

- Bugfixes in TouchGFX Designer:

    - Fixed Ctrl + A (select all) not working for CustomContainerDefinitions.

  - Fixed reordering of Screens selecting the first screen in the list and deleting the undo/redo history for the Screen that was moved.
  - Fixed bug where the undo/redo history would become broken after selecting the Application node.
  - Fixed application names not being allowed to start with a number or contain "-" or "_".
  - Fixed loading an application while on the CustomContainer tab resulting in erroneous content on the canvas.
  - Fixed pressing undo after moving multiple elements into a container resulting in a crash.
  - Fixed font files being locked when loading a project.
  - Fixed error not showing up on components that use text, when removing their Resource Text.
  - Fixed bug where loading a faulty application by double-clicking a TouchGFX file would cause the splash screen to get stuck.
  - Fixed faulty position code generation for ModalWindow.
  - Fixed missing "Move widget" interaction support for ScrollableContainer, ScrollList & ScrollWheel.
  - Fixed the ordering of the Recent Applications list. Now correctly updates when opening an application.
  - Fixed bug where inserting a widget could add an empty undo item to the undo/redo history.
  - Fixed missing header text and description in the properties pane for CustomContainerDefinitions.
  - Fixed bug where idle CPU usage was higher than expected.
  - Fixed bug where setting an interaction on a FlexButton inside a CustomContainer would generate faulty code.
  - Fixed bug where setting a mixin on a widget was not undo-able.
  - Fixed missing undo/redo functionality for adding styles to FlexButton.
  - Fixed wrong order of initializations when using numerous slider callbacks in interactions.
- Bugfixes in TouchGFX Core:
  - Fixed precision in CWR Painters for 4bpp and 2bpp.
  - Fixed precision in alpha blending formulaes for 8bpp, 16bpp and 24bpp.
- Update procedure:
  - For this release additional steps might be needed. Please refer to the Known Issues article for details: https://support.touchgfx.com/docs/miscellaneous/known-issues/#project-updater-issue

Version 4.10.0

- Release date: November 5th, 2018
- Requirements:
  - TouchGFX is now only available for STM32 microcontrollers.
- New TouchGFX Designer Features:
  - Added support for the following widgets: ImageProgress, BoxProgress, TextProgress, LineProgress, CircleProgress, Line, Circle, BoxWithBorder, FlexButton, ScrollList, ScrollWheel and SwipeContainer.
  - Canvas Buffer setting can be adjusted on screens.
  - Support for screen transition: CoverTransition.
  - Now logs the following system information for use in support scenarios: Username, Designer version, Designer installation path, Windows version, Current culture, Installed .NET versions.
  - It is now possible to import a UI with any resolution to an application (resolution check has been removed).
  - Added button to show/hide clipped widgets.
  - Improved performance when dragging and resizing widgets on the canvas.
- New TouchGFX Core Features:
  - Circle and AbstractShape now supports higher precision on arc start and arc end for smoother arcs.
  - The internal Q5 structure now uses 32 bit instead of 16 bits for increased value range.
  - Added Circle::getPrecision().
  - Added functons FadeAnimator::isFadeAnimationRunning(), MoveAnimator::isMoveAnimationRunning(), AnimatedImage::isAnimatedImageRunning() and ZoomAnimationImage::isZoomAnimationImageRunning(). The old isRunning() functions have been deprecated.
  - ListLayout::setDirection() and getDirection() added.
  - Updated roo gem from 1.13.1 to 2.7.1.
  - Pressing SHIFT-F3 will copy the screen to the clipboard (Windows only).
  - Pressing CTRL-F3 will save the next 50 screens to the screenshots folder.
  - Generated assets are now indented properly.
  - ScrollableContainer::setScrollbarsPermanentlyVisible() added.
- Bugfixes in TouchGFX Designer
  - Fixed ModalWindow widget not resizing when Screen or Custom Container size changes.
  - Fixed generating code failing if a files hidden attribute was set to hidden.

    - Fixed changing the casing of a screen or custom container name resulting in a recompilation error.
    - Fixed bug where internet loss would crash the Designer if no Online Applications are available.
    - Fixed ModalWindow widget position being generated incorrectly after loading a project.
    - Removed unnecessary recompilation when loading Designer project.
    - Fixed visual bug in ImagePicker where the "empty placeholder" would show up even though you have subfolders in current folder.
    - Fixed bug where the Designer was not using default credentials through proxy server.
    - Fixed bug where the Designer would not correctly report an error when trying to flash to a wrong target.
    - Fixed bug where having insufficient permissions to write to the chosen touchgfx path would crash the Designer.
    - Fixed bug where the Designer was incorrectly interpreting screen changes as an unsaved change.
    - Fixed a visual bug, where widgets inside a Container would not display properly when resizing the Container.
    - The Designer now closes a running Simulator process, when you load another application.
    - Fixed a bug where it was possible to drag widgets inside an instance of a Custom Container.
    - Circle did sometimes not render correctly, and invalidated rectangle was not calculated properly.
    - Fixed Circle when half line width was greater than radius.
- Bugfixes in TouchGFX Core:
    - Fixed erroneous calculation of x & y values in setValue in LineProgress.cpp.
    - Circle did sometimes not render correctly, and invalidated rectangle was not calculated properly.
    - Fixed Circle when half line width was greater than radius.
    - Fixed drawing lines longer than 2047 pixels, e.g. 1449 pixels wide and 1449 pixels high.
    - Fixed bug preventing some Arabic ligatures from being rendered correctly.
- Update procedure:
    - For this release additional steps might be needed. Please refer to the Known Issues article for details: https://support.touchgfx.com/docs/miscellaneous/known-issues/#project-updater-issue

Version 4.9.4

- Release date: January 25th, 2018
- Bugfixes:

- Reduced the time it takes to load an application in the Designer.

Version 4.9.3

- Release date: December 15th, 2017

- Bugfixes:

  - Designer now uses default Windows proxy settings.
  - Package manager updates available packages when online.
  - Improved error description when offline.
  - Set text interaction works with resource texts.
  - Project updater updates MSVS projects with correct image formats.
  - Text size calculated wrongly in Designer in rare occasions.
  - Recent files ordered by date.
  - Corrected initialization of counter in Wait For interaction.
  - Fixed drawing of child elements in list layout, when resized.
  - Fixed loading of application with list layout widgets.
  - .otf font files now correctly rendered.
  - Dragging containers could in rare cases introduce wrong coordinates.
  - Fixed null-termination of wildcard text buffers.
  - Button With Label text rendering correction.
  - tgfx.exe packager works for more complex file layouts.
  - Source code included for containers.
  - Additional minor Designer UI fixes and improvements.

Version 4.9.2

- Release date: November 20th, 2017

- Bugfixes:

  - Fixed Designer issue where dragging elements on the canvas would in some cases cause an exception.

Version 4.9.1

- Release date: November 16th, 2017
- Bugfixes:
  - Fixed several Designer issues with TextArea widgets when placed inside containers.
  - Fixed an issue with interactions triggered by "Another interaction is done" dissappearing when loading a project.
  - On PCs with certain security policy configurations, the Designer was not able to create new projects correctly.
  - Improved error handling in Designer if the asset generation, code compilation or post generation commands fail.
  - Fixed an issue where the TouchgfxPath in Designer project files was not interpreted correctly.
  - Some typography changes in Designer did not cause new code to be generated.
  - Fixed issue with ImageConverter when assets folder was under svn control.
  - ImageConverter could in certain cases fail to detect changes in assets.

Version 4.9.0

- Release date: November 8th, 2017
- New Features:
  - Added a package manager for handling board support packages, demos and examples. The Designer will now fetch these from an online repository.
  - All the old examples, demos and ports for various boards have been removed from the framework, and are now available as packages instead.
  - Substantially improved text handling in the Designer. It is now possible to work with translations and wildcards in the Designer, so it should no longer be necessary to edit the texts.xlsx file manually.
  - Designer is now much more flexible regarding application file structure, and is now able to auto-update IAR and Keil IDE projects regardless of file location.
  - Added Designer support for the ScrollableContainer and ListLayout widgets.
  - Added support for the SW4STM32 IDE.
  - Added support for version 8.10 of IAR Embedded Workbench.
  - Image converter now has an option to operate on folders, instead of being invoked once per .png file. This substantially speeds up the process of converting images. This mode is the default behavior for new projects.
  - The GNU Arm Embedded toolchain (GCC cross-compiler) has been updated to version 6-2017-q2-update (gcc version 6.3.1).

    - The GNU compiler for the PC simulator has been updated to version 6.3.0.
    - Added gcc core libs compiled with -mfloat-abi=hard for Cortex-M4f and Cortex-M7.
    - Increased number of widgets that can be registered as timer widgets from 24 to 32. Also added functions for obtaining information about which widgets are currently registered.
- Bugfixes:
    - AnimationTextureMapper::cancelMoveAnimation() is renamed to cancelAnimationTextureMapperAnimation() to avoid problems with MoveAnimator::cancelMoveAnimation().
    - Fixed bug in PainterRGB565Bitmap when rendering solid pixels from an ARGB8888 Bitmap.
    - Fixed rare bug in FontConvert if all used characters are missing from the font.
    - Fixed unitialized variables in the DMA class.
- Update procedure:
    - For this release additional steps might be needed.

Version 4.8.0

- Release date: March 10th, 2017
- Performance
    - LCD4bpp now draws characters up to 15% faster.
    - Canvas widgets now render slightly faster in certain situations.
- New Features:
    - TouchGFX Designer released. The core framework, Designer and environment shell are now bundled in a single installation.
    - Support for Farsi and Arabic ligatures where squences of up to three character are recognized.
    - Added support for Microsoft Visual Studio 2017.
    - TextArea and TextAreaWithWildcard(s) now support setWideTextAction() to automatically break lines and insert ellipsis at end of line, when the line is too long.
    - Added getter functions to Slider.
    - MoveAnimator and FadeAnimator can now clear the callback set for animation ended.
    - Errors from ImageConvert, TextConvert and FontConvert are now shown in the Error List window of Visual Studio.
    - Simulator applications are now Windows programs instead of Console programs.

- Bugfixes
  - AbstractShape::updateAbstractShapeCache() is now a public function and should be called after one or more calls to AbstractShape::setCorner(), to ensure shape is correct.
  - Simulator window can no longer be unintentionally resized.
  - F2 to highlight invalidated areas now works with old HALSDL.
  - PainterGRAY2Bitmap, PainterGRAY4Bitmap, PainterRGB565Bitmap and PainterRGB888Bitmap all failed to validate that painting was inside the size of the bitmap in some situations.
  - HALSDL2 (simulator) now uses 24bpp on screen to make colors in screenshots correct.
  - TiledImage::setOffset() now handles an empty bitmap correctly.
  - TiledImage::getSolidRect() would sometimes report wrong rect.
  - If text in a TextArea was rotated, resizeToCurrentText() and resizeHeightToCurrentText() would swap the width/height.
  - Function getTextHeight() would not take line spacing into account. Functions like resizeToCurrentText() and others that use the getTextHeight() function would not resize correctly.
  - SlideMenu::setState() did not handle EXPANDED state correctly.
- Update Procedure
  - Due to the addition of TouchGFX Designer, installation is now done via an .msi installer.
  - Compatible with existing 4.x applications and HAL ports.

Version 4.7.0

- Release date: December 14th, 2016
- New Features:
  - Source code for all the standard widgets and containers is now included. See the touchgfx/framework/source/touchgfx directory. Note that these classes are still present in the core library, and the source code files are not added to the IAR/Keil/gcc projects per default.
  - Optimized the handling of single frame buffer configuration on TFT controller based platforms, which in many cases eliminate the need for external RAM.
  - Substantial performance optimizations of the canvas widget system and all the standard painters. Expect a very significant increase in performance if many pixels are being drawn, and a smaller increase in performance for minor shapes (e.g. graph lines). The "PainterVerticalAlpha" used in our demos have also been updated.
  - The text converter tool will now combine identical translations across all languages, resulting in reduced footprint. The result of this process will be printed during asset generation. NOTE: This

behavior is enabled by default. If you have an existing project where you manipulate the text data structures (e.g. load a single language into RAM), this optimization might break your code. The optimization can be disabled by adding the following remap_identical_texts := no (for "make"-based generation) <RemapIdenticalTexts>no</RemapIdenticalTexts> (for MSVS)
  - Updated SDL version used by simulator from 1.2 to 2.0.4. SDL1.2 is still present in the distribution, but all examples and projects now use SDL2.
  - Support for skinning the simulator with .png files. If the .png files contain non-opaque areas, the simulator window will be shaped accordingly. See display_orientation_example for a code example.
  - On ST targets with Chrom-ART, the Box widget will now be drawn by DMA even when alpha < 255 (BLIT_OP_FILL_WITH_ALPHA support).
  - TextArea and TextArea with wildcard(s) now support setWideTextAction() to automatically wrap long lines.
  - Added the ability to display a "fallback" character in case a non-existing glyph is encountered at runtime. This is configured in the typography sheet of the text database.
  - Added options for forcing the inclusion of additional glyphs in a font. This makes it much easier to handle dynamic texts where the glyphs are not known at compile time. This is configured in the typography sheet of the text database.
  - Output from the TextConvert utility is now post-processed to give significant saving by mapping identical strings to the same memory areas.
  - Added built-in BitmapId called BITMAP_ANIMATION_STORAGE which can be used to refer to the animation storage when assigning a Bitmap to a widget.
  - Added dither algorithm selection from config/gcc/app.mk and config/msvs/Application.props.
  - It is possible to save a simulator screenshot programatically, by using: #ifdef SIMULATOR (static_cast<HALSDL2*>(HAL::getInstance()))->saveScreenshot(); #endif
  - ScrollableContainer now properly ignores invisible elements.
  - DigitalClock now supports a zero to be displayed in front of the hour indicator (if hour < 10).
  - The simulator can now highlight the areas being invalidated. Press F2 to toggle this feature.
  - Added Unicode::vsnprintf functions that take va_list arguments instead of ellipsis.
- Bugfixes
  - Unicode::sprintfFloat did not print <space> instead of '+' if the format string was "% f". Also, the sign of floating point numbers in range]-1..0[would not be printed with sign so for example -0.5 would print as 0.5.
  - Fixed a bug that could cause TextureMapper to read outside source bitmap memory area.
  - GPIO.cpp for STM32F769-Discovery and Eval boards had some incorrect GPIO pin manipulations (used for performance measurement).
  - Some methods in Slider.hpp were missing a virtual declaration.

- Fixed a bug in BoardConfiguration for STM32F769-Discovery board causing 24bpp color mode to be displayed incorrectly.
- AnimatedImage - setBitmap(..) should not be used and is now private For AnimatedImage use setBitmaps(..) instead.
- Project files and Makefile have been updated to allow the TouchGFX framework to be placed on another disk drive than the project being developed.

- TouchGFX Environment (version 2.8)

  - "make.exe" is now version 4.1 which allows for parallel compilation, by adding e.g. "-j8" to your make command. This substantially speeds up compilation. If your makefile is from TouchGFX 4.2.0 or earlier, you will need to either update it, or to use make-3.81.exe
  - g++ could in some cases report "There is no disk in the drive. Please insert a disk into drive E:.". This has been fixed by upgrading gcc from version 4.8.1 to version 4.9.3.

Version 4.6.1

- Release date: September 12th, 2016

- Performance

  - Optimization improvements of core library for GCC on Cortex-M4 and Cortex-M7, providing significant speeedup of especially TextureMapper and Canvas widgets compared to TouchGFX 4.6.0.

- New Features

  - New function in HALSDL to set title of simulator window see HALSDL::setWindowTitle().
  - BW_RLE format (1bpp displays) now compresses better. Remember to remove old generated files and re-generate assets.
  - STM32F756G-EVAL using IAR now supports flashing of external memory.

- Bugfixes

  - Added IAR linker redirect commands to fix linker errors when compiling a Cortex-M4 based target with IAR 7.x.
  - Assigning different memory buffers to CanvasWidgetRenderer using setupBuffer() could in rare cases result in memory corruption.
  - TextureMapper could in rare cases draw outside the framebuffer.
  - Setting the offset of a TiledImage did not work properly.

- Fixed two issues that would in some cases cause memory corruption when deleting dynamic bitmaps.
- Missing virtual method declarations in AnalogClock added.
- Fixed a problem in GCC linker script for LPC4088DisplayModule which caused texts and fonts to be placed in external flash.
- For those using fontconvert.out on its own, the output directory is now automatically created if it does not exist.
- ScrollableContainers could in rare cases send a wrong drag event to a child.
- Monochrome (1bpp) displays with width not divisible by 8 would not display text correctly.
- Slightly increased default touch sample rate on STM32F746G Discovery board.

Version 4.6.0

- Release date: June 14th, 2016

- New features

  - Added support for 2bpp grayscale displays.
  - Added support for 4bpp grayscale displays.
  - New widget TiledImage. Will display one or more repetions of an image. The number of repetitions depends on the size of the widget and the size of the image.
  - New widget RepeatButton. A button that will repeatedly fire click events when pressed.
  - New widget AnimationTextureMapper. TextureMapper with build in animation features. See animation_texture_mapper_example.
  - New containers AnalogClock and DigitalClock, see clock_example.
  - New containers ProgressIndicators, see progress_indicator_example.
  - New container ModalWindow. Creates a window on top of the main screen and a shade on the rest of the main screen. No clicks are passed on to the main screen as long as the modal window is visible. See example modal_window_example.
  - New container SlideMenu. Animating side/top/bottom-menu that has an activate button for sliding it in/out of the screen. A timeout can be set for automitical hiding when idle for a period of time.
  - Canvas Widget Line supports ROUND_CAP_ENDING and setCapPrecision() to control the round cap.
  - Simulator can now generate ticks very close to the frequency of the hardware.
  - Mouse X and Y coordinates are put in the title of the window in the simulator. (press F1 to (de)active this when running the simulator).
  - ST Cube drivers updated to version 1.4.0 for STM32F7 MCU and STM32F7 based boards.

    - Added support for the STM32769I-EVAL board.
    - Added support for the STM32F769I-Discovery board.
    - Screenshots made from the simulator (F3) are now saved under a name with timestamp to prevent old screenshots to be overwritten by accident.
    - Simulator now outputs canvas widget memory usage to easily find optimal canvas memory buffer size.

- Bugfixes

    - DMA drivers for ST boards: express DMA2D instance initialization for STM32F7. Fixed incorrect used of CLUT_CM for F4-Discovery.
    - DMA drivers for LPC17xx, LPC18xx, LPC43xx did not behave correctly if other DMA channels are in use simulatenously. They now properly look at flags for channel 0 only.
    - Touch controller drivers for ST boards now properly checks that initialization was OK before querying.
    - Mouse clicks in the simulator would not always be detected.
    - ImageConvert.exe has RGB565 as default (and sensible defaults for other opque formats)
    - ImageConvert would not work for a BW image scheduled for compression (BW_RLE) and rotation (.90. in filename) if the image would become too large if compressed (falling back to BW format).
    - All Makefiles now use abspath instead of realpath.
    - AnimatedImage now allows the animation to be restarted from the AnimationEnded callback function.
    - QSPI flash size corrected to 64MBytes for STM32756G-EVAL board.
    - Added D-cache invalidation to STM32F7HAL::flushFrameBuffer. This fixes occasional graphics errors on STM32F7 when in single frame buffer mode and fb was located in SRAM.
    - The otm8009a displays (STM32769-DISCO, STM32769-EVAL, STM32469-DISCO, STM32469-EVAL) are now using maximum display brightness.
    - Added a workaround for a bug in IAR 7.50.x regarding va_list name mangling.

- Update Procedure

    - Compatible with existing 4.x applications and HAL ports.

Version 4.5.1

- Release date: March 14th, 2016

- Bugfixes

- Fixed two IAR linker issues related to resolving the va_list symbol, which would cause some versions of IAR being unable to link the example projects.
- STM32F4-Discovery board would draw solid rectangles with the wrong color in 16bpp mode.
- The Canvas Widget Renderer no longer performs unaligned memory accesses.
- vApplicationIdleHook (FreeRTOS specific) no longer blocks, which previously prevented FreeRTOS from freeing memory if tasks were deleted.
- Arabic words with accent in the middle would not render properly.
- Added PixelDataWidget::getAlpha().
- Unicode::strncpy() with a char* as source would not copy characters with ascii codes above 127 properly.

Version 4.5.0

- Release date: February 2nd, 2016

- New features

  - Support for two new languages, Arabic and Hebrew, with right-to-left text rendering. RTL strings can be mixed with LTR texts and numbers.
  - Support for 24 bits per pixel framebuffers. Images look more detailed, but also consume more memory.
  - Bitmaps can now be created at runtime using method Bitmap::dynamicBitmapCreate. Useful for e.g. displaying .bmp files loaded from an SD card. See dynamic_bitmap_example.
  - Frame rate compensation feature which provides smoother animations if frame rate occasionally drops. Not enabled by default.
  - Bitmap caching is enhanced to allow removal of bitmaps from the cache to make room for caching of other bitmaps.
  - A new widget, PixelDataWidget, is introduced. This widget makes it possible to display raw pixel data obtained at runtime (e.g. video samples).
  - The simulator executable on windows now features an icon for easier identification in the task bar.
  - ST boards supported by TouchGFX can now have just their internal flash programmed from the command using 'make intflash' provided that ST-Link Utility Release 3.7+ is installed.
  - Unicode::snprintf() has been improved and updated substantially to support more of the standard format specifiers like %02d.
  - Unicode::snprintfFloat() added to support floats (in separate function because the "%f" va_args approach would force inclusion of doubles).

- Quality of image converter dithering has been improved (floating point arithmetics). Also added support for new types of dither algorithms, and can take into account hardware with various wiring of the low (unused) bits in 16/18 bit displays.
- touchgfx::ButtonWithLabel now contains a method, updateTextPosition(), that can be used to ensure horizontal text centering when changing label content (e.g. when changing language).
- touchgfx::TextArea has a new method, setBaselineY(), that allows placing texts according to a text baseline instead of upper left corner.
- The internal format of glyph encoding now stores the first pixel in the least significant bit instead of the most significant bit.
- Specification of color values has been switched from uint16_t to colortype to support seamless switching between 16 and 24 bit colors.
- The touchgfx::TextArea class now has a method, setIndentation(), that can prevent the glyph of characters from being cut off in the rare case where it extends under the previous character (similarly for touchgfx::Keyboard class which has a new setTextIndentation() method).
- STM32F7xx and STM32F4x9 ports now support DMA transfers of touchgfx::Box.
- The GPIO::VSYNC_FREQ signal was previously "toggled" exclusively on "VSYNC" interrupt (NXP LPC18xx, NXP LPC43xx, Freescale MK70F12, ST stm32f4x9). The signal is now high on "VSYNC" interrupt and low on "Front-Porch-Entered" interrupt.
- GCC support for Cortex-M3.

- Bugfixes

  - Fixed rare crash on STM32F7 caused by speculative caching of invalid QSPI memory region. Update your BoardConfiguration if yours is based on 4.4.x.
  - Fixed occasional display flickering on STM32F746G-DISCO board caused by cache access on FMC bank 1.
  - Handling of the character "%"" in touchgfx::TextAreaWithWildcards has been improved to prevent inserting %% in some special cases.
  - touchgfx::DragEvent and touchgfx::GestureEvent now use and report signed coordinates instead of unsigned. This makes more sense as drags/gestures are expressed in coordinates relative to the drawable receiving them.
  - snprintf("%x") would generate uppper case hex. Now "%X" generates uppercase hex and "%x" generates lower case hex, just like the standard snprintf().
  - Fixed randomness for demos when running on Linux.
  - Fixed redrawing when using heavily italicized fonts.
  - Pointer to ModelListener in Model class for all TouchGFX applications was not properly initialized (NULL).
  - Fixed support for heavily italicized fonts in touchgfx::TextArea.

  - Subtle error in the Image Converter where column 0 could get slightly incorrect pixel colors. As a result the entire image could be slightly wrong, probably not noticeable.
  - Minor error in Slider where values were not distributed evenly.
- Deprecated
  - LCD::drawGlyph() has been deprecated. Use LCD::drawString instead.
- Update Procedure
  - Compatible with existing 4.x applications and HAL ports.

Version 4.4.2

- Release date: November 26th, 2015
- Bugfixes:
  - Corrected rare GUI task hangup on STM32F7 targets when compiling with IAR 7.x

Version 4.4.1

- Release date: October 27th, 2015
- Bugfixes:
  - Corrected occasional GUI task hangup on STM32F7 targets when compiling with Keil 5.x
  - Fixed occasional tearing on STM32 F469 EVAL/Discovery boards when using DSI in landscape orientation and single framebuffer mode.
  - Modified IAR flash loader settings for STM32 F469 boards to enable programming of internal flash (Note: QuadSPI flash must still be programmed from ST-Link Utility as there are no IAR loaders for this)
  - GPIO class for perf. measurement for STM32F746G-EVAL boards now properly uses the BSP_LED functions. Note that only two signals are active on this board per default because LED2 and LED4 use IO Expander, making them unsuited for measuring performance.
  - Removed annoying "Get Alternative File" dialog popups in IAR Workbench when debugging Cortex-M7 applications.

Version 4.4.0

- Release date: October 6th, 2015
- New features
  - Added support for the Cortex-M7 core.
  - Introduced concept of "finger size" for touch input. When used, TouchGFX will attempt to find touchable widgets in the area surrounding the reported x,y coordinates, so users no longer have to click precisely on a widget. This feature makes it substantially easier to hit small buttons. See HAL::setFingerSize().
  - Supports Visual Studio 2015
  - Visual Studio projects for Demos and Examples now include Application.props under Resources for quick access. As always a rebuild might be required when altering the contents of Application.props.
  - Support for Bitmap Fonts in BDF format. If the requested font size is not available in the font file, the font converter will write the supported font size(s) in the error message. See the example monochrome_example for usage.
  - Generating assets now issues better error messages when spaces are detected in paths and file names.
  - All ST boards can now be flashed from the command line provided that ST-Link Utility Release 3.7 has been installed. Simply use 'make -f target/ST/<board>/Makefile flash' to build and flash your application to the connected board. If timeouts occur during flashing, go to Device Manager in Windows and disable "MBED microcontroller USB Device" under "Disk drives".
  - New touchgfx-env version 2.5 available with new gcc cross compiler version 4.9.3. The older version 4.8.4 could generate invalid code for Cortex-M7 cores in rare cases.
- Board support
  - Added support for the STM32F7xx processors
  - Added support for the STM32F746G-DISCO and STM32756G-EVAL boards
  - Added support for the STM32F469 processor with DSI displays
  - Added support for the STM32469I-EVAL and STM32469I-Discovery boards
- Bugfixes
  - TextureMapper and ScaleableImage now draws images correctly when using "rotate90".
  - Fixed potential initialization order bug in STM32F4DMA.cpp
  - Fixed bug that limited number of glyphs in a single font to 32768. Now supports 65536 glyphs per font as intended.
  - Fixed bug that caused hal.lockDMAToFrontPorch(false) to not have any effect in single framebuffer mode.

- ButtonWithLabel correctly center texts vertically if text contains newlines

Version 4.3.0

- Release date: June 8th, 2015
- New features
  - TextureMapper widget added. The TextureMapper is a highly optimized image renderer that can be used for displaying an image that is scaled and/or rotated in two or three dimensions during run time. This can be used for doing advanced rotation animations of images. See manual or texture_mapper_example for more information. LCD has new methods for drawing triangles and corresponding scan lines, drawTextureMapTriangle and drawTextureMapScanLine
  - Alpha Channel Dithering Images with alpha channel can now get the alpha channel dithered for smoother alpha gradients, see examples or Application Development section in manual for details
  - Compression of 1BPP (monochrome) bitmaps Added image format option of BW_RLE, which will cause bitmaps to be automatically run-length encoded if that takes up less space than the regular per-pixel format. Yields substantially smaller bitmap footprint in many cases. See advanced chapter in manual for details.
  - Slider widget added. See manual or slider_example for more information.
  - Makefiles has been updated to work with make-4.1.
  - Added support for the LPC4088 processor and the Embedded Artists LPC4088 Display Module board.
  - Individual bitmaps can now be placed in internal flash instead of external by having the bitmap file name include the string ".int."
  - MoveAnimator, FadeAnimator and ZoomAnimationImage now have a cancelMoveAnimation/cancelFadeAnimation/cancelZoomAnimation method.
- Update procedure
  - Compatible with existing 4.X applications. Just replace the touchgfx folder.
  - Check Known Issues in the documentation.
- Info
  - The evaluation version of TouchGFX is now distributed with source code for the hardware abstraction layer instead of a precompiled library. This makes it possible to port the evaluation version to custom hardware instead of it being limited to the supported eval boards only. Instead, the evaluation version now has a TouchGFX watermark which will appear occasionally.

    - Memory consumption reduced due to improved rendering algorithm. Will typically allow GUI task stack to be reduced by around 1400 bytes compared to version 4.2.0 (depending on actual application). Additionally the statically allocated memory is also reduced by around 1KB.
    - Maximum number of visible widgets limit of 150 removed.
    - Added two new demos for 640x480 and 480x272 resolutions showcasing new features, graphs, internationalization and custom widgets.
    - Drawable.setPosition() now calls setXY(), setWidth() and setHeight() for easier subclassing.
    - AbstractPainterRGB565 and AbstractPainterBW are recommended as base classes when implementing your own painters.
    - CanvasWidgets now have setAlpha() and getAlpha() methods. Your custom Painter classes must implement this, or inherit from the AbstractPainterRGB565 class
    - Maximum number of registered timer widgets increased from 16 to 24.
    - touchgfx-env updated to 2.4. The environment does not beep anymore.
    - Board Support Package for STM324x9I-EVAL is now based on the STMCubeF4 drivers.
- Bugfixes
    - Screen::handleGestureEvent now converts x/y to relative coordinates
    - Fixed bug when drawing several objects on the same canvas using moveTo() more than once.
    - ZoomAnimationImage movement relative to scaling did not use correct easing equation.
    - PainterRGB565 did not blend green alpha correctly.
    - RadioButtonGroup now initializes callbacks to zero.
    - ScalableImage now works with bitmaps with transparancy.
    - AnimatedImage would display the start and end of an animation twice.
    - Default implementation of CanvasWidget::getMinimalRect() returned coordinates relative to its parent, not itself.
    - ScrollableContainer erroneously unregistered itself as a timer widget at every tick, which made it difficult to use with other timer-based operations.
- Performance
    - ScalableImage and ZoomAnimationImage has been optimized for better performance.

Version 4.2.0

- Release date: January 14th, 2015
- Performance

- Substantially improved rendering performance, which in most cases will result in a 25% reduction of time it takes to render a frame.

  NOTE: This optimization does not necessarily work on all targets so it must be manually enabled. See the "Optimization" chapter in the porting guide for how to enable this optimization for existing portings. It is STRONGLY recommended that the optimization is enabled. This optimization is enabled for all appropriate evaluation boards in the 4.2.0 board packages.

- Major new features
  - Added CanvasWidgets for smooth, anti-aliased drawing of geometric shapes. Currently Line, Circle and a more generic Shape have been implemented. CanvasWidgets can be painted with a solid color (+ alpha), a bitmap (including alpha) or a custom painter. Read more on Canvas Widgets and Painters in the documentation.
  - Added support for the Keil compiler and uVision4 IDE. Please refer to the "Supported Hardware" section of the TouchGFX Distribution chapter in the documentation for a list of Keil-supported targets.
- New features
  - It is now possible to specify an animation start delay on ZoomAnimationImage, MoveAnimator and FadeAnimator.
  - Added Board support for 4.3" TouchGFX Demo board w. LPC4350 (No internal flash)
  - RadioButton and RadioButtonGroup widgets added. See app/examples/radio_button_example and documentation.
  - LPC43XX and LPC1788 can now fill rectangles using DMA.
  - Visual Studio 2013 is now supported.
  - Preliminary support for Visual Studio 2015 Preview version.
  - Improved performance when generating assets.
  - New canvas_widget_example added to the example directory.
  - The "using namespace touchgfx" present in various header files can now be avoided by defining the symbol NO_USING_NAMESPACE_TOUCHGFX in your project.
- TouchGFX env
  - The message displayed when starting a shell has been fixed with correct path to examples.
- Bugfixes
  - Fixed bug in simulator for 1bpp displays when width and/or height was not not a multiple of 8.
  - Fixed bug in ScrollableContainer where CANCEL events where not always delegated to correct child, causing e.g. buttons to remain pressed when dragging outside SC area.

    - Fixed bug when rendering chromArt fonts with a rotated display.
    - Fixed bug - Keyboard widget setTouchable(false) had no effect.
    - Freescale K70 DMA now checks the appropriate DONE bit in TCD0_CSR.
    - On ST processors fixed bug with rotated texts rendered by ChromArt when in non-native display orientation.
- Board support
    - Embedded Artists LPC4357DevKit board package: CPU clocked to 204Mhz (previously 96Mhz). Now uses SPIFI flash instead of NOR.
- Update procedure
    - Compatible with existing 4.X applications. Just replace the touchgfx folder.
- Info
    - Documentation has been updated.

Version 4.1.1

- Release date: October 29th, 2014
- New features
    - Mixin: MoveAnimator added. The MoveAnimator mixin makes the template class T able to animate a movement from its current position to a specified end position. See app/example/move_fade_example.
    - Mixin: FadeAnimator added. The FadeAnimator mixin makes the template class T able to animate an alpha fade from its current alpha value to a specified end alpha value. See app/example/move_fade_example.
    - ScalableImage and ZoomAnimationImage now support alpha per pixel bitmaps and alpha per bitmap
    - ScalableImage and ZoomAnimationImage now support ARGB8888 format bitmaps
- Bugfixes
    - Fixed a bug causing the Keyboard widget to render incorrectly in rare cases.
    - Fixed a bug causing drag event coordinates to be incorrect for widgets when placed in a Container with coords != {0,0} which itself was placed in a ScrollableContainer.
    - The Application class now properly keeps track of number of times registerTimerWidget vs. unregisterTimerWidget is called for a given widget, meaning that if registered several times it

now requires same number of unregisters before widget no longer receives tick events.
    - Some ZoomAnimationImage functions were not virtual as they should be.
    - Some widgets were missing certain getter functions.
- Update procedure
    - Compatible with existing 4.X applications. Just replace the touchgfx folder.

Version 4.1.0

- Release date: October 17th, 2014
- New features
    - Now supports monochrome 1BPP displays. See manual for details.
    - Support for dynamic screen orientation change (landscape/portrait)
    - Support for scaling images (See ScalableImage and ZoomAnimationImage drawables)
- Demo
    - Home Control Demo now support 640x480 mode.
    - Home Control Demo now supports STM324xI-EVAL 5.7" board.
- Board support changes
    - Added support for STM324xI-EVAL 5.7" board (IAR+gcc).
    - Added gcc support for the EmbeddedArtists LPC4357DevKit board.
    - Optimized SPIFI initialization for TouchGFX eval board.
- Bugfixes
    - Adding a persistent Drawable to a ScrollableContainer could cause assertion
    - Support for much larger fonts
- Update procedure
    - Compatible with existing 4.X applications. Just replace the touchgfx folder.

Version 4.0.0

- Release date: September 26th, 2014

- New features
  - TouchEvent refactoring (API breaking):
    - Drawable::setActive is renamed to Drawable::setTouchable
    - Drawable::isActive is renamed to Drawable::isTouchable
    - Drawable::hijackTouchEvent is deprecated
    - Drawables are now per default not touchables
    - TouchEvents are now always propagated to all containers children
  - Language specific typography and alignment columns support added to text converter. Read more about this feature in the documentation.
  - Font rendering has been vastly improved with regards to font shapes and kerning.
  - Simulator - assert check on new view/presenter/transition size when doing screen transition. Failed assert checks probably due to missing definition of view/presenter/transition in FrontEndHeap.
  - TextArea and ButtonWithLabel now support text rotated 0, 90, 180 or 270 degrees.
  - Text centering on ButtonWithLabel has been improved in special cases.
  - Hardware Accelerated text rendering (4 and 8bpp) on supported ST platforms.
  - Ability to cache all items in the bitmap database in external RAM.
  - Support for Freescales K70 MCU.
  - Translation Sheet: Instances of "\<" and ">" are converted into "<" and ">" respectively. This enables literal translated strings such as "<Not a wildcard>" using "\<Not a wildcard\>".
  - Support for NXP LPC18XX series of MCU's.
- Bugfixes
  - Rendering error of images with odd width and alpha value less than 255
  - Correct handling of TextArea::getTextHeight in case of non initialized textArea
  - TextAreaWithWildcard::getTextWidth now includes the width of the wildcard text
  - gcc Makefiles now includes *.BMP and* .PNG from image assets.
  - Do not trim leading and trailing white space from any translations in the texts sheet.
  - Font converter did not generate font data properly for 8bpp.
  - ButtonWithIcon::setBitmaps - Suppress IAR warning for intentional virtual function override.
  - ButtonWithIcon optimized draw functionality
  - In extremely rare cases text could be written slightly outside the text area
- Update procedure
  - Due to the TouchEvent refactoring you have to rename functions accordingly. You also need to state in any custom widget or containers if they need to receive touch events. If you were using

hijackTouchEvent to prevent children of getting touch events, you now need to make sure that all children is not touchable instead.
    - Main.cpp for simulators need to be updated by replacing the line: TypedText::registerTypedTextDatabase(TypedTextDatabase::getInstance(), TypedTextDatabase::getInstanceSize()) with: Texts::setLanguage(0) You can also specify a specific language from your text database e.g. Texts::setLanguage(GB) In that case you also need to: #include <texts/TextKeysAndLanguages.hpp>
    - Rebuild entire project.

- Info
    - The TouchGFX Manual has been updated considerably.

Version 3.1.0

- New features
    - Added support for FDI uEZGUI-1788-70WVT eval board (NXP LPC-1788 Cortex M3).
    - Added support for Mjolner TouchGFX Demo Board Rev. 1.1 eval board (NXP LPC-4353 Cortex M4/M0 4.3").

- Bugfixes
    - Visual Studio build now rebuild BitmapDatabase.h when new images are added to the assets/images folder.

- Update procedure
    - Only if using Visual Studio: Update TouchGFXReleasePath in your Visual Studio .props file. Simply edit the file in a text editor. The path should be extended with "touchgfx\". See the template_application for inspiration.
    - Only if using Visual Studio: Update your Visual Studio project file (.vcxproj file). Simply edit the file in a text editor. Replace all paths on the form "$(TouchGFXReleasePath)\framework\config\msvs\touchgfx_prebuild.targets" with "$(TouchGFXReleasePath)\config\msvs\touchgfx_prebuild.targets".

- Info

- Hardware Abstraction Layer architecture has been reworked so that all common code for various hardware components (MCUs and drivers) is now shared across different target boards. This greatly simplifies the porting effort for new/custom boards as long as they contain one or more hardware components already supported by TouchGFX.

Version 3.0.0

- New features
  - Visual Studio 2010/2012 support.
  - Added support for png images with alpha channel.
  - Added support for subfolders in assets/bitmaps folder
  - Added support for ST STM32F4X9I-EVAL eval board.
  - Added support for Robert Penners Easing Equations (see touchgfx/EasingEquations.hpp).
  - Image converter: Added sanity check of input image file names, must not start with digit and must be alphanumeric.
  - Image converter: Added checking against case insensitively file name duplicates in input list.
  - Text converter: Added build stopping sanity checks for bpp and font_size values.
  - ScrollableContainer: Now supports setScrollbarPadding, setScrollbarWidth, setScrollbarColor, and setScrollbarAlpha.
  - ScrollableContainer: Set default value of ScrollThreshold to 5 pixels, instead of 1.
  - Added support for alpha blending of fonts (TextArea::setAlpha(uint8_t alpha))
  - ImageConvert support two different output formats: RGB565 and ARGB8888
  - ImageConvert - two options added to control output format for images with/without an alpha channel
  - Touchgfx environment under MinGW is updated due to linker errors for large projects. g++ version is updated from 4.6.2 -> 4.8.1
  - Internal RAM footprint improvements
  - Structural changes of target library and hardware abstraction layers
- Bugfixes
  - Fill operation (Box widget) resulted in a crash on the lpc4357_emb_artist board
  - Textconvert & fontconvert: Different typographies may now have identical properties.
  - Imageconvert & fontconvert: Better error handling for POSIX compliant platforms
  - HALSDL: Do not overflow key data type.
  - LanguageXX.cpp files now end with a newline (removing warnings).
  - TextArea::draw now handles non initialized TypedText correctly.
- Update procedure
  - The folders assets/bitmaps and generated/bitmaps must be renamed to assets/images and generated/images.
  - Upgrade TouchGFX environment to version 2.0

    - Update any application Makefile to adhere with the Makefile specified in the updated template_application
    - Rebuild entire project
    - Convert bmp images that contains the former transparent color to png images that uses alpha channels. This can be done automatically using a free tool called imagemagick. More info and hints can be acquired by writing touchgfx-support@mjolner.com
    - Custom HAL implementations must be updated to conform with the new structure
- Info
    - The "magic" transparent color that was previously used for transparent color in the bmp format is no longer supported. Instead use png images with alpha channel.

Version 2.2.0

- New features
    - Added support for portrait mode with landscape displays at zero performance/resource cost.
    - Added kerning support.
    - Added Keyboard example (with IAR project for the Energy Micro DK3750 eval board)
    - Changed interface for blitCopy method in LCD.
    - Removed SyncBackBuffer method from HAL.
    - Removed clearLCD method from LCD.
    - Removed fillGradientRect method from LCD.
    - ScrollableContainer supports setScrollbarsVisible(bool visible).

Version 2.1.0

- First release of TouchGFX as a commercially available framework.

Version: 4.16

3rd Party Components in TouchGFX

TouchGFX uses different 3rd party libraries in its implementation.

To get an overview of what 3rd party components are used and their licenses, you can download a PDF file here:

- 3rd Party Components in TouchGFX PDF file

Version: 4.16

Cookie Policy

The TouchGFX documentation website does **not** install **any** cookies on your system.

Version: 4.16

Tutorial 1: Trying Out the Examples

Follow this tutorial to learn the very basics of TouchGFX. You will see how to install TouchGFX and how to run the provided examples on TouchGFX Simulator and on an STM32 Evaluation Kit.

Getting Started

First of all make sure you have TouchGFX Designer installed. Read more on how to download and install TouchGFX here.

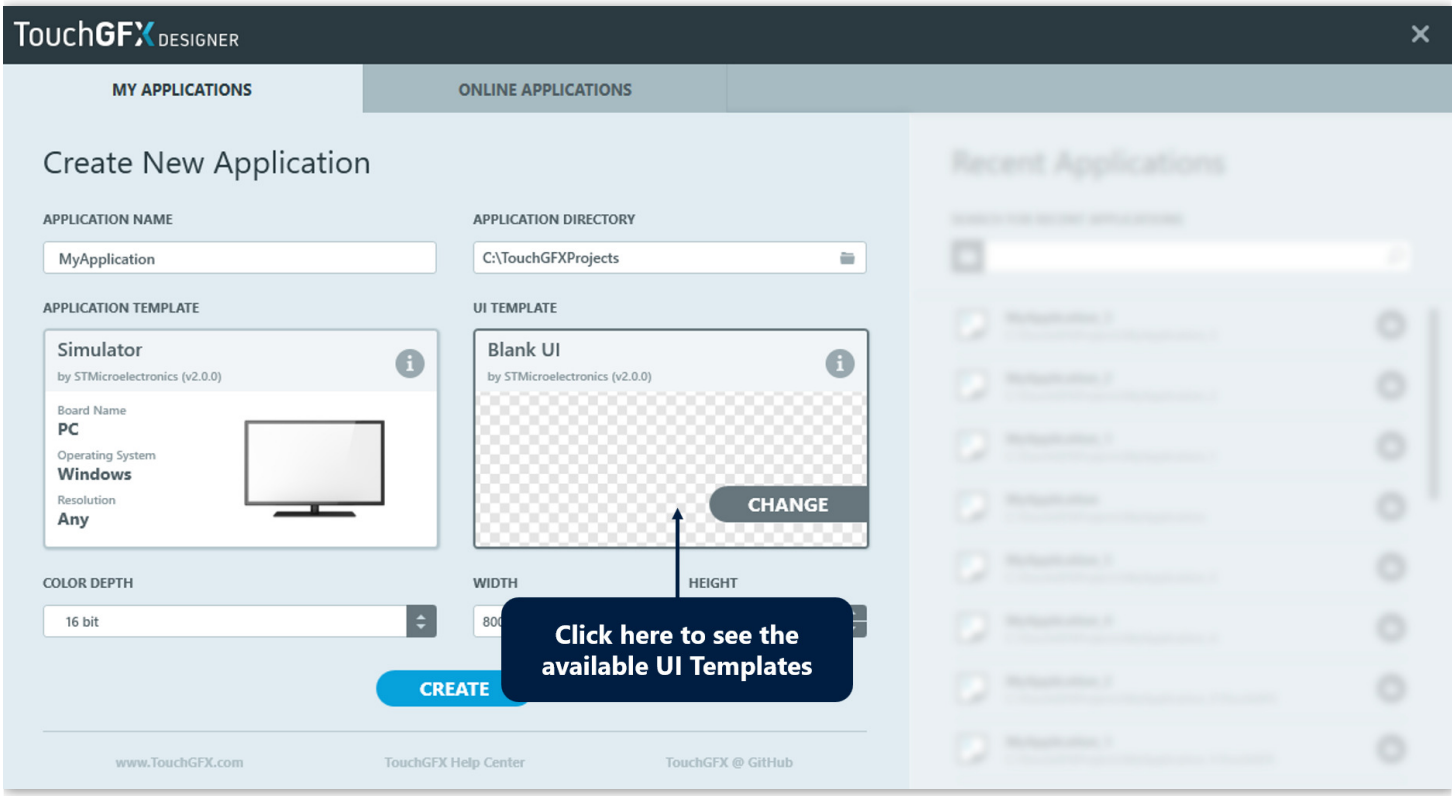
Running an Example Using TouchGFX Simulator

TouchGFX has a lot of UI examples available through TouchGFX Designer. These examples can help you learn more about specific TouchGFX topics, as they all focus on one particular TouchGFX topic or widget.

Selecting a UI Template

You can use the examples as starting points for your own projects or use them as reference examples. The examples can run either on your PC using TouchGFX Simulator, on a STM32 Evaluation Kit or even on your custom STM32 based hardware.

- To create a new example project simply select "File -> New" in the top bar menu in TouchGFX Designer or `CTRL + N` on your keyboard.
- Click the "Change" button in the "UI Template" section to select between all the available examples.

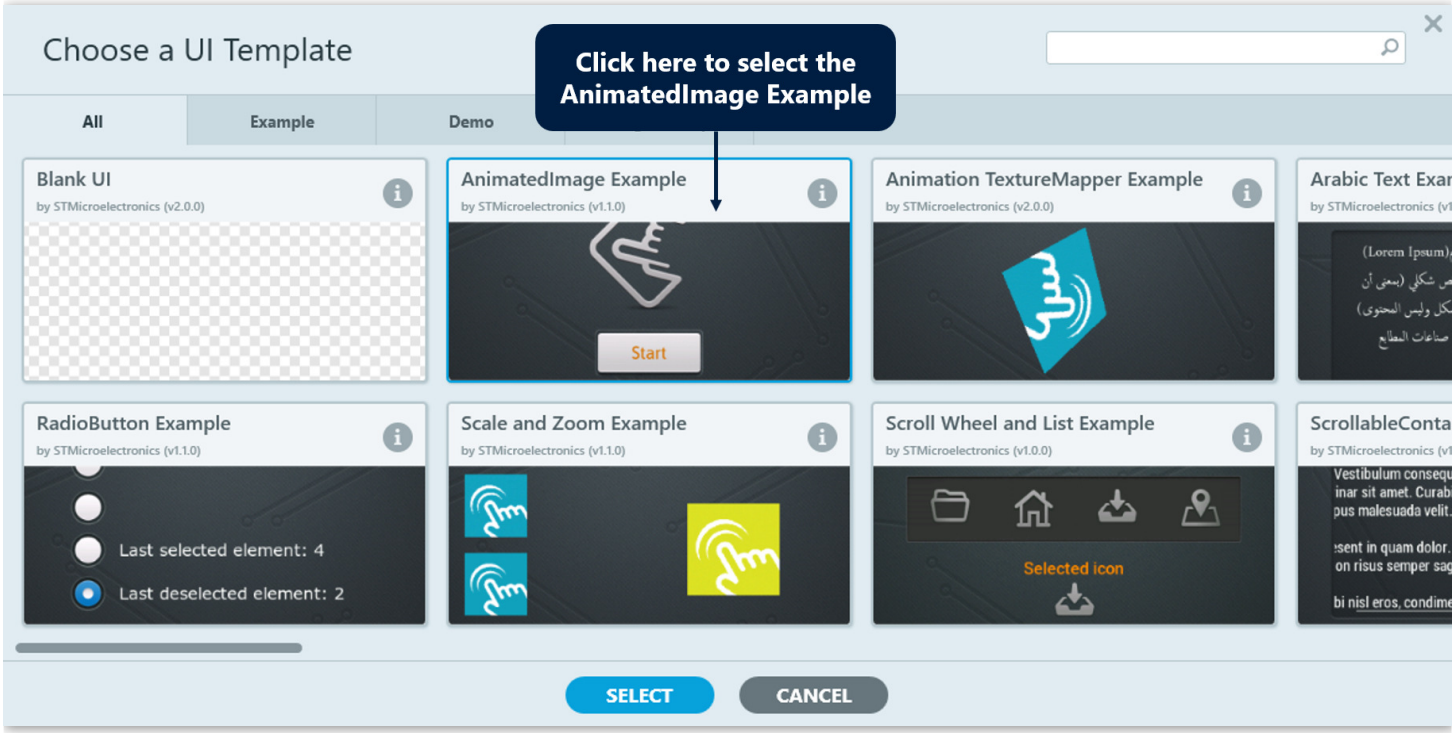


Selecting a UI Template

For this step in the tutorial, we will try out the examples in the Simulator, so leave the "Application Template" unchanged (with the "Simulator" Application Template selected).

TouchGFX Designer will now show you a window with the available UI examples.

- Select the "AnimatedImage Example".
- Click the blue "Select" button in the bottom.

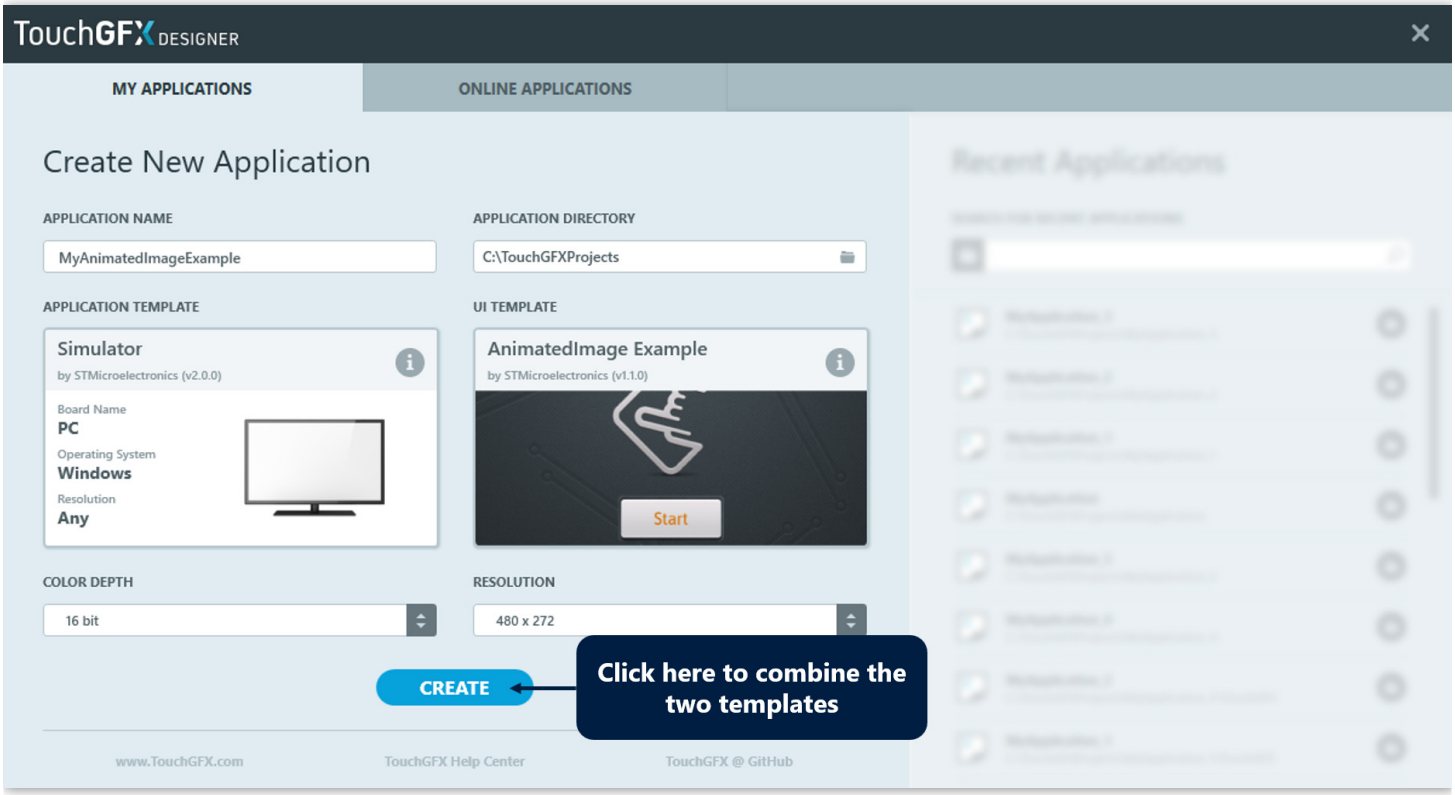


Selecting AnimatedImage Example

Creating a Project

After you have clicked "Select", TouchGFX Designer is ready to create a project for you. Here we have given the project the name "MyAnimatedImageExample".

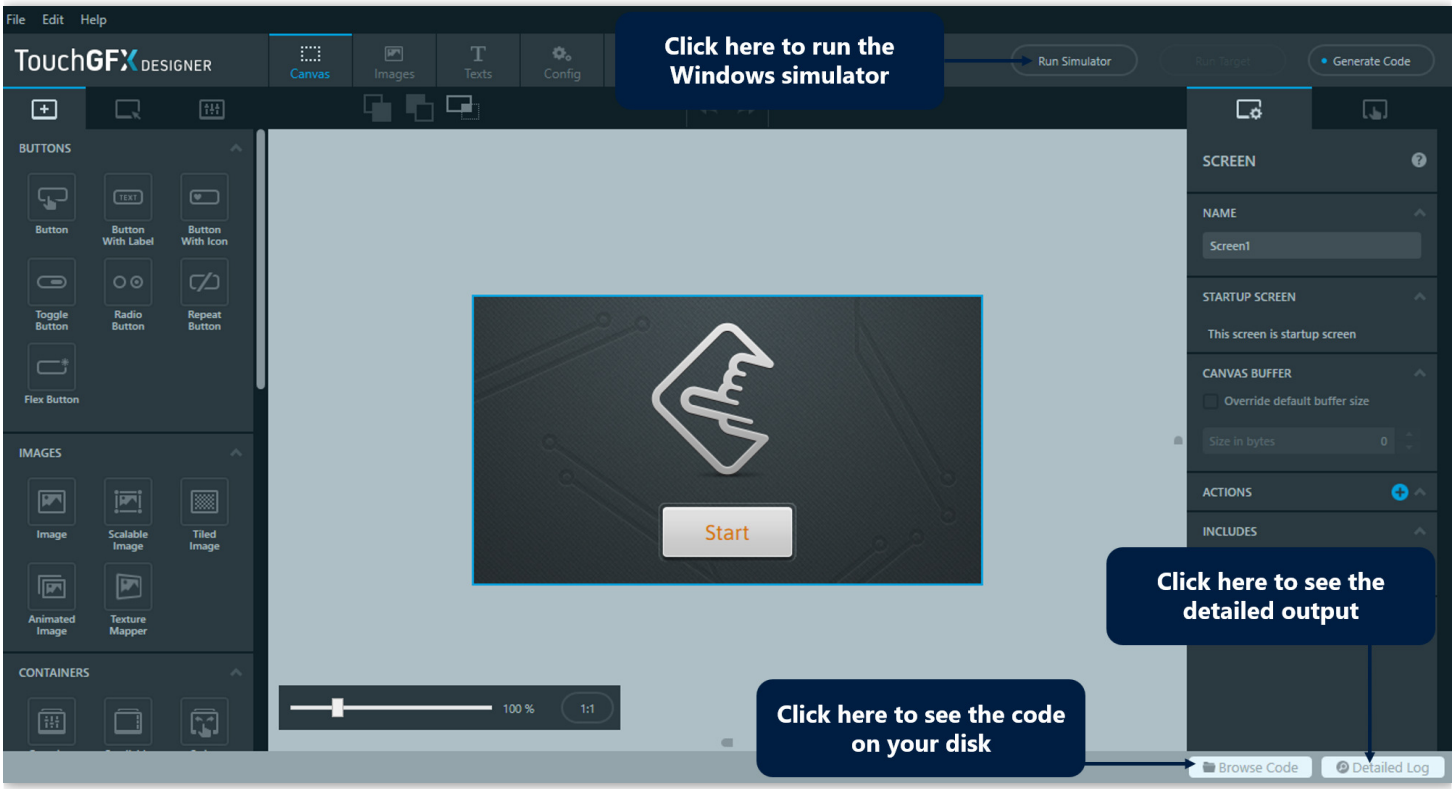
Click "Create" to create the project. TouchGFX Designer will now combine the UI Template you selected with the "Simulator" Application Template to create a complete project. This process takes various amounts of time, depending on your download speed.



Creating the project

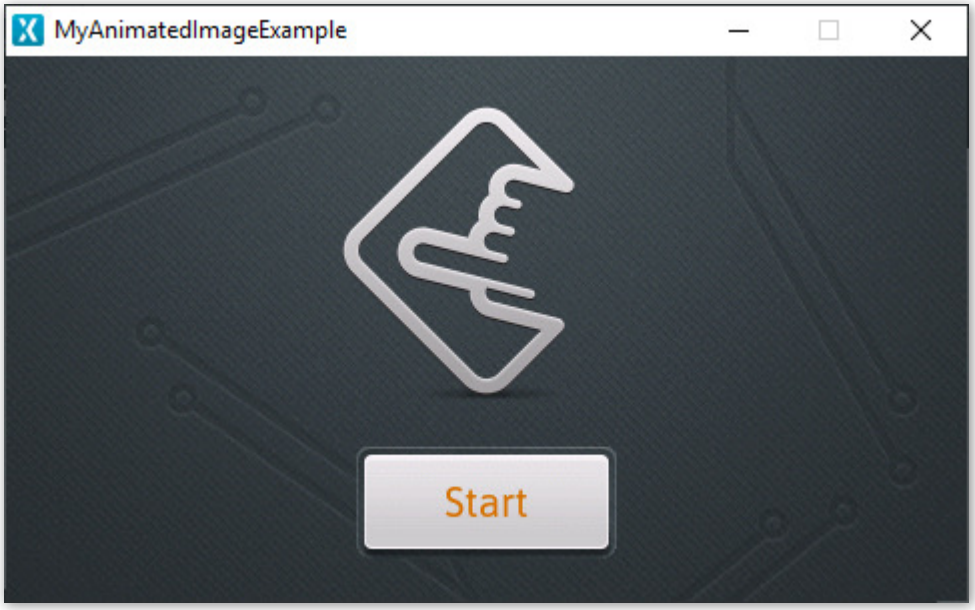
Running TouchGFX Simulator

TouchGFX Designer is now showing the combined project. To run the Windows simulator, click the "Run Simulator" button in the upper right part or `F5` on your keyboard.



The project is ready

TouchGFX Simulator is now showing as a regular Windows application. The titlebar shows the application name. Click the "Start" button to interact with the example.



The TouchGFX Simulator

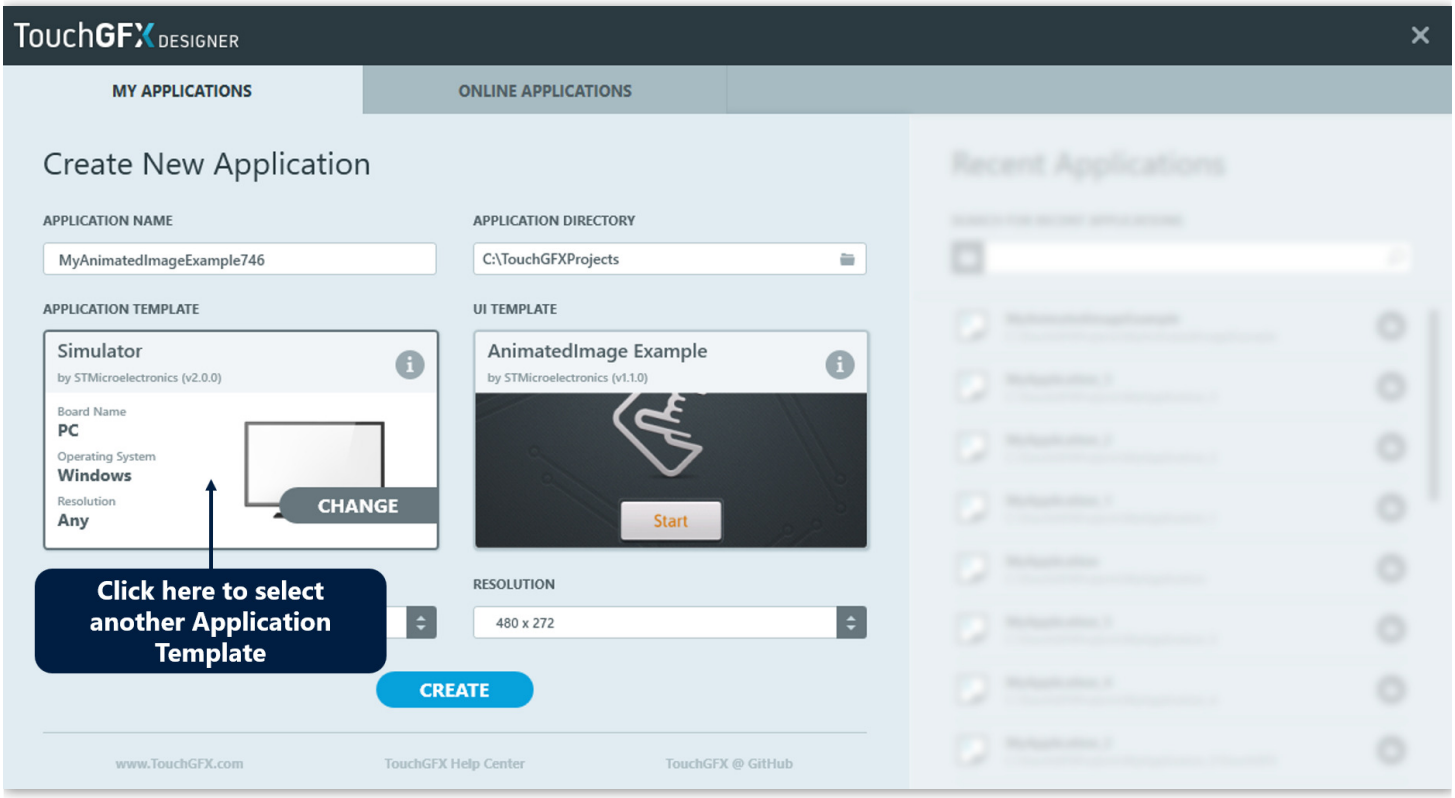
Before moving on with the tutorial you can try out some more examples if you want. Just create a new project and select a new UI Template as before.

Running an Example on an STM32 Evaluation Kit

In this step you will learn how to start a project for a STM32F746-Disco board and how to run one of the TouchGFX examples on that board. If you have no STM32 Evaluation Kit you can simply skip this step. If you have a different SMT32 Evaluation Kit have a look at the list of supported boards and see if you can find it there.

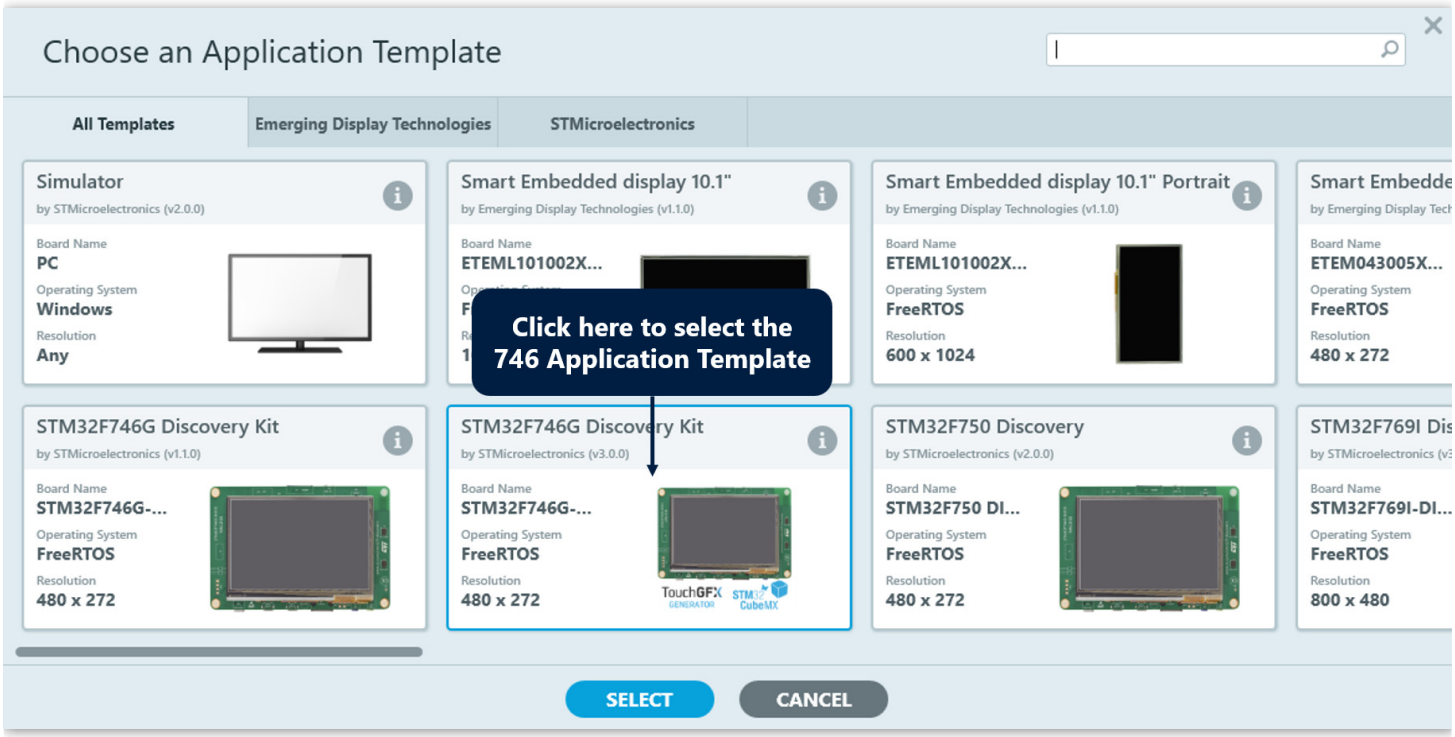
TouchGFX Designer comes with a list of premade Application Templates that matches a wide range of STM32 Evaluation Kit. To base your project on such a template, start out by creating a new project in TouchGFX Designer, by clicking "File -> New" in the top bar menu or `CTRL + N` on your keyboard.

Select the "AnimatedImage Example" as UI Template (if it is not already selected). Click the "Application Template" section to select another Application Template. The default Application Template "Simulator" will only allow you to run on Windows.



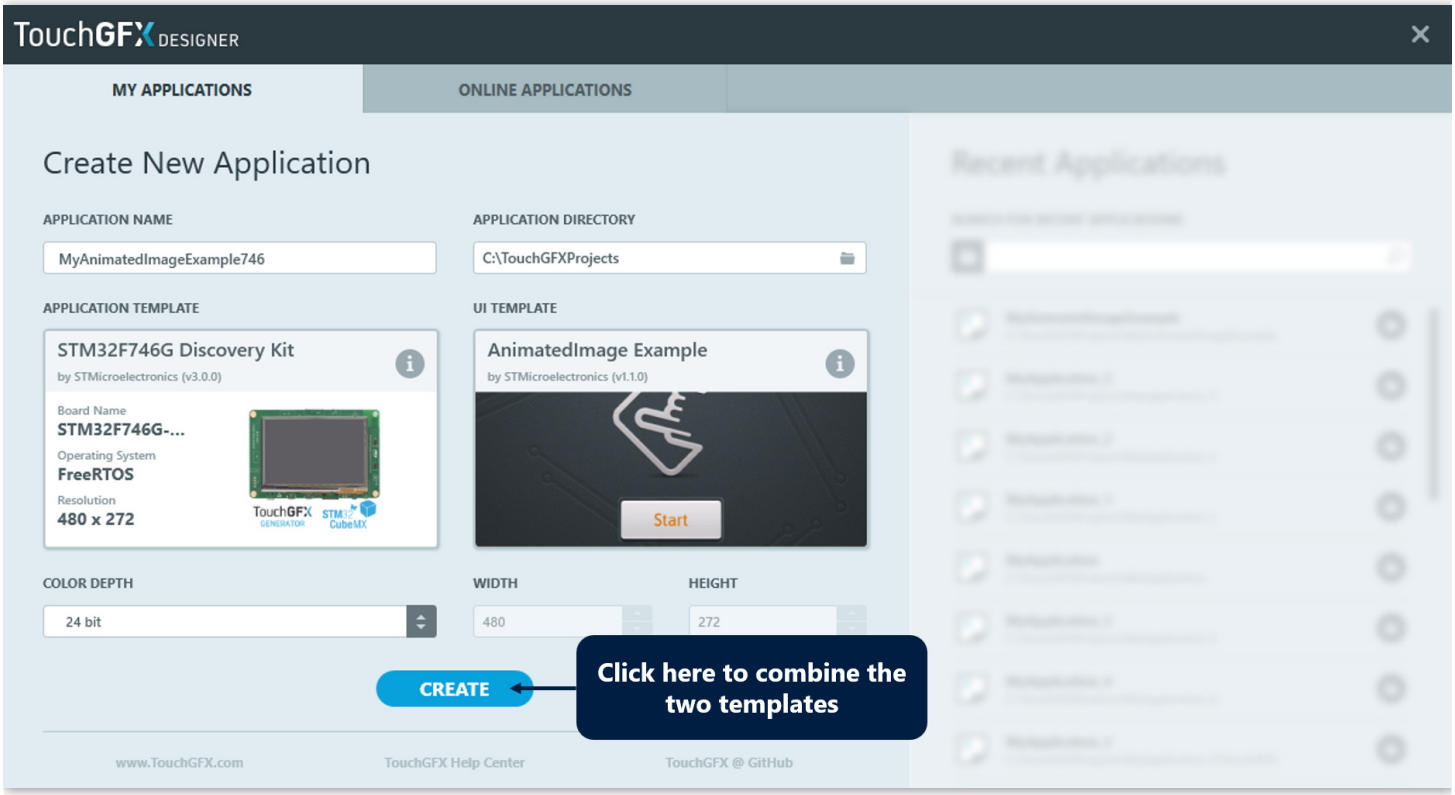
Selecting an Application Template

For this step we will use the STM32F746-Disco board, so click on the "STM32F746G Discovery Kit" and click "Select".



Select Discovery kit

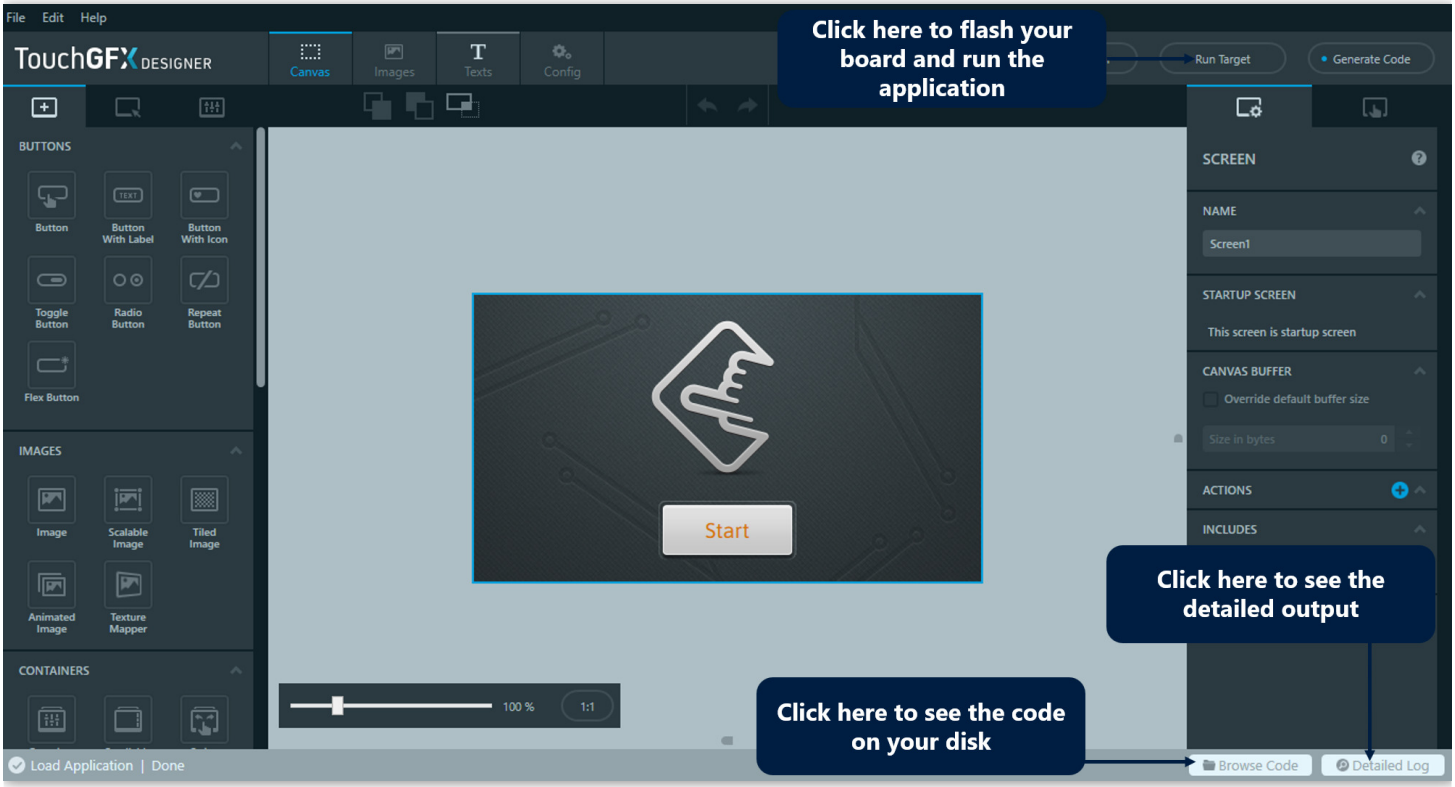
You are now ready to create the project. You can change the application name if you like. Here we have changed it to "MyAnimatedImageExample746". Click the "Create" button to continue.



Create the project

The look of the project is similar to what we saw in the previous step. The only difference is that we now also have a "Run Target" button next to the "Run Simulator" button. When you press this button (or `F6` on your keyboard), TouchGFX Designer compiles your project using the GNU ARM C-compiler and flashes the application to your target. This process can take minutes, depending on your computer speed and the complexity of the application. The progress will be output in the status bar in

the bottom of TouchGFX Designer. You can press the "Detailed Log" (or `ALT + L` on your keyboard) button if you want more details on the build and flashing step.



The project is ready

TouchGFX Designer will write "Flashing Done" in the status bar when flashing is completed. You should now see the application running on your board.

> **NOTE**
>
> **You will need to have Cube Programmer / ST-Link Utility installed to flash a target:**
>
> - **STM32CubeProgrammer**
> - **STM32 ST-LINK Utility**

If you click the "Browse Code" button in the bottom right, TouchGFX Designer will open a file browser showing the directory where the new project is located. Navigating to `build\bin` you can see the following files:

application.map	26-03-2020 15:40	Linker Address Map	891 KB
extflash.bin	26-03-2020 15:40	BIN File	0 KB
intflash.elf	26-03-2020 15:40	ELF File	1.053 KB
intflash.hex	26-03-2020 15:40	HEX File	437 KB
target.elf	26-03-2020 15:40	ELF File	1.054 KB
target.hex	26-03-2020 15:40	HEX File	437 KB

The binary files of the project

The `target.hex` file is the STM32 application for your board. This is the file TouchGFX Designer just programmed to your board.

You can also manually flash your board using Cube Programmer or ST-Link Utility. See the Compiling & Flashing page for more details.

Version: 4.16

Tutorial 2: Creating Your Own Application

Follow this tutorial to learn more about the basics of TouchGFX. You will learn how to add images to your application and use buttons. You will also see how to use texts and calculated numbers. In the last steps you will write code to enhance the look of the UI you have created with TouchGFX Designer. This tutorial assumes no knowledge of TouchGFX, but we assume a little experience with programming.

Step 1: Setting a Background Image

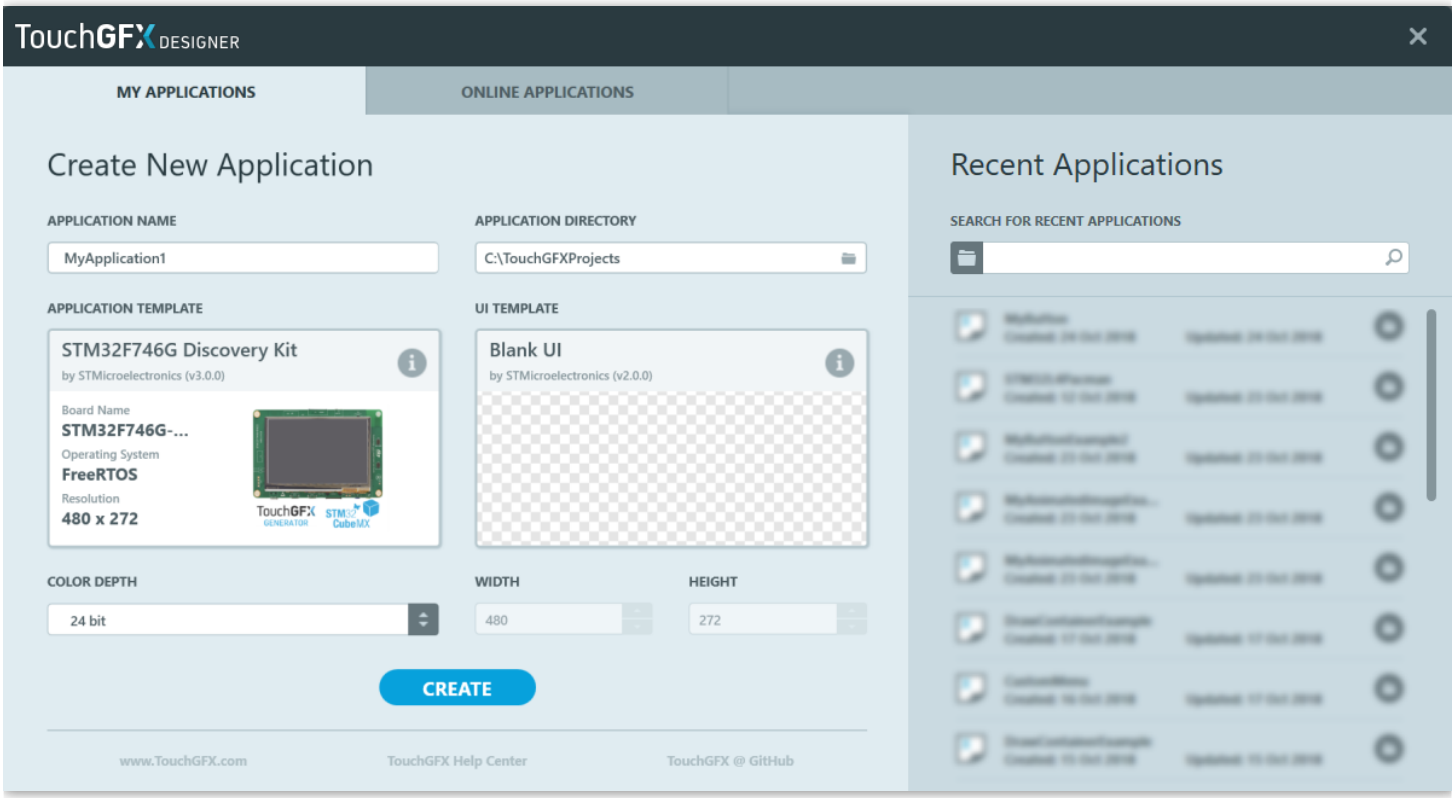
In this first step you will see how to insert a PNG image as a background. But first we will create a new project.

Starting a New Project

Start a new project in TouchGFX Designer. We will call the project "MyApplication1". The project is based on the "STM32F746G Discovery Kit" Application Template and the "Blank UI" UI template.

If you have a different STM32 Evaluation Kit, go ahead and look in the list presented in TouchGFX Designer when you are changing the Application Template to see if it is supported. If you do not have a supported board you can select the "Simulator" Application Template and just run the application on your computer.

Please be aware that this tutorial runs on a display with a resolution of 480x272. If you select an Application Template with a different resolution, the graphics will not fit the screen, but you should be able to complete the tutorial anyway.

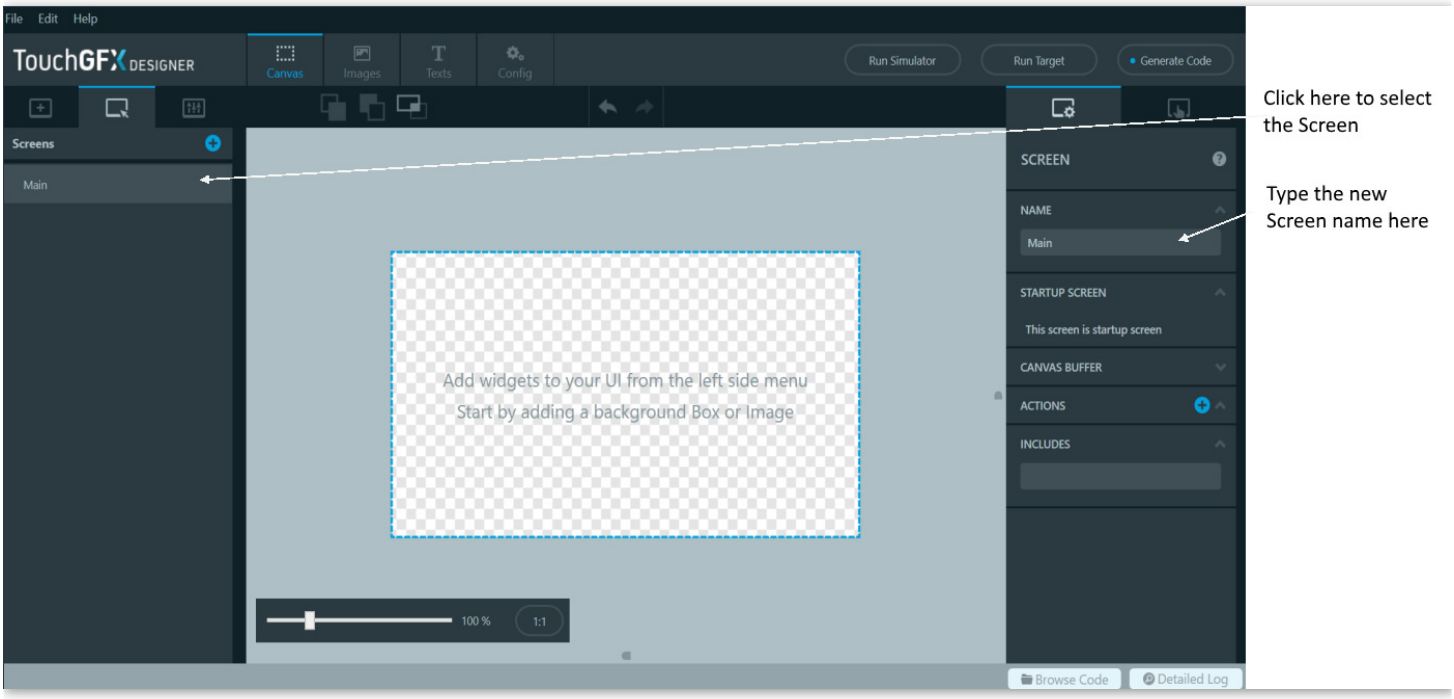


Creating the new project using the STM32F746 Application Template and Blank UI

Now that you have a newly created blank project let us start modifying it.

A TouchGFX application consists of a number of screens. The screens contain a number of widgets that make up the user interface. A Screen covers the whole display, so only one Screen is shown to the user at a time.

The first thing to do is to change the name of the initial Screen to "Main" as illustrated below:



Changing the name of the Screen

Inserting a Background

It is normally a good thing to cover the complete background of a Screen with one or more widgets. For example, this can be a Box or an Image. In this application we will use an Image.

Before we can use an image in TouchGFX Designer, we need to import the file. TouchGFX supports BMP and PNG images (though TouchGFX Designer only supports importing PNG images). PNG files are preferred over BMP files as they are smaller and supports transparent pixels.

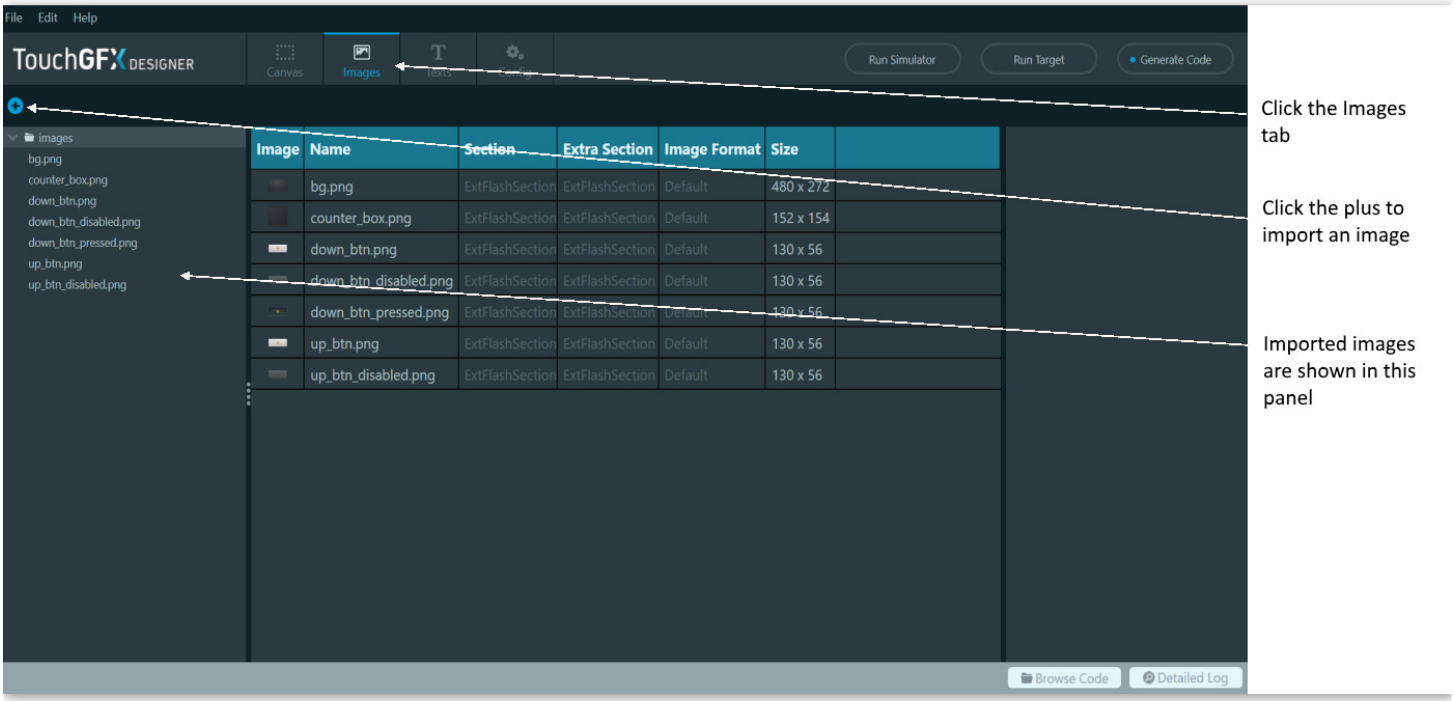
The images we will use in this tutorial can be downloaded from this link. Unzip the file to a directory on your disk.

We want to use the file named "background.png" as our background. To import that file:

- Select the Images tab and click the blue plus icon.
- Navigate to the unzipped folder and select the "background.png" file.
- Press open to import it.

You can also "drag & drop" images from File Explorer onto the image tab, or even directly on the canvas, to import them to your project.

Be aware that images imported to your project will be converted and compiled into your project and thus take up flash space. So only import the images that you need.



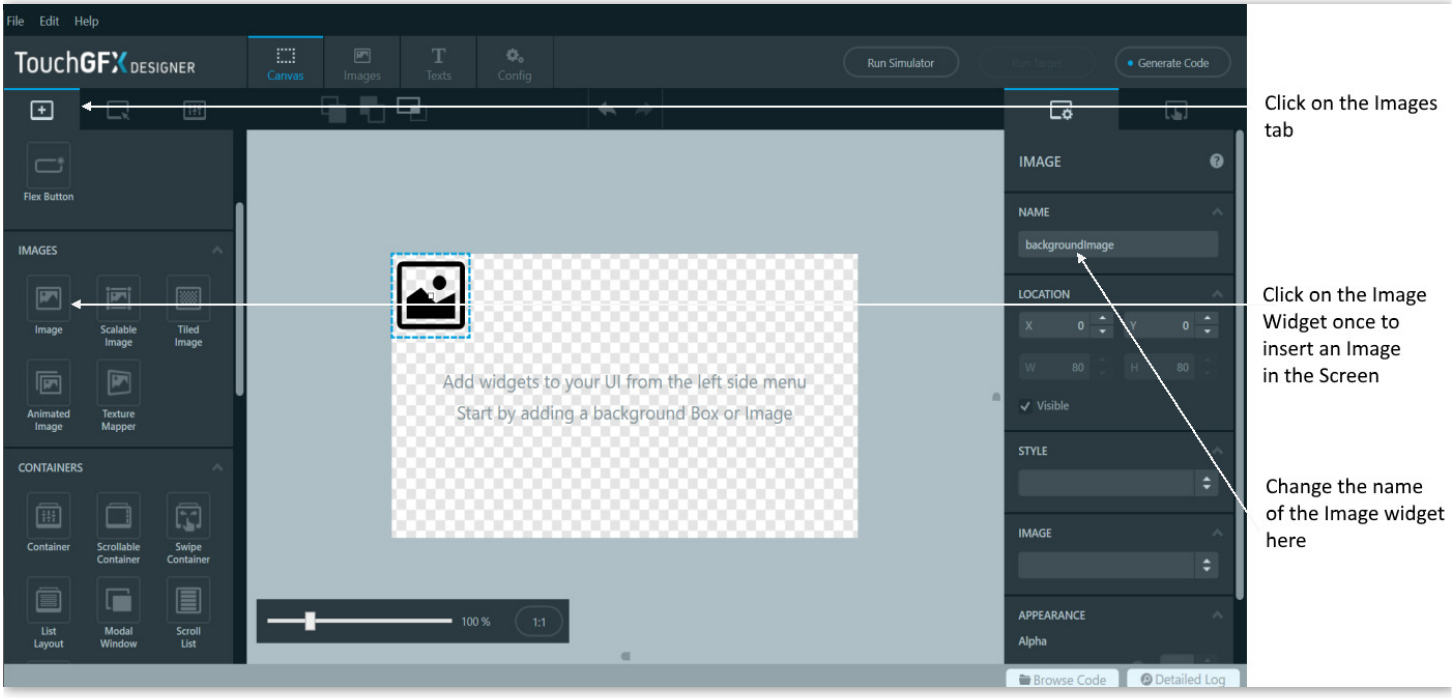
The image background.png is imported

We are now ready to use the image in our application. To do that we need an Image widget.

- Select the Widgets tab in the Canvas tab

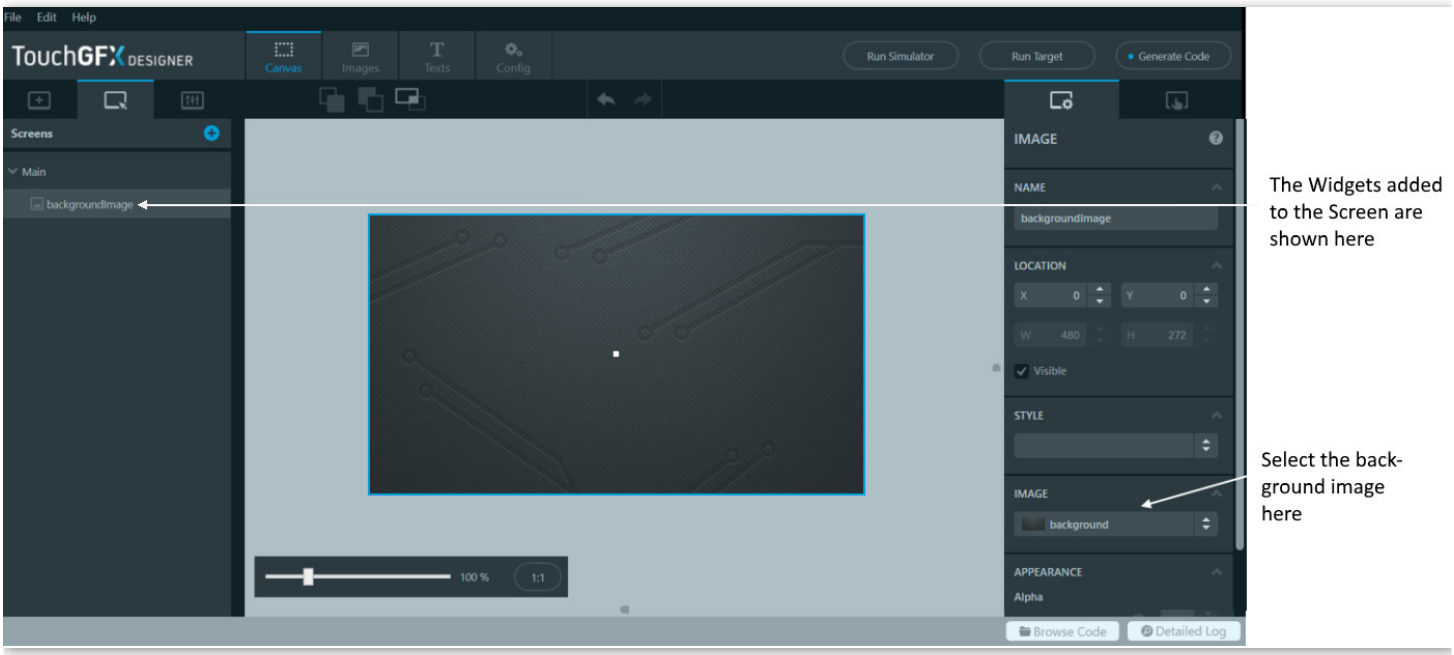
- Find the Image widget in the list of widgets
- Click it to insert an Image Widget on the Screen.

It is a good principle to change the name of widgets to something meaningful. In our case, something like "backgroundImage".



Inserting an Image widget

After inserting a widget we normally need to configure some of its properties like *Position* or *Color*. The properties of the selected Widget are shown to the right in the TouchGFX Designer. In this case we are satisfied with the position in the point 0,0, but we want to change the Image property to select the "background.png" file previously imported. Select the "background.png" in the Image drop-down list.



Selecting the imported image file as background

We have now created a simple application with one Screen consisting of only a background image covering the whole user interface.

Before moving on try to press the "Run Simulator" button to check that the project compiles and runs. You can still not interact with the application since we have not yet added any active widgets.

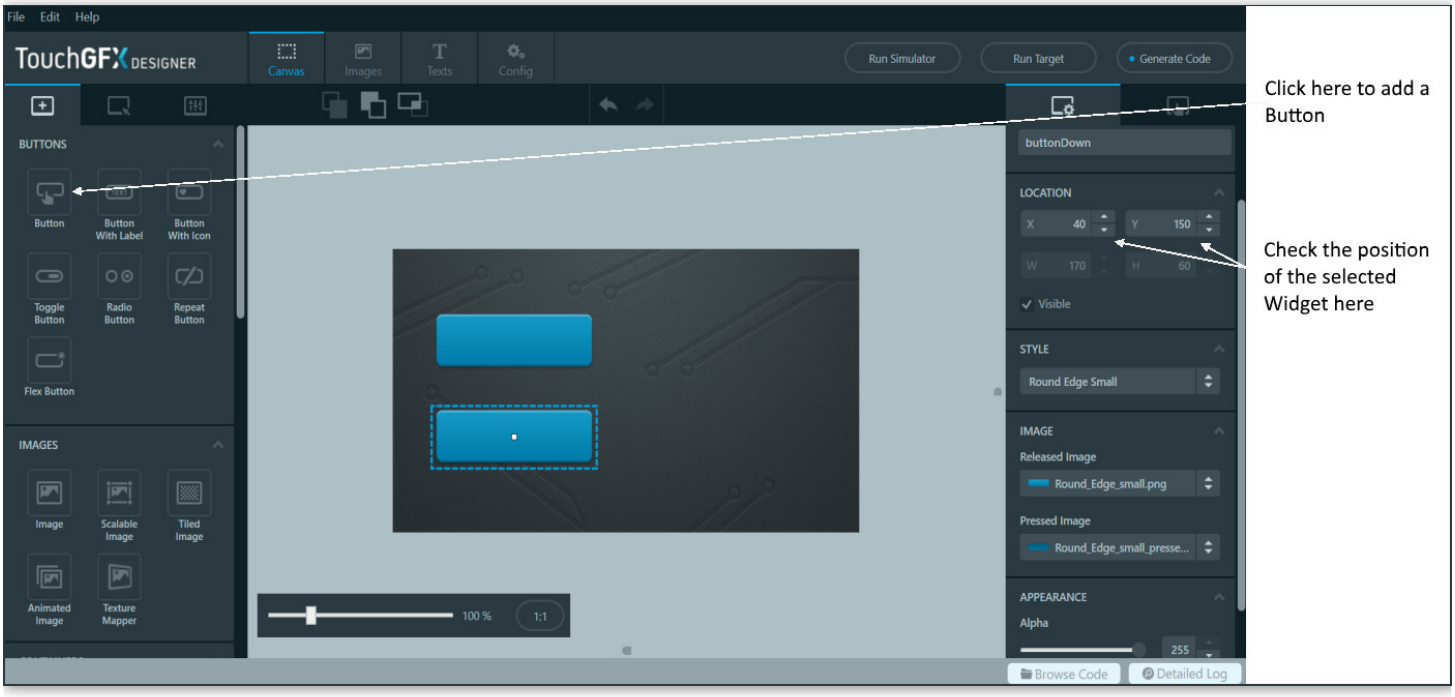
Step 2: Adding Buttons

In this step owe will add two buttons to the application and use different PNG files to give them a customized look.

Adding the Buttons

- Add a button to the Screen by clicking the Button widget in the Widgets tab.
- Move the new widget by dragging it with the mouse.
- Position the button at x=40, y=60.
- Name the new Widget "buttonUp".
- Add another Button at position x=40, y=150. Name this widget "buttonDown".

The project now looks like this:



Adding two buttons

You can use the small up/down button on the X and Y properties to fine-tune the position of the widgets. You can also select the button widget (by clicking it on the canvas) and adjust the position using the arrow keys on your keyboard.

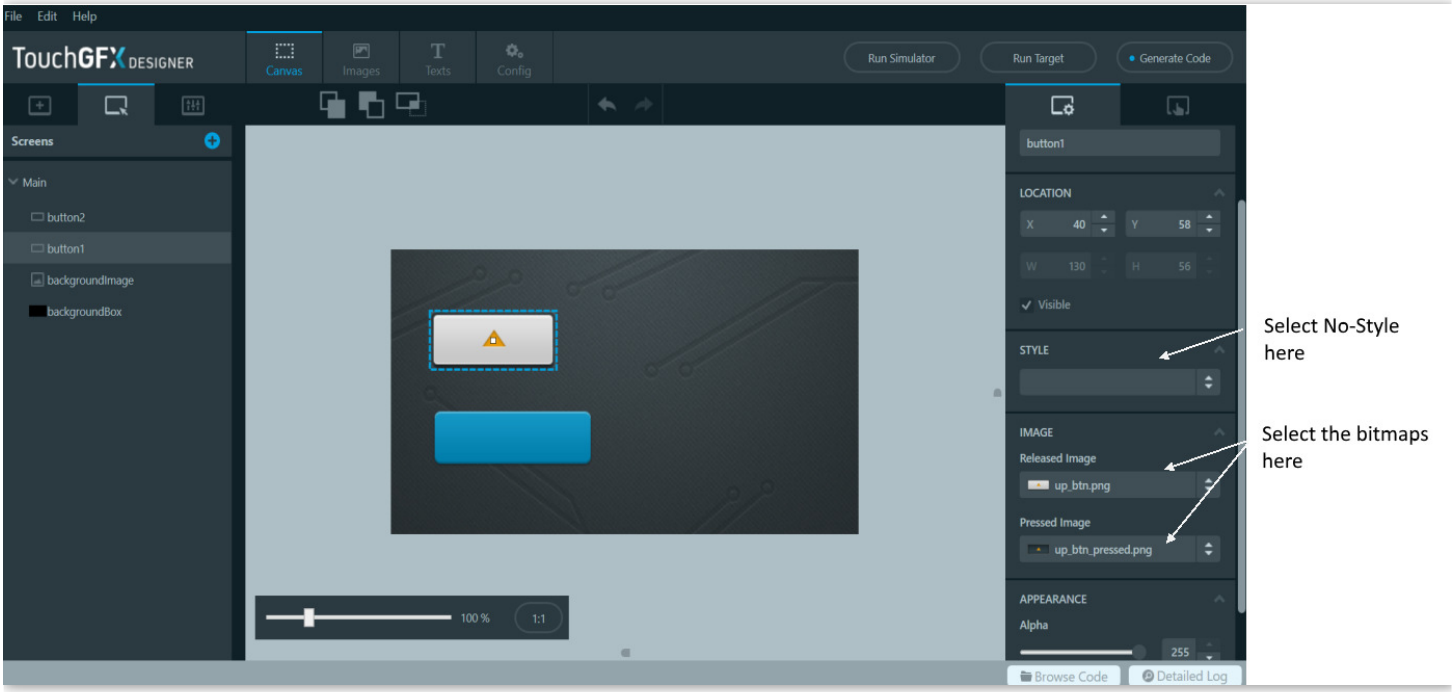
Changing the Look

We will now change the look of the buttons. A Button is made up of two images. One image is shown when the button is pressed, and another image is shown when the button is not pressed (released). Most widgets come with a set of predefined styles, which is basically a set of values for certain properties of the widget describing a particular look. These styles are good for fast prototyping, but most often you will replace them when creating a real application.

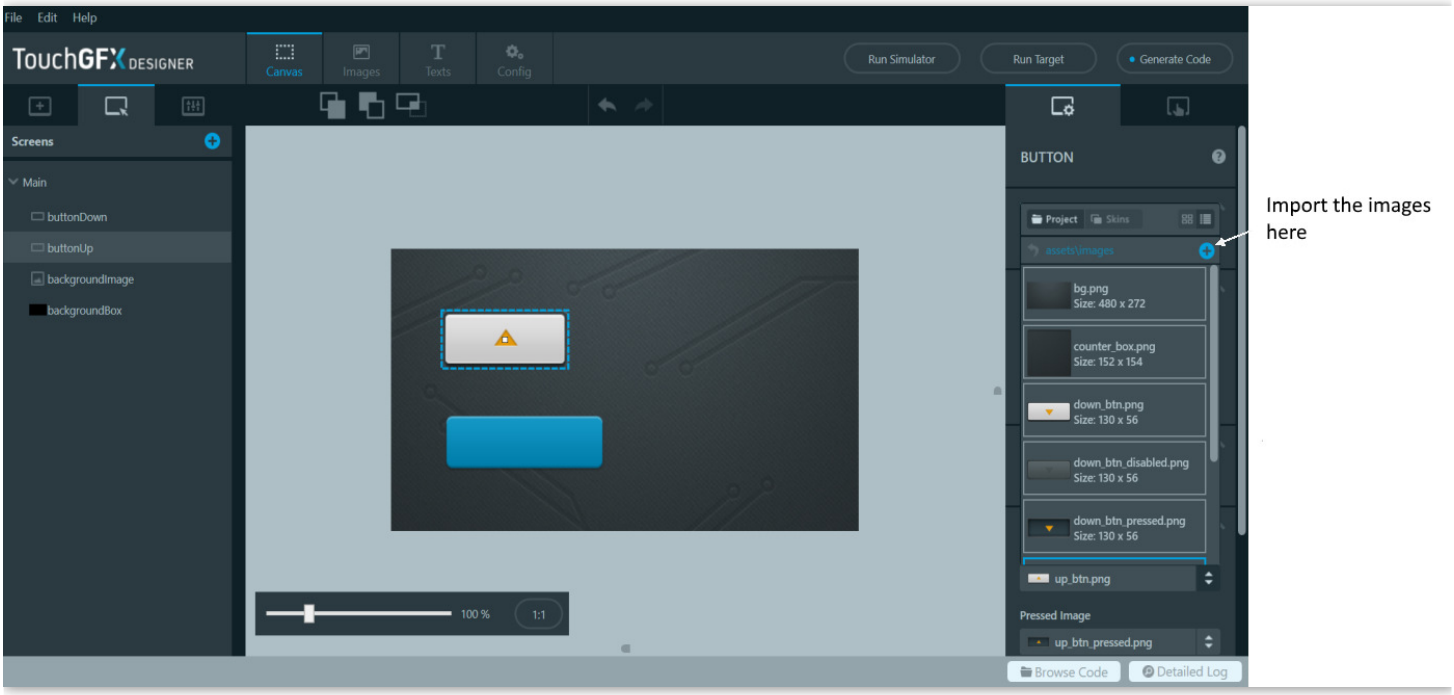
Go to the Images tab as in previous step and click the "plus" icon to import some images. This time import the images: "button_down_pressed.png", "button_down_released.png", "button_up_pressed.png", and "button_up_released.png".

Now select the "buttonUp" button. For that button, select "button_up_released.png" for the Release Image property. Select "button_up_pressed.png" for Pressed Image.

You can immediately see the look of the button on the canvas in TouchGFX Designer.



Setting bitmaps for buttons

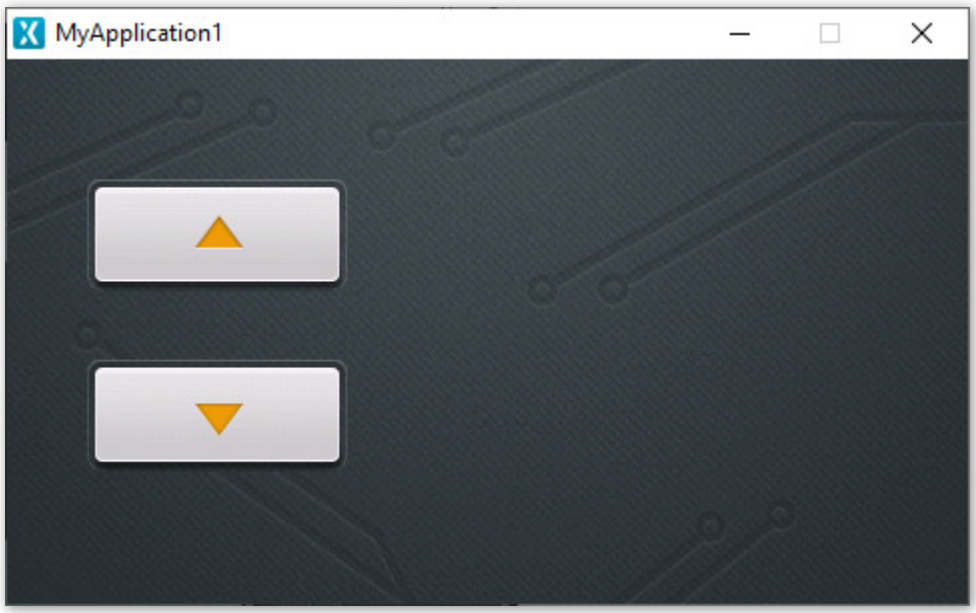


Setting bitmaps for buttons

For "buttonDown", select "button_down_released.png" for Released Image, "button_down_pressed.png" for Pressed Image.

You have now finished setting up the buttons. Click "Run Simulator" to try your application.

Try both buttons to verify that the buttons are configured correct.



Running the Simulator

TIP

Most widgets in TouchGFX uses images to define their size, meaning that they cannot directly be resized. This is done for performance reasons (see **General UI Component Performance**). If you want to change the size of such widgets, like for example the buttons in this tutorial, you will do this by creating a new set of images for the buttons and use them as Released and Pressed images instead.

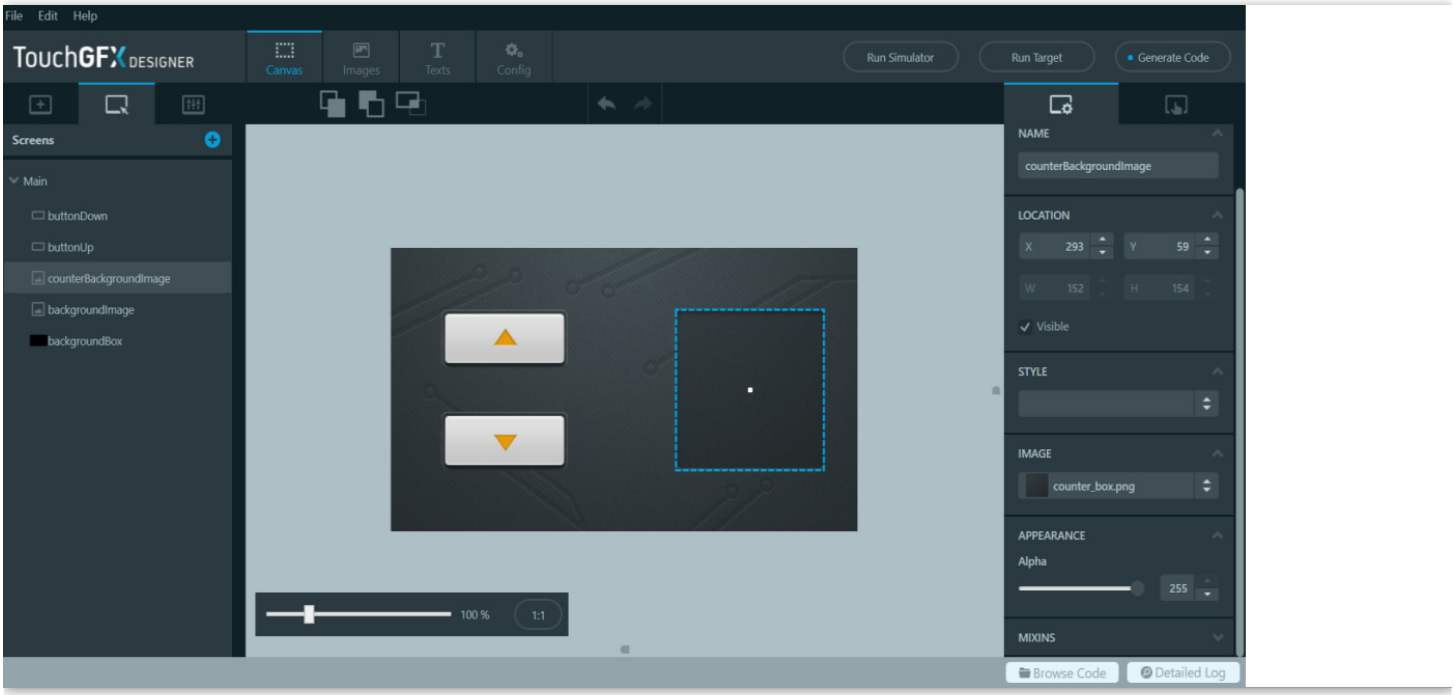
Step 3: Adding Text

In this step we will add a large TextArea widget to the application.

All text is shown using a TextArea widget, but before we add a TextArea to the application, we will add another Image to give the text a better background.

Text Background

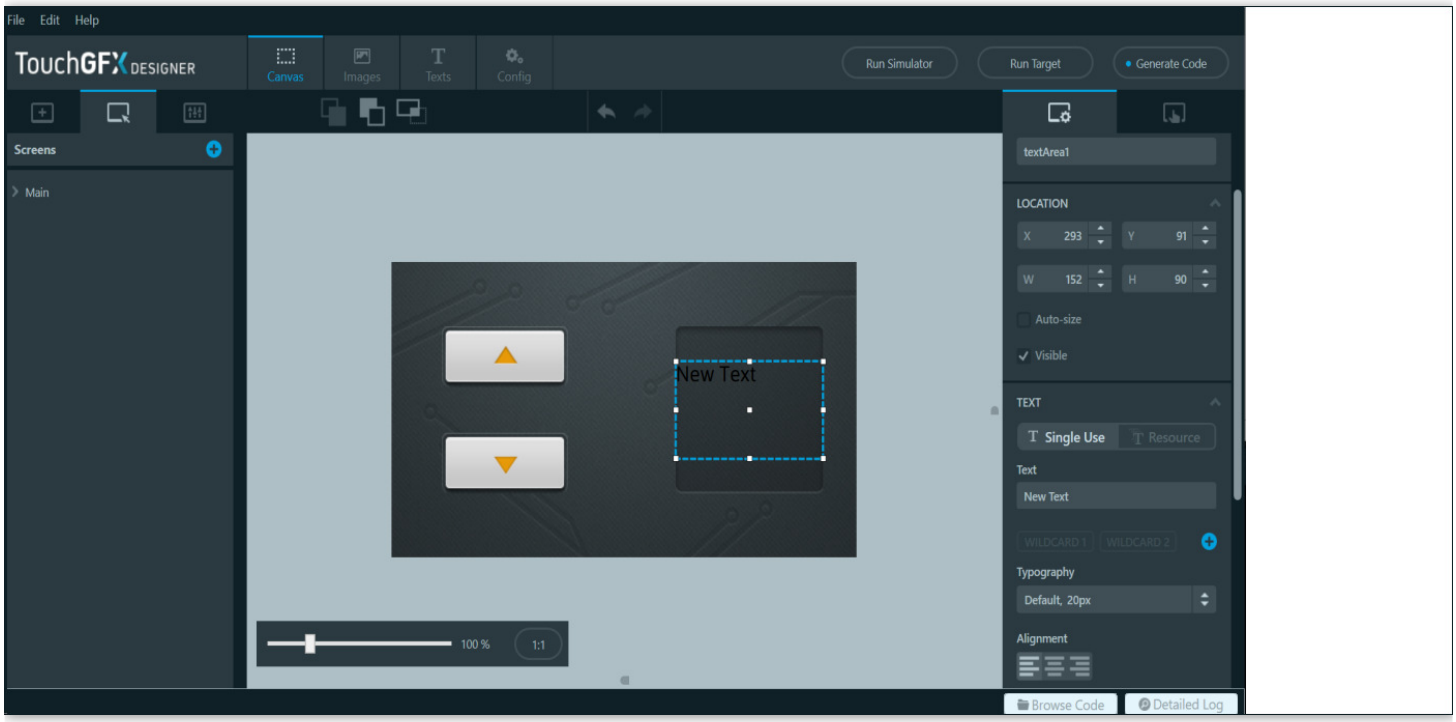
- Import another image file, "counter_box.png".
- Insert a new Image widget
- Name it "textBackground"
- Position it at x=250, y=59.
- Set *Image* property to "counter_box".



Added background for text

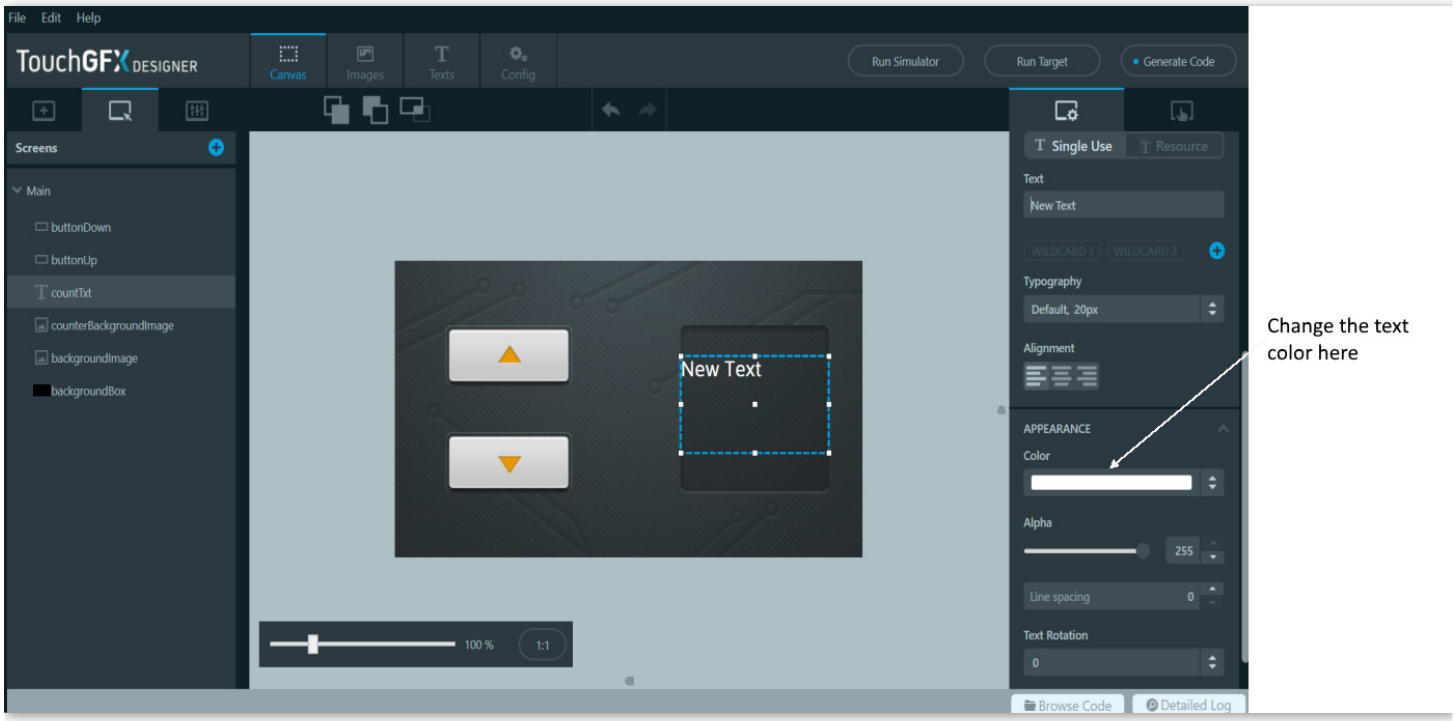
Adding the Text

We are now ready to add a TextArea widget. Click the TextArea icon in the Widgets tab. Rename the widget to "textCounter" and move the widget to position x=250, y=90. We want the widget to show a large text, so un-check the *Auto-size* property, and set the size to a fixed width=152, and height=90.



Added a TextArea

The default color of a TextArea widget is black, which is rather dark on our background. Select the *Color* property of "textCounter", and change the color to white.

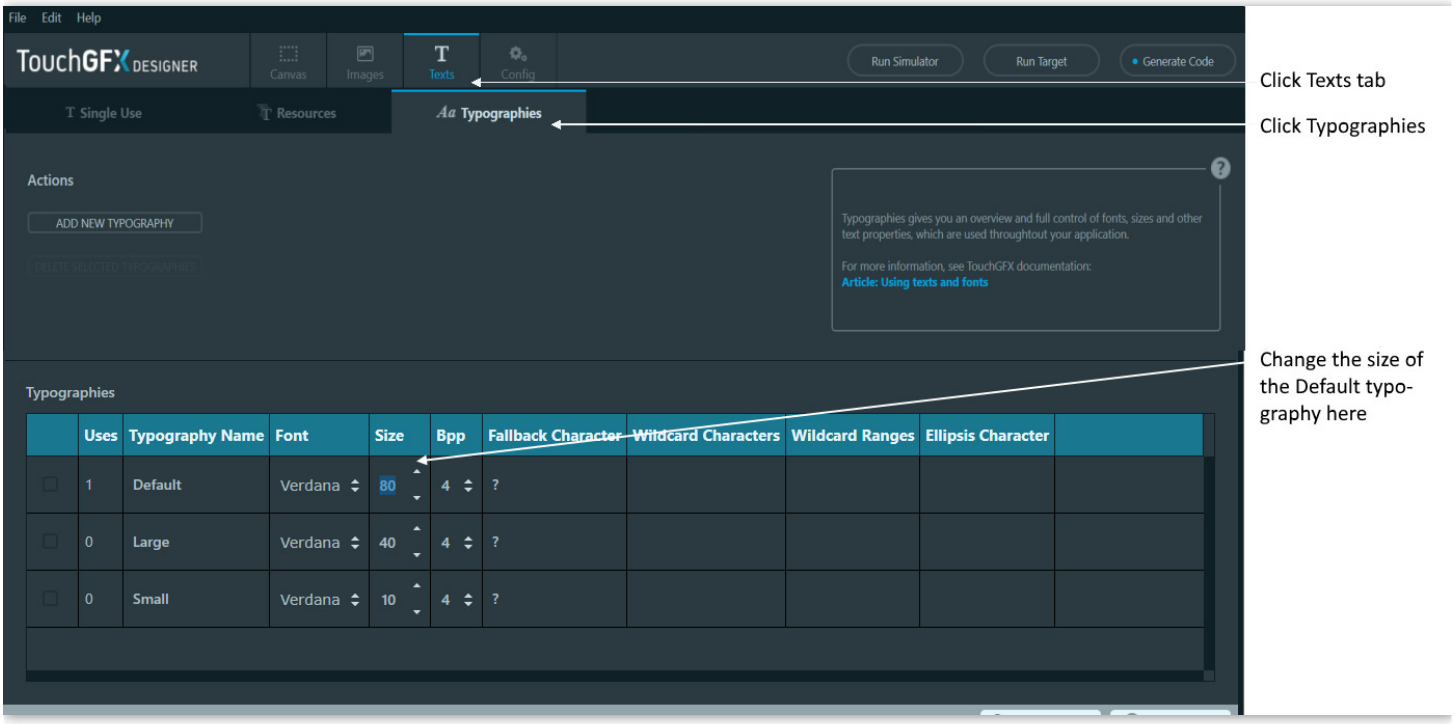


Changing text color

Changing the Text Typography

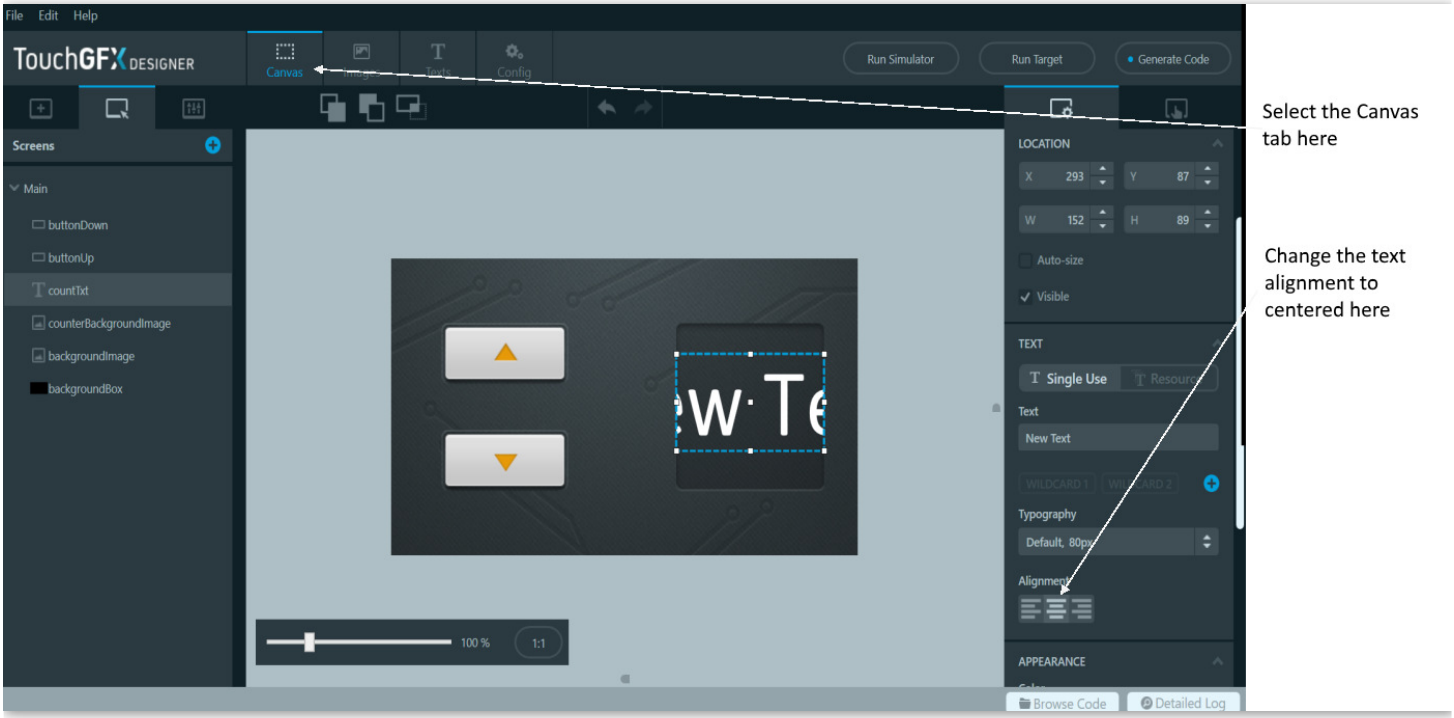
We want the text to be bigger. The way to do that is to change the *Typography* used for the text. A typography defines the *Font* (e.g. Verdana), the *Size*, and the *Alignment* (left, right, or center) for a text.

Select the Texts tab in the top of TouchGFX Designer, click Typographies, and update the size of the "Default" typography to 80.



Changing text size

Going back to the Screen (by clicking the "Canvas" tab in the top), we see that the text is much bigger now. In fact we cannot read the complete text "New Text". Click the centered icon under the *Alignment* property to get the text centered.

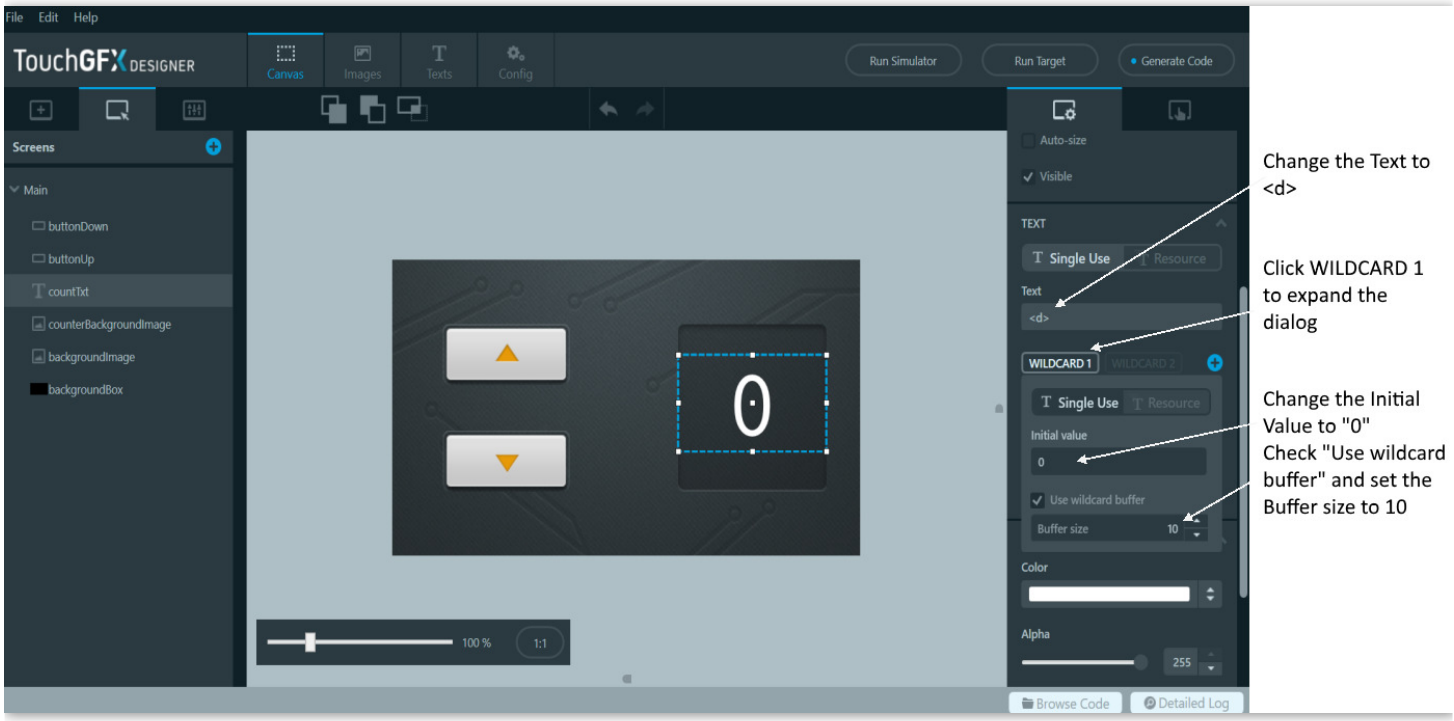


Changing text alignment

Using a Wildcard Text

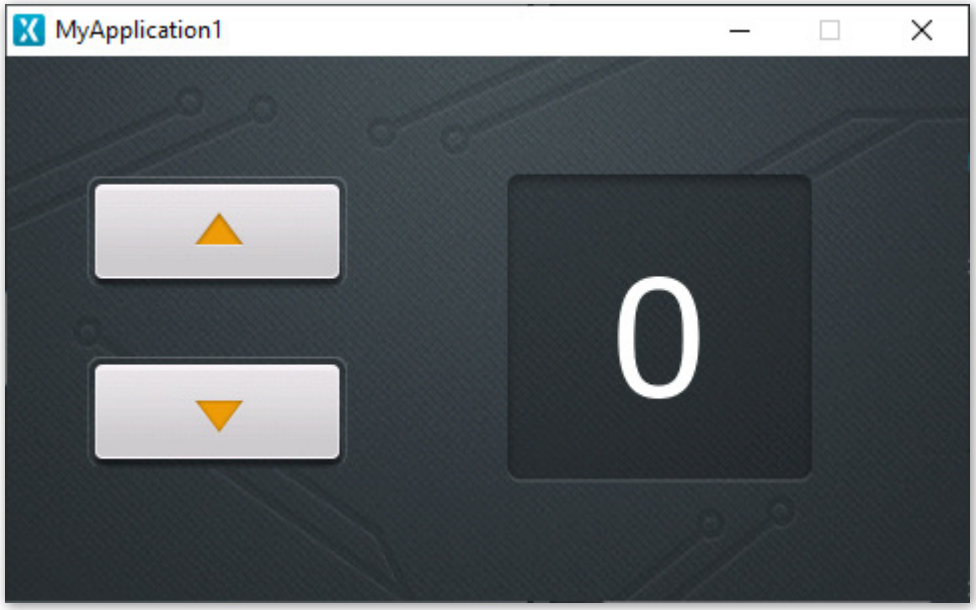
We want the TextArea to show a number that we can change with the buttons. To do that, we must change the text to include a "wildcard". A wildcard is a marker ("<d>") in the text that can be substituted with something else at runtime. We just want to show a number, so we will change the text to just "<d>". In other projects you can combine the dynamic parts with a fixed text, e.g. "Temperature: <temp> °C".

> **NOTE**
>
> **The text inside the <...> wildcard brackets are optional. You can use them to communicate to implementers or translators what kind of information will be inserted in the wildcard or you can leave it empty.**



Configure the wildcard text

Click "Run Simulator" to try your application.



Running the Simulator

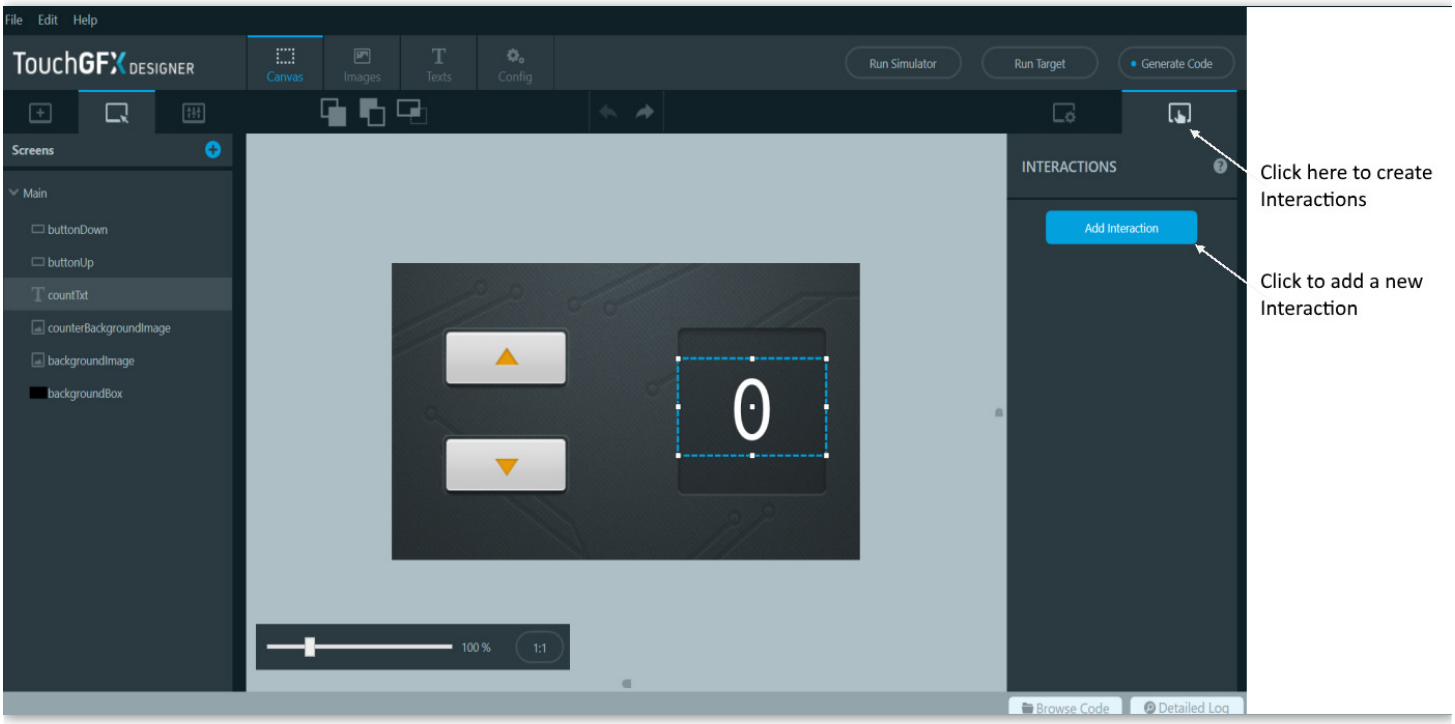
FURTHER READING

To learn more about using texts and fonts, read the **Texts and Fonts** page.

Step 4: Adding Code

With TouchGFX Designer it is easy to link actions to a Button through an Interaction. An Interaction links a Trigger (e.g. a button press) to an Action (e.g. running code or moving an element).

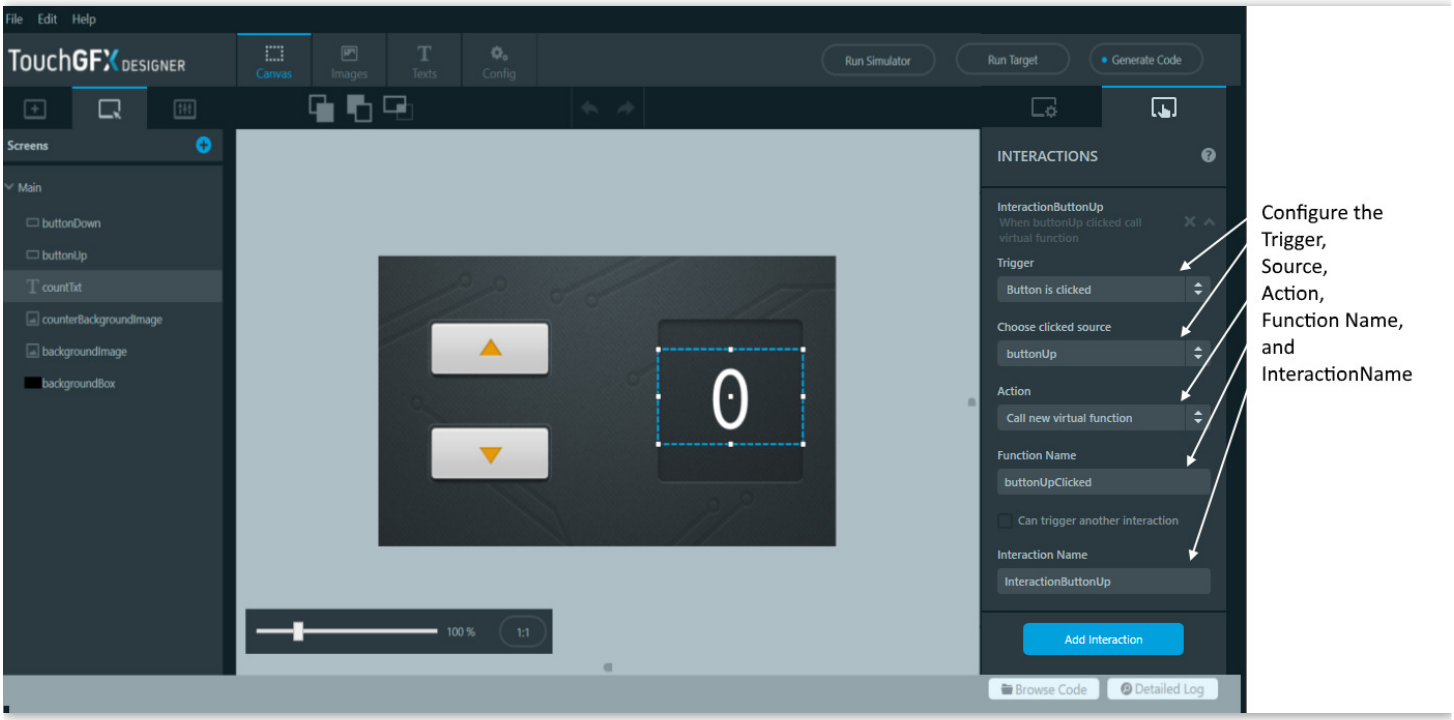
Select the Interactions tab in the upper right corner and click the "Add Interaction" button to create a new Interaction



Adding interactions

We will create two interactions, one for each of the buttons. We will setup both interactions to call a C++ method on the current Screen.

- Change the *Trigger* property to "Button is clicked".
- Set the *Choose clicked source* property to "buttonUp".
- Change the *Action* property to "Call new virtual function".
- For *Function Name*, type "buttonUpClicked".
- You should also give the Interaction an informative name, so that you can recognize it later on.



Configure the Button interaction

Create a similar Interaction with "buttonDown" as "clicked source":

- Change the *Trigger* property to "Button is clicked".
- Set the *Choose clicked source* property to "buttonDown".
- Change the *Action* property to "Call new virtual function".
- For *Function Name*, type "buttonDownClicked".
- You should also give the Interaction an informative name, so that you can recognize it later on.

If you either click the "Generate Code" button or "Run Simulator" button, TouchGFX Designer will update the generated code with the information you entered in the interactions just created. This means that it will create two new virtual functions in the view base class for this screen.

Let us investigate this more and see how we can have our own code executed. Click the "Browse Code" button in the bottom bar. This will give you a File Explorer placed in your application folder. Navigate to the folder:

`MyApplication1/generated/gui_generated/include/gui_generated/main_screen/`

and open the file `MainViewBase.hpp`. If you like you can also open one of the project files and find the file here:

IDE	Path to project file
Visual Studio	`simulator/msvs/Application.sln`
CubeIDE	`ProjectName.ioc`

IDE	Path to project file
IAR Embedded Workbench 8	`target/IAR8.x/application.eww`
KEIL uVision v5	`target/Keil/application.uvprojx`

> **NOTE**
>
> Not all project files are present as default. To change toolchain you need to use the CubeMX tool. Read more on this on the **Using IDEs with TouchGFX** page.

The new virtual methods are found in the public part of the `MainViewBase` class. The generated methods have empty implementations. The intention is that the programmer implements these methods in the subclass `MainView`.

MainViewBase.hpp

```
//********************************************************************************/
/********** THIS FILE IS GENERATED BY TOUCHGFX DESIGNER, DO NOT MODIFY **********/
/********************************************************************************/
#ifndef MAINVIEWBASE_HPP
#define MAINVIEWBASE_HPP

#include <gui/common/FrontendApplication.hpp>
#include <mvp/View.hpp>
#include <gui/main_screen/MainPresenter.hpp>
#include <touchgfx/widgets/Image.hpp>
#include <touchgfx/widgets/Button.hpp>
#include <touchgfx/widgets/TextAreaWithWildcard.hpp>

class MainViewBase : public touchgfx::View<MainPresenter>
{
public:
  MainViewBase();
  virtual ~MainViewBase() {}
  virtual void setupScreen();

  /*
    * Custom Action Handlers
    */
  virtual void buttonUpClicked()
  {
    // Override and implement this function in MainView
  }

  virtual void buttonDownClicked()
  {
    // Override and implement this function in MainView
  }
```

```
...
```

Implementing the Virtual Methods

The remaining task is now to implement these two methods to change the counter value when the user presses the buttons. To do that, declare the methods again in the `MainView` class. This class can be found in:

`MyApplication1/gui/include/gui/main_screen/MainView.hpp`

Open this file and insert the two function declarations in the class:

MainView.hpp

```
#ifndef MAIN_VIEW_HPP
#define MAIN_VIEW_HPP

#include <gui_generated/main_screen/MainViewBase.hpp>
#include <gui/main_screen/MainPresenter.hpp>

class MainView : public MainViewBase
{
public:
  MainView();
  virtual ~MainView() {}
  virtual void setupScreen();
  virtual void tearDownScreen();
  virtual void buttonUpClicked();
  virtual void buttonDownClicked();
}
```

The next task is to implement the two methods by adding the implementation in the .cpp file. This file is located in:

`MyApplication1/gui/src/main_screen/MainView.cpp`

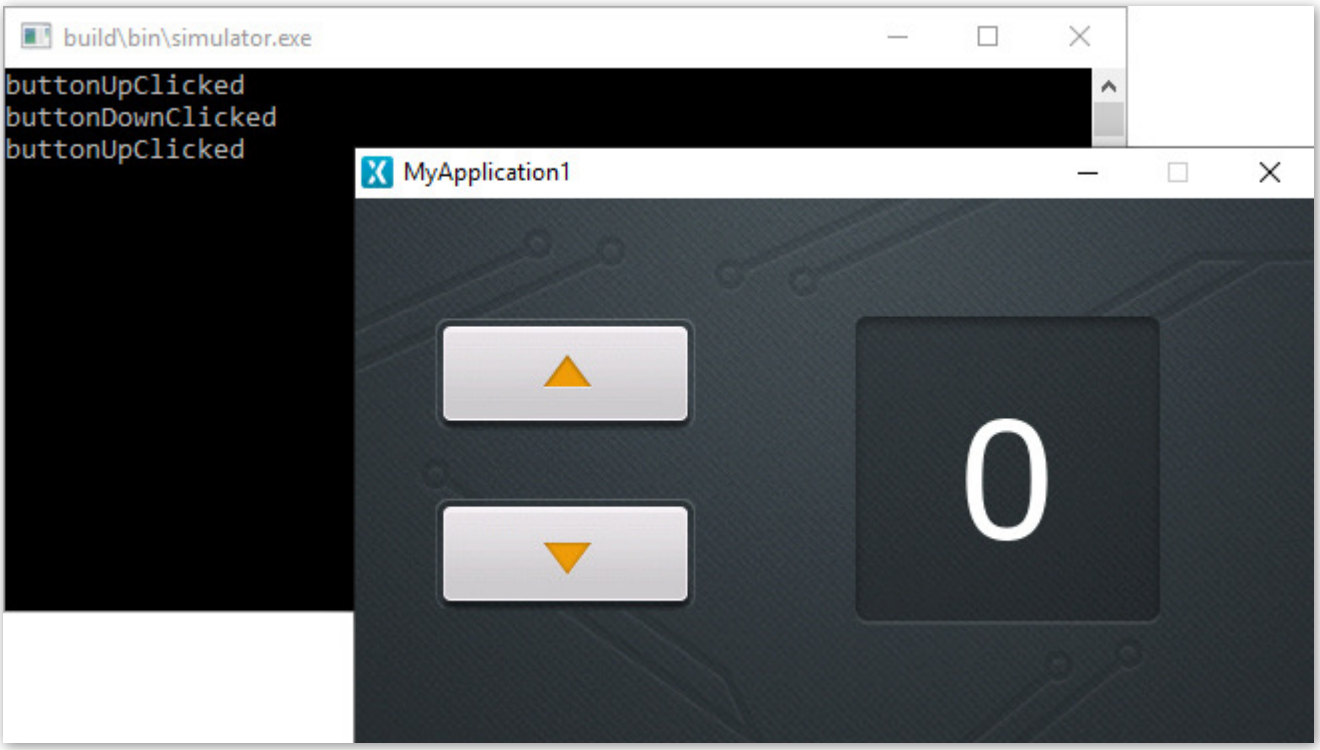
In the implementation below we have added calls to `touchgfx_printf`. This function is useful to print out lines of text when running the simulator. When running on target, the line will have no effect.

MainView.cpp

```
#include <gui/main_screen/MainView.hpp>

MainView::MainView()
{
```

```
}

void MainView::setupScreen()
{
  MainViewBase::setupScreen();
}

void MainView::tearDownScreen()
{
  MainViewBase::tearDownScreen();
}

void MainView::buttonUpClicked()
{
  touchgfx_printf("buttonUpClicked\n")
}

void MainView::buttonDownClicked()
{
  touchgfx_printf("buttonDownClicked\n")
}
```

Click "Run Simulator" in TouchGFX Designer again to run the new code. Click the buttons a couple of times to see that the interactions and methods are working as expected:



Running the Simulator

Updating the Counter Value

The last task is to write C++ code in the new methods to update the counter value when the user presses the button. To do that we first add a new integer variable, `counter`, in the `MainView` class:

MainView.hpp

```
#ifndef MAIN_VIEW_HPP
#define MAIN_VIEW_HPP

#include <gui_generated/main_screen/MainViewBase.hpp>
#include <gui/main_screen/MainPresenter.hpp>

class MainView : public MainViewBase
{
public:
  MainView();
  virtual ~MainView() {}
  virtual void setupScreen();
  virtual void tearDownScreen();
  virtual void buttonUpClicked();
  virtual void buttonDownClicked();
protected:
  int counter;
}
```

In the `buttonUpClicked` method we increment the counter value. The new value is then converted to a string and copied to the 10 characters buffer we configured for the text in the previous step:

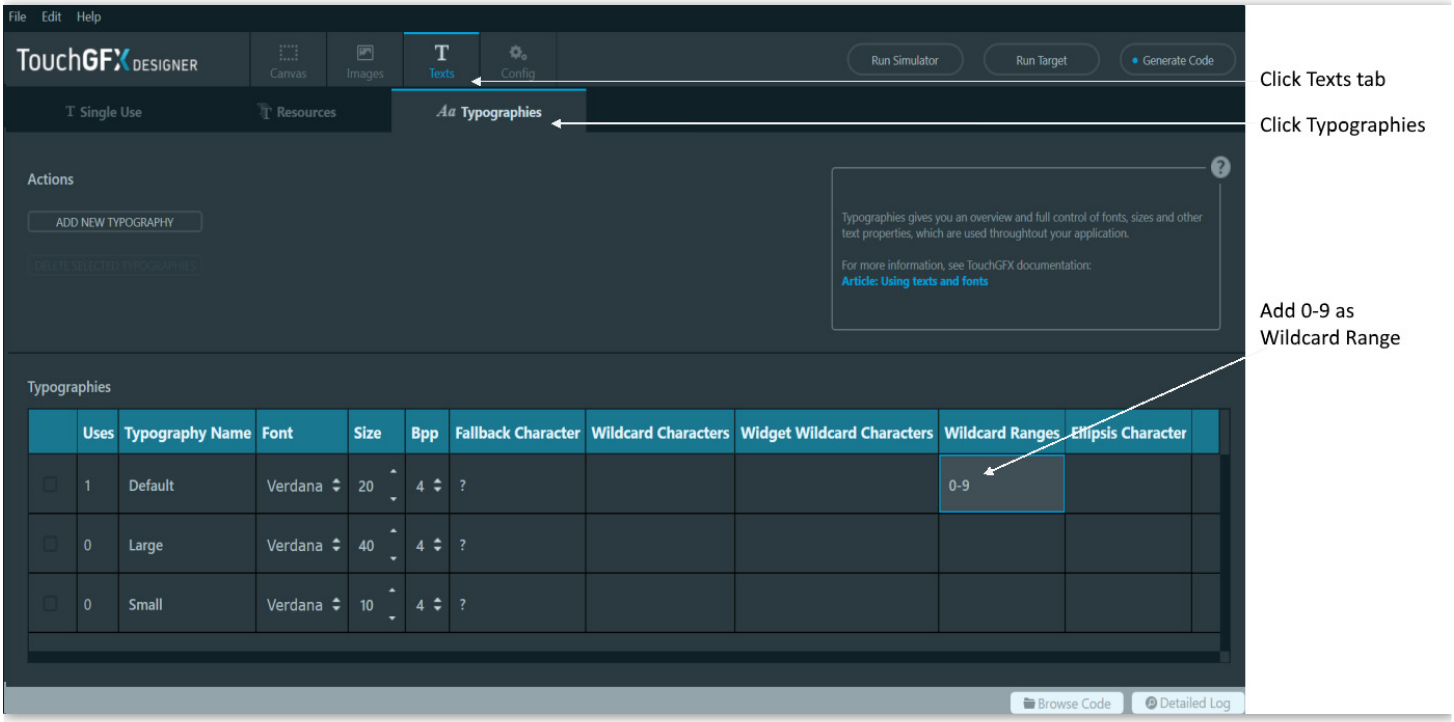
MainView.cpp

```
#include <gui/main_screen/MainView.hpp>

MainView::MainView()
{

}

void MainView::setupScreen()
{
  MainViewBase::setupScreen();
}

void MainView::tearDownScreen()
{
  MainViewBase::tearDownScreen();
}

void MainView::buttonUpClicked()
{
  touchgfx_printf("buttonUpClicked\n")
```

```
    counter++;
    Unicode::snprintf(textCounterBuffer, TEXTCOUNTER_SIZE, "%d", counter);
    // Invalidate text area, which will result in it being redrawn in next tick.
    textCounter.invalidate();
}

void MainView::buttonDownClicked()
{
    touchgfx_printf("buttonDownClicked\n")

    counter--;
    Unicode::snprintf(textCounterBuffer, TEXTCOUNTER_SIZE, "%d", counter);
    // Invalidate text area, which will result in it being redrawn in next tick.
    textCounter.invalidate();
}
```

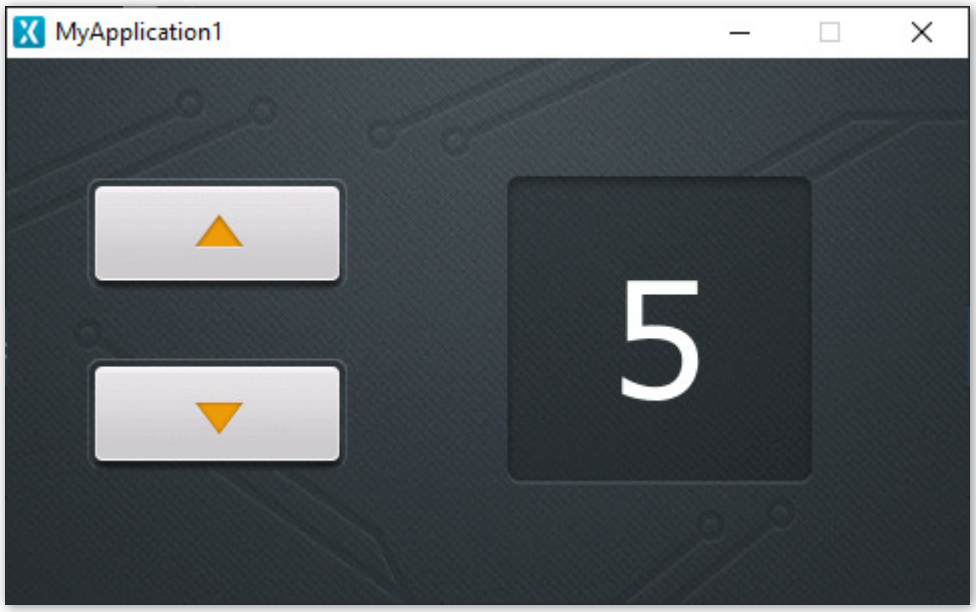
Note that we call `invalidate()` on the `textCounter` widget after updating it. This ensures that the TextArea is redrawn after the counter value has been updated.

We need one more thing before the application is finished. TouchGFX only included the characters needed from the used fonts, so we need to tell TouchGFX Designer to include the characters 0-9 in the "Default" typography. To do that, go back to TouchGFX Designer and click the "Texts" tab, then the "Typographies" tab. In the "Wildcard Ranges" column for the Default typography, add the range "0-9".



Setting the Wildcard Range for the Default typography

Now click "Run Simulator" again and click the up button a few times:



Running the Simulator

As the program is now, it will not handle negative numbers correctly. This can be fixed, either by inserting a guard in the `buttonDownClicked()` function to ensure the counter does not go below 0 or by adding the character "-" to the used typography. This can be accomplished simply by adding a minus ("-") in the Wildcard Characters cell for the Default typography.

This step concludes tutorial 2.

FURTHER READING

- Read more about texts on the **Texts and Fonts** page.

Version: 4.16

Tutorial 3: Applications with Multiple Screens

In this tutorial, you will learn how to create multiple screens in an application and share data between the two screens. We will create an application that simulates a clock, which will use one screen to set hour and minute and pass the time to another screen which has a running clock.

You will also learn how to use the TouchGFX Designer to create animations based on interaction such as screen changes.

The images we will use in this tutorial can be downloaded from this link. Unzip the file to a directory on your disk.

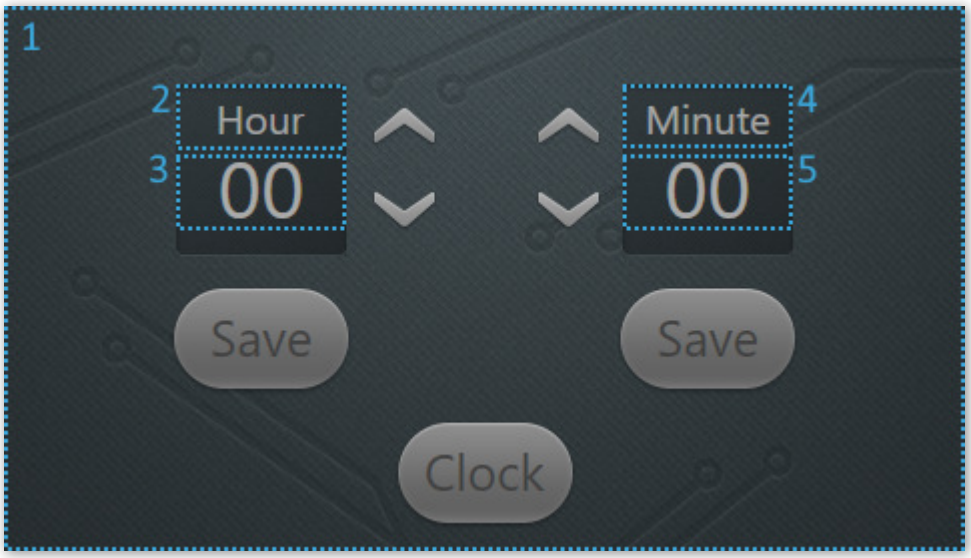
Step 1: Setting up the Two Screens

Create a new project just as was done in tutorial 2. Again the graphics is designed for a resoultion of 480x272. If you base the project on the Application Template for the "STM32F746G Discovery Kit", your project will have this resolution. If you have a demo board with a different resolution you should" be able to modify the graphics to match it yourself. If you have no demo board simply base it on the "Simulator" Application Template and make sure to enter 480x272 as the resolution.

Setting up Screen1

The first screen to create is Screen1, which is the screen where the hour and minute values can be set and sent to the screen with the running clock. Screen1 is shown below and contains two text boxes which contain the desired time for the clock. To adjust the value in the boxes the up and down arrows is used. For saving the selected values and thereby pass them on to the clock the save button below the respective boxes is used. Lastly switching to the Screen2 with the clock is done by pressing the Clock button.

Let us start out by inserting the background and the text areas for our application. Insert the widgets and update the properties as listed in the picture and table below.



Screenshot of screen1 with the location of the background and the text areas highlighted

Location	Widget	Properties
1	Image	• Name: background • Location: ○ X: 0, Y: 0 • Image:background_Screen1.png

2	TextArea	• Name: textAreaHourCaption
		• Location:
		  ○ X: 86, Y: 46
		  ○ W: 85, H: 24
		• Text:
		  ○ Text: Hour
		  ○ Typography: 20px
		  ○ Alignment: Center
		• Appearance:
		  ○ Color: #FFABABAB

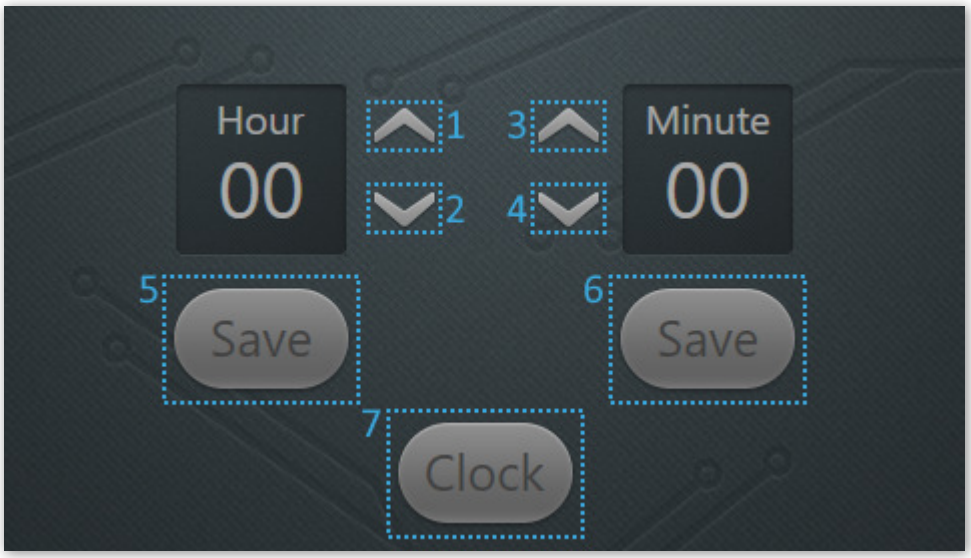
Location	Widget	Properties

3	TextArea	• Name: textAreaHour
		• Location:
		  ◦ X: 87, Y: 70
		  ◦ W: 83, H: 50
		• Text:
		  ◦ Text: <value>
		  ◦ Wildcard 1:
		  ▪ Initial Value: 00
		  ▪ Buffer size: 3
		  ◦ Typography: 40px
		  ◦ Alignment: Center
		• Appearance:
		  ◦ Color: #FFABABAB

4	TextArea	• Name: textAreaMinuteCaption
		• Location:
		  ◦ X: 309, Y: 46
		  ◦ W: 85, H: 24
		• Text:
		  ◦ Text: Minute
		  ◦ Typography: 20px
		  ◦ Alignment: Center
		• Appearance:
		  ◦ Color: #FFABABAB

5	TextArea	• Name: textAreaMinute
		• Location:
		  ◦ X: 311, Y: 70
		  ◦ W: 83, H: 50
		• Text:
		  ◦ Text: <value>
		  ◦ Wildcard 1:
		  ▪ Initial Value: 00
		  ▪ Buffer size: 3
		  ◦ Typography: 40px
		  ◦ Alignment: Center
		• Appearance:
		  ◦ Color: #FFABABAB

Then insert the buttons to the application as shown in the picture and table below.



Screenshot of screen 1 with the location of the buttons highlighted

Location	Widget	Properties

1	Button	• Name: buttonHourUp
		• Location:
		  ◦ X: 184, Y: 51
		• Image:
		  ◦ Released Image: Up_arrow.png
		  ◦ Pressed Image: Up_arrow_pressed.png

2	Button	• Name: buttonHourDown
		• Location:
		  ◦ X: 184, Y: 93
		• Image:
		  ◦ Released Image: Down_arrow.png
		  ◦ Pressed Image: Down_arrow_pressed.png

3	Button	• Name: buttonMinuteUp
		• Location:
		  ◦ X: 266, Y: 51
		• Image:
		  ◦ Released Image: Up_arrow.png
		  ◦ Pressed Image: Up_arrow_pressed.png

Location	Widget	Properties

4	Button	• Name: buttonMinuteDown
		• Location:
		  ○ X: 266, Y: 93
		• Image:
		  ○ Released Image: Down_arrow.png
		  ○ Pressed Image: Down_arrow_pressed.png

5	Button With Label	• Name: buttonSaveHour
		• Location:
		  ○ X: 80, Y: 137
		• Text:
		  ○ Text: Save
		  ○ Typography: 25px
		  ○ Alignment: Center
		• Text Appearance:
		  ○ Released Color: #FF424242
		  ○ Pressed Color: #FFA6A6A6
		• Image:
		  ○ Released Image: btn_round.png
		  ○ Pressed Image: btn_round_pressed.png

6	Button With Label	• Name: buttonSaveMinute
		• Location:
		  ○ X: 303, Y: 137
		• Text:
		  ○ Text: Save
		  ○ Typography: 25px
		  ○ Alignment: Center
		• Text Appearance:
		  ○ Released Color: #FF424242
		  ○ Pressed Color: #FFA6A6A6
		• Image:
		  ○ Released Image: btn_round.png
		  ○ Pressed Image: btn_round_pressed.png

Location	Widget	Properties

7	Button With Label	• Name: buttonClock
		• Location:
		  ◦ X: 192, Y: 204
		• Text:
		  ◦ Text: Clock
		  ◦ Typography: 25px
		  ◦ Alignment: Center
		• Text Appearance:
		  ◦ Released Color: #FF424242
		  ◦ Pressed Color: #FFA6A6A6
		• Image:
		  ◦ Released Image: btn_round.png
		  ◦ Pressed Image: btn_round_pressed.png

With the graphical elements set up next step is to add the triggers for the buttons, which adjust the selected values and saves them:

Interaction	Properties
Hour up button is clicked	• Trigger: Button is clicked • Clicked Source: buttonHourUp • Action: Call new virtual function • Function Name: buttonHourUpClicked
Hour down button is clicked	• Trigger: Button is clicked • Clicked Source: buttonHourDown • Action: Call new virtual function • Function Name: buttonHourDownClicked
Minute up button is clicked	• Trigger: Button is clicked • Clicked Source: buttonMinuteUp • Action: Call new virtual function • Function Name: buttonMinuteUpClicked
Minute down button is clicked	• Trigger: Button is clicked • Clicked Source: buttonMinuteDown • Action: Call new virtual function • Function Name: buttonMinuteDownClicked

Interaction	Properties
Save Hour button is clicked	• Trigger: Button is clicked • Clicked Source: buttonSaveHour • Action: Call new virtual function • Function Name: buttonSaveHourClicked
Save Minute button is clicked	• Trigger: Button is clicked • Clicked Source: buttonSaveMinute • Action: Call new virtual function • Function Name: buttonSaveMinuteClicked

If you press "Generate Code" or "Run Simulator" the specified virtual functions will be generated by the Designer. Let us start by integrating the four functions for the arrow buttons. To keep track of the values for hour and minute, two counters are also added.

Now add the following code:

Screen1View.hpp

```
...
public:
...
    virtual void buttonHourUpClicked();
    virtual void buttonHourDownClicked();
    virtual void buttonMinuteUpClicked();
    virtual void buttonMinuteDownClicked();

protected:
    int16_t hour;
    int16_t minute;
...
```

When pressing the arrows the corresponding value changes to fit within the values for a clock.

The logic for the four functions should be added in the following way:

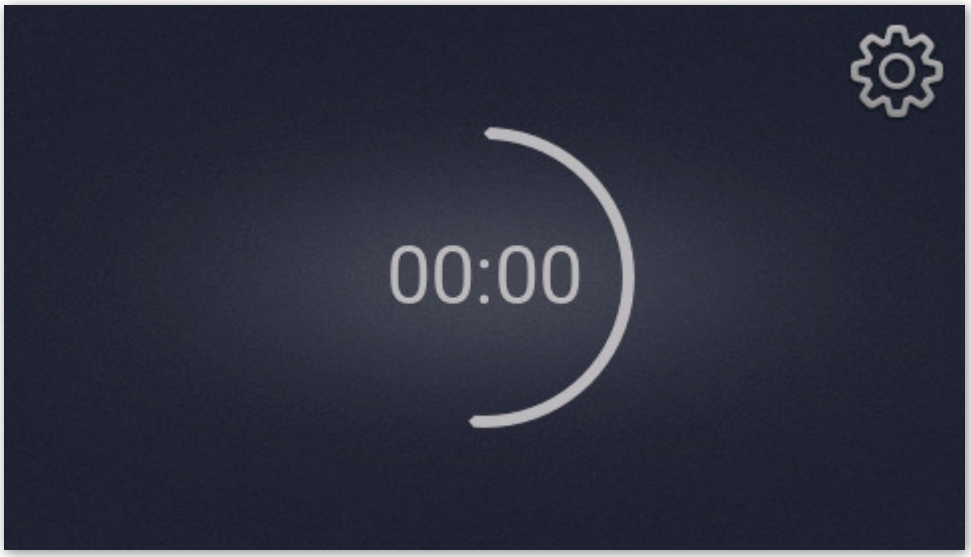
Screen1View.cpp

```
...
void Screen1View::buttonHourUpClicked()
{
    hour = (hour + 1) % 24; // Keep new value in range 0..23
    Unicode::snprintf(textAreaHourBuffer, TEXTAREAHOUR_SIZE, "%02d", hour);
    textAreaHour.invalidate();
```

```
}

void Screen1View::buttonHourDownClicked()
{
    hour = (hour + 24 - 1) % 24; // Keep new value in range 0..23
    Unicode::snprintf(textAreaHourBuffer, TEXTAREAHOUR_SIZE, "%02d", hour);
    textAreaHour.invalidate();
}

void Screen1View::buttonMinuteUpClicked()
{
    minute = (minute + 1) % 60; // Keep new value in range 0..59
    Unicode::snprintf(textAreaMinuteBuffer, TEXTAREAMINUTE_SIZE, "%02d", minute);
    textAreaMinute.invalidate();
}

void Screen1View::buttonMinuteDownClicked()
{
    minute = (minute + 60 - 1) % 60; // Keep new value in range 0..59
    Unicode::snprintf(textAreaMinuteBuffer, TEXTAREAMINUTE_SIZE, "%02d", minute);
    textAreaMinute.invalidate();
}
```

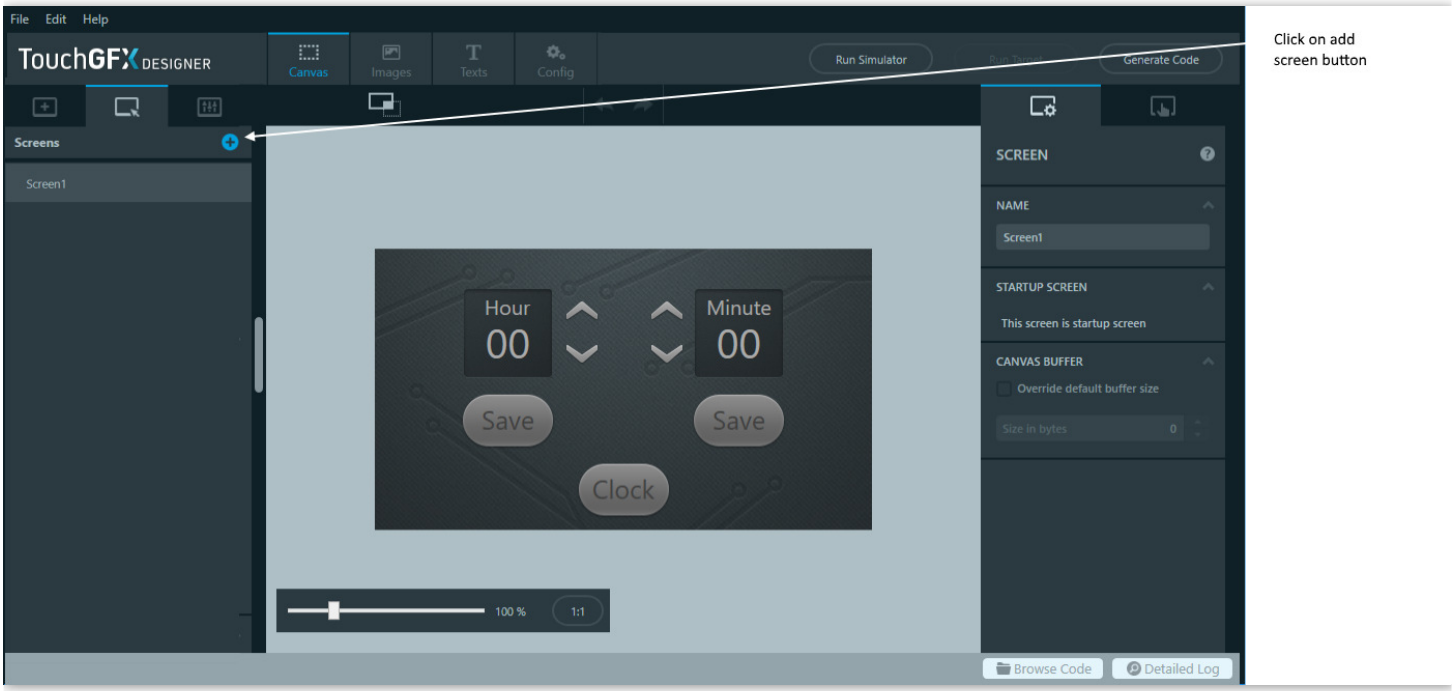
Setting up Screen2

The second screen, Screen2, is where the running clock is placed, starting from the values saved in Screen1. Other than the clock Screen2 also consist off a circle animating around the clock, indicating that the clock is running. Finally to return to Screen1 a button is implemented which changes the screen to Screen1.



Screenshot of screen2 when running the application with the simulator

Before adding elements to Screen2, a new screen has to be created. This is done with the add screen button in the TouchGFX Designer illustrated in the picture below.



The location of the add screen button in the TouchGFX Designer

When entering "screen2", both the clock and the circle animates into their position, by moving into view, with the clock moving in from the left and the circle from the right. The two widgets are therefore initially placed outside the canvas in the TouchGFX Designer.

The placement of the widgets should be done as in the picture and table below.

The loaction of the Wigets in screen2

Location	Widget	Properties
1	Image	• Name: background • Location: ○ X: 0, Y: 0 • Image:background_Screen2.png

Location	Widget	Properties

1	Button	• Name: buttonSettings
		• Location:
		  ○ X: 422, Y: 10
		• Image:
		  ○ Released Image: configuration.png
		  ○ Pressed Image: configuration.png

3	TextArea	• Name: textClock
		• Location:
		  ○ X: -156, Y: 109
		  ○ W: 156, H: 54
		• Text:
		  ○ Text: <hour>:<min>
		  ○ Wildcard 1:
		  ■ Initial Value: 00
		  ■ Buffer size: 3
		  ○ Wildcard 2:
		  ■ Initial Value: 00
		  ■ Buffer size: 3
		  ○ Typography: 40px
		  ○ Alignment: Center
		• Appearance:
		  ○ Color: #FFABABAB

4	Circle	• Name: circle
		• Location:
		  ○ X: 479, Y: 36
		  ○ W: 200, H: 200
		• Image & Color:
		  ○ Color: #FFBABABA
		• Appearance:
		  ○ Center Position:
		  ■ X: 100, Y: 100
		  ○ Start & End Angle:
		  ■ Start: 0, End: 180
		  ○ Radius: 72
		  ○ Line Width: 6
		  ○ Cap Style: Triangle

Switching between Screens

Now we need to add functionality to switch between the two screens. For this we assign interactinos to the Clock button on Screen1 and the Configuration button on Screen2. Also, we want the two elements that are placed outside the screen area on Screen2 to move into place when Screen2 is entered.

To do this add the following interaction to Screen1:

Interaction	Properties
Change to "Screen2"	• Trigger: Button is clicked • Clicked Source: buttonClock • Action: Change screen • Screen: Screen2 • Transition: Cover • Transition Direction: North

Also add the following interaction to Screen2:

Interaction	Properties
Change to "Screen1"	• Trigger: Button is clicked • Clicked Source: buttonSettings • Action: Change screen • Screen: Screen1 • Transition: Slide • Transition Direction: South
Move circle into place	• Trigger: Screen is entered • Action: Move widget • Widget to move: circle • Position: 140, 36 • Easing: Cubic, Out • Duration: 750ms
Move text clock into place	• Trigger: Screen is entered • Action: Move widget • Widget to move: textClock • Position: 162, 109 • Easing: Cubic, Out • Duration: 750ms

To update the clock and animate the circle at runtime the virtual function `handleTickEvent` is used.

`handleTickEvent` is called periodically by the TouchGFX framework enabling it to update elements in the active screen dynamically, which in this case will be the clock and circle.

Similar to Screen1 an hour and a minute counter is used to keep track of the clock. Since `handleTickEvent` is called more frequently than the clock should be updated, a tickCounter is added to determine the number of ticks between the clock updates. For updating the angle of the arc in the circle the functions addStart and addEnd are used. The `handleTickEvent` function and the variables are added as shown to Screen2View.hpp as shown below.

Screen2View.hpp

```
public:
...
    virtual void handleTickEvent();

protected:
    int16_t hour;
    int16_t minute;
    int16_t tickCount;
    int16_t addStart;
    int16_t addEnd;
...
```

The implementation of `handleTickEvent` in `Screen2View.cpp`, thereby, the code which updates the clock and the circle is shown below

Screen2View.cpp

```
...
void Screen2View::handleTickEvent()
{
    if (tickCount == 60)
    {
        minute++;
        hour = (hour + (minute / 60)) % 24;
        minute %= 60;

        Unicode::snprintf(textClockBuffer1, TEXTCLOCKBUFFER1_SIZE, "%02d", hour);
        Unicode::snprintf(textClockBuffer2, TEXTCLOCKBUFFER2_SIZE, "%02d", minute);

        textClock.invalidate();

        tickCount = 0;
    }

    if (!textClock.isMoveAnimationRunning())
```

```
    {
        tickCount++;
        if (circle.getArcStart() + 340 == circle.getArcEnd())
        {
            addStart = 2;
            addEnd = 1;
        }
        else if (circle.getArcStart() + 20 == circle.getArcEnd())
        {
            addStart = 1;
            addEnd = 2;
        }
        circle.invalidate();
        circle.setArc(circle.getArcStart() + addStart, circle.getArcEnd() + addEnd);
        circle.invalidate();
    }
}
...
```

As learned in tutorial 2 we need to add the characters used in the wildcards. In this case you need to add "0-9" in the column *Wildcard Ranges* for the typography used for the TextAreas.

FURTHER READING

In this step the widget Circle has been used. Read more about the Circle widget on the **Circle** page.

Step 2: Saving Data

In this step, we will show how to save data when switching between screens and how to retrieve the saved data.

TouchGFX applications follows the Model-View-Presenter design pattern, so in order to persist data manipulated in a view (i.e. a Screen), the data should be sent to the model (through a presenter). More information on the Model-View-Presenter design pattern can be found on the Model-View-Presenter Design Pattern page.

Adding Hour and Minute to the Model

The model is responsible for holding data for the application. Temporary data, such as button states, currently visible widget etc. should not be in the model.

To save and retrieve data via the model, add protected hour and minute values to the model, as well as public functions to access these values:

Model.hpp

```
...
public:
    void saveHour(int16_t saveHour)
    {
        hour = saveHour;
    }

    void saveMinute(int16_t saveMinute)
    {
        minute = saveMinute;
    }

    int16_t getHour()
    {
        return hour;
    }

    int16_t getMinute()
    {
        return minute;
    }

protected:
    int16_t hour;
    int16_t minute;
...
```

NOTE

The `model.hpp` file should also include `#include <touchgfx/hal/types.hpp>` in order to enable the usage of the type `int16_t`

Make sure to initialize hour and minute in the constructor:

Model.cpp

```
...
Model::Model() : modelListener(0), hour(0), minute(0)
{
}
...
```

With this code, the hour and minute have a place in the model. Since the model is available to all presenters, this the recommended way to share information between presenters (and views). The model is also where the UI is able to connect to the rest of the system, such as hardware peripherals and other software modules.

Accessing the Model from the View

Now, in order to access the data in the model from the view, the presenter should provide functions to allow `Screen1View` to load and save the data from the model as follows:

Screen1Presenter.hpp

```
...
public:
    void saveHour(int16_t hour)
    {
        model->saveHour(hour);
    }

    void saveMinute(int16_t minute)
    {
        model->saveMinute(minute);
    }

    int16_t getHour()
    {
        return model->getHour();
    }

    int16_t getMinute()
    {
        return model->getMinute();
    }
...
```

Since Screen2 should also be able to access the data in the model, add the same lines to `Screen2Presenter.hpp`.

Data from the Model

The code to access hour and minute in the Model is now in place, and `Screen1` and `Screen2` should be updated to get these values from the Model instead of using local variables only.

Updating Screen1

Now we can initialize the hour and minute in Screen1View with the values from the model and initialize the buffers for the Text Areas:

Screen1View.cpp

```
...
void Screen1View::setupScreen()
```

```
{
    Screen1ViewBase::setupScreen();

    hour = presenter->getHour();
    minute = presenter->getMinute();

    Unicode::snprintf(textAreaHourBuffer, TEXTAREAHOUR_SIZE, "%02d", hour);
    Unicode::snprintf(textAreaMinuteBuffer, TEXTAREAMINUTE_SIZE, "%02d", minute);
}
...
```

For saving the hour and minute values, the virtual functions, that were created under interactions, for the two save buttons, is implemented in `Screen1View.hpp` and stores the values in the model (via the presenter):

Screen1View.hpp

```
...
public:
    virtual void buttonSaveHourClicked()
    {
        presenter->saveHour(hour);
    }

    virtual void buttonSaveMinuteClicked()
    {
        presenter->saveMinute(minute);
    }
...
```

Screen1 now gets the initial values for hour and minute from the model.

Updating Screen2

Screen2 also needs to synchronize its values with the model.

Similar to Screen1, the initial value shown in the text clock must match the data from the Model.

Screen2View.cpp

```
...
void Screen2View::setupScreen()
{
    Screen2ViewBase::setupScreen();

    hour = presenter->getHour();
    minute = presenter->getMinute();
```

```
    Unicode::snprintf(textClockBuffer1, TEXTCLOCKBUFFER1_SIZE, "%02d", hour);
    Unicode::snprintf(textClockBuffer2, TEXTCLOCKBUFFER2_SIZE, "%02d", minute);
}
...
```

This fetches the hour and minute from the Model. The updated values must be sent back to the Model when we leave the screen (to go to the configuration screen on `Screen1`):

Screen2View.cpp

```
...
void Screen2View::tearDownScreen()
{
    presenter->saveHour(hour);
    presenter->saveMinute(minute);

    Screen2ViewBase::tearDownScreen();
}
...
```

This will send the updated values of hour and minute to the model just before going to the configuration screen.

This concludes the small application and thereby tutorial 3.

Version: 4.16

Tutorial 4: Creating a Scroll Wheel with Custom Behavior

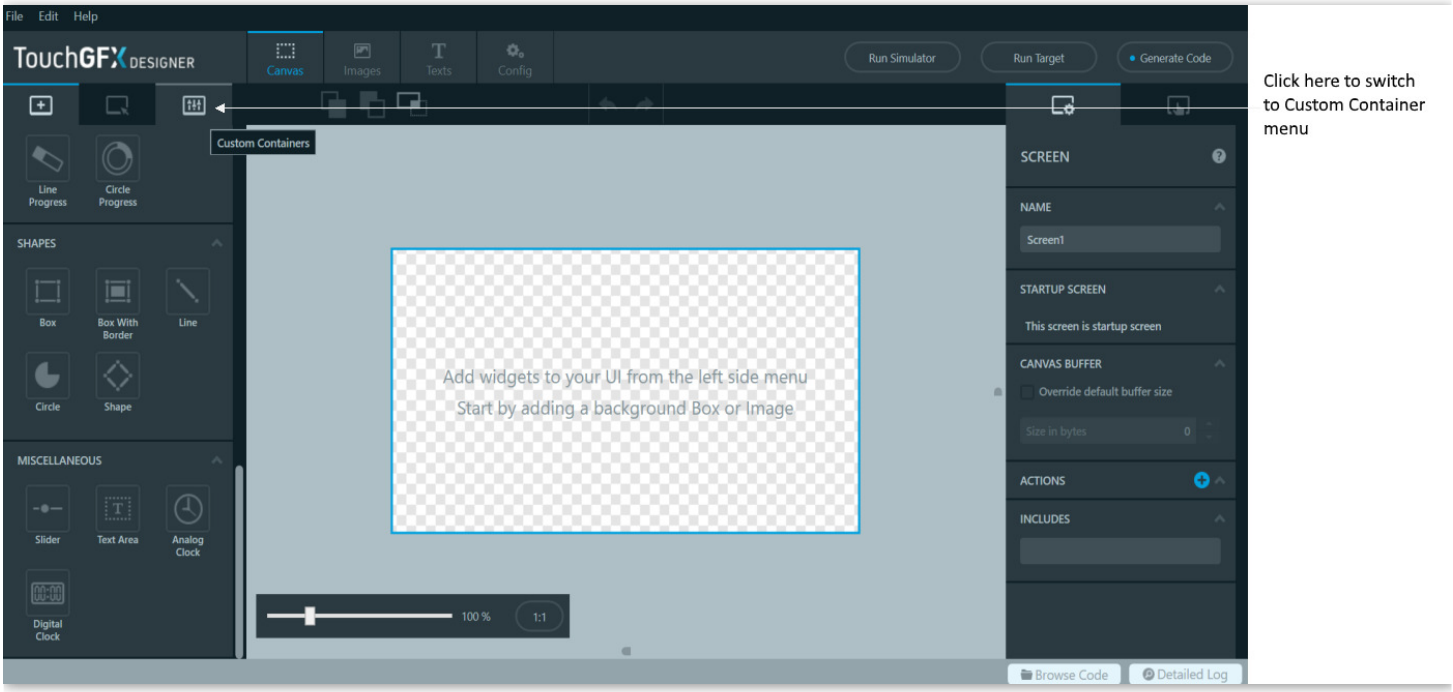
In this tutorial, you will learn how to create and configure the two widgets - Custom Container and Scroll Wheel. A Custom Container is a widget that enables you to create a custom widget by combining multiple other widgets and add specific behavior for the widgets in the Custom Container. The Scroll Wheel is a widget used for creating a scrollable menu, consisting of multiple selectable items. The tutorial will also teach how user code can be created to change the behavior of a widget.

More information about the Custom Container and Scroll Wheel can be found on the two pages, Custom Containers and ScrollWheel.

The graphics for the tutorial can be downloaded from this link. Unzip the file in the images folder under assets, which for the project used in this tutorial is MyApplication2\assets\images.

Step 1: Creating a Custom Container

Similar to step 1 in tutorial 2, start by creating a new project with the TouchGFX Designer. When the new project is ready, change from the screens tab in the TouchGFX Designer to the Custom Container.

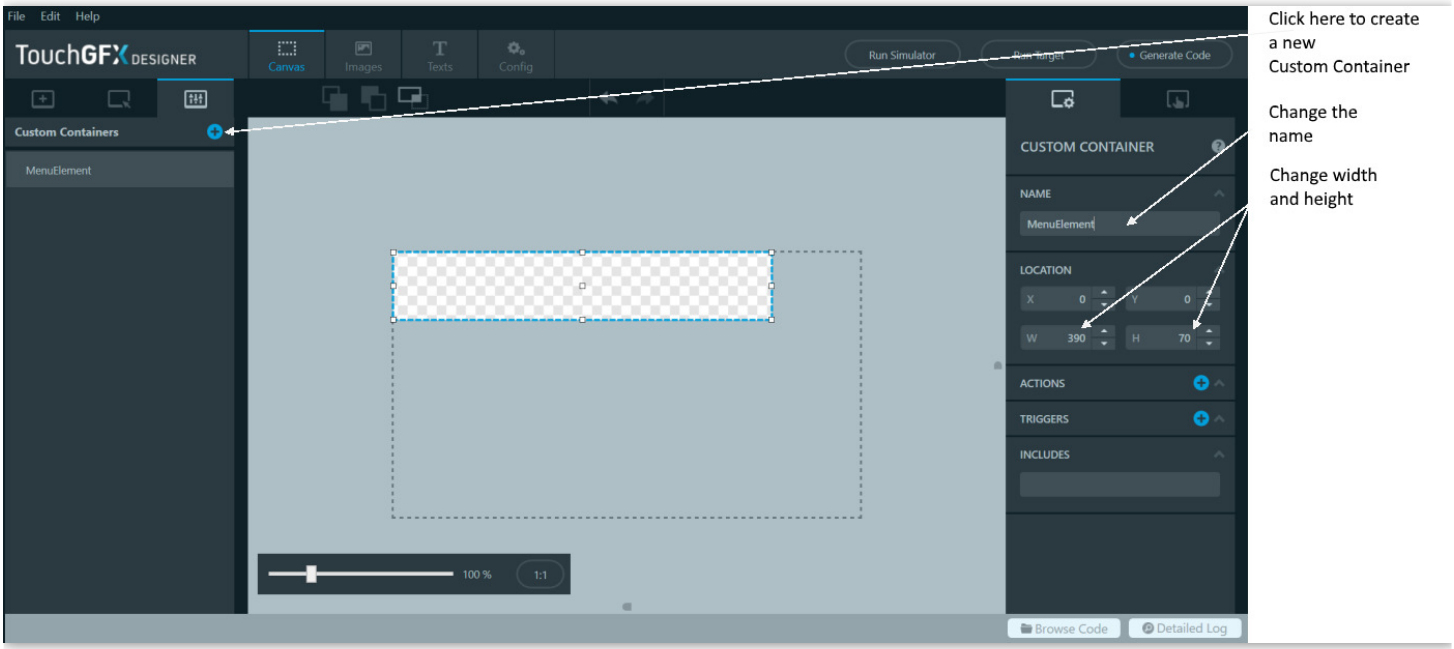


Selecting the Custom Container menu

The tab for creating a Custom Container is similar to the Screens tab and creating a new Custom Container is done in the same way as creating a new screen. After a Custom Container is created

widgets, except Custom Container, can be added, along with edit the size and name of the Custom Container.

In the Custom Container tab, use the blue plus icon to create a new Custom Container and rename it "MenuElement", change the width (W) to 390 and height (H) to 70.



Creating a Custom Container and setting it properties

Adding Widgets to the Custom Container

With Custom Container created and its properties set, widgets can be added to the Custom Container. The Custom Container is going to consist of an image and a text area with a wildcard.

> **FURTHER READING**
>
> To read more about how to use text with wildcards, read the wildcards section on the **Texts and Fonts page**.

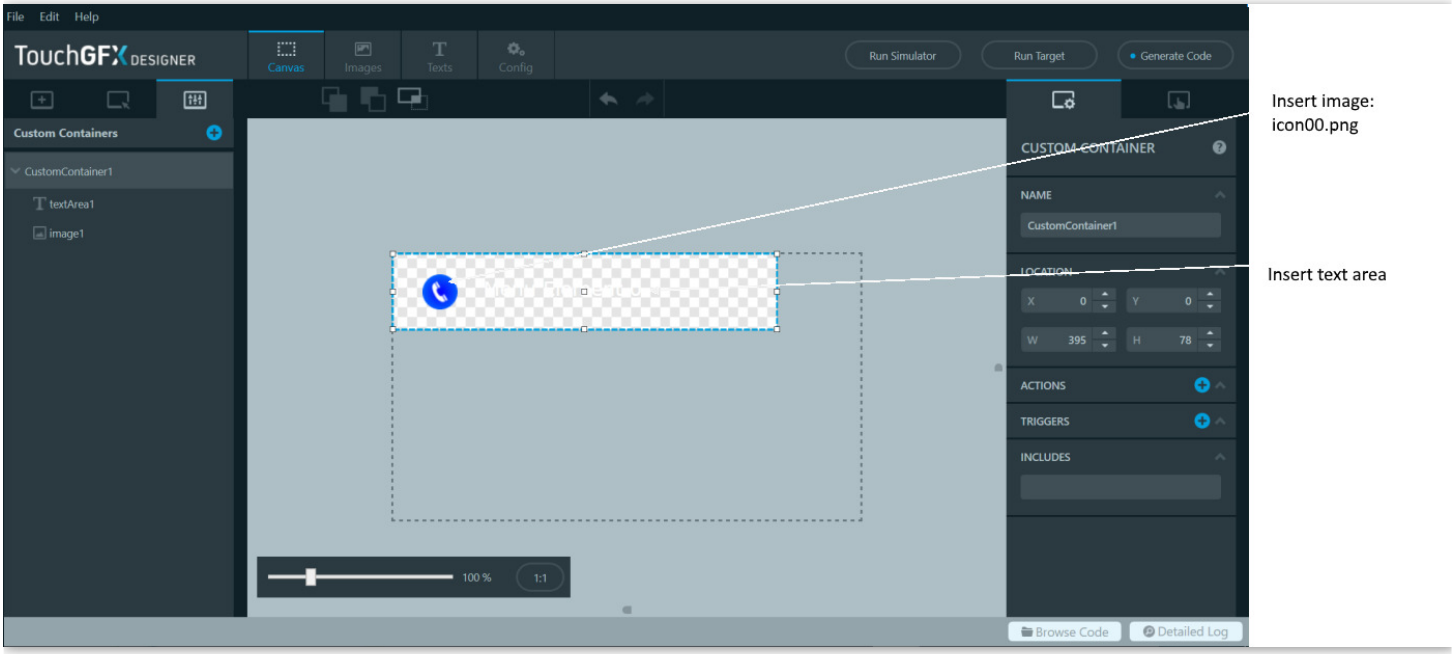
The two widgets are inserted in the following way:

Widget	Properties

Image	• Name: icon
	• Image: icon00.png
	• Location:
	  ○ X: 34
	  ○ Y: 17

Widget	Properties

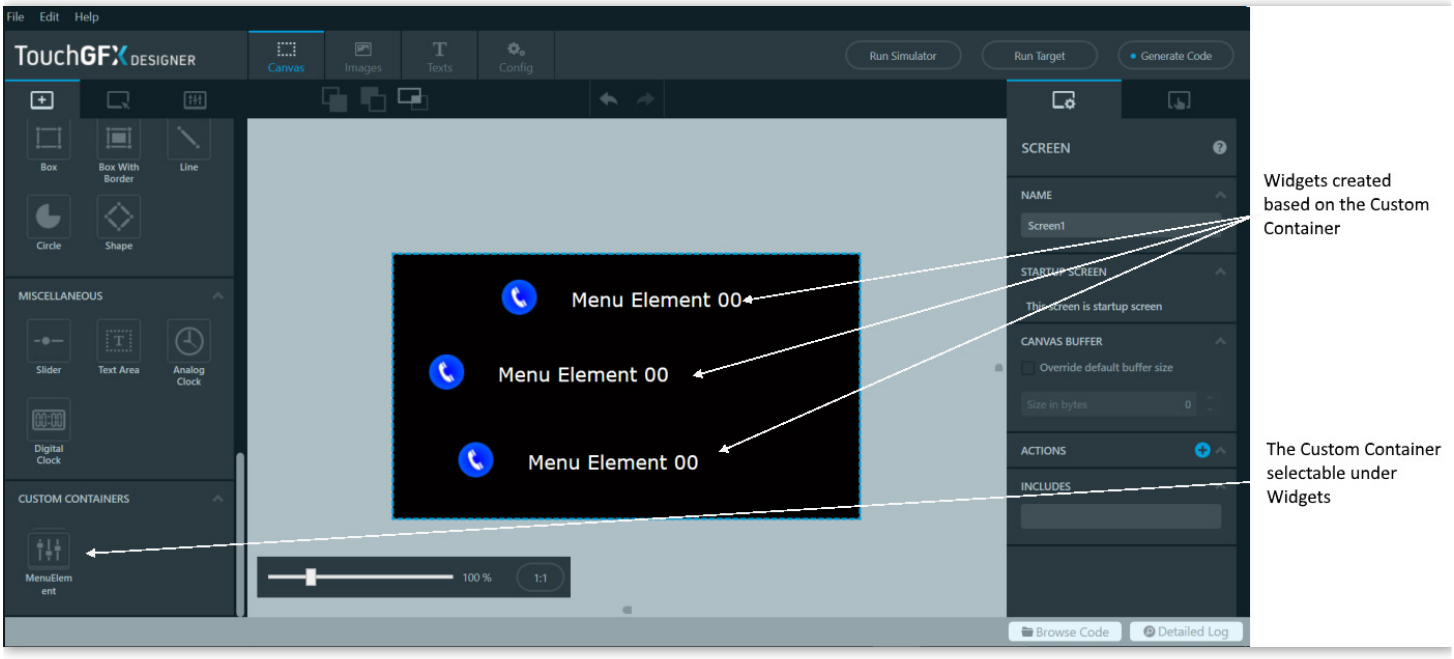
TextArea	• Name: text
	• Location:
	  ◦ X: 93, Y: 23
	• Text:
	  ◦ Text: Menu Element <value>
	  ◦ Wildcard 1:
	  ▪ Initial Value: 00
	  ▪ Buffer size: 3
	  ◦ Typography: 20px
	  ◦ Alignment: Left
	• Appearance:
	  ◦ Color: #FFFFFFFF



Adding content to the Custom Container

Adding the Custom Container to a Screen

Going back to the Screens tab, it is now possible to select the "MenuElement" in the widget menu under Custom Container. Place a black box as background and add a couple of the created Custom Container on the canvas.



Inserting the Custom Container as a widget on a screen

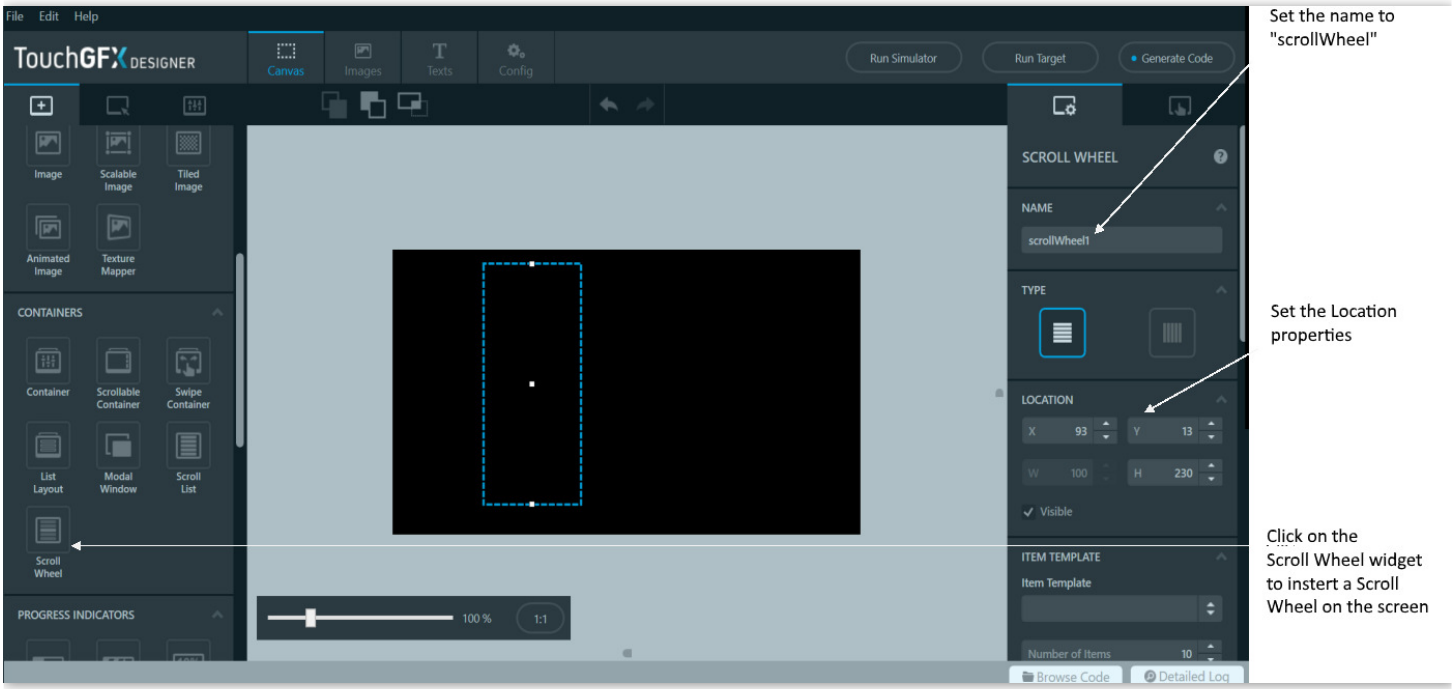
Step 2: Creating a Scroll Wheel

In this step of tutorial 4, we will create a Scroll Wheel by using the Custom Container, "MenuElement", created in step 1. As described in step 1, the Scroll Wheel is used to create a scrollable menu containing multiple selectable items. The items in the wheel are dynamically updated when scrolling and when selecting an item, it is moved into focus.

Adding items to the Scroll Wheel is done by selecting a Custom Container to use as the "Item Template". The concept of "Item Template" works by using the widgets in the Custom Container as the foundation for the items in the Scroll Wheel and use user code to update the widgets in the items at runtime.

Creating the Scroll Wheel

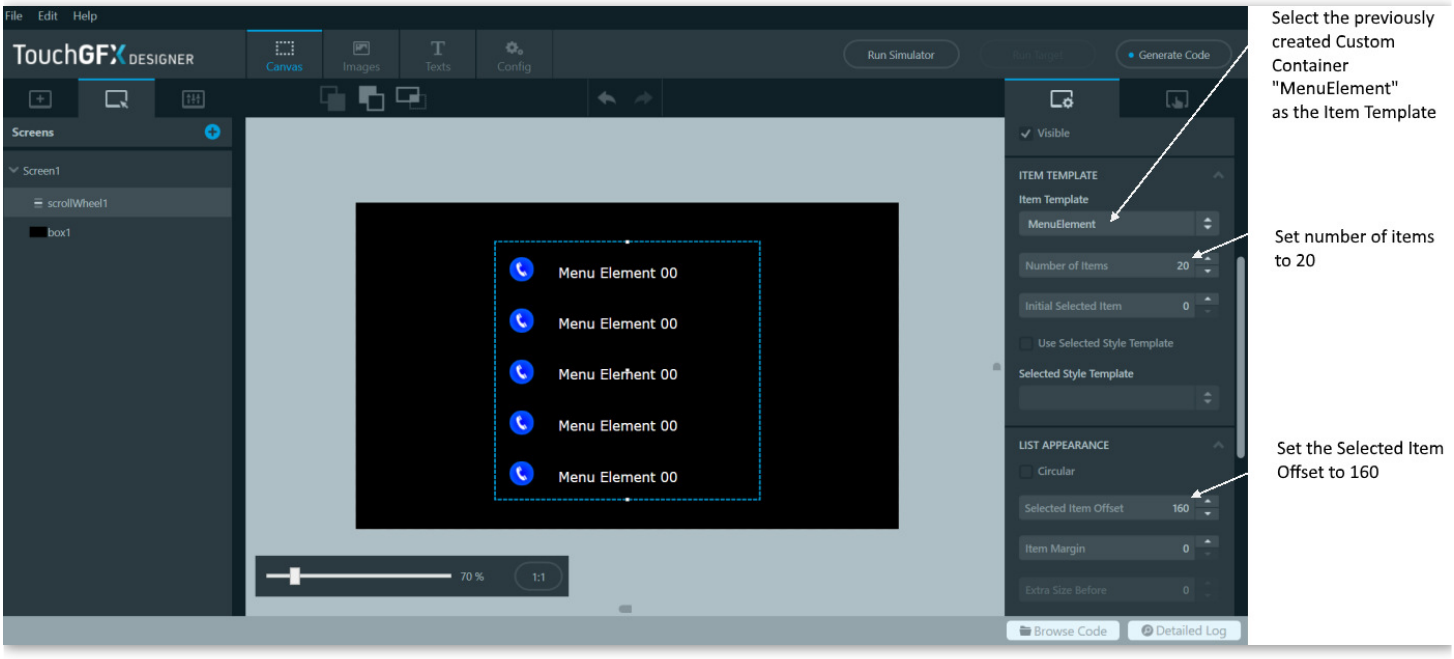
Before creating the Scroll Wheel, remove the Custom Containers on the screen, leaving only the black box as the background. Select the Scroll Wheel located in the widgets tab under section Containers. Create a Scroll Wheel and set the location properties to X = 208, Y = 45 and H = 390 and change the name to "scrollWheel".



Inserting the Scroll Wheel and setting the name and location properties

Adding Items to the Scroll Wheel

Select "MenuElement" created in step 1 as the "Item Template", which is done with the drop-down list under the Scroll Wheel property "Item Template". The number of items in the Scroll Wheel is also set under "Item Template". Set this to 20 items. Since the Scroll Wheel works by having a selected item in focus, setting where the selected item is positioned, is done by setting "Selected Item Offset" under the property "List Apperance". We want the selected item to be in the middle of the Scroll Wheel and are therefore setting "Selected Item Offset" to 160.

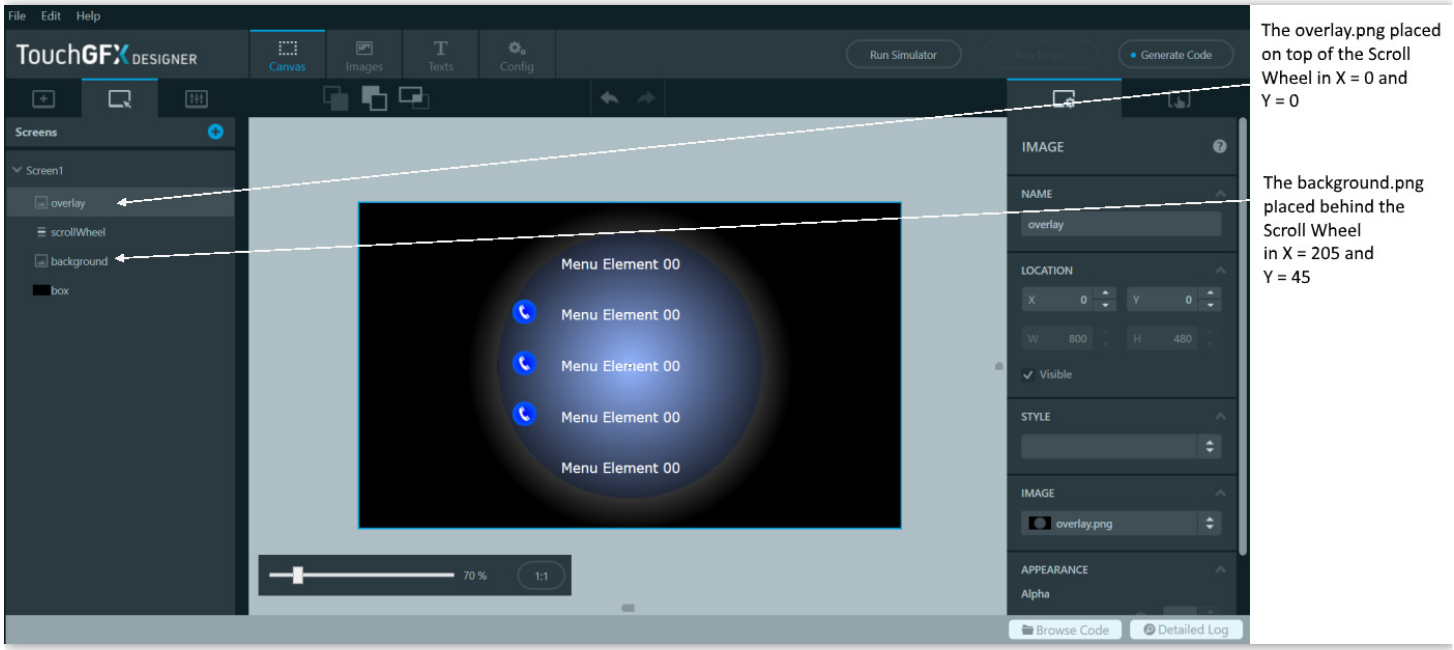


Adding the Custom Container to the Scroll Wheel and adjusting its properties

To highlight the area of the "scrollWheel", the two pictures background.png, and overlay.png from the .zip file downloaded in step 1 are used and is added to the application as Image widgets. The two

images is a background that highlights the area of "scrollWheel" and an overlay which hides the items in "scrollWheel" when the are moved to the edge of "scrollWheel".

The image background.png are placed in X = 205 and Y = 45 and is placed behind "scrollWheel" so the items in "scrollWheel" is drawn op top of the background. The overlay.png is placed in X = 0 and Y = 0 on top of "scrollWheel" meaning that the items are drawn beneath overlay.png thereby hiding the items where overlay.png is not transparent.



Graphics added to the screen with the Scroll Wheel

Since we only have adjusted the static properties for "scrollWheel", logic has not been added to it. Running the application will, therefore, result in a scrollable menu consisting of 20 items that all show the same. In the next step, we will add the logic to the "scrollWheel" with user code which updates the items in the wheel at runtime.

Step 3: Adding User Code to Scroll Wheel

With the Scroll Wheel, "scrollWheel", created and configured in the TouchGFX Designer, this step will create the logic, via user code, that updates the items in "scrollWheel", so they display different graphics based on the position of the item in the wheel. This step will, therefore, work with integrating designer generated code with user code. A more detailed description of integrating designer code with user code can be found on the Code Structure page.

Change Image and Text in MenuElement

Since the items in the Scroll Wheel are based on the Custom Container "MenuElement", created in step 1, user code for changing the icon and updating the wildcard needs to be added to "MenuElement". When a Custom Container is created in the TouchGFX Designer it generates a .hpp

and .cpp file with the same name as the Custom Container which is where the user code should be integrated. The location of the files generated for "MenuElement" in the example application are:

`MyApplication2\gui\include\gui\containers\MenuElement.hpp`

`MyApplication2\gui\src\containers\MenuElement.cpp`

Enabling the items in "scrollWheel" to change their text and icon is done by adding the function `setNumber(int no)` to "MenuElement". The function uses the variable `no` to decide which icon the Image widget should show and change the Wildcard in the Text Area widget to the value of `no`.

The declaration and implementation of `setNumber(int no)` is done in the `MenuElement.hpp` which is shown below.

MenuElement.hpp

```
#ifndef MENUELEMENT_HPP
#define MENUELEMENT_HPP

#include <gui_generated/containers/MenuElementBase.hpp>
#include <BitmapDatabase.hpp>

class MenuElement : public MenuElementBase
{
public:
    MenuElement();
    virtual ~MenuElement() {}

    virtual void initialize();

    void setNumber(int no)
    {
        Unicode::itoa(no, textBuffer, TEXT_SIZE, 10);
        switch (no % 7)
        {
        case 0:
        icon.setBitmap(Bitmap(BITMAP_ICON00_ID));
            break;
        case 1:
        icon.setBitmap(Bitmap(BITMAP_ICON01_ID));
            break;
        case 2:
        icon.setBitmap(Bitmap(BITMAP_ICON02_ID));
            break;
        case 3:
        icon.setBitmap(Bitmap(BITMAP_ICON03_ID));
            break;
        case 4:
        icon.setBitmap(Bitmap(BITMAP_ICON04_ID));
            break;
```

```
        case 5:
        icon.setBitmap(Bitmap(BITMAP_ICON05_ID));
            break;
        case 6:
        icon.setBitmap(Bitmap(BITMAP_ICON06_ID));
            break;
        }
    }
protected:
};

#endif // MENUELEMENT_HPP
```

With the code added to update the content of the MenuElement, the next thing to do is to add code which updates the items in the Scroll Wheel.

Update the Items in the Scroll Wheel

When creating a Scroll Wheel, the TouchGFX Designer generates a virtual function which is called each time a new item in the Scroll Wheel becomes visible. Overriding this function in the user code enables the code to interact with the items in the Scroll Wheel.

The name of the function is the name of the Scroll Wheel appended with UpdatedItem. For the tutorial, the function is called `scrollWheelUpdateItem(MenuElement& item, int16_t itemIndex)`.

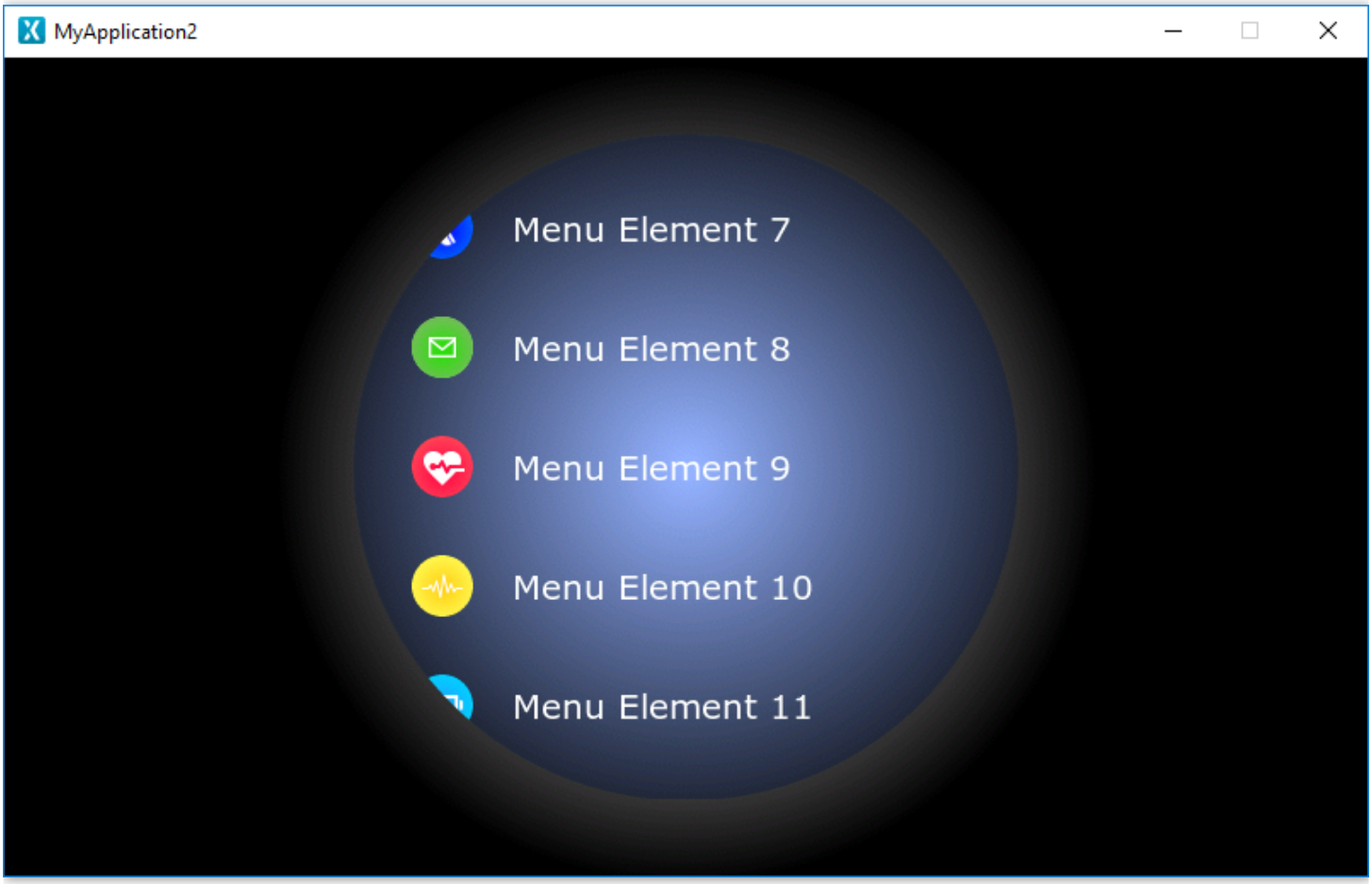
The parameter `itemIndex` is an index value informing which item is currently being updated and `item` is a reference to an instance of MenuElement which is currently visible in the Scroll Wheel. With `itemIndex` informing which item is being updated, `setNumber()` is called for item which will change the content of the item being updated based on the value of `itemIndex`. The code used for updating the Scroll Wheel items is shown below.

Screen1View.hpp

```
#ifndef SCREEN1VIEW_HPP
#define SCREEN1VIEW_HPP

#include <gui_generated/screen1_screen/Screen1ViewBase.hpp>
#include <gui/screen1_screen/Screen1Presenter.hpp>

class Screen1View : public Screen1ViewBase
{
public:
    Screen1View();
    virtual ~Screen1View() {}
    virtual void setupScreen();
    virtual void tearDownScreen();
```

```
    virtual void scrollWheelUpdateItem(MenuElement& item, int16_t itemIndex)
    {
        item.setNumber(itemIndex);
    }
protected:
};

#endif // SCREEN1VIEW_HPP
```

Running the simulator for the application now shows that the text for the items contains the value of their index and the icons change based on which item is showing. The images below shows an example of the simulator running with the code implemented.



Running the simulator

Step 4: Adding Custom Behavior to Scroll Wheel

In the last step of tutorial 4, we will add custom behavior for the Scroll Wheel, by making it move in a circular pattern when scrolling through the items, thereby moving in a pattern similar to a dial.

Add Custom Behavior to MenuElement

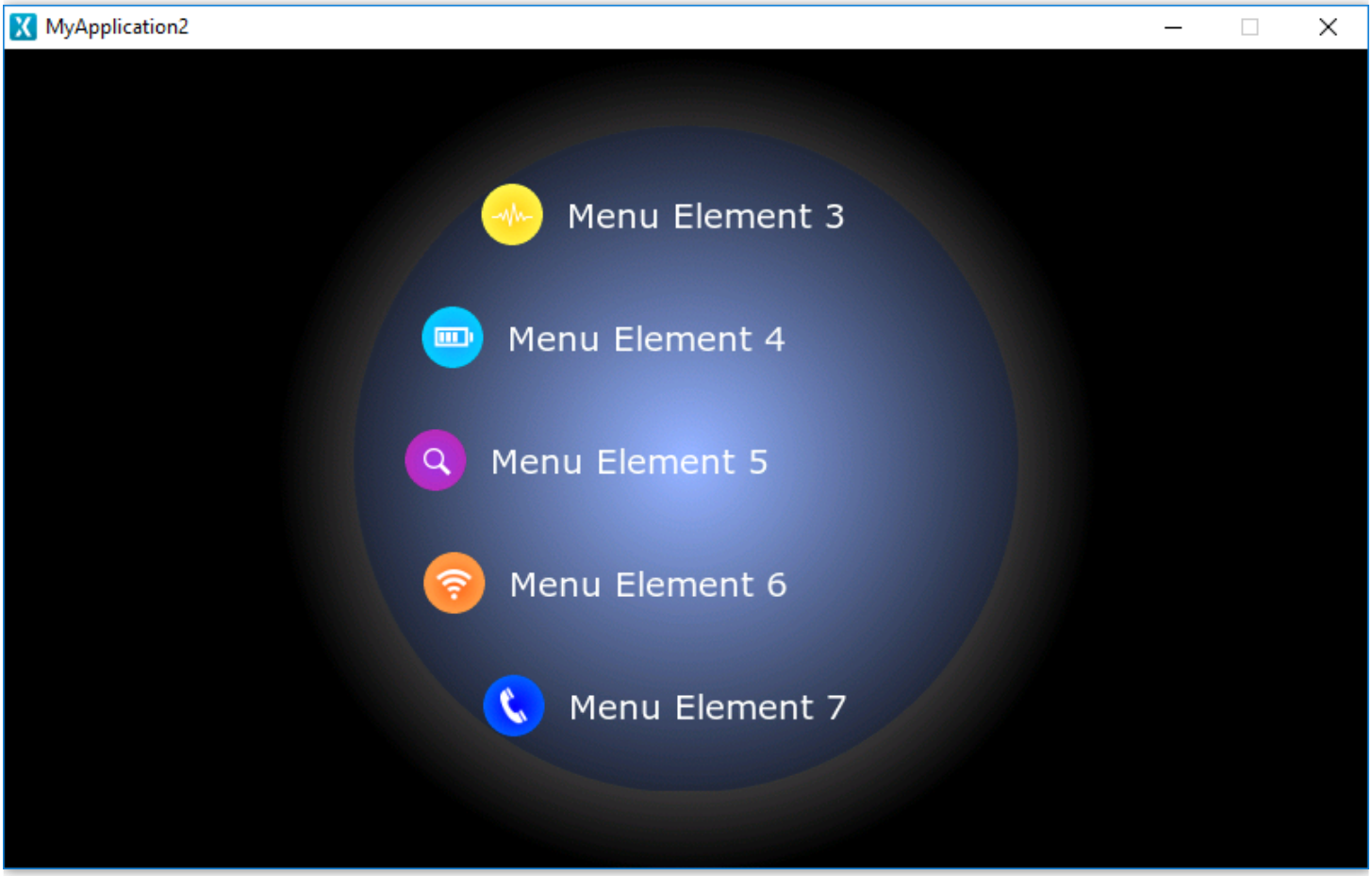
Getting the Scroll Wheel to move in a dial pattern, is done by shifting the horizontal position of the Image and Text widget for each item that is visible in the Scroll Wheel. To do this the function `setY()` for "MenuElement" is overridden in `MenuElement.hpp`. The `setY()` function is called for a Custom Container each time it is moved in the vertical direction and is used for redrawing the Custom Container in its new position. By overriding `setY()`, we are able to rearrange the Image and Text widget each time the Scroll Wheel is moved. The image below describes how to implement the new `setY()` function and shift the the two widgets in `MenuElement.hpp`. Note that `math.h` needs to be included.

MenuElement.hpp

```
#ifndef MENUELEMENT_HPP
#define MENUELEMENT_HPP

#include <gui_generated/containers/MenuElementBase.hpp>
#include <BitmapDatabase.hpp>
#include <math.h>

class MenuElement : public MenuElementBase
{
public:
    MenuElement();
    virtual ~MenuElement() {}

    virtual void initialize();

    //Adjusts the position of the text and the icon, based in the calculated offset(x)
    void offset(int16_t x)
    {
        icon.moveTo(30 + x, icon.getY());
        text.moveTo(80 + x, text.getY());
    }

    //The new declaration and implementation of the setY() function
    virtual void setY(int16_t y)
    {
        //set Y as normal
        MenuElementBase::setY(y);

        const int circleRadius = 250;

        //center around middle of background
        y = y + getHeight() / 2 - 390 /2;

        //calculate x
        float x_f = circleRadius - sqrtf((float)((circleRadius * circleRadius) - (y * y)))
        int16_t x = (int16_t)(x_f + 0.5f);

        offset(x);
    }
```

```
...
```

With the new `setY()` function implemented, running the simulator shows that the Scroll Wheel is now moving in a dial pattern aligning with the curve from the overlay.



Running the simulator

This concludes tutorial 4.

FURTHER READING

To learn more about the concepts that have been used throughout the tutorial the chapters below discuss some concepts that you have worked with:

- **The ScrollWheel page** describes how to create and configure the Scroll Wheel in TouchGFX Designer and how to create the logic in user code.
- **The Custom Containers page** discusses the Custom Container concept and usage.

Version: 4.16

Tutorial 5: Creating Custom Triggers and Actions

With TouchGFX Designer it is possible to define your own interaction components with custom triggers and actions. Each Screen in your application can contain a collection of actions *(these are simply void methods in C++)* that you can call from within TouchGFX Designer as well as in code, while custom containers can also have a collection of triggers *(which is equal to a callback in C++)* which your application can react to.

In this tutorial, we will go through this functionality to learn the possibilities this gives us to create more clean and dynamic TouchGFX applications.

Adding a Custom Action to a Screen

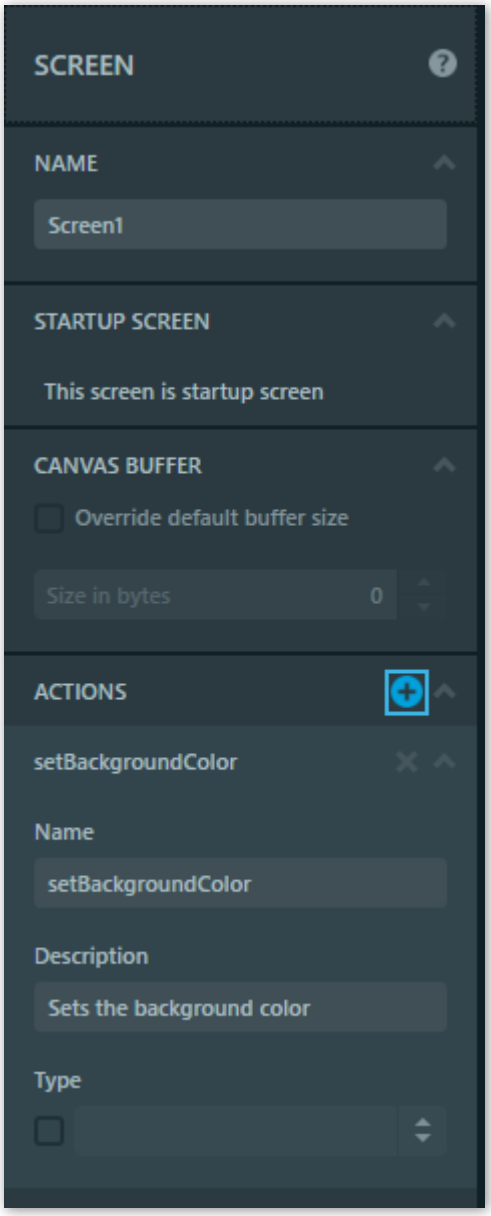
In this section we will:

- Create a new application with a background box and a button
- Add a custom action to the application
- Change the background color using the custom action when the button is pressed

Let's start out by creating a new blank application with dimensions 480x272 and inserting a Box for the background (let's name this "background") and a Button (name this one "button"). You should have something similar to the image below:

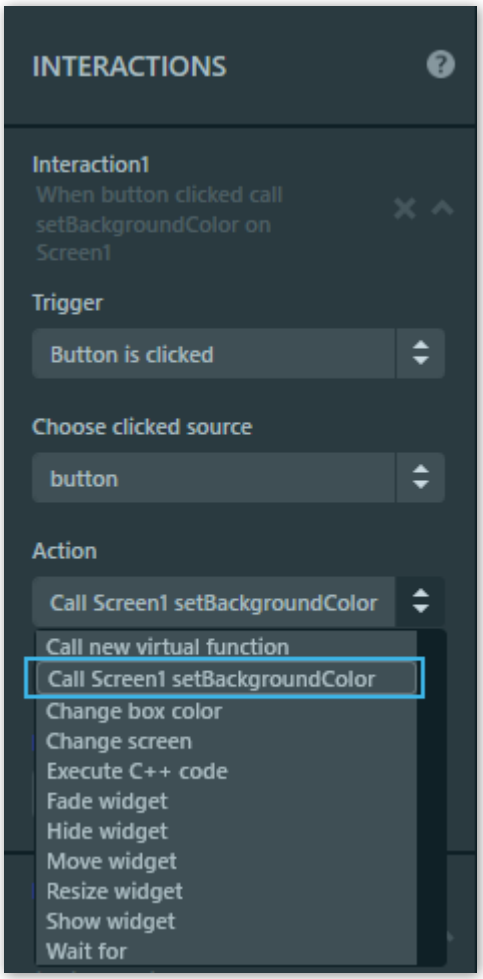
Adding a white background Box and a Button

Next, let's add a custom action to our Screen. You can do this from the properties tab of the Screen by selecting the Screen and pressing the + button in the "ACTIONS" group. Name the action "setBackgroundColor" and give it a description like "Sets the background color". This generates a virtual method in `Screen1ViewBase.hpp` called `setBackgroundColor()` with an empty implementation in `Screen1ViewBase.cpp`.



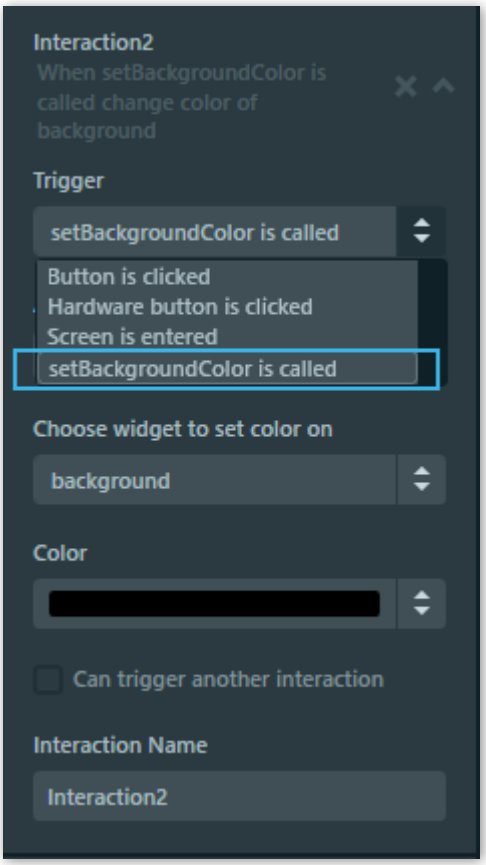
Adding a new custom action to a Screen

You can add functionality to this method by overriding it in user code in the `Screen1View.cpp` file or by creating interactions through TouchGFX Designer. Let's try out the latter by going to the interactions tab for the screen and adding an interaction that calls our new method when our button is clicked.



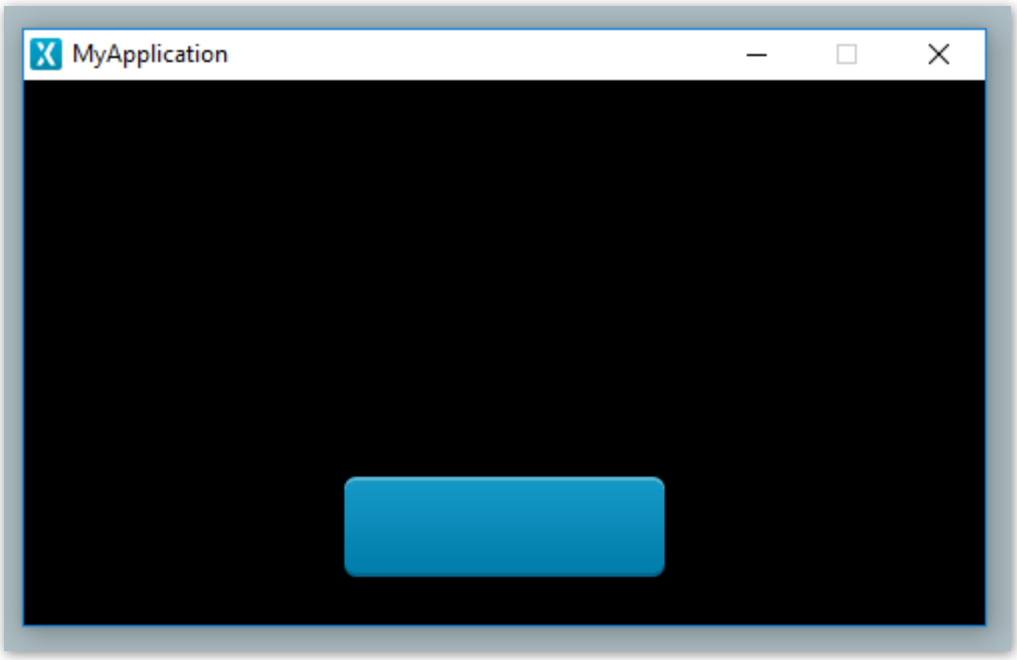
Call setBackgroundColor when button is clicked

Now we specify what actually happens when `setBackgroundColor` is called. This is done by using our new custom action as a trigger in another interaction. Let's start out by simply setting the background Box color to black by using the action "*Change box color*" when the trigger "*setBackgroundColor is called*" happens.



Implementing functionality for custom action setBackgroundColor

Now run the simulator and press the button; the background should turn black. You have successfully created your first custom action.



Pressing button turns background black

Passing a Value to a Custom Action

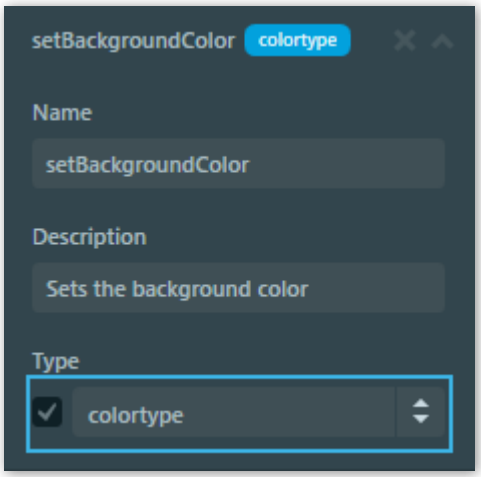
Building upon the application we have just created, this section will expand upon the custom action concept by:

- Adding a parameter to the setBackgroundColor custom action
- Passing a random color to setBackgroundColor
- Using this to change the background to random colors when pressing the button

Let's make this application a little more interesting by passing a value to our `setBackgroundColor` custom action to make it more dynamic.

Goto the interactions tab fot the Screen and delete the two current interactions by pressing the x button for each of them, as we will be setting up new ones.

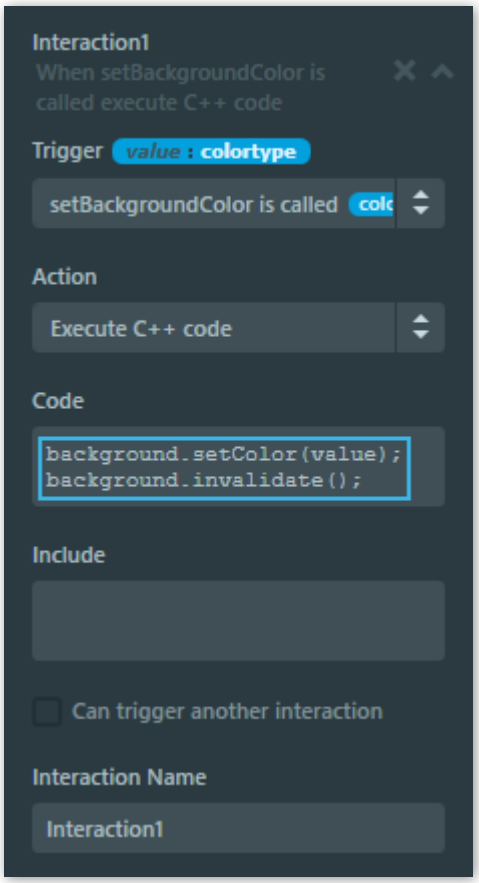
Go to the properties tab for the Screen to the custom action called `setBackgroundColor` and check off the checkbox for type and input "colortype" which will be the type of the parameter we are going to pass to the action (colortype is the built-in TouchGFX type for describing colors). It is not possible to name the parameter and it will be named "*value*".



Setting up a parameter for a custom action

Next let's setup an interaction which uses our newly added parameter value. We do this by using the trigger "setBackgroundColor is called" and the action "Execute C++ code". We want to use our new parameter to set the color of our background Box, so the code to execute should be:

```
background.setColor(value);
background.invalidate();
```

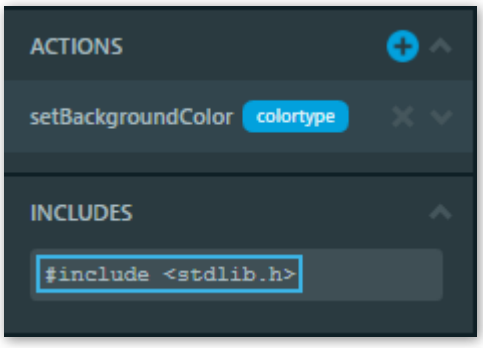


Using the passed value to change color of background

Notice that the trigger displays the name and type of the parameter "value : colortype".

Next, let's set up the interaction that actually calls setBackgroundColor when our button is clicked. Add another interaction with trigger "Button is clicked" and action "Call Screen1 setBackgroundColor" and notice that the value property also displays which type it expects. Let's pass a random color to setBackgroundColor by utilizing the randomization method `rand()` in `stdlib.h` to get three random numbers between 0 and 255 and using those to specify the color. To gain access to `rand()` we need to include it into our application. Luckily for us, it is also possible to supply your own includes from within TouchGFX Designer for both screens and custom containers. Go to the properties tab for the screen and under the "INCLUDES" group, input:

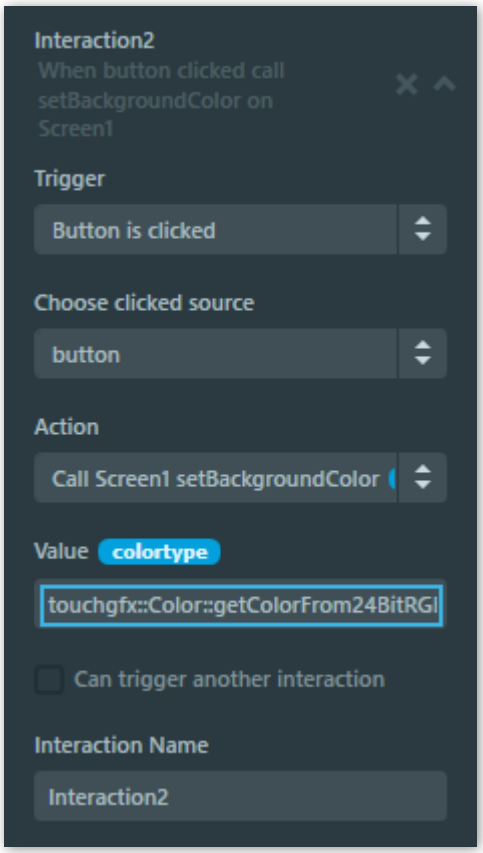
```
#include <stdlib.h>
```



Including stdlib to gain access to rand()

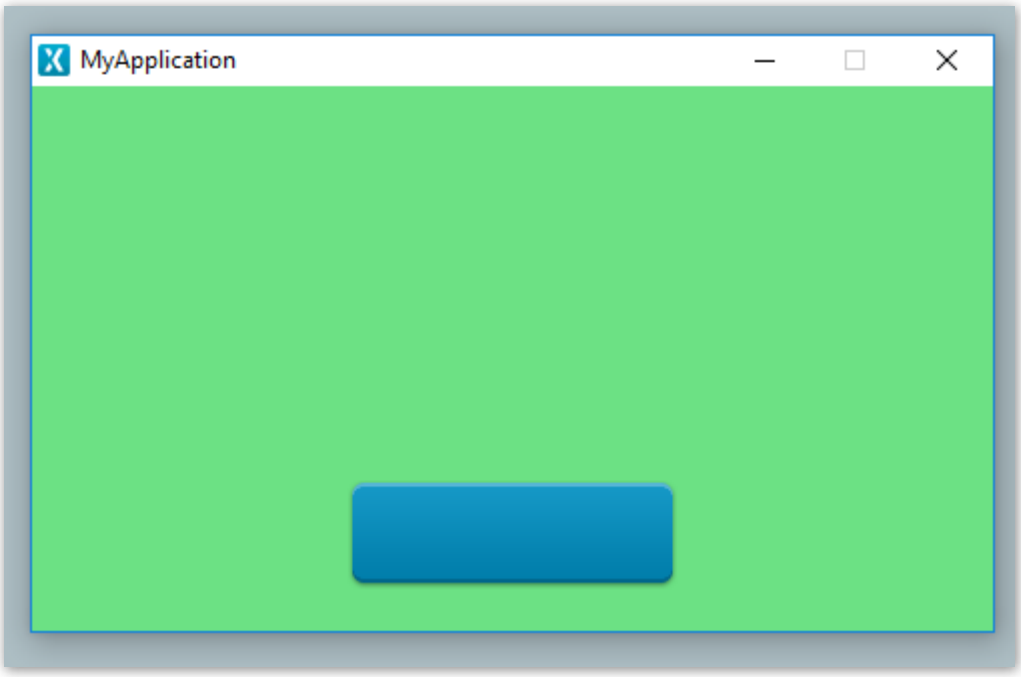
Next, for the value property we are going to input:

```
touchgfx::Color::getColorFrom24BitRGB(rand()%256, rand()%256, rand()%256)
```



Passing a random color when button is clicked

Now run the simulator and try pressing the button a couple of times. You should see the background changing to random colors.



Resulting random color when clicking the button

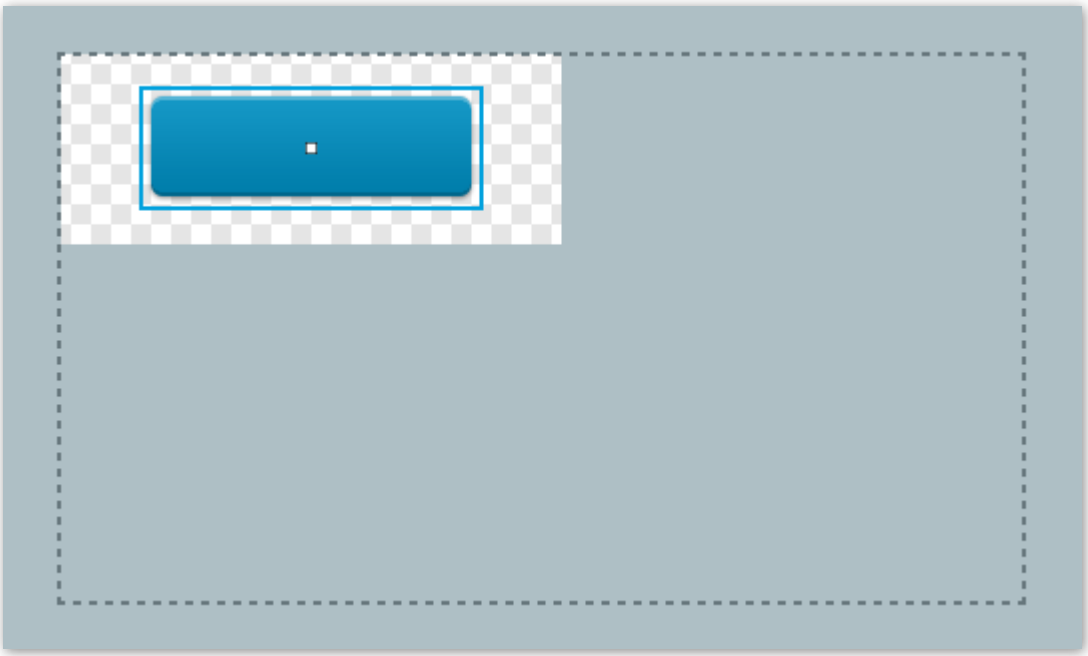
Using Custom Triggers in Custom Containers

Finally, Custom Containers can also define a collection of custom triggers so in this section we will expand upon the application by doing the following:

- Create a new custom container called ColorEmitter
- Add a custom trigger to ColorEmitter called "colorChanged"
- Use the colorChanged trigger to signal out random colors to the application when the button is pressed
- Set up interactions in the screen to listen for the colorChanged trigger
- Use whatever color the ColorEmitter sends out to set the color of the background box

Let's try using a custom trigger to signal some event in our application. Instead of our button interaction passing the random color to `setBackgroundColor`, let's try and make a custom container send out the random color to our Screen, and then let the Screen use whatever value the custom container communicated. This should end up being a simple example of different UI components communicating with each other in an application to make smaller, more reusable components.

First, let's create a new custom container and call it "ColorEmitter". Insert a button and call it "button". You should have something similar to the image below:

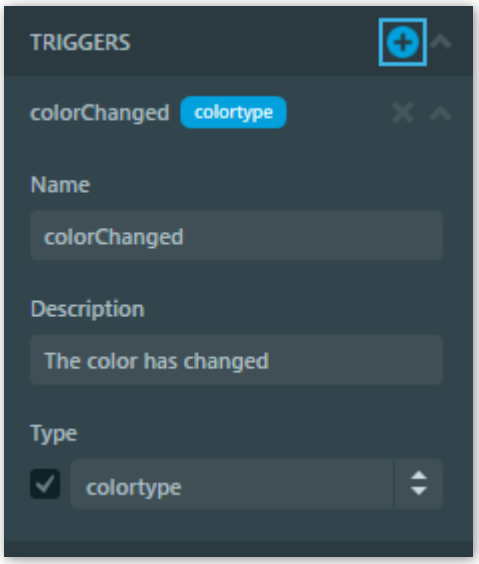


ColorEmitter custom container

Now, let's make it so that whenever the button is clicked, the ColorEmitter will emit a random color to the world. Anyone can then decide to listen for this emit and use the color for something. In our case, we just want to mimic the behavior we had before by using the color to set the background color.

To make the custom container emit a color, first we need to create a custom trigger. Go to the properties tab for the custom container and press the + button in the "TRIGGERS" group. Name the

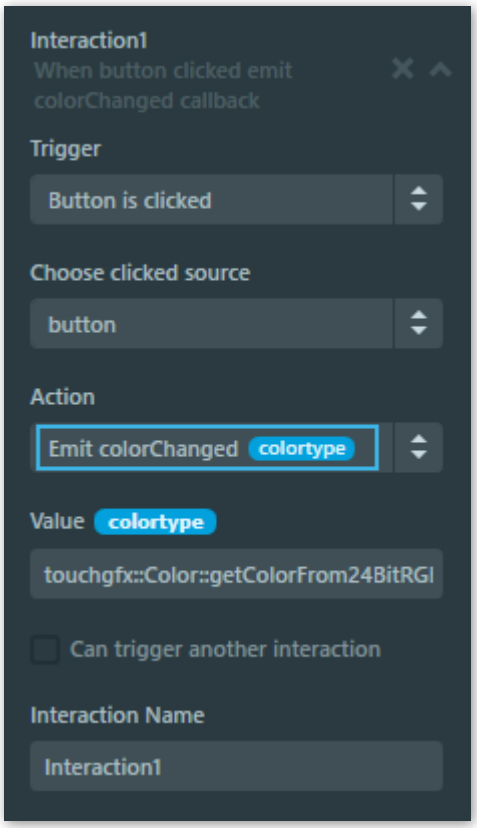
trigger "colorChanged", give it description "The color has changed" and give it the type "colortype".



Adding a custom trigger to a custom container

Next, go to the interactions tab for the custom container and create a new interaction. Use trigger "Button is clicked" and action "Emit colorChanged". Now we want to communicate a random color, so use the same code from earlier for the value property:

```
touchgfx::Color::getColorFrom24BitRGB(rand()%256, rand()%256, rand()%256)
```

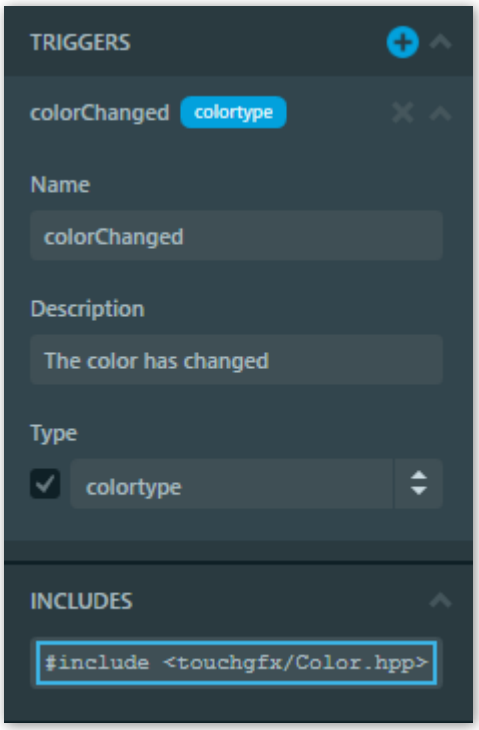


Emitting a custom trigger

However, this won't work initially since the `touchgfx::Color` namespace is not automatically included within our custom container. So like earlier, we are going to supply our own include for the

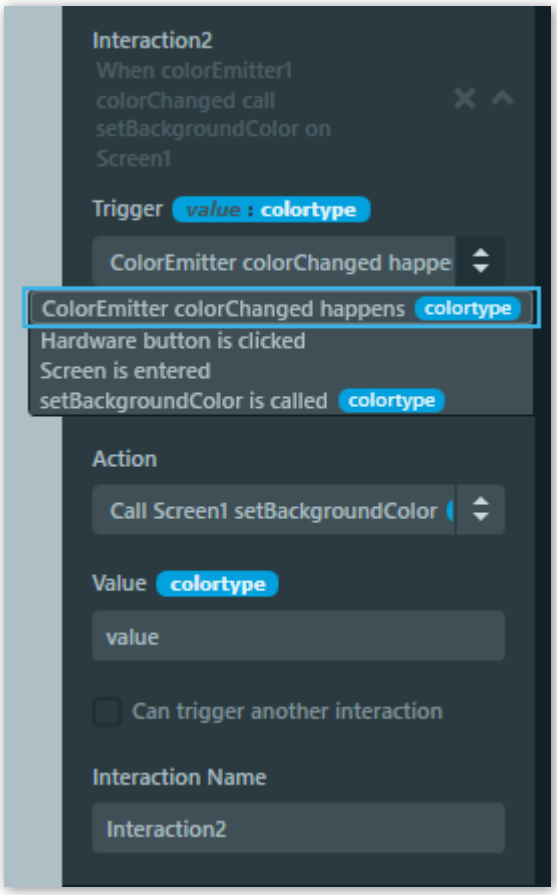
custom container. Go to the properties tab for the custom container and under the "INCLUDES" group, input:

```
#include <touchgfx/Color.hpp>
```



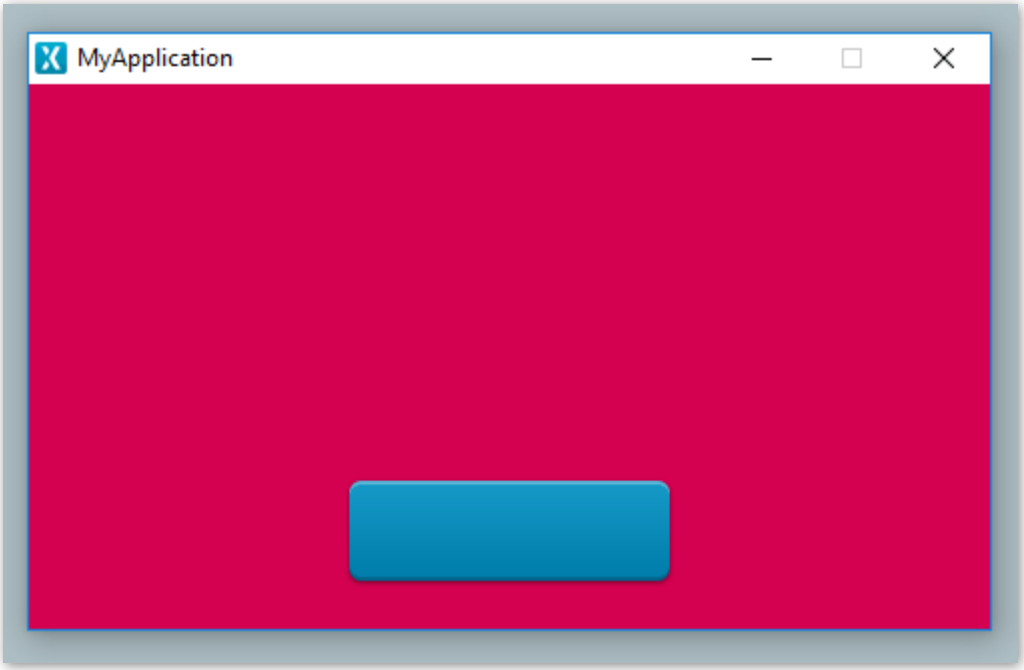
Supplying extra includes

Now we want to replace our old button with the new ColorEmitter custom container we have created. Select Screen1 and delete the button here. This should give a domain error in the interaction that was using this button, so delete that interaction also as we will create a new one for the ColorEmitter. Now insert an instance of our ColorEmitter on Screen1, then create a new interaction on Screen1. For the trigger, you should see an option called "ColorEmitter colorChanged happens". Select that one and for the action use "Call Screen1 setBackgroundColor". Now we need to use the value from the colorChanged emit, which will always be named "value" (like discussed earlier in this article). Therefore, input "value" into the value property.



Setting up an interaction to listen for colorChanged custom trigger

Now run simulator and try pressing the button again. The same behavior should be present, with the background changing to random colors. But now, instead of just having all the functionality implemented in the Screen, we've successfully created our own communication between the Screen and some of its smaller, reusable components, namely our simple ColorEmitter.



Resulting random color when button is clicked

Version: 4.16

AbstractButton

This class defines an abstract interface for button-like elements. A button is a clickable element that has two states: pressed and released. A button also has an action that is executed when the button goes from state pressed to state released.

Inherits from: Widget, Drawable

Inherited by: Button, RadioButton, TouchArea

Public Functions

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const

	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const

	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractButton

AbstractButton ( )

Sets this Widget touchable so the user can interact with buttons.

executeAction

virtual void executeAction ( )

Executes the previously set action.

See also:

setAction

getPressedState

virtual bool getPressedState ( ) const

Function to determine if the **AbstractButton** is currently pressed.

Returns:

true if button is pressed, false otherwise.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Updates the current state of the button.

The state can be either pressed or released, and if the new state is different from the current state, the button is also invalidated to force a redraw.

If the button state is changed from **ClickEvent::PRESSED** to **ClickEvent::RELEASED**, the associated action (if any) is also executed.

Parameters:

event Information about the click.

Reimplements: **touchgfx::Drawable::handleClickEvent**

Reimplemented by: **touchgfx::RadioButton::handleClickEvent**, **touchgfx::RepeatButton::handleClickEvent**, **touchgfx::ToggleButton::handleClickEvent**, **touchgfx::TouchArea::handleClickEvent**

setAction

void setAction (GenericCallback< const AbstractButton & > & callback)

Associates an action with the button.

The action is performed when the **AbstractButton** is in the pressed state, goes to the released.

Parameters:

callback The callback to be executed. The callback will be executed with a reference to the **AbstractButton**.

See also:

GenericCallback, **handleClickEvent**, **ClickEvent**

Protected Attributes Documentation

action

GenericCallback< const AbstractButton & > * action

The callback to be executed when this **AbstractButton** is clicked.

pressed

bool pressed

Is the button pressed or released? True if pressed.

Version: 4.16

AbstractButtonContainer

An abstract button container. The AbstractButtonContainer defines pressed/not pressed state, the alpha value, and the action Callback of a button. AbstractButtonContainer is used as superclass for classes defining a specific button behavior.

See: ClickButtonTrigger, RepeatButtonTrigger, ToggleButtonTrigger, TouchButtonTrigger

Inherits from: Container, Drawable

Inherited by: ClickButtonTrigger, RepeatButtonTrigger, ToggleButtonTrigger, TouchButtonTrigger

Public Functions

	AbstractButtonContainer()
virtual void	**executeAction**()
	Executes the previously set action.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
bool	**getPressed**()
	Gets the pressed state.
void	**setAction**(**GenericCallback**< const **AbstractButtonContainer** & > & callback)
	Sets an action callback to be executed by the subclass of AbstractContainerButton.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setPressed**(bool isPressed)
	Sets the pressed state to the given state.

Protected Functions

virtual void	**handleAlphaUpdated**()

	Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

GenericCallback< const **AbstractButtonContainer** & > *	**action**
	The action to be executed.
uint8_t	**alpha**
	The current alpha value. 255 denotes solid, 0 denotes completely invisible.
bool	**pressed**
	True if pressed.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0

	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.

int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()

Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractButtonContainer

AbstractButtonContainer ( )

executeAction

virtual void executeAction ( )

Executes the previously set action.

See also:

setAction

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getPressed

bool getPressed ( )

Gets the pressed state.

Returns:

True if it succeeds, false if it fails.

See also:

setPressed

setAction

void setAction (GenericCallback< const AbstractButtonContainer & > & callback)

Sets an action callback to be executed by the subclass of AbstractContainerButton.

Parameters:

callback The callback.

See also:

executeAction

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setPressed

void setPressed (bool isPressed)

Sets the pressed state to the given state.

A subclass of **AbstractButtonContainer** should implement handlePressedUpdate() to handle the new pressed state.

Parameters:

isPressed True if is pressed, false if not.

See also:

getPressed, **handlePressedUpdated**

Protected Functions Documentation

handleAlphaUpdated

virtual void handleAlphaUpdated ( )

Handles what should happen when the alpha is updated.

handlePressedUpdated

virtual void handlePressedUpdated ( )

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

action

GenericCallback< const AbstractButtonContainer & > * action

The action to be executed.

alpha

uint8_t alpha

The current alpha value. 255 denotes solid, 0 denotes completely invisible.

pressed

bool pressed

True if pressed.

Version: 4.16

AbstractClock

Superclass of clock widgets. Allows the hour, minute and second of the clock to be set and read.

See: AnalogClock, DigitalClock

Inherits from: Container, Drawable

Inherited by: AnalogClock, DigitalClock

Public Functions

	AbstractClock()
bool	**getCurrentAM**() const
	Is the current time a.m.
uint8_t	**getCurrentHour**() const
	Gets the current hour.
uint8_t	**getCurrentHour12**() const
	Gets current hour 12, i.e.
uint8_t	**getCurrentHour24**() const
	Gets current hour 24, i.e.
uint8_t	**getCurrentMinute**() const
	Gets the current minute.
uint8_t	**getCurrentSecond**() const
	Gets the current second.
virtual void	**setTime12Hour**(uint8_t hour, uint8_t minute, uint8_t second, bool am)
	Sets the time with input format as 12H.
virtual void	**setTime24Hour**(uint8_t hour, uint8_t minute, uint8_t second)
	Sets the time with input format as 24H.

Protected Functions

virtual void	**updateClock**() =0
	Update the visual representation of the clock on the display.

Protected Attributes

uint8_t	**currentHour**
	Local copy of the current hour.
uint8_t	**currentMinute**
	Local copy of the current minute.
uint8_t	**currentSecond**
	Local copy of the current second.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0

	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.

int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()

Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractClock

AbstractClock ( )

getCurrentAM

bool getCurrentAM ( ) const

Is the current time a.m.

or p.m.? True for a.m. and false for p.m.

Returns:

True if a.m., false if p.m.

getCurrentHour

uint8_t getCurrentHour ( ) const

Gets the current hour.

Returns:

The current hour in range 0-23.

See also:

getCurrentHour24, **getCurrentHour12**

getCurrentHour12

uint8_t getCurrentHour12 ( ) const

Gets current hour 12, i.e.

between 1 and 12.

Returns:

The current hour in range 1-12.

See also:

getCurrentHour24, **getCurrentAM**

getCurrentHour24

uint8_t getCurrentHour24 ( ) const

Gets current hour 24, i.e.

between 0 and 23.

Returns:

The current hour in range 0-23.

getCurrentMinute

uint8_t getCurrentMinute ( ) const

Gets the current minute.

Returns:

The current minute in range 0-59.

getCurrentSecond

uint8_t getCurrentSecond ( ) const

Gets the current second.

Returns:

The current second in range 0-59.

setTime12Hour

virtual void setTime12Hour (uint8_t hour ,
uint8_t minute ,
uint8_t second ,
bool am
)

Sets the time with input format as 12H.

Note that this does not affect any selected presentation formats.

Parameters:

hour	The hours, value should be between 1 and 12.
minute	The minutes, value should be between 0 and 59.
second	The seconds, value should be between 0 and 59.
am	AM/PM setting. True = AM, false = PM.

NOTE

all values passed are saved modulo the values limit. For example minutes=62 is treated as minutes=2.

setTime24Hour

virtual void setTime24Hour (uint8_t hour ,
uint8_t minute ,
uint8_t second
)

Sets the time with input format as 24H.

Note that this does not affect any selected presentation formats.

Parameters:

hour	The hours, value should be between 0 and 23.
minute	The minutes, value should be between 0 and 59.
second	The seconds, value should be between 0 and 59.

NOTE

all values passed are saved modulo the values limit. For example minutes=62 is treated as minutes=2.

Protected Functions Documentation

updateClock

virtual void updateClock ( ) =0

Update the visual representation of the clock on the display.

Reimplemented by: **touchgfx::AnalogClock::updateClock**, **touchgfx::DigitalClock::updateClock**

Protected Attributes Documentation

currentHour

uint8_t currentHour

Local copy of the current hour.

currentMinute

uint8_t currentMinute

Local copy of the current minute.

currentSecond

uint8_t currentSecond

Local copy of the current second.

Version: 4.16

AbstractDataGraph

An abstract data graph.

Inherits from: Container, Drawable

Inherited by: AbstractDataGraphWithY

Public Classes

class	**GraphClickEvent**
	An object of this type is passed with each callback that is sent when the graph is clicked.
class	**GraphDragEvent**
	An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions

	AbstractDataGraph(int16_t capacity)
	Initializes a new instance of the **AbstractDataGraph** class.
void	**addBottomElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area below the graph.
void	**addGraphElement**(**AbstractGraphElement** & d)
	Adds a graph element which will display the graph.
void	**addLeftElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area to the left of the graph.
void	**addRightElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area to the right of the graph.
void	**addTopElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area above the graph.
virtual void	**clear**()
	Clears the graph to its blank/initial state.

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

int16_t **getGapBeforeIndex**() const

Gets gap before index as set using setGapBeforeIndex().

int16_t **getGraphAreaHeight**() const

Gets graph area height.

int16_t **getGraphAreaHeightIncludingPadding**() const

Gets graph area height including padding (but not margin).

int16_t **getGraphAreaMarginBottom**() const

Gets graph margin bottom.

int16_t **getGraphAreaMarginLeft**() const

Gets graph margin left.

int16_t **getGraphAreaMarginRight**() const

Gets graph margin right.

int16_t **getGraphAreaMarginTop**() const

Gets graph margin top.

int16_t **getGraphAreaPaddingBottom**() const

Gets graph area padding bottom.

int16_t **getGraphAreaPaddingLeft**() const

Gets graph area padding left.

int16_t **getGraphAreaPaddingRight**() const

Gets graph area padding right.

int16_t **getGraphAreaPaddingTop**() const

Gets graph area padding top.

int16_t **getGraphAreaWidth**() const

Gets graph area width.

int16_t **getGraphAreaWidthIncludingPadding**() const

Gets graph area width including padding (but not margin).

virtual int **getGraphRangeXMax**() const =0

Gets the maximum x coordinate for the graph.

virtual int **getGraphRangeXMin**() const =0

Gets the minimum x coordinate for the graph.

virtual float **getGraphRangeYMaxAsFloat**() const =0

Gets maximum y coordinate for the graph.

virtual int **getGraphRangeYMaxAsInt**() const =0

Gets maximum y coordinate for the graph.

virtual float **getGraphRangeYMinAsFloat**() const =0

Gets minimum y coordinate for the graph.

virtual int **getGraphRangeYMinAsInt**() const =0

Gets minimum y coordinate for the graph.

int16_t **getMaxCapacity**() const

Gets the capacity (max number of points) of the graph.

virtual bool **getNearestIndexForScreenX**(int16_t x, int16_t & index) const

Gets graph index nearest to the given screen x coordinate.

virtual bool **getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)

Gets graph index nearest to the given screen position.

int **getScale**() const

Gets the scaling factor previously set using setScale().

int16_t **getUsedCapacity**() const

Gets the number of point used by the graph.

virtual void **handleClickEvent**(const **ClickEvent** & evt)

Defines the event handler interface for ClickEvents.

virtual void **handleDragEvent**(const **DragEvent** & evt)

Defines the event handler interface for DragEvents.

float **indexToDataPointXAsFloat**(int16_t index) const

Get the data point x value for the given graph point index.

int **indexToDataPointXAsInt**(int16_t index) const

Get the data point x value for the given graph point index.

float	**indexToDataPointYAsFloat**(int16_t index) const
	Get the data point y value for the given graph point index.
int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.
int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setGraphRangeX**(int min, int max) =0

	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions

int	**convertToGraphScale**(int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0

	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.
void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)

	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**
	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**

	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**
	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**
	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**
	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**

	The graph area top padding.
int16_t	**usedCapacity**
	The number of used points in the graph.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.

virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.

virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()

	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractDataGraph

AbstractDataGraph (int16_t capacity)

Initializes a new instance of the **AbstractDataGraph** class.

Parameters:

capacity The capacity.

addBottomElement

void addBottomElement (AbstractGraphDecoration & d)

Adds an element to be shown in the area below the graph.

Labels and titles can be added here.

Parameters:

d an **AbstractGraphElement** to add.

See also:

GraphLabelsX, **GraphTitle**

addGraphElement

void addGraphElement (AbstractGraphElement & d)

Adds a graph element which will display the graph.

Several graph elements can be added. Examples of graph elements are lines, dots, histograms as well as horizontal and vertical grid lines.

Parameters:

d an **AbstractGraphElement** to add.

See also:

GraphElementGridX, **GraphElementGridY**, **GraphElementArea**, **GraphElementBoxes**, **GraphElementDiamonds**, **GraphElementDots**, **GraphElementHistogram**, **GraphElementLine**,

addLeftElement

void addLeftElement (AbstractGraphDecoration & d)

Adds an element to be shown in the area to the left of the graph.

Labels and titles can be added here.

Parameters:

d an **AbstractGraphElement** to add.

See also:

GraphLabelsY, **GraphTitle**

addRightElement

void addRightElement (AbstractGraphDecoration & d)

Adds an element to be shown in the area to the right of the graph.

Labels and titles can be added here.

Parameters:

d an **AbstractGraphElement** to add.

See also:

GraphLabelsY, **GraphTitle**

addTopElement

void addTopElement (AbstractGraphDecoration & d)

Adds an element to be shown in the area above the graph.

Labels and titles can be added here.

Parameters:

d an **AbstractGraphElement** to add.

See also:

GraphLabelsX, **GraphTitle**

clear

virtual void clear ( )

Clears the graph to its blank/initial state.

Reimplemented by: **touchgfx::DataGraphScroll::clear**, **touchgfx::DataGraphWrapAndOverwrite::clear**

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getGapBeforeIndex

int16_t getGapBeforeIndex ( ) const

Gets gap before index as set using setGapBeforeIndex().

Returns:

The gap before index.

See also:

setGapBeforeIndex

getGraphAreaHeight

int16_t getGraphAreaHeight ( ) const

Gets graph area height.

This is the height of the actual graph area and is the same as the height of the graph widget where graph area margin and graph area padding has been removed.

Returns:

The graph area height.

getGraphAreaHeightIncludingPadding

int16_t getGraphAreaHeightIncludingPadding ( ) const

Gets graph area height including padding (but not margin).

This is the height of the actual graph area and is the same as the height of the graph widget where graph area margin has been removed.

Returns:

The graph area height including graph padding.

getGraphAreaMarginBottom

int16_t getGraphAreaMarginBottom ( ) const

Gets graph margin bottom.

Returns:

The graph margin bottom.

See also:

setGraphAreaMargin

getGraphAreaMarginLeft

int16_t getGraphAreaMarginLeft ( ) const

Gets graph margin left.

Returns:

The graph margin left.

See also:

setGraphAreaMargin

getGraphAreaMarginRight

int16_t getGraphAreaMarginRight ( ) const

Gets graph margin right.

Returns:

The graph margin right.

See also:

setGraphAreaMargin

getGraphAreaMarginTop

int16_t getGraphAreaMarginTop ( ) const

Gets graph margin top.

Returns:

The graph margin top.

See also:

setGraphAreaMargin

getGraphAreaPaddingBottom

int16_t getGraphAreaPaddingBottom ( ) const

Gets graph area padding bottom.

Returns:

The graph area padding bottom.

See also:

setGraphAreaPadding

getGraphAreaPaddingLeft

int16_t getGraphAreaPaddingLeft ( ) const

Gets graph area padding left.

Returns:

The graph area padding left.

See also:

setGraphAreaPadding

getGraphAreaPaddingRight

int16_t getGraphAreaPaddingRight ( ) const

Gets graph area padding right.

Returns:

The graph area padding right.

See also:

setGraphAreaPadding

getGraphAreaPaddingTop

int16_t getGraphAreaPaddingTop ( ) const

Gets graph area padding top.

Returns:

The graph areapadding top.

See also:

setGraphAreaPadding

getGraphAreaWidth

int16_t getGraphAreaWidth ( ) const

Gets graph area width.

This is the width of the actual graph area and is the same as the width of the graph widget where graph area margin and graph area padding has been removed.

Returns:

The graph area width.

getGraphAreaWidthIncludingPadding

int16_t getGraphAreaWidthIncludingPadding ( ) const

Gets graph area width including padding (but not margin).

This is the width of the actual graph area and is the same as the width of the graph widget where graph area margin has been removed.

Returns:

The graph width including graph padding.

getGraphRangeXMax

virtual int getGraphRangeXMax ( ) const =0

Gets the maximum x coordinate for the graph.

Returns:

The maximum x coordinate .

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getGraphRangeXMax**

getGraphRangeXMin

virtual int getGraphRangeXMin ( ) const =0

Gets the minimum x coordinate for the graph.

Returns:

The minimum x coordinate .

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getGraphRangeXMin**

getGraphRangeYMaxAsFloat

virtual float getGraphRangeYMaxAsFloat ( ) const =0

Gets maximum y coordinate for the graph.

Returns:

The maximum y coordinate.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getGraphRangeYMaxAsFloat**

getGraphRangeYMaxAsInt

virtual int getGraphRangeYMaxAsInt ( ) const =0

Gets maximum y coordinate for the graph.

Returns:

The maximum y coordinate.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getGraphRangeYMaxAsInt**

getGraphRangeYMinAsFloat

virtual float getGraphRangeYMinAsFloat ( ) const =0

Gets minimum y coordinate for the graph.

Returns:

The minimum y coordinate.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getGraphRangeYMinAsFloat**

getGraphRangeYMinAsInt

virtual int getGraphRangeYMinAsInt ( ) const =0

Gets minimum y coordinate for the graph.

Returns:

The minimum y coordinate.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getGraphRangeYMinAsInt**

getMaxCapacity

int16_t getMaxCapacity ( ) const

Gets the capacity (max number of points) of the graph.

Returns:

The capacity.

getNearestIndexForScreenX

```
virtual bool getNearestIndexForScreenX ( int16_t x , const
                                         int16_t & index const
                                         ) const
```

Gets graph index nearest to the given screen x coordinate.

The index of the graph point closest to the given x coordinate is handed back.

Parameters:

x	The x coordinate.
index	Zero-based index of the.

Returns:

True if it succeeds, false if it fails.

See also:

getNearestIndexForScreenXY

getNearestIndexForScreenXY

```
virtual bool getNearestIndexForScreenXY ( int16_t x ,
                                          int16_t y ,
                                          int16_t & index
                                          )
```

Gets graph index nearest to the given screen position.

The distance to each point on the graph is measured and the index of the point closest to the given position handed back.

Parameters:

x	The x coordinate.
y	The y coordinate.
index	Zero-based index of the point closest to the given position.

Returns:

True if it succeeds, false if it fails.

See also:

getNearestIndexForScreenX

getScale

int getScale ( ) const

Gets the scaling factor previously set using setScale().

Returns:

The scaling factor.

See also:

setScale

getUsedCapacity

int16_t getUsedCapacity ( ) const

Gets the number of point used by the graph.

Returns:

The number of point used by the graph.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

indexToDataPointXAsFloat

float indexToDataPointXAsFloat (int16_t index)

Get the data point x value for the given graph point index.

Parameters:

index Zero-based index of the point.

Returns:

The data point x value.

indexToDataPointXAsInt

int indexToDataPointXAsInt (int16_t index)

Get the data point x value for the given graph point index.

Parameters:

index Zero-based index of the point.

Returns:

The data point x value.

indexToDataPointYAsFloat

float indexToDataPointYAsFloat (int16_t index)

Get the data point y value for the given graph point index.

Parameters:

index Zero-based index of the point.

Returns:

The data point y value.

indexToDataPointYAsInt

int indexToDataPointYAsInt (int16_t index)

Get the data point y value for the given graph point index.

Parameters:

index Zero-based index of the point.

Returns:

The data point y value.

indexToGlobalIndex

virtual int32_t indexToGlobalIndex (int16_t index)

Convert an index to global index.

The index is the index of any data point, The global index is a value that keeps growing whenever a new data point is added the the graph.

Parameters:

index Zero-based index of the point.

Returns:

The global index.

Reimplemented by: **touchgfx::DataGraphScroll::indexToGlobalIndex**, **touchgfx::DataGraphWrapAndClear::indexToGlobalIndex**, **touchgfx::DataGraphWrapAndOverwrite::indexToGlobalIndex**

indexToScreenX

int16_t indexToScreenX (int16_t index)

Get the screen x coordinate for the given graph point index.

Parameters:

index Zero-based index of the point.

Returns:

The screen x coordinate.

indexToScreenY

int16_t indexToScreenY (int16_t index)

Get the screen y coordinate for the given graph point index.

Parameters:

index Zero-based index of the point.

Returns:

The screen x coordinate.

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.All graph elements have to take this alpha into consideration.

See also:

getAlpha

setClickAction

void setClickAction (GenericCallback< const AbstractDataGraph &, const GraphClickEvent & > & callback)

Sets an action to be executed when the graph is clicked.

Parameters:

callback The callback.

See also:

GraphClickEvent

setDragAction

void setDragAction (GenericCallback< const AbstractDataGraph &, const GraphDragEvent & > & callback)

Sets an action to be executed when the graph is dragged.

Parameters:

callback The callback.

See also:

GraphDragEvent

setGapBeforeIndex

void setGapBeforeIndex (int16_t index)

Makes gap before the specified index.

This can be used to split a graph in two, but for some graph types, e.g. histograms, this has no effect. Only one gap can be specified at a time. Specifying a new gap automatically removes the previous gap.

Parameters:

index Zero-based index where the gap should be placed.

setGraphAreaMargin

void setGraphAreaMargin (int16_t top , int16_t left , int16_t right , int16_t bottom)

Sets graph position inside the widget by reserving a margin around the graph.

These areas to the left, the right, above and below are used for optional axis and titles.

Parameters:

top	The top margin in pixels.
left	The left margin in pixels.
right	The right margin in pixels.
bottom	The bottom margin in pixels.

NOTE

The graph is automatically invalidated when the graph margins are changed.

See also:

GraphLabelsX, **GraphLabelsY**, **GraphTitle**

setGraphAreaPadding

```
void setGraphAreaPadding ( int16_t top ,
                           int16_t left ,
                           int16_t right ,
                           int16_t bottom
                           )
```

Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.

that extend around the actual data point). The set padding will also work to make a gap between the graph and any labels that might have been added to the graph. To reserve an area that the graph will not be drawn in, use setGraphAreaMargin.

Parameters:

top	The top padding in pixels.
left	The left padding in pixels.
right	The right padding in pixels.
bottom	The bottom padding in pixels.

NOTE

The graph is automatically invalidated when the margins are set.

See also:

setGraphAreaMargin

setGraphRange

```
void setGraphRange ( int   xMin ,
                     int   xMax ,
                     float yMin ,
                     float yMax
                     )
```

Sets minimum and maximum x and y coordinate ranges for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

xMin The minimum x coordinate.
xMax The maximum x coordinate.
yMin The minimum y coordinate.
yMax The maximum y coordinate.

See also:

setGraphRangeX, **setGraphRangeY**

setGraphRange

```
void setGraphRange ( int xMin ,
                     int xMax ,
                     int yMin ,
                     int yMax
                     )
```

Sets minimum and maximum x and y coordinate ranges for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

xMin The minimum x coordinate.
xMax The maximum x coordinate.
yMin The minimum y coordinate.
yMax The maximum y coordinate.

See also:

setGraphRangeX, **setGraphRangeY**

setGraphRangeX

```
virtual void setGraphRangeX ( int min ,  =0
                              int max   =0
                              )    =0
```

Sets minimum and maximum x coordinates for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

min The minimum x coordinate.
max The maximum x coordinate.

NOTE

The graph as well as the area above and below are automatically redrawn (invalidated).

Reimplemented by: **touchgfx::AbstractDataGraphWithY::setGraphRangeX**

setGraphRangeY

```
virtual void setGraphRangeY ( float min ,  =0
                              float max   =0
                              )     =0
```

Sets minimum and maximum y coordinates for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

min The minimum y coordinate.
max The maximum y coordinate.

NOTE

The graph as well as the area to the left and to the right of the graph are automatically redrawn (invalidated)

Reimplemented by: **touchgfx::AbstractDataGraphWithY::setGraphRangeY**

setGraphRangeY

```
virtual void setGraphRangeY ( int min ,  =0
                              int max   =0
                              )    =0
```

Sets minimum and maximum y coordinates for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

min The minimum y coordinate.
max The maximum y coordinate.

NOTE

The graph as well as the area to the left and to the right of the graph are automatically redrawn (invalidated)

Reimplemented by: **touchgfx::AbstractDataGraphWithY::setGraphRangeY**

setHeight

virtual void setHeight (int16_t height)

Sets the height of this drawable.

Parameters:

height The new height.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

Reimplements: **touchgfx::Drawable::setHeight**

setScale

virtual void setScale (int scale)

Sets a scaling factor to be multiplied on each added element.

Since the graph only stores integer values internally, it is possible to set a scale to e.g. 1000 and make the graph work as if there are three digits of precision. The addDataPoint() will multiply the argument with the scaling factor and store this value.

By setting the scale to 1 it is possible to simply use integer values for the graph.

Parameters:

scale The scaling factor.

NOTE

Calling setScale after adding points to the graph has undefined behaviour. The scale should be set as the first thing before other settings of the graph is being set.

See also:

getScale

Reimplemented by: **touchgfx::AbstractDataGraphWithY::setScale**

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

Reimplements: **touchgfx::Drawable::setWidth**

float2scaled

static int float2scaled (float f ,
int scale
)

Multiply a floating point value with a constant and round the result.

Parameters:

f the value to scale.
scale The scale.

Returns:

The product of the two numbers, rounded to nearest integer value.

int2scaled

```
static int int2scaled ( int i ,
                        int scale
                        )
```

Multiply an integer value with a constant.

Parameters:

i	the value to scale.
scale	The scale.

Returns:

The product of the two numbers.

scaled2float

```
static float scaled2float ( int i ,
                            int scale
                            )
```

Divide a floating point number with a constant.

Parameters:

i	The number to divide.
scale	The divisor (scale).

Returns:

The number divided by the scale.

scaled2int

```
static int scaled2int ( int i ,
                        int scale
                        )
```

Divide an integer with a constant and round the result.

Parameters:

i	The number to divide.
scale	The divisor (scale).

Returns:

The number divided by the scale, rounded to nearest integer.

Protected Functions Documentation

convertToGraphScale

```
int convertToGraphScale ( int value ,  const
                          int scale    const
                          )    const
```

Converts a number with one scale to a number that has the same scale as the graph.

Parameters:

value The value to convert.
scale The scale.

Returns:

The given data converted to the graph scale.

NOTE

For internal use.

float2scaled

```
int float2scaled ( float f )
```

Same as **float2scaled(float,int)** using the graph's scale.

Parameters:

f The floating point value to scale.

Returns:

The scaled value.

NOTE

For internal use.

getGraphRangeYMaxScaled

virtual int getGraphRangeYMaxScaled ( ) const =0

Gets maximum y coordinate for the graph.

Returns:

The maximum y coordinate.

NOTE

The returned value is left scaled.For internal use.

See also:

AbstractDataGraph::getGraphRangeYMaxAsInt,
AbstractDataGraph::getGraphRangeYMaxAsFloat

Reimplemented by: touchgfx::AbstractDataGraphWithY::getGraphRangeYMaxScaled

getGraphRangeYMinScaled

virtual int getGraphRangeYMinScaled ( ) const =0

Gets minimum y coordinate for the graph.

Returns:

The minimum y coordinate.

NOTE

The returned value is left scaled.For internal use.

See also:

AbstractDataGraph::getGraphRangeYMinAsInt,
AbstractDataGraph::getGraphRangeYMinAsFloat

Reimplemented by: touchgfx::AbstractDataGraphWithY::getGraphRangeYMinScaled

getXAxisOffsetScaled

virtual int getXAxisOffsetScaled ( ) const

Get x axis offset as a scaled value.

Returns:

The x axis offset (left scaled).

NOTE

For internal use.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getXAxisOffsetScaled**

getXAxisScaleScaled

virtual int getXAxisScaleScaled ( ) const

Get x axis scale as a scaled value.

Returns:

The x axis scale (left scaled).

NOTE

For internal use.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::getXAxisScaleScaled**

indexToDataPointXScaled

virtual int indexToDataPointXScaled (int16_t index)

Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.

Parameters:

index Zero-based index of the data point.

Returns:

The data point x value scaled.

NOTE

For internal use.

See also:

indexToDataPointXAsInt, **indexToDataPointXAsFloat**

Reimplemented by: touchgfx::AbstractDataGraphWithY::indexToDataPointXScaled

indexToDataPointYScaled

virtual int indexToDataPointYScaled (int16_t index)

Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.

Parameters:

index Zero-based index of the data point.

Returns:

The data point y value scaled.

NOTE

For internal use.

See also:

indexToDataPointYAsInt, indexToDataPointYAsFloat

Reimplemented by: touchgfx::AbstractDataGraphWithY::indexToDataPointYScaled

indexToScreenXQ5

virtual CWRUtil::Q5 indexToScreenXQ5 (int16_t index)

Gets screen x coordinate for a specific data point added to the graph.

Parameters:

index The index of the element to get the x coordinate for.

Returns:

The screen x coordinate for the specific data point.

Reimplemented by: touchgfx::AbstractDataGraphWithY::indexToScreenXQ5

indexToScreenYQ5

virtual CWRUtil::Q5 indexToScreenYQ5 (int16_t index)

Gets screen y coordinate for a specific data point added to the graph.

Parameters:

index The index of the element to get the y coordinate for.

Returns:

The screen x coordinate for the specific data point.

Reimplemented by: **touchgfx::AbstractDataGraphWithY::indexToScreenYQ5**

int2scaled

int int2scaled (int i)

Same as **int2scaled(int,int)** using the graph's scale.

Parameters:

i The integer value to scale.

Returns:

The scaled integer.

NOTE

For internal use.

invalidateAllXAxisPoints

void invalidateAllXAxisPoints ( )

Invalidate all x axis points.

Similar to invalidateXAxisPointAt, this function will iterate all visible x values and invalidate them in turn.

See also:

invalidateXAxisPointAt

invalidateGraphArea

void invalidateGraphArea ( )

Invalidate entire graph area (the center of the graph).

This is often useful when a graph is cleared or the X or Y range is changed.

invalidateGraphPointAt

void invalidateGraphPointAt (int16_t index)

Invalidate point at a given index.

This will call the function **invalidateGraphPointAt()** on every element added to the graphArea which in turn is responsible for invalidating the part of the screen occupied by its element.

Parameters:

index Zero-based index of the element to invalidate.

invalidateXAxisPointAt

void invalidateXAxisPointAt (int16_t index)

Invalidate x axis point at the given index.

Since the y axis is often static, the x axis can change, and all labels need to be updated wihtout redrawing the entire graph.

Parameters:

index The x index to invalidate.

See also:

invalidateAllXAxisPoints

scaled2float

float scaled2float (int i)

Same as **scaled2float(int,int)** using the graph's scale.

Parameters:

i The scaled value to convert to a floating point value.

Returns:

The unscaled value.

NOTE

For internal use.

scaled2int

int scaled2int (int i)

Same as **scaled2int(int,int)** using the graph's scale.

Parameters:

i The scaled value to convert to an integer.

Returns:

The unscaled value.

NOTE

For internal use.

setGraphRangeScaled

void setGraphRangeScaled (int xMin ,
int xMax ,
int yMin ,
int yMax
)

Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.

Parameters:

xMin The minimum x coordinate.
xMax The maximum x coordinate.
yMin The minimum y coordinate.
yMax The maximum y coordinate.

NOTE

For internal use.

See also:

setGraphRange

setGraphRangeYScaled

virtual void setGraphRangeYScaled (int min , =0
int max =0
) =0

Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.

Parameters:

min The minimum y coordinate.
max The maximum y coordinate.

NOTE

For internal use.

See also:

setGraphRangeY

Reimplemented by: **touchgfx::AbstractDataGraphWithY::setGraphRangeYScaled**

updateAreasPosition

void updateAreasPosition ( )

Updates the position of all elements in all area after a change in size of the graph area and/or label padding.

NOTE

The entire graph area is invalidated.

valueToScreenXQ5

virtual CWRUtil::Q5 valueToScreenXQ5 (int x)

Gets screen x coordinate for an absolute value.

Parameters:

x The x value.

Returns:

The screen x coordinate for the given value.

Reimplemented by: touchgfx::AbstractDataGraphWithY::valueToScreenXQ5

valueToScreenYQ5

virtual CWRUtil::Q5 valueToScreenYQ5 (int y)

Gets screen y coordinate for an absolute value.

Parameters:

y The y value.

Returns:

The screen y coordinate for the given value.

Reimplemented by: touchgfx::AbstractDataGraphWithY::valueToScreenYQ5

xScreenRangeToIndexRange

```
virtual bool xScreenRangeToIndexRange ( int16_t    xLo ,      const =0
                                        int16_t    xHi ,      const =0
                                        int16_t &  indexLow ,  const =0
                                        int16_t &  indexHigh   const =0
                                        )          const =0
```

Gets index range for screen x coordinate range taking the current graph range into account.

Parameters:

xLo	The low screen x coordinate.
xHi	The high screen x coordinate.
indexLow	The low element index.
indexHigh	The high element index.

Returns:

True if the range from low index to high index is legal.

NOTE

For internal use.

Reimplemented by: touchgfx::AbstractDataGraphWithY::xScreenRangeToIndexRange

Protected Attributes Documentation

alpha

uint8_t alpha

The alpha of the entire graph.

bottomArea

Container bottomArea

The area below the graph.

bottomPadding

int16_t bottomPadding

The graph area bottom padding.

clickAction

GenericCallback< const AbstractDataGraph &, const GraphClickEvent & > * clickAction

The callback to be executed when this Graph is clicked.

dataScale

int dataScale

The data scale applied to all values.

dragAction

GenericCallback< const AbstractDataGraph &, const GraphDragEvent & > * dragAction

The callback to be executed when this Graph is dragged.

gapBeforeIndex

int16_t gapBeforeIndex

The graph is disconnected (there is a gap) before this element index.

graphArea

Container graphArea

The graph area (the center area)

leftArea

Container leftArea

The area to the left of the graph.

leftPadding

int16_t leftPadding

The graph area left padding.

maxCapacity

int16_t maxCapacity

Maximum number of points in the graph.

rightArea

Container rightArea

The area to the right of the graph.

rightPadding

int16_t rightPadding

The graph area right padding.

topArea

Container topArea

The area above the graph.

topPadding

int16_t topPadding

The graph area top padding.

usedCapacity

int16_t usedCapacity

The number of used points in the graph.

Version: 4.16

AbstractDataGraphWithY

Abstract helper class used to implement graphs with the same distance between the x values (i.e. x is ignored).

Inherits from: AbstractDataGraph, Container, Drawable

Inherited by: DataGraphScroll, DataGraphWrapAndClear, DataGraphWrapAndOverwrite

Public Functions

	AbstractDataGraphWithY(int16_t capacity, int * values)
	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)
	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.

virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const
	Get x coordinate axis scale value.
virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.
virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.
void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.

virtual int16_t	**addValue**(int value) =0
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.
virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)

	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes

uint32_t	**dataCounter**
	The data counter of how many times addDataPoint() has been called.
int	**xOffset**
	The x axis offset (real value of data point at index 0)
int	**xScale**
	The x axis scale (increment between two data points)
int *	**yValues**
	The values of the graph.

Additional inherited members

Public Classes inherited from AbstractDataGraph

class	**GraphClickEvent**
	An object of this type is passed with each callback that is sent when the graph is clicked.
class	**GraphDragEvent**
	An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

	AbstractDataGraph(int16_t capacity)
	Initializes a new instance of the **AbstractDataGraph** class.
void	**addBottomElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area below the graph.
void	**addGraphElement**(**AbstractGraphElement** & d)
	Adds a graph element which will display the graph.
void	**addLeftElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area to the left of the graph.
void	**addRightElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area to the right of the graph.
void	**addTopElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area above the graph.
virtual void	**clear**()
	Clears the graph to its blank/initial state.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
int16_t	**getGapBeforeIndex**() const
	Gets gap before index as set using setGapBeforeIndex().
int16_t	**getGraphAreaHeight**() const
	Gets graph area height.
int16_t	**getGraphAreaHeightIncludingPadding**() const
	Gets graph area height including padding (but not margin).
int16_t	**getGraphAreaMarginBottom**() const
	Gets graph margin bottom.
int16_t	**getGraphAreaMarginLeft**() const
	Gets graph margin left.
int16_t	**getGraphAreaMarginRight**() const
	Gets graph margin right.
int16_t	**getGraphAreaMarginTop**() const

	Gets graph margin top.
int16_t	**getGraphAreaPaddingBottom**() const
	Gets graph area padding bottom.
int16_t	**getGraphAreaPaddingLeft**() const
	Gets graph area padding left.
int16_t	**getGraphAreaPaddingRight**() const
	Gets graph area padding right.
int16_t	**getGraphAreaPaddingTop**() const
	Gets graph area padding top.
int16_t	**getGraphAreaWidth**() const
	Gets graph area width.
int16_t	**getGraphAreaWidthIncludingPadding**() const
	Gets graph area width including padding (but not margin).
int16_t	**getMaxCapacity**() const
	Gets the capacity (max number of points) of the graph.
virtual bool	**getNearestIndexForScreenX**(int16_t x, int16_t & index) const
	Gets graph index nearest to the given screen x coordinate.
virtual bool	**getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)
	Gets graph index nearest to the given screen position.
int	**getScale**() const
	Gets the scaling factor previously set using setScale().
int16_t	**getUsedCapacity**() const
	Gets the number of point used by the graph.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.

int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const
	Get the data point y value for the given graph point index.
int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.
int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)

	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.
void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.

float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)
	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**
	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**

	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**
	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**
	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**
	The graph area top padding.
int16_t	**usedCapacity**
	The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.

virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.

Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.

bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractDataGraphWithY

```
AbstractDataGraphWithY ( int16_t capacity ,
                         int *   values
                         )
```

Initializes a new instance of the **AbstractDataGraphWithY** class.

Parameters:

capacity The capacity.
values Address where to store the y values of the graph.

addDataPoint

```
int16_t addDataPoint ( float y )
```

Adds a new data point to the end of the graph.

The part of the graph that is changed, is automatically redrawn (invalidated).

Parameters:

y The new data point.

Returns:

The index of the just added value.

addDataPoint

```
int16_t addDataPoint ( int y )
```

Adds a new data point to the end of the graph.

The part of the graph that is changed, is automatically redrawn (invalidated).

Parameters:

y The new data point.

Returns:

The index of the just added value.

getGraphRangeXMax

virtual int getGraphRangeXMax ( ) const

Gets the maximum x coordinate for the graph.

Returns:

The maximum x coordinate .

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeXMax**

getGraphRangeXMin

virtual int getGraphRangeXMin ( ) const

Gets the minimum x coordinate for the graph.

Returns:

The minimum x coordinate .

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeXMin**

getGraphRangeYMaxAsFloat

virtual float getGraphRangeYMaxAsFloat ( ) const

Gets maximum y coordinate for the graph.

Returns:

The maximum y coordinate.

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeYMaxAsFloat**

getGraphRangeYMaxAsInt

virtual int getGraphRangeYMaxAsInt ( ) const

Gets maximum y coordinate for the graph.

Returns:

The maximum y coordinate.

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeYMaxAsInt**

getGraphRangeYMinAsFloat

virtual float getGraphRangeYMinAsFloat ( ) const

Gets minimum y coordinate for the graph.

Returns:

The minimum y coordinate.

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeYMinAsFloat**

getGraphRangeYMinAsInt

virtual int getGraphRangeYMinAsInt ( ) const

Gets minimum y coordinate for the graph.

Returns:

The minimum y coordinate.

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeYMinAsInt**

getXAxisOffsetAsFloat

virtual float getXAxisOffsetAsFloat ( ) const

Get x coordinate axis offset value.

This is the real x value of the first data point added to the graph (i.e. the data point at index 0).

Returns:

The x axis offset.

getXAxisOffsetAsInt

virtual int getXAxisOffsetAsInt ( ) const

Get x coordinate axis offset value.

This is the real x value of the first data point added to the graph (i.e. the data point at index 0).

Returns:

The x axis offset.

getXAxisScaleAsFloat

virtual float getXAxisScaleAsFloat ( ) const

Get x coordinate axis scale value.

This is the real x value increment between two data points added to the graph.

Returns:

The x axis scale.

getXAxisScaleAsInt

virtual int getXAxisScaleAsInt ( ) const

Get x coordinate axis scale value.

This is the real x value increment between two data points added to the graph.

Returns:

The x axis scale.

setGraphRangeX

virtual void setGraphRangeX (int min ,
int max
)

Sets minimum and maximum x coordinates for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

min The minimum x coordinate.
max The maximum x coordinate.

NOTE

The graph as well as the area above and below are automatically redrawn (invalidated).

Reimplements: **touchgfx::AbstractDataGraph::setGraphRangeX**

setGraphRangeY

```
virtual void setGraphRangeY ( float min ,
                              float max
                              )
```

Sets minimum and maximum y coordinates for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

min The minimum y coordinate.
max The maximum y coordinate.

NOTE

The graph as well as the area to the left and to the right of the graph are automatically redrawn (invalidated)

Reimplements: **touchgfx::AbstractDataGraph::setGraphRangeY**

setGraphRangeY

```
virtual void setGraphRangeY ( int min ,
                              int max
                              )
```

Sets minimum and maximum y coordinates for the graph.

This can be used to zoom in or out and only show parts of the graph.

Parameters:

min The minimum y coordinate.
max The maximum y coordinate.

NOTE

The graph as well as the area to the left and to the right of the graph are automatically redrawn (invalidated)

Reimplements: **touchgfx::AbstractDataGraph::setGraphRangeY**

setGraphRangeYAuto

```
void setGraphRangeYAuto ( bool showXaxis =true,
                          int  margin =0
                          )
```

Automatic adjust min and max y coordinate to show entire graph.

It is possible to ensure that the x axis (i.e. y=0) is included in the new range. If the graph range is changed, the graph is automatically redrawn (invalidated).

Parameters:

showXaxis	(Optional) True to ensure that the x axis is visible (default is true).
margin	(Optional) The margin to add above/below the max/min y value (default is no margin).

NOTE

This takes the current visible x coordinate range into account.

setScale

```
virtual void setScale ( int scale )
```

Sets a scaling factor to be multiplied on each added element.

Since the graph only stores integer values internally, it is possible to set a scale to e.g. 1000 and make the graph work as if there are three digits of precision. The **addDataPoint()** will multiply the argument with the scaling factor and store this value.

By setting the scale to 1 it is possible to simply use integer values for the graph.

Parameters:

scale	The scaling factor.

NOTE

Calling setScale after adding points to the graph has undefined behaviour. The scale should be set as the first thing before other settings of the graph is being set.

See also:

getScale

Reimplements: **touchgfx::AbstractDataGraph::setScale**

setXAxisOffset

virtual void setXAxisOffset (float offset)

Set x coordinate axis offset value.

This is the real x value of the first data point added to the graph (i.e. the data point at index 0).

Parameters:

offset The x axis offset.

setXAxisOffset

virtual void setXAxisOffset (int offset)

Set x coordinate axis offset value.

This is the real x value of the first data point added to the graph (i.e. the data point at index 0).

Parameters:

offset The x axis offset.

setXAxisScale

virtual void setXAxisScale (float scale)

Set x coordinate axis scale value.

This is the real x value increment between two data points added to the graph.

Parameters:

scale The x axis scale.

setXAxisScale

virtual void setXAxisScale (int scale)

Set x coordinate axis scale value.

This is the real x value increment between two data points added to the graph.

Parameters:

scale The x axis scale.

Protected Functions Documentation

addDataPointScaled

int16_t addDataPointScaled (int y)

Same as **addDataPoint(int)** except the passed argument is assumed scaled.

Adds a data point scaled.

Parameters:

y The y coordinate.

Returns:

The index of the added data point.

addValue

virtual int16_t addValue (int value)

Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).

Parameters:

value The value to add to the array.

Returns:

The index of the newly added value.

Reimplemented by: **touchgfx::DataGraphScroll::addValue**, **touchgfx::DataGraphWrapAndClear::addValue**, **touchgfx::DataGraphWrapAndOverwrite::addValue**

beforeAddValue

virtual void beforeAddValue ( )

This function is called before a new value (data point) is added.

This allows for invalidation to be calculated based on the global data counter before it is increased as a result of adding the new point.

Reimplemented by: **touchgfx::DataGraphScroll::beforeAddValue**, **touchgfx::DataGraphWrapAndClear::beforeAddValue**, **touchgfx::DataGraphWrapAndOverwrite::beforeAddValue**

getGraphRangeYMaxScaled

virtual int getGraphRangeYMaxScaled ( ) const

Gets maximum y coordinate for the graph.

Returns:

The maximum y coordinate.

NOTE

The returned value is left scaled.For internal use.

See also:

AbstractDataGraph::getGraphRangeYMaxAsInt, **AbstractDataGraph::getGraphRangeYMaxAsFloat**

Reimplements: **touchgfx::AbstractDataGraph::getGraphRangeYMaxScaled**

getGraphRangeYMinScaled

virtual int getGraphRangeYMinScaled ( ) const

Gets minimum y coordinate for the graph.

Returns:

The minimum y coordinate.

NOTE

The returned value is left scaled.For internal use.

See also:

AbstractDataGraph::getGraphRangeYMinAsInt, **AbstractDataGraph::getGraphRangeYMinAsFloat**

Reimplements: touchgfx::AbstractDataGraph::getGraphRangeYMinScaled

getXAxisOffsetScaled

virtual int getXAxisOffsetScaled ( ) const

Get x axis offset as a scaled value.

Returns:

The x axis offset (left scaled).

NOTE

For internal use.

Reimplements: touchgfx::AbstractDataGraph::getXAxisOffsetScaled

getXAxisScaleScaled

virtual int getXAxisScaleScaled ( ) const

Get x axis scale as a scaled value.

Returns:

The x axis scale (left scaled).

NOTE

For internal use.

Reimplements: touchgfx::AbstractDataGraph::getXAxisScaleScaled

indexToDataPointXScaled

virtual int indexToDataPointXScaled (int16_t index)

Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.

Parameters:

index Zero-based index of the data point.

Returns:

The data point x value scaled.

NOTE

For internal use.

See also:

indexToDataPointXAsInt, **indexToDataPointXAsFloat**

Reimplements: **touchgfx::AbstractDataGraph::indexToDataPointXScaled**

indexToDataPointYScaled

virtual int indexToDataPointYScaled (int16_t index)

Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.

Parameters:

index Zero-based index of the data point.

Returns:

The data point y value scaled.

NOTE

For internal use.

See also:

indexToDataPointYAsInt, **indexToDataPointYAsFloat**

Reimplements: **touchgfx::AbstractDataGraph::indexToDataPointYScaled**

indexToScreenXQ5

virtual CWRUtil::Q5 indexToScreenXQ5 (int16_t index)

Gets screen x coordinate for a specific data point added to the graph.

Parameters:

index The index of the element to get the x coordinate for.

Returns:

The screen x coordinate for the specific data point.

Reimplements: touchgfx::AbstractDataGraph::indexToScreenXQ5

indexToScreenYQ5

virtual CWRUtil::Q5 indexToScreenYQ5 (int16_t index)

Gets screen y coordinate for a specific data point added to the graph.

Parameters:

index The index of the element to get the y coordinate for.

Returns:

The screen x coordinate for the specific data point.

Reimplements: touchgfx::AbstractDataGraph::indexToScreenYQ5

realIndex

virtual int16_t realIndex (int16_t index)

Get the real index in the yValues array of the given index.

Normally this is just the 'i' but e.g. **DataGraphScroll** does not, for performance reasons.

Parameters:

index Zero-based index.

Returns:

The index in the yValues array.

Reimplemented by: touchgfx::DataGraphScroll::realIndex

setGraphRangeYScaled

virtual void setGraphRangeYScaled (int min ,
int max
)

Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.

Parameters:

min The minimum y coordinate.

max The maximum y coordinate.

NOTE

For internal use.

See also:

setGraphRangeY

Reimplements: touchgfx::AbstractDataGraph::setGraphRangeYScaled

setXAxisOffsetScaled

virtual void setXAxisOffsetScaled (int offset)

Set x coordinate axis offset value with a pre-scaled offset value.

This is the real x value of the first data point added to the graph (i.e. the data point at index 0).

Parameters:

offset The x axis offset.

NOTE

For internal use.

See also:

setXAxisOffset

setXAxisScaleScaled

virtual void setXAxisScaleScaled (int scale)

Set x coordinate axis scale value using a pre-scaled value.

This is the real x value increment between two data points added to the graph.

Parameters:

scale The x axis scale.

NOTE

For internal use.

See also:

setXAxisScale

valueToScreenXQ5

virtual CWRUtil::Q5 valueToScreenXQ5 (int x)

Gets screen x coordinate for an absolute value.

Parameters:

x The x value.

Returns:

The screen x coordinate for the given value.

Reimplements: touchgfx::AbstractDataGraph::valueToScreenXQ5

valueToScreenYQ5

virtual CWRUtil::Q5 valueToScreenYQ5 (int y)

Gets screen y coordinate for an absolute value.

Parameters:

y The y value.

Returns:

The screen y coordinate for the given value.

Reimplements: touchgfx::AbstractDataGraph::valueToScreenYQ5

xScreenRangeToIndexRange

virtual bool xScreenRangeToIndexRange (int16_t xLo , const
int16_t xHi , const
int16_t & indexLow , const
int16_t & indexHigh const
) const

Gets index range for screen x coordinate range taking the current graph range into account.

Parameters:

xLo	The low screen x coordinate.
xHi	The high screen x coordinate.
indexLow	The low element index.
indexHigh	The high element index.

Returns:

True if the range from low index to high index is legal.

NOTE

For internal use.

Reimplements: touchgfx::AbstractDataGraph::xScreenRangeToIndexRange

Protected Attributes Documentation

dataCounter

uint32_t dataCounter

The data counter of how many times addDataPoint() has been called.

xOffset

int xOffset

The x axis offset (real value of data point at index 0)

xScale

int xScale

The x axis scale (increment between two data points)

yValues

int * yValues

The values of the graph.

Version: 4.16

AbstractDirectionProgress

An abstract class for progress indicators that need a horizontal or vertical direction to be specified.

Inherits from: AbstractProgressIndicator, Container, Drawable

Inherited by: BoxProgress, ImageProgress

Public Types

enum	**DirectionType** { RIGHT, LEFT, DOWN, UP }
	Values that represent directions.

Public Functions

	AbstractDirectionProgress()

virtual **DirectionType**	**getDirection**() const
	Gets the current direction for the progress indicator.

virtual void	**setDirection**(**DirectionType** direction)
	Sets a direction for the progress indicator.

Protected Attributes

DirectionType	**progressDirection**
	The progress direction.

Additional inherited members

Public Functions inherited from AbstractProgressIndicator

	AbstractProgressIndicator()

	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.
virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the actual progress indicator relative to the background image.

virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.
void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()

Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.
int	**rangeMin**
	The range minimum.
uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0

	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.

int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()

Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

DirectionType

enum DirectionType

Values that represent directions.

RIGHT	Progress should be from left to right.
LEFT	Progress should be from right to left.
DOWN	Progress should be down (top to bottom)
UP	Progress should be up (bottom to top)

Public Functions Documentation

AbstractDirectionProgress

AbstractDirectionProgress ( )

getDirection

virtual DirectionType getDirection ( ) const

Gets the current direction for the progress indicator.

Returns:

The direction.

See also:

setDirection

setDirection

virtual void setDirection (DirectionType direction)

Sets a direction for the progress indicator.

This will re-calculate the current value according to the new direction.

Parameters:

direction The direction.

See also:

getDirection

Protected Attributes Documentation

progressDirection

DirectionType progressDirection

The progress direction.

Version: 4.16

AbstractGraphDecoration

Helper class used for adding labels around the graph. Currently empty.

Inherits from: AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Inherited by: GraphLabelsBase, GraphTitle

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	color
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	getScale() const
	Gets the scaling factor set using setScale.
virtual void	invalidateGraphPointAt(int16_t index) =0
	Invalidate the point at the given index.
void	setScale(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	convertToGraphScale(const AbstractDataGraph * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	getGraph() const
	Gets a pointer to the the graph containing the GraphElement.
int	getGraphRangeYMaxScaled(const AbstractDataGraph * graph) const
	Gets maximum y coordinate for the graph.
int	getGraphRangeYMinScaled(const AbstractDataGraph * graph) const
	Gets minimum y coordinate for the graph.
int	getGraphXAxisOffsetScaled(const AbstractDataGraph * graph) const
	Get x axis offset as a scaled value.
int	getGraphXAxisScaleScaled(const AbstractDataGraph * graph) const
	Get x axis scale as a scaled value.

CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const

	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

virtual void **translateRectToAbsolute**(**Rect** & r) const

	Helper function for converting a region of this **Drawable** to absolute coordinates.

virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.

Drawable *	**parent**
	Pointer to this drawable's parent.

Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.

bool	**touchable**
	True if this drawable should receive touch events.

bool	**visible**
	True if this drawable should be drawn.

Version: 4.16

AbstractGraphElement

An abstract graph element. Declares a couple of useful functions to help subclasses which use CWR (CanvasWidget Renderer).

Inherits from: CanvasWidget, Widget, Drawable

Inherited by: AbstractGraphElementNoCWR, GraphElementArea, GraphElementDiamonds, GraphElementDots, GraphElementLine

Public Functions

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
virtual void	**invalidateGraphPointAt**(int16_t index) =0
	Invalidate the point at the given index.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.

int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes

int	**dataScale**

	The scaling factor.

Additional inherited members

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0

	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractGraphElement

AbstractGraphElement ( )

getScale

int getScale ( ) const

Gets the scaling factor set using setScale.

Returns:

The scaling factor.

See also:

setScale

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplemented by: **touchgfx::GraphElementGridBase::invalidateGraphPointAt**, **touchgfx::GraphElementArea::invalidateGraphPointAt**, **touchgfx::GraphElementLine::invalidateGraphPointAt**, **touchgfx::GraphElementVerticalGapLine::invalidateGraphPointAt**, **touchgfx::GraphElementHistogram::invalidateGraphPointAt**, **touchgfx::GraphElementBoxes::invalidateGraphPointAt**, **touchgfx::GraphElementDots::invalidateGraphPointAt**, **touchgfx::GraphElementDiamonds::invalidateGraphPointAt**, **touchgfx::GraphLabelsBase::invalidateGraphPointAt**, **touchgfx::GraphLabelsX::invalidateGraphPointAt**, **touchgfx::GraphTitle::invalidateGraphPointAt**

setScale

void setScale (int scale)

Sets a scaling factor to be multiplied on each added element.

Since only integer values are stored internally, it is possible to set a scale to e.g. 1000 and make elements work as if there are three digits of precision.

By setting the scale to 1 it is possible to simply use integer values for the graph.

Parameters:

scale The scaling factor.

NOTE

Calling setScale should be done as the first thing as any new scaling factor will not be applied to already set scaled values.

See also:

getScale

Protected Functions Documentation

convertToGraphScale

```
int convertToGraphScale ( const AbstractDataGraph * graph , const
                          int                       value , const
                          int                       scale   const
                          )                         const
```

Converts a number with one scale to a number that has the same scale as the graph.

Parameters:

value The value to convert.
scale The scale.
graph The graph.

Returns:

The given data converted to the graph scale.

NOTE

For internal use.

getGraph

AbstractDataGraph * getGraph ( ) const

Gets a pointer to the the graph containing the GraphElement.

Returns:

A pointer to the graph.

getGraphRangeYMaxScaled

int getGraphRangeYMaxScaled (const AbstractDataGraph * graph)

Gets maximum y coordinate for the graph.

Parameters:

graph The graph.

Returns:

The maximum y coordinate.

NOTE

The returned value is left scaled.For internal use.

See also:

AbstractDataGraph::getGraphRangeYMaxAsInt,
AbstractDataGraph::getGraphRangeYMaxAsFloat

getGraphRangeYMinScaled

int getGraphRangeYMinScaled (const AbstractDataGraph * graph)

Gets minimum y coordinate for the graph.

Parameters:

graph The graph.

Returns:

The minimum y coordinate.

NOTE

The returned value is left scaled.For internal use.

See also:

AbstractDataGraph::getGraphRangeYMinAsInt,
AbstractDataGraph::getGraphRangeYMinAsFloat

getGraphXAxisOffsetScaled

int getGraphXAxisOffsetScaled (const AbstractDataGraph * graph)

Get x axis offset as a scaled value.

Parameters:

graph The graph.

Returns:

The x axis offset (left scaled).

NOTE

For internal use.

getGraphXAxisScaleScaled

int getGraphXAxisScaleScaled (const AbstractDataGraph * graph)

Get x axis scale as a scaled value.

Parameters:

graph The graph.

Returns:

The x axis scale (left scaled).

NOTE

For internal use.

indexToScreenXQ5

CWRUtil::Q5 indexToScreenXQ5 (const AbstractDataGraph * graph , const
int16_t index const

```
)                                   const
```

Gets screen x coordinate for a specific data point added to the graph.

Parameters:

graph The graph.
index The index of the element to get the x coordinate for.

Returns:

The screen x coordinate for the specific data point.

indexToScreenYQ5

```
CWRUtil::Q5 indexToScreenYQ5 ( const AbstractDataGraph * graph ,  const
                               int16_t                   index   const
                               )                         const
```

Gets screen y coordinate for a specific data point added to the graph.

Parameters:

graph The graph.
index The index of the element to get the y coordinate for.

Returns:

The screen x coordinate for the specific data point.

isCenterInvisible

```
bool isCenterInvisible ( const AbstractDataGraph * graph ,  const
                         int16_t                   index   const
                         )                         const
```

Query if the center of a given data point index is visible inside the graph area.

Parameters:

graph The graph.
index The data point index.

Returns:

True if center invisible, false if not.

rectAround

```
Rect rectAround ( CWRUtil::Q5 xQ5 ,          const
                  CWRUtil::Q5 yQ5 ,          const
                  CWRUtil::Q5 diameterQ5     const
                  )           const
```

Find the screen rectangle around a given point with the specified diameter.

Parameters:

xQ5	The screen x coordinate (in Q5).
yQ5	The screen y coordinate (in Q5).
diameterQ5	The diameter (in Q5).

Returns:

A **Rect** containing the point (and diameter).

rectFromQ5Coordinates

```
Rect rectFromQ5Coordinates ( CWRUtil::Q5 screenXminQ5 ,  const
                             CWRUtil::Q5 screenYminQ5 ,  const
                             CWRUtil::Q5 screenXmaxQ5 ,  const
                             CWRUtil::Q5 screenYmaxQ5    const
                             )           const
```

Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.

Parameters:

screenXminQ5	The minimum screen x coordinate (in Q5).
screenYminQ5	The maximum screen y coordinate (in Q5).
screenXmaxQ5	The minimum screen x coordinate (in Q5).
screenYmaxQ5	The maximum screen y coordinate (in Q5).

Returns:

A **Rect** containing the Q5 rectangle.

roundQ5

```
CWRUtil::Q5 roundQ5 ( CWRUtil::Q5 q5 )
```

Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.

Parameters:

q5 The **CWRUtil::Q5** value to round.

Returns:

The nearest integer as a **CWRUtil::Q5** value.

valueToScreenXQ5

CWRUtil::Q5 valueToScreenXQ5 (const AbstractDataGraph * graph , const
int x const
) const

Gets graph screen x for x value.

Parameters:

graph The graph.
x The x value.

Returns:

The graph screen x for value.

valueToScreenYQ5

CWRUtil::Q5 valueToScreenYQ5 (const AbstractDataGraph * graph , const
int y const
) const

Gets graph screen y for y value.

Parameters:

graph The graph.
y The y value.

Returns:

The graph screen y for value.

xScreenRangeToIndexRange

bool xScreenRangeToIndexRange (int16_t xLow , const
int16_t xHigh , const
int16_t & elementLow , const
int16_t & elementHigh const

) const

Gets graph element range for screen x coordinate range.

Parameters:

xLow	The low.
xHigh	The high.
elementLow	The element low.
elementHigh	The element high.

Returns:

True if it succeeds, false if it fails.

Protected Attributes Documentation

dataScale

int dataScale

The scaling factor.

Version: 4.16

AbstractGraphElementNoCWR

An abstract graph element. Declares a couple of useful functions to help subclasses which do not use CWR (CanvasWidget Renderer).

Inherits from: AbstractGraphElement, CanvasWidget, Widget, Drawable

Inherited by: AbstractGraphDecoration, GraphElementBoxes, GraphElementGridBase, GraphElementHistogram, GraphElementVerticalGapLine

Public Functions

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes

colortype	**color**

	The currently assigned color.

Additional inherited members

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
virtual void	**invalidateGraphPointAt**(int16_t index) =0
	Invalidate the point at the given index.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const

	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const

	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const

	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)

	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractGraphElementNoCWR

AbstractGraphElementNoCWR ( )

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: touchgfx::CanvasWidget::drawCanvasWidget

Reimplemented by: touchgfx::GraphTitle::drawCanvasWidget

getColor

virtual colortype getColor ( ) const

Gets the color of the graph element.

Returns:

The color.

See also:

setColor

setColor

virtual void setColor (colortype newColor)

Sets the color of the graph element.

Parameters:

newColor The new color.

See also:

getColor

Protected Functions Documentation

normalizeRect

void normalizeRect (Rect & rect)

Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.

Parameters:

rect The rectangle.

setPainter

virtual void setPainter (AbstractPainter & painter)

Protected function to prevent users from setting a painter.

Parameters:

painter The painter.

Reimplements: **touchgfx::CanvasWidget::setPainter**

Protected Attributes Documentation

color

colortype color

The currently assigned color.

Version: 4.16

AbstractPainter

An abstract class for creating painter classes for drawing canvas widgets. All canvas widgets need a painter to fill the shape drawn with a CanvasWidgetRenderer. The painter must provide the color of a pixel on a given coordinate, which will the be blended into the framebuffer depending on the position of the canvas widget and the transparency of the given pixel.

The AbstractPainter also implements a function which will blend each pixel in a scanline snippet into the framebuffer, but for better performance, the function should be reimplemented in each painter.

Inherited by: AbstractPainterABGR2222, AbstractPainterARGB2222, AbstractPainterARGB8888, AbstractPainterBGRA2222, AbstractPainterBW, AbstractPainterGRAY2, AbstractPainterGRAY4, AbstractPainterRGB565, AbstractPainterRGB888, AbstractPainterRGBA2222

Public Functions

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers) =0
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions

void	**setWidgetAlpha**(const uint8_t alpha)

	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainter

AbstractPainter ( )

Initializes a new instance of the **AbstractPainter** class.

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

render

```
virtual void render ( uint8_t *       ptr ,    =0
                      int             x ,      =0
                      int             xAdjust , =0
                      int             y ,      =0
                      unsigned        count ,  =0
                      const uint8_t * covers   =0
                      )               =0
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplemented by: **touchgfx::AbstractPainterABGR2222::render**, **touchgfx::AbstractPainterARGB2222::render**, **touchgfx::AbstractPainterARGB8888::render**, **touchgfx::AbstractPainterBGRA2222::render**, **touchgfx::AbstractPainterBW::render**, **touchgfx::AbstractPainterGRAY2::render**, **touchgfx::AbstractPainterGRAY4::render**, **touchgfx::AbstractPainterRGB565::render**, **touchgfx::AbstractPainterRGB888::render**, **touchgfx::AbstractPainterRGBA2222::render**, **touchgfx::PainterABGR2222::render**, **touchgfx::PainterABGR2222Bitmap::render**, **touchgfx::PainterARGB2222::render**, **touchgfx::PainterARGB2222Bitmap::render**, **touchgfx::PainterARGB8888::render**, **touchgfx::PainterARGB8888Bitmap::render**, **touchgfx::PainterARGB8888L8Bitmap::render**, **touchgfx::PainterBGRA2222::render**, **touchgfx::PainterBGRA2222Bitmap::render**,

touchgfx::PainterBW::render, touchgfx::PainterBWBitmap::render, touchgfx::PainterGRAY2::render, touchgfx::PainterGRAY2Bitmap::render, touchgfx::PainterGRAY4::render, touchgfx::PainterGRAY4Bitmap::render, touchgfx::PainterRGB565::render, touchgfx::PainterRGB565Bitmap::render, touchgfx::PainterRGB565L8Bitmap::render, touchgfx::PainterRGB888::render, touchgfx::PainterRGB888Bitmap::render, touchgfx::PainterRGB888L8Bitmap::render, touchgfx::PainterRGBA2222::render, touchgfx::PainterRGBA2222Bitmap::render

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setOffset

void setOffset (uint16_t offsetX ,
uint16_t offsetY
)

Sets the offset of the area being drawn.

This allows **render()** to calculate the x, y relative to the widget, and not just relative to the invalidated area.

Parameters:

offsetX The offset x coordinate of the invalidated area relative to the widget.
offsetY The offset y coordinate of the invalidated area relative to the widget.

NOTE

Used by **CanvasWidgetRenderer** - should not be overwritten.

~AbstractPainter

virtual ~AbstractPainter ( )

Finalizes an instance of the **AbstractPainter** class.

Protected Functions Documentation

setWidgetAlpha

void setWidgetAlpha (const uint8_t alpha)

Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

Parameters:

alpha The alpha.

NOTE

Used internally by **CanvasWidgetRenderer**.

compatibleFramebuffer

static FORCE_INLINE_FUNCTION bool compatibleFramebuffer (Bitmap::BitmapFormat format)

Helper function to check if the provided bitmap format matches the current framebuffer format.

Parameters:

format A bitmap format.

Returns:

True if the format matches the framebuffer format, false otherwise.

Protected Attributes Documentation

areaOffsetX

int16_t areaOffsetX

The offset x coordinate of the area being drawn.

areaOffsetY

int16_t areaOffsetY

The offset y coordinate of the area being drawn.

painterAlpha

uint8_t painterAlpha

The alpha value for the painter.

widgetAlpha

uint8_t widgetAlpha

The alpha of the widget using the painter.

Version: 4.16

AbstractPainterABGR2222

The AbstractPainterABGR2222 class is an abstract class for creating a painter to draw on a ABGR2222 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterABGR2222, PainterABGR2222Bitmap

Public Functions

	AbstractPainterABGR2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.

virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.

virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)

	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterABGR2222

AbstractPainterABGR2222 ( )

mixColors

FORCE_INLINE_FUNCTION uint8_t mixColors (uint8_t newpix ,
uint8_t bufpix ,
uint8_t alpha
)

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

newpix	The new pixel value.
bufpix	The buffer pixel value.
alpha	The alpha to apply to the new pixel.

Returns:

The result of blending the two colors into a new color.

mixColors

```
FORCE_INLINE_FUNCTION uint8_t mixColors ( uint8_t R ,
                                          uint8_t G ,
                                          uint8_t B ,
                                          uint8_t bufpix ,
                                          uint8_t alpha
                                          )
```

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

R	The red color.
G	The green color.
B	The blue color.
bufpix	The buffer pixel value.
alpha	The alpha of the R,G,B.

Returns:

The result of blending the two colors into a new color.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.

xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterABGR2222::render, touchgfx::PainterABGR2222Bitmap::render

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterABGR2222Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: **touchgfx::PainterABGR2222::renderNext**, **touchgfx::PainterABGR2222Bitmap::renderNext**

renderPixel

```
virtual void renderPixel ( uint8_t * p ,
                           uint8_t   red ,
                           uint8_t   green ,
                           uint8_t   blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterARGB2222

The AbstractPainterARGB2222 class is an abstract class for creating a painter to draw on a ARGB2222 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterARGB2222, PainterARGB2222Bitmap

Public Functions

	AbstractPainterARGB2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.

virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.

virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)

	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterARGB2222

AbstractPainterARGB2222 ( )

mixColors

FORCE_INLINE_FUNCTION uint8_t mixColors (uint8_t newpix ,
uint8_t bufpix ,
uint8_t alpha
)

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

newpix	The new pixel value.
bufpix	The buffer pixel value.
alpha	The alpha to apply to the new pixel.

Returns:

The result of blending the two colors into a new color.

mixColors

```
FORCE_INLINE_FUNCTION uint8_t mixColors ( uint8_t R ,
                                          uint8_t G ,
                                          uint8_t B ,
                                          uint8_t bufpix ,
                                          uint8_t alpha
                                          )
```

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

R	The red color.
G	The green color.
B	The blue color.
bufpix	The buffer pixel value.
alpha	The alpha of the R,G,B.

Returns:

The result of blending the two colors into a new color.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.

xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: **touchgfx::AbstractPainter::render**

Reimplemented by: **touchgfx::PainterARGB2222::render**, **touchgfx::PainterARGB2222Bitmap::render**

Protected Functions Documentation

renderInit

```
virtual bool renderInit ( )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: **touchgfx::PainterARGB2222Bitmap::renderInit**

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: **touchgfx::PainterARGB2222::renderNext**, **touchgfx::PainterARGB2222Bitmap::renderNext**

renderPixel

```
virtual void renderPixel ( uint8_t * p ,
                           uint8_t   red ,
                           uint8_t   green ,
                           uint8_t   blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterARGB8888

The AbstractPainterARGB8888 class is an abstract class for creating a painter to draw on a ARGB8888 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterARGB8888, PainterARGB8888Bitmap, PainterARGB8888L8Bitmap

Public Functions

	AbstractPainterARGB8888()

virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue, uint8_t alpha)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**

Current x coordinate relative to the widget.

int **currentY**

Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

AbstractPainter()

Initializes a new instance of the **AbstractPainter** class.

virtual uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

virtual void **setAlpha**(uint8_t newAlpha)

Sets the opacity (alpha value).

void **setOffset**(uint16_t offsetX, uint16_t offsetY)

Sets the offset of the area being drawn.

virtual **~AbstractPainter**()

Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void **setWidgetAlpha**(const uint8_t alpha)

Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

FORCE_INLINE_FUNCTION bool **compatibleFramebuffer**(**Bitmap::BitmapFormat** format)

Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t **areaOffsetX**

The offset x coordinate of the area being drawn.

int16_t **areaOffsetY**

The offset y coordinate of the area being drawn.

uint8_t **painterAlpha**

The alpha value for the painter.

uint8_t **widgetAlpha**

The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterARGB8888

```
AbstractPainterARGB8888 ( )
```

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterARGB8888::render, touchgfx::PainterARGB8888Bitmap::render, touchgfx::PainterARGB8888L8Bitmap::render

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterARGB8888Bitmap::renderInit, touchgfx::PainterARGB8888L8Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: **touchgfx::PainterARGB8888::renderNext**, **touchgfx::PainterARGB8888Bitmap::renderNext**, **touchgfx::PainterARGB8888L8Bitmap::renderNext**

renderPixel

```
virtual void renderPixel ( uint16_t * p ,
                           uint8_t    red ,
                           uint8_t    green ,
                           uint8_t    blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

NOTE

Will set the alpha value to 255 (solid)

renderPixel

```
virtual void renderPixel ( uint16_t * p ,
                           uint8_t    red ,
                           uint8_t    green ,
                           uint8_t    blue ,
                           uint8_t    alpha
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.
alpha	The alpha.

NOTE

The *alpha* value is written to the 32bit framebuffer, just like the color is.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterBGRA2222

The AbstractPainterBGRA2222 class is an abstract class for creating a painter to draw on a BGRA2222 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterBGRA2222, PainterBGRA2222Bitmap

Public Functions

	AbstractPainterBGRA2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.

virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.

virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)

	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterBGRA2222

AbstractPainterBGRA2222 ( )

mixColors

FORCE_INLINE_FUNCTION uint8_t mixColors (uint8_t newpix ,
uint8_t bufpix ,
uint8_t alpha
)

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

newpix	The new pixel value.
bufpix	The buffer pixel value.
alpha	The alpha to apply to the new pixel.

Returns:

The result of blending the two colors into a new color.

mixColors

```
FORCE_INLINE_FUNCTION uint8_t mixColors ( uint8_t R ,
                                          uint8_t G ,
                                          uint8_t B ,
                                          uint8_t bufpix ,
                                          uint8_t alpha
                                          )
```

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

R	The red color.
G	The green color.
B	The blue color.
bufpix	The buffer pixel value.
alpha	The alpha of the R,G,B.

Returns:

The result of blending the two colors into a new color.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.

xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterBGRA2222::render, touchgfx::PainterBGRA2222Bitmap::render

Protected Functions Documentation

renderInit

```
virtual bool renderInit ( )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterBGRA2222Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: **touchgfx::PainterBGRA2222::renderNext**, **touchgfx::PainterBGRA2222Bitmap::renderNext**

renderPixel

```
virtual void renderPixel ( uint8_t * p ,
                           uint8_t   red ,
                           uint8_t   green ,
                           uint8_t   blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterBW

The AbstractPainterBW class is an abstract class for creating a painter to draw on a BW display using CanvasWidgetRenderer. Pixels are either set or removed, alpha blending (and transparency) is not supported.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterBW, PainterBWBitmap

Public Functions

	AbstractPainterBW()
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & color) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint16_t	**currentX**
	Current x coordinate relative to the widget.
uint16_t	**currentY**
	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterBW

```
AbstractPainterBW ( )
```

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: **touchgfx::AbstractPainter::render**

Reimplemented by: **touchgfx::PainterBW::render**, **touchgfx::PainterBWBitmap::render**

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: **touchgfx::PainterBWBitmap::renderInit**

renderNext

virtual bool renderNext (uint8_t & color)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

color The color (0 or 1).

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: **touchgfx::PainterBW::renderNext**, **touchgfx::PainterBWBitmap::renderNext**

Protected Attributes Documentation

currentX

uint16_t currentX

Current x coordinate relative to the widget.

currentY

uint16_t currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterGRAY2

The AbstractPainterGRAY2 class is an abstract class for creating a painter to draw on a GRAY2 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterGRAY2, PainterGRAY2Bitmap

Public Functions

	AbstractPainterGRAY2()

virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.

virtual bool	**renderNext**(uint8_t & gray, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint8_t * p, uint16_t offset, uint8_t gray)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**

	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.

uint8_t **painterAlpha**

The alpha value for the painter.

uint8_t **widgetAlpha**

The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterGRAY2

```
AbstractPainterGRAY2 ( )
```

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterGRAY2::render, touchgfx::PainterGRAY2Bitmap::render

Protected Functions Documentation

renderInit

```
virtual bool renderInit (  )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterGRAY2Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & gray ,  =0
                          uint8_t & alpha  =0
                          )              =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

gray The gray color (0-3).
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: touchgfx::PainterGRAY2::renderNext, touchgfx::PainterGRAY2Bitmap::renderNext

renderPixel

```
virtual void renderPixel ( uint8_t * p ,
                           uint16_t  offset ,
```

```
                        uint8_t   gray
                        )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
offset	The offset to the pixel from the given pointer.
gray	The gray color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterGRAY4

The AbstractPainterGRAY4 class is an abstract class for creating a painter to draw on a GRAY4 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterGRAY4, PainterGRAY4Bitmap

Public Functions

	AbstractPainterGRAY4()
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & gray, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.
virtual void	**renderPixel**(uint8_t * p, uint16_t offset, uint8_t gray)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**

	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.

uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterGRAY4

AbstractPainterGRAY4 ( )

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterGRAY4::render, touchgfx::PainterGRAY4Bitmap::render

Protected Functions Documentation

renderInit

```
virtual bool renderInit (  )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterGRAY4Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & gray ,  =0
                          uint8_t & alpha  =0
                          )              =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

gray The gray color (0-15).
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: touchgfx::PainterGRAY4::renderNext, touchgfx::PainterGRAY4Bitmap::renderNext

renderPixel

```
virtual void renderPixel ( uint8_t * p ,
                           uint16_t  offset ,
```

```
                        uint8_t   gray
                        )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
offset	The offset to the pixel from the given pointer.
gray	The gray color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterRGB565

The AbstractPainterRGB565 class is an abstract class for creating a painter to draw on a RGB565 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterRGB565, PainterRGB565Bitmap, PainterRGB565L8Bitmap

Public Functions

	AbstractPainterRGB565()
FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t newpix, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t R, uint16_t G, uint16_t B, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Public Attributes

const uint16_t	**BMASK**
	Mask for blue (0000000000011111)
const uint16_t	**GMASK**
	Mask for green (0000011111100000)
const uint16_t	**RMASK**
	Mask for red (1111100000000000)

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)

	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterRGB565

AbstractPainterRGB565 ( )

mixColors

FORCE_INLINE_FUNCTION uint16_t mixColors (uint16_t newpix ,

```
                                    uint16_t bufpix ,
                                    uint8_t  alpha
                                    )
```

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

newpix	The new pixel value.
bufpix	The buffer pixel value.
alpha	The alpha to apply to the new pixel.

Returns:

The result of blending the two colors into a new color.

mixColors

```
FORCE_INLINE_FUNCTION uint16_t mixColors ( uint16_t R ,
                                           uint16_t G ,
                                           uint16_t B ,
                                           uint16_t bufpix ,
                                           uint8_t  alpha
                                           )
```

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

R	The red color (0-31 shifted into **RMASK**).
G	The green color (0-63 shifted into **GMASK**).
B	The blue color (0-31 shifted into **BMASK**).
bufpix	The buffer pixel value.
alpha	The alpha of the R,G,B.

Returns:

The result of blending the two colors into a new color.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
```

```
                int             y ,
                unsigned        count ,
                const uint8_t * covers
                )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainter::render**

Reimplemented by: **touchgfx::PainterRGB565::render**, **touchgfx::PainterRGB565Bitmap::render**, **touchgfx::PainterRGB565L8Bitmap::render**

Protected Functions Documentation

renderInit

```
virtual bool renderInit (  )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterRGB565Bitmap::renderInit, touchgfx::PainterRGB565L8Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: touchgfx::PainterRGB565::renderNext, touchgfx::PainterRGB565Bitmap::renderNext, touchgfx::PainterRGB565L8Bitmap::renderNext

renderPixel

```
virtual void renderPixel ( uint16_t * p ,
                           uint8_t    red ,
                           uint8_t    green ,
                           uint8_t    blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

Public Attributes Documentation

BMASK

const uint16_t BMASK = 0x001F

Mask for blue (0000000000011111)

GMASK

const uint16_t GMASK = 0x07E0

Mask for green (0000011111100000)

RMASK

const uint16_t RMASK = 0xF800

Mask for red (1111100000000000)

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterRGB888

The AbstractPainterRGB888 class is an abstract class for creating a painter to draw on a RGB888 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterRGB888, PainterRGB888Bitmap, PainterRGB888L8Bitmap

Public Functions

	AbstractPainterRGB888()

virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**

	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.

uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterRGB888

AbstractPainterRGB888 ( )

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterRGB888::render, touchgfx::PainterRGB888Bitmap::render, touchgfx::PainterRGB888L8Bitmap::render

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterRGB888Bitmap::renderInit, touchgfx::PainterRGB888L8Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: touchgfx::PainterRGB888::renderNext, touchgfx::PainterRGB888Bitmap::renderNext,

touchgfx::PainterRGB888L8Bitmap::renderNext

renderPixel

```
virtual void renderPixel ( uint16_t * p ,
                           uint8_t    red ,
                           uint8_t    green ,
                           uint8_t    blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPainterRGBA2222

The AbstractPainterRGBA2222 class is an abstract class for creating a painter to draw on a RGBA2222 display using CanvasWidgetRenderer.

See: AbstractPainter

Inherits from: AbstractPainter

Inherited by: PainterRGBA2222, PainterRGBA2222Bitmap

Public Functions

	AbstractPainterRGBA2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha) =0
	Get the color of the next pixel in the scan line to blend into the framebuffer.

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Additional inherited members

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.

virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.

virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)

	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

AbstractPainterRGBA2222

AbstractPainterRGBA2222 ( )

mixColors

FORCE_INLINE_FUNCTION uint8_t mixColors (uint8_t newpix ,
uint8_t bufpix ,
uint8_t alpha
)

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

newpix	The new pixel value.
bufpix	The buffer pixel value.
alpha	The alpha to apply to the new pixel.

Returns:

The result of blending the two colors into a new color.

mixColors

```
FORCE_INLINE_FUNCTION uint8_t mixColors ( uint8_t R ,
                                          uint8_t G ,
                                          uint8_t B ,
                                          uint8_t bufpix ,
                                          uint8_t alpha
                                          )
```

Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Parameters:

R	The red color.
G	The green color.
B	The blue color.
bufpix	The buffer pixel value.
alpha	The alpha of the R,G,B.

Returns:

The result of blending the two colors into a new color.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.

xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: touchgfx::AbstractPainter::render

Reimplemented by: touchgfx::PainterRGBA2222::render, touchgfx::PainterRGBA2222Bitmap::render

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplemented by: touchgfx::PainterRGBA2222Bitmap::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,     =0
                          uint8_t & green ,   =0
                          uint8_t & blue ,    =0
                          uint8_t & alpha     =0
                          )                   =0
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplemented by: **touchgfx::PainterRGBA2222::renderNext**, **touchgfx::PainterRGBA2222Bitmap::renderNext**

renderPixel

```
virtual void renderPixel ( uint8_t * p ,
                           uint8_t   red ,
                           uint8_t   green ,
                           uint8_t   blue
                           )
```

Renders (writes) the specified color into the framebuffer.

Parameters:

p	pointer into the framebuffer where the given color should be written.
red	The red color.
green	The green color.
blue	The blue color.

Protected Attributes Documentation

currentX

int currentX

Current x coordinate relative to the widget.

currentY

int currentY

Current y coordinate relative to the widget.

Version: 4.16

AbstractPartition

This type defines an abstract interface to a storage partition for allocating memory slots of equal size. The "partition" is not aware of the actual types stored in the partition memory, hence it provides no mechanism for deleting C++ objects when clear()'ed.

Inherited by: Partition< ListOfTypes, NUMBER_OF_ELEMENTS >

Public Functions

template \<typename T \> void *	**allocate**()
	Gets the address of the next available storage slot.
virtual void *	**allocate**(uint16_t size)
	Gets the address of the next available storage slot.
template \<typename T \> void *	**allocateAt**(uint16_t index)
	Gets the address of the specified storage slot.
virtual void *	**allocateAt**(uint16_t index, uint16_t size)
	Gets the address of the specified index.
template \<typename T \> T &	**at**(const uint16_t index)
	Gets the object at the specified index.
template \<typename T \> const T &	**at**(const uint16_t index) const
	const version of **at()**.
virtual uint16_t	**capacity**() const =0
	Gets the capacity, i.e.
virtual void	**clear**()
	Prepares the Partition for new allocations.
void	**dec**()
	Decreases number of allocations.

virtual uint32_t	**element_size**() =0
	Access to concrete element-size.
template \<class T \> **Pair**< T *, uint16_t >	**find**(const void * pT)
	Determines if the specified object could have been previously allocated in the partition.
virtual uint16_t	**getAllocationCount**() const
	Gets allocation count.
virtual uint16_t	**indexOf**(const void * address)
	Determines index of previously allocated location.
virtual	**~AbstractPartition**()
	Finalizes an instance of the **AbstractPartition** class.

Protected Functions

	AbstractPartition()
	Initializes a new instance of the **AbstractPartition** class.
virtual const void *	**element**(uint16_t index) const =0
	Access to stored element, const version.
virtual void *	**element**(uint16_t index) =0
	Access to stored element.

Public Functions Documentation

allocate

void * allocate ( )

Gets the address of the next available storage slot.

The slot size is determined from the size of type T.

Template Parameters:

T Generic type parameter.

Returns:

The address of an empty storage slot.

NOTE

Asserts if T is too large, or the storage is depleted.

allocate

virtual void * allocate (uint16_t size)

Gets the address of the next available storage slot.

The slot size is compared with the specified size.

Parameters:

size The size.

Returns:

The address of an empty storage slot which contains minimum 'size' bytes.

NOTE

Asserts if 'size' is too large, or the storage is depleted.

allocateAt

void * allocateAt (uint16_t index)

Gets the address of the specified storage slot.

The slot size is determined from the size of type T.

Template Parameters:

T Generic type parameter.

Parameters:

index Zero-based index of the.

Returns:

The address of the appropriate storage slot.

NOTE

Asserts if T is too large.

allocateAt

```
virtual void * allocateAt ( uint16_t index ,
                            uint16_t size
                            )
```

Gets the address of the specified index.

Parameters:

index Zero-based index of the.
size The size.

Returns:

The address of the appropriate storage slot which contains minimum 'size' bytes.

NOTE

Asserts if 'size' is too large.

at

```
T & at ( const uint16_t index )
```

Gets the object at the specified index.

Template Parameters:

T Generic type parameter.

Parameters:

index The index into the **Partition** storage where the returned object is located.

Returns:

A typed reference to the object at the specified index.

at

```
const T & at ( const uint16_t index )
```

const version of **at()**.

Template Parameters:

T Generic type parameter.

Parameters:

index Zero-based index of the.

Returns:

A T&

capacity

virtual uint16_t capacity ( ) const =0

Gets the capacity, i.e.

the maximum allocation count.

Returns:

The maximum allocation count.

Reimplemented by: **touchgfx::Partition::capacity**

clear

virtual void clear ( )

Prepares the Partition for new allocations.

Any objects present in the **Partition** shall not be used after invoking this method.

dec

void dec ( )

Decreases number of allocations.

element_size

virtual uint32_t element_size ( ) =0

Access to concrete element-size.

Used internally.

Returns:

An uint32_t.

Reimplemented by: **touchgfx::Partition::element_size**

find

Pair< T *, uint16_t > find (const void * pT)

Determines if the specified object could have been previously allocated in the partition.

Since the **Partition** concept is loosely typed this method shall be used with care. The method does not guarantee that the found object at the returned index is a valid object. It only tests whether or not the object is within the bounds of the current partition allocations.

Template Parameters:

T Generic type parameter.

Parameters:

pT Pointer to the object to look up.

Returns:

If the object seems to be allocated in the **Partition**, a **Pair** object containing a typed pointer to the object and an index into the **Partition** storage is returned. Otherwise, a Pair<0, 0> is returned.

getAllocationCount

virtual uint16_t getAllocationCount ( ) const

Gets allocation count.

Returns:

The currently allocated storage slots.

indexOf

virtual uint16_t indexOf (const void * address)

Determines index of previously allocated location.

Since the **Partition** concept is loosely typed this method shall be used with care. The method does not guarantee that the found object at the returned index is a valid object. It only tests whether or not the object is within the bounds of the current partition allocations.

Parameters:

address The location address to lookup.

Returns:

An uint16_t.

~AbstractPartition

virtual ~AbstractPartition ( )

Finalizes an instance of the **AbstractPartition** class.

Protected Functions Documentation

AbstractPartition

AbstractPartition ( )

Initializes a new instance of the **AbstractPartition** class.

element

virtual const void * element (uint16_t index)

Access to stored element, const version.

Parameters:

index Zero-based index of the.

Returns:

null if it fails, else a void*.

Reimplemented by: **touchgfx::Partition::element**

element

virtual void * element (uint16_t index)

Access to stored element.

Used internally.

Parameters:

index Zero-based index of the.

Returns:

null if it fails, else a void*.

Reimplemented by: **touchgfx::Partition::element**

Version: 4.16

AbstractProgressIndicator

The AbstractProgressIndicator declares methods that provides the basic mechanisms and tools to implement a progress indicator. For more specific implementations see classes that inherit from AbstractProgressIndicator.

See: BoxProgress, CircleProgress, ImageProgress, LineProgress, TextProgress

Inherits from: Container, Drawable

Inherited by: AbstractDirectionProgress, CircleProgress, Gauge, LineProgress, TextProgress

Public Functions

	AbstractProgressIndicator()
	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.
virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().

virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the actual progress indicator relative to the background image.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.
void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes

int	**animationDuration**

	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.
int	**rangeMin**
	The range minimum.
uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const

	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AbstractProgressIndicator

AbstractProgressIndicator ( )

Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.

getProgress

virtual uint16_t getProgress (uint16_t range =100)

Gets the current progress based on the range set by setRange() and the value set by setValue().

Parameters:

range (Optional) The range, default is 100.

Returns:

The progress.

See also:

setRange, **setValue**, **getValue**

getProgressIndicatorHeight

virtual int16_t getProgressIndicatorHeight ( ) const

Gets progress indicator height.

Returns:

The progress indicator height.

See also:

setProgressIndicatorPosition

getProgressIndicatorWidth

virtual int16_t getProgressIndicatorWidth ( ) const

Gets progress indicator width.

Returns:

The progress indicator width.

See also:

setProgressIndicatorPosition

getProgressIndicatorX

virtual int16_t getProgressIndicatorX ( ) const

Gets progress indicator x coordinate.

Returns:

The progress indicator x coordinate.

See also:

setProgressIndicatorPosition

getProgressIndicatorY

virtual int16_t getProgressIndicatorY ( ) const

Gets progress indicator y coordinate.

Returns:

The progress indicator y coordinate.

See also:

setProgressIndicatorPosition

getRange

virtual void getRange (int & min , const
int & max const
) const

Gets the range set by setRange().

Parameters:

min The minimum input value.
max The maximum input value.

See also:

setRange

getRange

```
virtual void getRange ( int &       min ,  const
                        int &       max ,  const
                        uint16_t &  steps  const
                        )           const
```

Gets the range set by setRange().

Parameters:

min	The minimum input value.
max	The maximum input value.
steps	The steps in which to report progress.

See also:

setRange

getRange

```
virtual void getRange ( int &       min ,     const
                        int &       max ,     const
                        uint16_t &  steps ,   const
                        uint16_t &  minStep   const
                        )           const
```

Gets the range set by setRange().

Parameters:

min	The minimum input value.
max	The maximum input value.
steps	The steps in which to report progress.
minStep	The step which the minimum input value is mapped to.

See also:

setRange

getValue

virtual int getValue ( ) const

Gets the current value set by setValue().

Returns:

The value.

See also:

setValue

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the Drawable instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

setBackground

virtual void setBackground (const Bitmap & bitmapBackground)

Sets the background image.

The width and height of the progress indicator widget is updated according to the dimensions of the bitmap.

Parameters:

bitmapBackground The background bitmap.

setEasingEquation

virtual void setEasingEquation (EasingEquation easingEquation)

Sets easing equation to be used in updateValue.

Parameters:

easingEquation The easing equation.

See also:

updateValue

setProgressIndicatorPosition

```
virtual void setProgressIndicatorPosition ( int16_t x ,
                                            int16_t y ,
                                            int16_t width ,
                                            int16_t height
                                            )
```

Sets the position and dimensions of the actual progress indicator relative to the background image.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the box progress indicator.
height	The height of the box progress indicator.

See also:

getProgressIndicatorX, getProgressIndicatorY, getProgressIndicatorWidth, getProgressIndicatorHeight

Reimplemented by: touchgfx::BoxProgress::setProgressIndicatorPosition, touchgfx::CircleProgress::setProgressIndicatorPosition, touchgfx::ImageProgress::setProgressIndicatorPosition, touchgfx::LineProgress::setProgressIndicatorPosition, touchgfx::TextProgress::setProgressIndicatorPosition, touchgfx::Gauge::setProgressIndicatorPosition

setRange

```
virtual void setRange ( int      min ,
                        int      max ,
                        uint16_t steps =0,
                        uint16_t minStep =0
                        )
```

Sets the range for the progress indicator.

The range is the values that are given to the progress indicator while progressing through the task at hand. If an app needs to work through 237 items to finish a task, the range should be set to (0,

237) assuming that 0 items is the minimum. Though the minimum is often 0, it is possible to customize this.

The steps parameter is used to specify at what granularity you want the progress indicator to report a new progress value. If the 237 items to be reported as 0%, 10%, 20%, ... 100%, the steps should be set to 10 as there are ten steps from 0% to 100%. If you want to update a widget which is 150 pixels wide, you might want to set steps to 150 to get a new progress value for every pixel. If you are updating a clock and want this to resemble an analog clock, you might want to use 12 or perhaps 60 as number of steps.

The minStep parameter is used when the minimum input value (min) should give a progress different from 0. For example, if progress is a clock face, you want to count from 0..1000 and you want progress per minute, but want to make sure that 0 is not a blank clock face, but instead you want 1 minute to show, use

```
setRange(0, 1000, 60, 1)
```

to make sure that as values progress from 0 to 1000, **getProgress()** start from 1 and goes up to 60. Another example could be a **BoxProgress** with a **TextProgress** on top and you want to make sure that "0%" will always show in the box, use something like

```
setRange(0, 1000, 200, 40)
```

if your box is 200 pixels wide and "0%" is 40 pixels wide.

Parameters:

min	The minimum input value.
max	The maximum input value.
steps	(Optional) The steps in which to report progress.
minStep	(Optional) The step which the minimum input value is mapped to.

See also:

setValue, **getProgress**

setValue

virtual void setValue (int value)

Sets the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. If a callback function has been set using setValueSetAction, that callback will be called (unless the new value is

the same as the current value).

Parameters:

value The value.

NOTE

if value is equal to the current value, nothing happens, and the callback will not be called.

See also:

getValue, **updateValue**, **setValueSetAction**

Reimplemented by: **touchgfx::BoxProgress::setValue**, **touchgfx::CircleProgress::setValue**, **touchgfx::ImageProgress::setValue**, **touchgfx::LineProgress::setValue**, **touchgfx::TextProgress::setValue**, **touchgfx::Gauge::setValue**

setValueSetAction

void setValueSetAction (GenericCallback< const AbstractProgressIndicator & > & callback)

Sets callback that will be triggered every time a new value is assigned to the progress indicator.

This can happen directly from **setValue()** or during a gradual change initiated using **updateValue()**.

Parameters:

callback The callback.

See also:

setValue, **updateValue**

setValueUpdatedAction

void setValueUpdatedAction (GenericCallback< const AbstractProgressIndicator & > & callback)

Sets callback that will be triggered when updateValue has finished animating to the final value.

Parameters:

callback The callback.

See also:

updateValue, **setValueSetAction**

updateValue

```
virtual void updateValue ( int      value ,
                           uint16_t duration
                           )
```

Update the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. The value is changed gradually in the given number of ticks using the easing equation set in setEasingEquation. Function **setValue()** is called for every new value during the change of value, and if a callback function has been set using setValueSetAction, that callback will be called for every new value. The callback set using setValueUpdatedCallback is called when the animation has finished.

Parameters:

value	The value.
duration	The duration.

NOTE

If duration is 0, setValue will be called immediately and the valueUpdated action is called immediately.

See also:

setValue, **setEasingEquation**, **setValueSetAction**, **setValueUpdatedAction**

Protected Attributes Documentation

animationDuration

int animationDuration

Duration of the animation.

animationEndValue

int animationEndValue

The animation end value.

animationStartValue

int animationStartValue

The animation start value.

animationStep

int animationStep

The current animation step.

background

Image background

The background image.

currentValue

int currentValue

The current value.

equation

EasingEquation equation

The equation used in updateValue()

progressIndicatorContainer

Container progressIndicatorContainer

The container that holds the actual progress indicator.

rangeMax

int rangeMax

The range maximum.

rangeMin

int rangeMin

The range minimum.

rangeSteps

uint16_t rangeSteps

The range steps.

rangeStepsMin

uint16_t rangeStepsMin

The range steps minimum.

valueSetCallback

GenericCallback< const AbstractProgressIndicator & > * valueSetCallback

New value assigned Callback.

valueUpdatedCallback

GenericCallback< const AbstractProgressIndicator & > * valueUpdatedCallback

Animation ended Callback.

Version: 4.16

AbstractShape

Simple widget capable of drawing a abstractShape. The abstractShape can be scaled and rotated around 0,0. The shapes points (corners) are calculated with regards to scaling and rotation to allow for faster redrawing. Care must be taken to call updateAbstractShapeCache() after updating the shape, the scale of the shape or the rotation of the shape.

Inherits from: CanvasWidget, Widget, Drawable

Inherited by: Shape< POINTS >

Public Classes

struct	**ShapePoint**
	Defines an alias for a single point.

Public Functions

	AbstractShape()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
int	**getAngle**() const
	Gets the current angle of the abstractShape.
template \<typename T \> void	**getAngle**(T & angle)
	Gets the abstractShape's angle.
virtual **CWRUtil::Q5**	**getCornerX**(int i) const =0
	Gets the x coordinate of a corner (a point) of the shape.
virtual **CWRUtil::Q5**	**getCornerY**(int i) const =0
	Gets the y coordinate of a corner (a point) of the shape.
virtual int	**getNumPoints**() const =0

	Gets number of points used to make up the shape.
template \<typename T \> void	**getOrigin**(T & dx, T & dy) const
	Gets the position of the shapes (0,0).
template \<typename T \> void	**getScale**(T & x, T & y) const
	Gets the x scale and y scale of the shape as previously set using setScale.
template \<typename T \> void	**moveOrigin**(T x, T y)
	Sets the position of the shape's (0,0) in the widget.
template \<typename T \> void	**setAngle**(T angle)
	Sets the absolute angle to turn the **AbstractShape**.
virtual void	**setCorner**(int i, **CWRUtil::Q5** x, **CWRUtil::Q5** y) =0
	Sets one of the points (a corner) of the shape in CWRUtil::Q5 format.
template \<typename T \> void	**setOrigin**(T x, T y)
	Sets the position of the shape's (0,0) in the widget.
template \<typename T \> void	**setScale**(T newXScale, T newYScale)
	Scale the **AbstractShape** the given amounts in the x direction and the y direction.
template \<typename T \> void	**setScale**(T scale)
	Scale the **AbstractShape** the given amount in the x direction and the y direction.
template \<typename T \> void	**setShape**(const **ShapePoint**< T > * points)
	Sets a shape the struct Points.
template \<typename T \> void	**setShape**(**ShapePoint**< T > * points)
	Sets a shape the struct Points.
void	**updateAbstractShapeCache**()
	Updates the **AbstractShape** cache.

template \<typename T \> void	**updateAngle**(T angle)
	Sets the absolute angle to turn the **AbstractShape**.
template \<typename T \> void	**updateScale**(T newXScale, T newYScale)
	Scale the **AbstractShape** the given amount in the x direction and the y direction.

Protected Functions

virtual **CWRUtil::Q5**	**getCacheX**(int i) const =0
	Gets cached x coordinate of a point/corner.
virtual **CWRUtil::Q5**	**getCacheY**(int i) const =0
	Gets cached y coordinate of a point/corner.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual void	**setCache**(int i, **CWRUtil::Q5** x, **CWRUtil::Q5** y) =0
	Sets the cached coordinates of a given point/corner.

Additional inherited members

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const

	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.

int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.

virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**

	True if this drawable should receive touch events.
bool **visible**	
	True if this drawable should be drawn.

Public Functions Documentation

AbstractShape

AbstractShape ( )

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::CanvasWidget::drawCanvasWidget**

getAngle

int getAngle ( ) const

Gets the current angle of the abstractShape.

Returns:

The angle of the AbstractShaperounded down to the precision of int.

getAngle

void getAngle (T & angle)

Gets the abstractShape's angle.

Template Parameters:

T Generic type parameter.

Parameters:

angle The current **AbstractShape** rotation angle rounded down to the precision of T.

getCornerX

virtual CWRUtil::Q5 getCornerX (int i)

Gets the x coordinate of a corner (a point) of the shape.

Parameters:

i Zero-based index of the corner.

Returns:

The corner x coordinate in **CWRUtil::Q5** format.

Reimplemented by: **touchgfx::Shape::getCornerX**

getCornerY

virtual CWRUtil::Q5 getCornerY (int i)

Gets the y coordinate of a corner (a point) of the shape.

Parameters:

i Zero-based index of the corner.

Returns:

The corner y coordinate in **CWRUtil::Q5** format.

Reimplemented by: **touchgfx::Shape::getCornerY**

getNumPoints

virtual int getNumPoints ( ) const =0

Gets number of points used to make up the shape.

Returns:

The number of points.

Reimplemented by: **touchgfx::Shape::getNumPoints**

getOrigin

void getOrigin (T & dx , const
T & dy const
) const

Gets the position of the shapes (0,0).

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

dx The x coordinate rounded down to the precision of T.
dy The y coordinate rounded down to the precision of T.

getScale

void getScale (T & x , const
T & y const
) const

Gets the x scale and y scale of the shape as previously set using setScale.

Default is 1 for both x scale and y scale.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x Scaling of x coordinates rounded down to the precision of T.
y Scaling of y coordinates rounded down to the precision of T.

See also:

setScale

moveOrigin

```
void moveOrigin ( T x ,
                  T y
                  )
```

Sets the position of the shape's (0,0) in the widget.

This means that all coordinates initially used when created the shape are moved relative to these given offsets. Subsequent calls to **moveOrigin()** or **setOrigin()** will replace the old values for origin. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The absolute x coordinate of the shapes position (0,0).
y The absolute y coordinate of the shapes position (0,0).

NOTE

The area containing the AbstractShape is invalidated before and after the change.

See also:

setOrigin

setAngle

```
void setAngle ( T angle )
```

Sets the absolute angle to turn the **AbstractShape**.

0 degrees means no rotation and 90 degrees is rotate the shape clockwise. Repeated calls to setAngle(10) will therefore not rotate the shape by an additional 10 degrees. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter.

Parameters:

angle The absolute angle to turn the abstractShape to relative to 0 (straight up).

NOTE

The area containing the AbstractShape is not invalidated.

See also:

updateAngle

setCorner

```
virtual void setCorner ( int              i ,   =0
                         CWRUtil::Q5      x ,   =0
                         CWRUtil::Q5      y     =0
                         )                =0
```

Sets one of the points (a corner) of the shape in CWRUtil::Q5 format.

Parameters:

i Zero-based index of the corner.
x The x coordinate in **CWRUtil::Q5** format.
y The y coordinate in **CWRUtil::Q5** format.

NOTE

Remember to call updateAbstractShapeCache() after calling setCorner one or more times, to make sure that the cached outline of the shape is correct.

See also:

updateAbstractShapeCache

Reimplemented by: **touchgfx::Shape::setCorner**

setOrigin

```
void setOrigin ( T x ,
                 T y
                 )
```

Sets the position of the shape's (0,0) in the widget.

This means that all coordinates initially used when created the shape are moved relative to these given offsets. Subsequent calls to **setOrigin()** or **moveOrigin()** will replace the old values for origin. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The absolute x coordinate of the shapes position (0,0).
y The absolute y coordinate of the shapes position (0,0).

NOTE

The area containing the **AbstractShape** is not invalidated.

See also:

moveOrigin

setScale

```
void setScale ( T newXScale ,
                T newYScale
                )
```

Scale the **AbstractShape** the given amounts in the x direction and the y direction.

The new scaling factors do not multiply to previously set scaling factors, so scaling by 2 and later scaling by 2 again will not scale by 4, only by 2. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

newXScale The new scale in the x direction.
newYScale The new scale in the y direction.

NOTE

The area containing the **AbstractShape** is not invalidated.

See also:

getScale, **updateScale**

setScale

```
void setScale ( T scale )
```

Scale the **AbstractShape** the given amount in the x direction and the y direction.

The new scaling factors do not multiply to previously set scaling factors, so scaling by 2 and later scaling by 2 again will not scale by 4, only by 2. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

scale The scale in the x direction.

NOTE

The area containing the AbstractShape is not invalidated.

See also:

getScale

setShape

void setShape (const ShapePoint< T > * points)

Sets a shape the struct Points.

The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

points The points that make up the shape. The pointer must point to an array of **getNumPoints()** elements of type **ShapePoint**.

NOTE

The area containing the shape is not invalidated.

setShape

void setShape (ShapePoint< T > * points)

Sets a shape the struct Points.

The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

points The points that make up the shape. The pointer must point to an array of **getNumPoints()** elements of type **ShapePoint**.

NOTE

The area containing the shape is not invalidated.

updateAbstractShapeCache

void updateAbstractShapeCache ( )

Updates the **AbstractShape** cache.

The cache is used to be able to quickly redraw the **AbstractShape** without calculating the points that make up the abstractShape (with regards to scaling and rotation).

updateAngle

void updateAngle (T angle)

Sets the absolute angle to turn the **AbstractShape**.

0 degrees means no rotation and 90 degrees is rotate the shape clockwise. Repeated calls to setAngle(10) will therefore not rotate the shape by an additional 10 degrees. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter.

Parameters:

angle The angle to turn the abstractShape.

NOTE

The area containing the AbstractShape is invalidated before and after the change.

See also:

setAngle

updateScale

void updateScale (T newXScale ,
T newYScale
)

Scale the **AbstractShape** the given amount in the x direction and the y direction.

The new scaling factors do not multiply to previously set scaling factors, so scaling by 2 and later scaling by 2 again will not scale by 4, only by 2. The cached outline of the shape is automatically updated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

newXScale The new scale in the x direction.
newYScale The new scale in the y direction.

NOTE

The area containing the AbstractShape is invalidated before and after the change.

See also:

setScale

Protected Functions Documentation

getCacheX

virtual CWRUtil::Q5 getCacheX (int i)

Gets cached x coordinate of a point/corner.

Parameters:

i Zero-based index of the point/corner.

Returns:

The cached x coordinate, or zero if nothing is cached for the given i.

Reimplemented by: **touchgfx::Shape::getCacheX**

getCacheY

virtual CWRUtil::Q5 getCacheY (int i)

Gets cached y coordinate of a point/corner.

Parameters:

i Zero-based index of the point/corner.

Returns:

The cached y coordinate, or zero if nothing is cached for the given i.

Reimplemented by: **touchgfx::Shape::getCacheY**

getMinimalRect

virtual Rect getMinimalRect ( ) const

Gets minimal rectangle containing the shape drawn by this widget.

Default implementation returns the size of the entire widget, but this function should be overwritten in subclasses and return the minimal rectangle containing the shape. See classes such as **Circle** for example implementations.

Returns:

The minimal rectangle containing the shape drawn.

Reimplements: **touchgfx::CanvasWidget::getMinimalRect**

setCache

```
virtual void setCache ( int              i ,   =0
                        CWRUtil::Q5      x ,   =0
                        CWRUtil::Q5      y     =0
                        )                =0
```

Sets the cached coordinates of a given point/corner.

The coordinates in the cache are the coordinates from the corners after rotation and scaling has been applied to the coordinate.

Parameters:

i Zero-based index of the corner.

x The x coordinate.

y The y coordinate.

Reimplemented by: **touchgfx::Shape::setCache**

Version: 4.16

AnalogClock

An analog clock. Should be supplied with images for the background, hour hand, minute hand and the optional second hand. You setup the AnalogClock by specifying the rotation point of each hand as well as the global rotation point of the clock. You can customize the behavior of the AnalogClock in respect to animations and relations between the hands e.g. if the hour hand should move gradually towards the next hour as the minute hand progresses (setHourHandMinuteCorrection())

Inherits from: AbstractClock, Container, Drawable

Public Functions

	AnalogClock()
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual uint16_t	**getAnimationDuration**()
	Gets the animation duration.
virtual bool	**getHourHandMinuteCorrection**() const
	Gets hour hand minute correction.
virtual bool	**getMinuteHandSecondCorrection**() const
	Gets minute hand second correction.
virtual void	**initializeTime12Hour**(uint8_t hour, uint8_t minute, uint8_t second, bool am)
	Sets the time with input format as 12H.
virtual void	**initializeTime24Hour**(uint8_t hour, uint8_t minute, uint8_t second)
	Sets the time with input format as 24H.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setAnimation**(uint16_t duration =10, **EasingEquation** animationProgressionEquation =**EasingEquations::backEaseInOut**)
	Setup the clock to use animation for hand movements.
virtual void	**setBackground**(const **BitmapId** backgroundBitmapId)

	Sets the background image of the clock.
virtual void	**setBackground**(const **BitmapId** backgroundBitmapId, int16_t rotationCenterX, int16_t rotationCenterY)
	Sets the background image of the clock and the rotation center of the clock.
virtual void	**setHourHandMinuteCorrection**(bool active)
	Sets whether hour hand minute correction should be active.
virtual void	**setMinuteHandSecondCorrection**(bool active)
	Sets whether minute hand second correction should be active.
virtual void	**setRotationCenter**(int16_t rotationCenterX, int16_t rotationCenterY)
	Sets the rotation center of the clock.
virtual void	**setupHourHand**(const **BitmapId** hourHandBitmapId, int16_t rotationCenterX, int16_t rotationCenterY)
	Sets up the hour hand.
virtual void	**setupMinuteHand**(const **BitmapId** minuteHandBitmapId, int16_t rotationCenterX, int16_t rotationCenterY)
	Sets up the minute hand.
virtual void	**setupSecondHand**(const **BitmapId** secondHandBitmapId, int16_t rotationCenterX, int16_t rotationCenterY)
	Sets up the second hand.

Protected Functions

virtual bool	**animationEnabled**() const
	Is animation enabled for the hands?
virtual float	**convertHandValueToAngle**(uint8_t steps, uint8_t handValue, uint8_t secondHandValue =0) const
	Convert hand value to angle.
virtual void	**setupHand**(**TextureMapper** & hand, const **BitmapId** bitmapId, int16_t rotationCenterX, int16_t rotationCenterY)
	Sets up a given the hand.
virtual void	**updateClock**()
	Update the visual representation of the clock on the display.

Protected Attributes

uint16_t	**animationDuration**
	The duration of hand animations. If 0 animations are disabled.
EasingEquation	**animationEquation**
	The easing equation used by hand animations.
Image	**background**
	The background image of the **AnalogClock**.
int16_t	**clockRotationCenterX**
	The x coordinate of the rotation point of the hands.
int16_t	**clockRotationCenterY**
	The y coordinate of the rotation point of the hands.
AnimationTextureMapper	**hourHand**
	TextureMapper used for drawing the hourHand.
bool	**hourHandMinuteCorrectionActive**
	Is hour hand minute correction active.
uint8_t	**lastHour**
	The last know hour value.
uint8_t	**lastMinute**
	The last know minute value.
uint8_t	**lastSecond**
	The last know second value.
AnimationTextureMapper	**minuteHand**
	TextureMapper used for drawing the minuteHand.
bool	**minuteHandSecondCorrectionActive**
	Is minute hand second correction active.
AnimationTextureMapper	**secondHand**
	TextureMapper used for drawing the secondHand.

Additional inherited members

Public Functions inherited from AbstractClock

	AbstractClock()
bool	**getCurrentAM**() const
	Is the current time a.m.
uint8_t	**getCurrentHour**() const
	Gets the current hour.
uint8_t	**getCurrentHour12**() const
	Gets current hour 12, i.e.
uint8_t	**getCurrentHour24**() const
	Gets current hour 24, i.e.
uint8_t	**getCurrentMinute**() const
	Gets the current minute.
uint8_t	**getCurrentSecond**() const
	Gets the current second.
virtual void	**setTime12Hour**(uint8_t hour, uint8_t minute, uint8_t second, bool am)
	Sets the time with input format as 12H.
virtual void	**setTime24Hour**(uint8_t hour, uint8_t minute, uint8_t second)
	Sets the time with input format as 24H.

Protected Attributes inherited from AbstractClock

uint8_t	**currentHour**
	Local copy of the current hour.
uint8_t	**currentMinute**
	Local copy of the current minute.
uint8_t	**currentSecond**
	Local copy of the current second.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const

	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()

	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const

	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AnalogClock

AnalogClock ( )

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getAnimationDuration

virtual uint16_t getAnimationDuration ( )

Gets the animation duration.

Returns:

The animation duration.

See also:

setAnimation

getHourHandMinuteCorrection

virtual bool getHourHandMinuteCorrection ( ) const

Gets hour hand minute correction.

Returns:

true if hour hand minute correction is active.

See also:

setHourHandMinuteCorrection

getMinuteHandSecondCorrection

```
virtual bool getMinuteHandSecondCorrection ( ) const
```

Gets minute hand second correction.

Returns:

true if minute hand second correction is active.

See also:

setHourHandMinuteCorrection

initializeTime12Hour

```
virtual void initializeTime12Hour ( uint8_t hour ,
                                    uint8_t minute ,
                                    uint8_t second ,
                                    bool    am
                                    )
```

Sets the time with input format as 12H.

No animations are performed regardless of the animation settings. This is often useful when setting up the **AnalogClock** where you do not want an initial animation.

Parameters:

hour	The hours, value should be between 1 and 12.
minute	The minutes, value should be between 0 and 59.
second	The seconds, value should be between 0 and 59.
am	AM/PM setting. True = AM, false = PM.

NOTE

that this does not affect any selected presentation formats.

See also:

setTime12Hour

initializeTime24Hour

```
virtual void initializeTime24Hour ( uint8_t hour ,
                                    uint8_t minute ,
                                    uint8_t second
                                    )
```

Sets the time with input format as 24H.

No animations are performed regardless of the animation settings. This is often useful when setting up the **AnalogClock** where you do not want an initial animation.

Parameters:

hour	The hours, value should be between 0 and 23.
minute	The minutes, value should be between 0 and 59.
second	The seconds, value should be between 0 and 59.

NOTE

that this does not affect any selected presentation formats.

See also:

setTime24Hour

setAlpha

```
virtual void setAlpha ( uint8_t newAlpha )
```

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha	The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display. The alpha value is reflected in the background image

See also:

getAlpha

setAnimation

```
virtual void setAnimation ( uint16_t          duration =10,
                            EasingEquation    animationProgressionEquation
                                              =EasingEquations::backEaseInOut
                          )
```

Setup the clock to use animation for hand movements.

Parameters:

duration	(Optional) The animation duration, default is 10.
animationProgressionEquation	(Optional) The animation progression equation, default is **EasingEquations::backEaseInOut**.

setBackground

```
virtual void setBackground ( const BitmapId backgroundBitmapId )
```

Sets the background image of the clock.

The clock rotation center is automatically set to the background image center. The clock rotation center is the point that the clock hands rotates around. The size of the AnalocClock widget is set to the size of the bitmap.

Parameters:

backgroundBitmapId	Identifier for the background bitmap.

setBackground

```
virtual void setBackground ( const BitmapId backgroundBitmapId ,
                             int16_t        rotationCenterX ,
                             int16_t        rotationCenterY
                           )
```

Sets the background image of the clock and the rotation center of the clock.

The clock rotation center is the point that the clock hands rotates around. The size of the AnalocClock widget is set to the size of the bitmap.

Parameters:

backgroundBitmapId	Identifier for the background bitmap.
rotationCenterX	The rotation center x coordinate.
rotationCenterY	The rotation center y coordinate.

See also:

setBackground(BitmapId), **setRotationCenter**

setHourHandMinuteCorrection

virtual void setHourHandMinuteCorrection (bool active)

Sets whether hour hand minute correction should be active.

If set to true the hour hand will be positioned between the current hour and the next depending on the minute hands position.

Parameters:

active true to use hour hand correction.

See also:

getHourHandMinuteCorrection

setMinuteHandSecondCorrection

virtual void setMinuteHandSecondCorrection (bool active)

Sets whether minute hand second correction should be active.

If set to true the minute hand will be positioned between the current minute and the next depending on the second hands position.

Parameters:

active true to use.

See also:

setMinuteHandSecondCorrection

setRotationCenter

virtual void setRotationCenter (int16_t rotationCenterX ,
int16_t rotationCenterY
)

Sets the rotation center of the clock.

The clock rotation center is the point that the clock hands rotates around.

Parameters:

rotationCenterX	The rotation center x coordinate.
rotationCenterY	The rotation center y coordinate.

setupHourHand

```
virtual void setupHourHand ( const BitmapId hourHandBitmapId ,
                             int16_t         rotationCenterX ,
                             int16_t         rotationCenterY
                             )
```

Sets up the hour hand.

The specified rotation center is the point of the hand that is to be placed on top of the clock rotation center. That is the point that the hand rotates around. The rotation point is relative to the supplied bitmap and can be placed outside of it.

Parameters:

hourHandBitmapId	Identifier for the hour hand bitmap.
rotationCenterX	The hand rotation center x coordinate.
rotationCenterY	The hand rotation center y coordinate.

NOTE

If no hour hand is setup it will just be omitted.

setupMinuteHand

```
virtual void setupMinuteHand ( const BitmapId minuteHandBitmapId ,
                               int16_t         rotationCenterX ,
                               int16_t         rotationCenterY
                               )
```

Sets up the minute hand.

The specified rotation center is the point of the hand that is to be placed on top of the clock rotation center. That is the point that the hand rotates around. The rotation point is relative to the supplied bitmap but can be placed outside of it.

Parameters:

minuteHandBitmapId	Identifier for the minute hand bitmap.
rotationCenterX	The hand rotation center x coordinate.

rotationCenterY	The hand rotation center y coordinate.

NOTE

If no minute hand is setup it will just be omitted.

setupSecondHand

```
virtual void setupSecondHand ( const BitmapId secondHandBitmapId ,
                               int16_t        rotationCenterX ,
                               int16_t        rotationCenterY
                               )
```

Sets up the second hand.

The specified rotation center is the point of the hand that is to be placed on top of the clock rotation center. That is the point that the hand rotates around. The rotation point is relative to the supplied bitmap but can be placed outside of it.

Parameters:

secondHandBitmapId	Identifier for the second hand bitmap.
rotationCenterX	The hand rotation center x coordinate.
rotationCenterY	The hand rotation center y coordinate.

NOTE

If no second hand is setup it will just be omitted.

Protected Functions Documentation

animationEnabled

```
virtual bool animationEnabled (  ) const
```

Is animation enabled for the hands?

Returns:

true if animation is enabled.

convertHandValueToAngle

```
virtual float convertHandValueToAngle ( uint8_t steps ,                const
                                        uint8_t handValue ,            const
                                        uint8_t secondHandValue =0     const
                                        )       const
```

Convert hand value to angle.

Parameters:

steps	Number of steps the primary hand value is divided into, i.e. 60 for minutes/seconds and 12 for hour.
handValue	The actual value for the hand in question (in the range [0; steps]).
secondHandValue	(Optional) If the angle should be corrected for a secondary hand its value should be specified here (in the range [0; 60]). This is the case when setHourHandMinuteCorrection(true) or setMinuteHandSecondCorrection(true) is selected.

Returns:

The converted value to angle.

setupHand

```
virtual void setupHand ( TextureMapper & hand ,
                         const BitmapId    bitmapId ,
                         int16_t           rotationCenterX ,
                         int16_t           rotationCenterY
                         )
```

Sets up a given the hand.

Parameters:

hand	Reference to the hand being setup.
bitmapId	The bitmap identifier for the given hand.
rotationCenterX	The hand rotation center x coordinate.
rotationCenterY	The hand rotation center y coordinate.

updateClock

```
virtual void updateClock ( )
```

Update the visual representation of the clock on the display.

Reimplements: **touchgfx::AbstractClock::updateClock**

Protected Attributes Documentation

animationDuration

uint16_t animationDuration

The duration of hand animations. If 0 animations are disabled.

animationEquation

EasingEquation animationEquation

The easing equation used by hand animations.

background

Image background

The background image of the **AnalogClock**.

clockRotationCenterX

int16_t clockRotationCenterX

The x coordinate of the rotation point of the hands.

clockRotationCenterY

int16_t clockRotationCenterY

The y coordinate of the rotation point of the hands.

hourHand

AnimationTextureMapper hourHand

TextureMapper used for drawing the hourHand.

hourHandMinuteCorrectionActive

bool hourHandMinuteCorrectionActive

Is hour hand minute correction active.

lastHour

uint8_t lastHour

The last know hour value.

lastMinute

uint8_t lastMinute

The last know minute value.

lastSecond

uint8_t lastSecond

The last know second value.

minuteHand

AnimationTextureMapper minuteHand

TextureMapper used for drawing the minuteHand.

minuteHandSecondCorrectionActive

bool minuteHandSecondCorrectionActive

Is minute hand second correction active.

secondHand

AnimationTextureMapper secondHand

TextureMapper used for drawing the secondHand.

Version: 4.16

AnimatedImage

A widget capable of basic animation using a range of bitmaps. The AnimatedImage is capable of running the animation from start to end or, in reverse order, end to start. It is capable of doing a single animation or looping the animation until stopped or paused.

Inherits from: Image, Widget, Drawable

Public Functions

	AnimatedImage(const **BitmapId** & start, const **BitmapId** & end, const uint8_t & updateInterval =1)
	Constructs an **AnimatedImage**.
	AnimatedImage(const uint8_t & updateInterval =1)
	Constructs an **AnimatedImage** without initializing bitmaps.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
bool	**isAnimatedImageRunning**() const
	Gets the running state of the **AnimatedImage**.
bool	**isReverse**()
	Query if this object is running in reverse.
virtual void	**pauseAnimation**()
	Toggles the running state of an animation.
virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this Image and updates the width and height of this widget to match those of the Bitmap.
virtual void	**setBitmapEnd**(const **Bitmap** & bitmap)
	Sets the end bitmap for this **AnimatedImage** sequence.
void	**setBitmaps**(**BitmapId** start, **BitmapId** end)
	Sets the bitmaps that are used by the animation.

void	**setDoneAction**(**GenericCallback**< const **AnimatedImage** & > & callback)
	Associates an action to be performed when the animation of the **AnimatedImage** is done.
void	**setUpdateTicksInterval**(uint8_t updateInterval)
	Sets the update interval.
virtual void	**startAnimation**(const bool rev, const bool reset =false, const bool loop =false)
	Starts the animation with the given parameters for animation direction, normal or reverse, whether to restart the animation and finally if the animation should loop automatically upon completion.
virtual void	**stopAnimation**()
	Stops and resets the animation.

Protected Attributes

GenericCallback< const **AnimatedImage** & > *	**animationDoneAction**
	Pointer to the callback to be executed when animation is done.
BitmapId	**endId**
	Id of last bitmap in animation.
bool	**loopAnimation**
	If true, continuously loop animation.
bool	**reverse**
	If true, run in reverse direction (last to first).
bool	**running**
	If true, animation is running.
BitmapId	**startId**
	Id of first bitmap in animation.
uint8_t	**ticksSinceUpdate**
	Number of ticks since last animation update.
uint8_t	**updateTicksInterval**
	Number of ticks between each animation update (image change).

Additional inherited members

Public Functions inherited from Image

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getBitmap**() const
	Gets the **Bitmap** currently assigned to the **Image** widget.
BitmapId	**getBitmapId**() const
	Gets the BitmapId currently assigned to the **Image** widget.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
	Image(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

Protected Attributes inherited from Image

uint8_t	**alpha**
	The Alpha for this image.
Bitmap	**bitmap**
	The **Bitmap** to display.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

AnimatedImage

```
AnimatedImage ( const BitmapId & start ,
                const BitmapId & end ,
                const uint8_t &  updateInterval =1
                )
```

Constructs an **AnimatedImage**.

The start and the end specifies the range of bitmaps to be used for animation. The update interval defines how often the animation should be updated. The animation will iterate over the bitmaps that lies between the IDs of start and end, both included.

Parameters:

start	Defines the start of the range of images in the animation.
end	Defines the end of the range of images in the animation.
updateInterval	(Optional) Defines the number of ticks between each animation step. Higher value results in a slower animation. Default is to update the image on every tick.

AnimatedImage

AnimatedImage (const uint8_t & updateInterval =1)

Constructs an **AnimatedImage** without initializing bitmaps.

Parameters:

updateInterval	(Optional) Defines the number of ticks between each animation step. Higher value results in a slower animation.

NOTE

The bitmaps to display must be configured through set setBitmaps function before this widget displays anything.

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the Drawable instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

isAnimatedImageRunning

bool isAnimatedImageRunning ( ) const

Gets the running state of the **AnimatedImage**.

Returns:

true if the animation is currently running, false otherwise.

isReverse

bool isReverse ( )

Query if this object is running in reverse.

Returns:

true if the animation is performed in reverse order.

pauseAnimation

virtual void pauseAnimation ( )

Toggles the running state of an animation.

Pauses the animation if the animation is running. Continues the animation if previously paused.

See also:

stopAnimation

setBitmap

virtual void setBitmap (const Bitmap & bitmap)

Sets the bitmap for this Image and updates the width and height of this widget to match those of the Bitmap.

Parameters:

bitmap The bitmap instance.

NOTE

The user code must call invalidate() in order to update the image on the display.This only sets the start image.

See also:

setBitmaps, setEndBitmap

Reimplements: **touchgfx::Image::setBitmap**

setBitmapEnd

virtual void setBitmapEnd (const Bitmap & bitmap)

Sets the end bitmap for this **AnimatedImage** sequence.

Parameters:

bitmap The bitmap.

See also:

setBitmaps, **setBitmap**

setBitmaps

void setBitmaps (BitmapId start ,
BitmapId end
)

Sets the bitmaps that are used by the animation.

The animation will iterate over the bitmaps that lies between the IDs of start and end, both inclusive.

Parameters:

start Defines the start of the range of images in the animation.
end Defines the end of the range of images in the animation.

See also:

setBitmap, SetBitmapEnd

setDoneAction

void setDoneAction (GenericCallback< const AnimatedImage & > & callback)

Associates an action to be performed when the animation of the **AnimatedImage** is done.

If the animation is set to loop at the end, the action is also triggered when the animation starts over.

Parameters:

callback The callback is executed when done. The callback is given the animated image.

setUpdateTicksInterval

void setUpdateTicksInterval (uint8_t updateInterval)

Sets the update interval.

The value specifies the number of ticks between each step of the animation. The default update interval for animated images is 1, which means results in the fastest possible animation.

Parameters:

updateInterval	Defines the number of ticks between each animation step. Higher value results in a slower animation.

startAnimation

```
virtual void startAnimation ( const bool rev ,
                              const bool reset =false,
                              const bool loop =false
                              )
```

Starts the animation with the given parameters for animation direction, normal or reverse, whether to restart the animation and finally if the animation should loop automatically upon completion.

Parameters:

rev	Defines if the animation should be performed in reverse order.
reset	(Optional) Defines if the animation should reset and start from the first (or last if reverse order) bitmap.
loop	(Optional) Defines if the animation should loop or do a single animation.

stopAnimation

virtual void stopAnimation ( )

Stops and resets the animation.

If the animation should not reset to the first image in the animation sequence, use **pauseAnimation()**.

See also:

startAnimation, **pauseAnimation**

Protected Attributes Documentation

animationDoneAction

GenericCallback< const AnimatedImage & > * animationDoneAction

Pointer to the callback to be executed when animation is done.

endId

BitmapId endId

Id of last bitmap in animation.

loopAnimation

bool loopAnimation

If true, continuously loop animation.

reverse

bool reverse

If true, run in reverse direction (last to first).

running

bool running

If true, animation is running.

startId

BitmapId startId

Id of first bitmap in animation.

ticksSinceUpdate

uint8_t ticksSinceUpdate

Number of ticks since last animation update.

updateTicksInterval

uint8_t updateTicksInterval

Number of ticks between each animation update (image change).

Version: 4.16

AnimatedImageButtonStyle

An animated image button style. An animated image button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show the first or last image of an animated image depending on the state of the button (pressed or released). When the state changes the button will show the sequence of images in forward or reversed order.

The AnimatedImageButtonStyle will set the size of the enclosing container (normally AbstractButtonContainer) to the size of the first Bitmap. This can be overridden by calling setWidth/setHeight after setting the bitmaps.

The position of the bitmap can be adjusted with setBitmapXY (default is upper left corner).

Template Parameters:

- **T** Generic type parameter. Typically a AbstractButtonContainer subclass.

See: AbstractButtonContainer

Inherits from: T

Public Functions

	AnimatedImageButtonStyle()
void	**setBitmaps**(const **Bitmap** & bitmapStart, const **Bitmap** & bitmapEnd) Sets the bitmaps.
void	**setBitmapXY**(uint16_t x, uint16_t y) Sets bitmap x and y.
void	**setUpdateTicksInterval**(uint8_t updateInterval) Sets update ticks interval.

Protected Functions

virtual void	**handleAlphaUpdated**() Handles what should happen when the alpha is updated.

virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

AnimatedImage	**buttonAnimatedImage**
	The button animated image.

Public Functions Documentation

AnimatedImageButtonStyle

```
AnimatedImageButtonStyle ( )
```

setBitmaps

```
void setBitmaps ( const Bitmap & bitmapStart ,
                  const Bitmap & bitmapEnd
                  )
```

Sets the bitmaps.

Parameters:

bitmapStart The bitmap start.
bitmapEnd The bitmap end.

setBitmapXY

```
void setBitmapXY ( uint16_t x ,
                   uint16_t y
                   )
```

Sets bitmap x and y.

Parameters:

x An uint16_t to process.
y An uint16_t to process.

setUpdateTicksInterval

void setUpdateTicksInterval (uint8_t updateInterval)

Sets update ticks interval.

Parameters:

updateInterval The update interval.

Protected Functions Documentation

handleAlphaUpdated

virtual void handleAlphaUpdated ( )

Handles what should happen when the alpha is updated.

handlePressedUpdated

virtual void handlePressedUpdated ( )

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

buttonAnimatedImage

AnimatedImage buttonAnimatedImage

The button animated image.

Version: 4.16

AnimationSetting

Information about how a specific animation parameter should be animated.

Public Attributes

bool	**animationActive**
	Should this animation be performed?
uint16_t	**animationDelay**
	A delay before the actual animation start. Expressed in ticks.
uint16_t	**animationDuration**
	The complete duration of the animation. Expressed in ticks.
float	**animationEnd**
	The animation end value.
EasingEquation	**animationProgressionEquation**
	EasingEquation expressing the development of the value during the animation.
float	**animationStart**
	The animation start value.

Public Attributes Documentation

animationActive

bool animationActive

Should this animation be performed?

animationDelay

uint16_t animationDelay

A delay before the actual animation start. Expressed in ticks.

animationDuration

uint16_t animationDuration

The complete duration of the animation. Expressed in ticks.

animationEnd

float animationEnd

The animation end value.

animationProgressionEquation

EasingEquation animationProgressionEquation

EasingEquation expressing the development of the value during the animation.

animationStart

float animationStart

The animation start value.

Version: 4.16

AnimationTextureMapper

A TextureMapper with animation capabilities. Note that the angles of the TextureMapper is normalized to lie in the range [0; 2PI[at the beginning at the animation. The end angles should be relative to this and are limited to values in the range [-32.7; 32.7].

Inherits from: TextureMapper, Image, Widget, Drawable

Protected Classes

struct	**AnimationSetting**
	Information about how a specific animation parameter should be animated.

Public Types

enum	**AnimationParameter** { X_ROTATION, Y_ROTATION, Z_ROTATION, SCALE }
	Values that represent different animation parameter.

Public Functions

	AnimationTextureMapper()
virtual void	**cancelAnimationTextureMapperAnimation**()
	Cancel move animation.
virtual uint16_t	**getAnimationStep**()
	Gets the current animation step measured in ticks since the call to startAnimation().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual bool	**isTextureMapperAnimationRunning**() const
	Gets whether or not the animation is running.
void	**setTextureMapperAnimationEndedAction**(**GenericCallback**< const **AnimationTextureMapper** & > & callback)

	Associates an action to be performed when the animation ends.
void	**setTextureMapperAnimationStepAction**(**GenericCallback**< const **AnimationTextureMapper** & > & callback)
	Associates an action to be performed for every step in the animation.
virtual void	**setupAnimation**(**AnimationParameter** parameter, float endValue, uint16_t duration, uint16_t delay, **EasingEquation** progressionEquation =&**EasingEquations::linearEaseNone**)
	Sets up the animation for a specific parameter (angle/scale) for the next animation.
virtual void	**startAnimation**()
	Starts the animation from the current position to the specified end angles/scale, as specified by one or more calls to **setupAnimation()**.

Public Attributes

const int	**NUMBER_OF_ANIMATION_PARAMETERS**
	Number of animation parameters.

Protected Attributes

uint16_t	**animationCounter**
	Counter that is equal to the current step in the animation.
bool	**animationRunning**
	Boolean that is true if the animation is running.
AnimationSetting	**animations**
	Descriptions of the animation of specific animation parameters.
GenericCallback< const **AnimationTextureMapper** & > *	**textureMapperAnimationEndedCallback**
	Callback that is executed after the animation ends.
GenericCallback< const **AnimationTextureMapper** & > *	**textureMapperAnimationStepCallback**
	Callback that is executed after every step of the animation.

Additional inherited members

Public Types inherited from TextureMapper

enum	**RenderingAlgorithm** { NEAREST_NEIGHBOR, BILINEAR_INTERPOLATION }
	Rendering algorithm to use when scaling the bitmap.

Public Functions inherited from TextureMapper

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual float	**getBitmapPositionX**() const
	Gets bitmap position x coordinate.
virtual float	**getBitmapPositionY**() const
	Gets bitmap position y coordinate.
virtual float	**getCameraDistance**() const
	Gets camera distance.
virtual float	**getCameraX**() const
	Gets camera x coordinate.
virtual float	**getCameraY**() const
	Gets camera y coordinate.
virtual float	**getOrigoX**() const
	Gets transformation origo x coordinate.
virtual float	**getOrigoY**() const
	Gets transformation origo y coordinate.
virtual float	**getOrigoZ**() const
	Gets transformation origo z coordinate.
virtual **RenderingAlgorithm**	**getRenderingAlgorithm**() const
	Gets the algorithm used when rendering.
virtual float	**getScale**() const
	Gets the scale of the image.

virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual float	**getX0**() const
	Get the x coordinate of the top left corner of the transformed bitmap.
virtual float	**getX1**() const
	Get the x coordinate of the top right corner of the transformed bitmap.
virtual float	**getX2**() const
	Get the x coordinate of the bottom right of the transformed bitmap.
virtual float	**getX3**() const
	Get the x coordinate of the bottom left corner of the transformed bitmap.
virtual float	**getXAngle**() const
	Get the x angle.
virtual float	**getY0**() const
	Get the y coordinate of the top left corner of the transformed bitmap.
virtual float	**getY1**() const
	Get the y coordinate of the top right corner of the transformed bitmap.
virtual float	**getY2**() const
	Get the y coordinate of the bottom right corner of the transformed bitmap.
virtual float	**getY3**() const
	Get the y coordinate of the bottom left corner of the transformed bitmap.
virtual float	**getYAngle**() const
	Get the y angle.
virtual float	**getZ0**() const
	Get the z coordinate of the top left corner of the transformed bitmap.
virtual float	**getZ1**() const
	Get the z coordinate of the top right corner of the transformed bitmap.
virtual float	**getZ2**() const

	Get the z coordinate of the bottom right corner of the transformed bitmap.
virtual float	**getZ3**() const
	Get the z coordinate of the bottom left corner of the transformed bitmap.
virtual float	**getZAngle**() const
	Get the z angle.
void	**invalidateBoundingRect**() const
	Invalidate the bounding rectangle of the transformed bitmap.
virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this **TextureMapper** and updates the width and height of this widget to match those of the **Bitmap**.
virtual void	**setBitmapPosition**(float x, float y)
	Sets the position of the bitmap within the **TextureMapper**.
virtual void	**setBitmapPosition**(int x, int y)
	Sets the position of the bitmap within the **TextureMapper**.
virtual void	**setCamera**(float x, float y)
	Sets the camera coordinate.
virtual void	**setCameraDistance**(float d)
	Sets camera distance.
virtual void	**setOrigo**(float x, float y)
	Sets the transformation origo (center) in two dimensions.
virtual void	**setOrigo**(float x, float y, float z)
	Sets the transformation origo (center).
virtual void	**setRenderingAlgorithm**(**RenderingAlgorithm** algorithm)
	Sets the render algorithm to be used.
virtual void	**setScale**(float scale)
	Sets the scale of the image.
	TextureMapper(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **TextureMapper** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

virtual void	**updateAngles**(float newXAngle, float newYAngle, float newZAngle)
	Updates the angles of the image.
virtual void	**updateXAngle**(float newXAngle)
	Updates the x angle.
virtual void	**updateYAngle**(float newYAngle)
	Updates the y angle.
virtual void	**updateZAngle**(float newZAngle)
	Updates the z angle.

Protected Functions inherited from TextureMapper

void	**applyTransformation**()
	Transform the bitmap using the supplied origo, scale, rotation and camera.
void	**drawTriangle**(const **Rect** & invalidatedArea, uint16_t *fb, const float* triangleXs, const float *triangleYs, const float* triangleZs, const float *triangleUs, const float* triangleVs) const
	The **TextureMapper** will draw the transformed bitmap by drawing two triangles.
Rect	**getBoundingRect**() const
	Gets bounding rectangle of the transformed bitmap.
RenderingVariant	**lookupRenderVariant**() const
	Returns the rendering variant based on the bitmap format, alpha value and rendering algorithm.

Protected Attributes inherited from TextureMapper

float	**cameraDistance**
	The camera distance.
RenderingAlgorithm	**currentRenderingAlgorithm**
	The current rendering algorithm.
float	**imageX0**
	The coordinate for the image points.
float	**imageX1**
	The coordinate for the image points.

float	**imageX2**
	The coordinate for the image points.
float	**imageX3**
	The coordinate for the image points.
float	**imageY0**
	The coordinate for the image points.
float	**imageY1**
	The coordinate for the image points.
float	**imageY2**
	The coordinate for the image points.
float	**imageY3**
	The coordinate for the image points.
float	**imageZ0**
	The coordinate for the image points.
float	**imageZ1**
	The coordinate for the image points.
float	**imageZ2**
	The coordinate for the image points.
float	**imageZ3**
	The coordinate for the image points.
float	**scale**
	The scale.
uint16_t	**subDivisionSize**
	The size of the affine sub divisions.
float	**xAngle**
	The angle x.
float	**xBitmapPosition**
	The bitmap position x.
float	**xCamera**
	The camera x coordinate.

float	**xOrigo**
	The origo x coordinate.
float	**yAngle**
	The angle y.
float	**yBitmapPosition**
	The bitmap position y.
float	**yCamera**
	The camera y coordinate.
float	**yOrigo**
	The origo y coordinate.
float	**zAngle**
	The angle z.
float	**zOrigo**
	The origo z coordinate.
const int	**MINIMAL_CAMERA_DISTANCE**
	The minimal camera distance.

Public Functions inherited from Image

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getBitmap**() const
	Gets the **Bitmap** currently assigned to the **Image** widget.
BitmapId	**getBitmapId**() const
	Gets the BitmapId currently assigned to the **Image** widget.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
	Image(const **Bitmap** & bitmap =**Bitmap**())

	Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this **Image** and updates the width and height of this widget to match those of the **Bitmap**.

Protected Attributes inherited from Image

uint8_t	**alpha**
	The Alpha for this image.
Bitmap	**bitmap**
	The **Bitmap** to display.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)

	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.

bool **visible**	
	True if this drawable should be drawn.

Public Types Documentation

AnimationParameter

enum AnimationParameter

Values that represent different animation parameter.

X_ROTATION	Rotation around the X axis.
Y_ROTATION	Rotation around the Y axis.
Z_ROTATION	Rotation around the Z axis.
SCALE	Scaling of the image.

Public Functions Documentation

AnimationTextureMapper

AnimationTextureMapper ( )

cancelAnimationTextureMapperAnimation

virtual void cancelAnimationTextureMapperAnimation ( )

Cancel move animation.

Stops any running animation at the current position regardless of the progress made so far. Disables all animation parameters set using setupAnimation and mark the animation as stopped.

getAnimationStep

virtual uint16_t getAnimationStep ( )

Gets the current animation step measured in ticks since the call to startAnimation().

The steps during the initial delay are also counted.

Returns:

The current animation step.

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the Drawable instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

isTextureMapperAnimationRunning

virtual bool isTextureMapperAnimationRunning ( ) const

Gets whether or not the animation is running.

Returns:

true if the animation is running.

setTextureMapperAnimationEndedAction

void setTextureMapperAnimationEndedAction (GenericCallback< const AnimationTextureMapper & > & callback)

Associates an action to be performed when the animation ends.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **AnimationTextureMapper**.

See also:

GenericCallback

setTextureMapperAnimationStepAction

void setTextureMapperAnimationStepAction (GenericCallback< const AnimationTextureMapper & > & callback)

Associates an action to be performed for every step in the animation.

Will not be called during the delay period.

Parameters:

callback	The callback to be executed. The callback will be given a reference to the **AnimationTextureMapper**.

See also:

GenericCallback

setupAnimation

```
virtual void setupAnimation ( AnimationParameter parameter ,
                              float              endValue ,
                              uint16_t           duration ,
                              uint16_t           delay ,
                              EasingEquation     progressionEquation
                                                 =&EasingEquations::linearEaseNone
                              )
```

Sets up the animation for a specific parameter (angle/scale) for the next animation.

The specific parameter is chosen using the AnimationType enum. AnimationTypes that are not setup using this method will keep their value during the animation.

Parameters:

parameter	The parameter of the **TextureMapper** that should be animated.
endValue	The end value for the parameter.
duration	The duration for the animation of the parameter. Specified in ticks.
delay	The delay before the animation of the parameter starts. Specified in ticks.
progressionEquation	(Optional) the progression equation for the animation of this parameter. Default is **EasingEquations::linearEaseNone**.

startAnimation

```
virtual void startAnimation ( )
```

Starts the animation from the current position to the specified end angles/scale, as specified by one or more calls to **setupAnimation()**.

Public Attributes Documentation

NUMBER_OF_ANIMATION_PARAMETERS

const int NUMBER_OF_ANIMATION_PARAMETERS = SCALE + 1

Number of animation parameters.

Protected Attributes Documentation

animationCounter

uint16_t animationCounter

Counter that is equal to the current step in the animation.

animationRunning

bool animationRunning

Boolean that is true if the animation is running.

animations

AnimationSetting animations

Descriptions of the animation of specific animation parameters.

textureMapperAnimationEndedCallback

GenericCallback< const AnimationTextureMapper & > * textureMapperAnimationEndedCallback

Callback that is executed after the animation ends.

textureMapperAnimationStepCallback

GenericCallback< const AnimationTextureMapper & > * textureMapperAnimationStepCallback

Callback that is executed after every step of the animation.

Version: 4.16

Application

The Application class is the main interface for manipulating screen contents. It holds a pointer to the currently displayed Screen, and delegates draw requests and events to that Screen. Additionally it contains some global application settings.

Inherits from: UIEventListener

Inherited by: MVPApplication

Protected Types

typedef **Vector**< **Rect**, 8 >	**RectVector_t**
	Type to ensure the same number of rects are in the Vector.

Public Functions

DebugPrinter *	**getDebugPrinter**()
	Returns the DebugPrinter object associated with the application.
Application *	**getInstance**()
	Gets the single instance application.
void	**invalidateDebugRegion**()
	Sets the debug string to be displayed onscreen on top of the framebuffer.
void	**setDebugPrinter**(**DebugPrinter** * printer)
	Sets the DebugPrinter object to be used by the application to print debug messages.
void	**setDebugString**(const char * string)
	Sets the debug string to be displayed onscreen on top of the framebuffer.
virtual void	**appSwitchScreen**(uint8_t screenId)
	An application specific function for switching screen.
virtual void	**cacheDrawOperations**(bool enableCache)
	This function allows for deferring draw operations to a later time.

void	**clearAllTimerWidgets**()
	Clears all currently registered timer widgets.
void	**copyInvalidatedAreasFromTFTToClientBuffer**()
	This function copies the parts that were updated in the previous frame (in the tft buffer) to the active framebuffer (client buffer).
virtual void	**draw**()
	Initiate a draw operation of the entire screen.
virtual void	**draw**(**Rect** & rect)
	Initiate a draw operation of the specified region of the screen.
Screen *	**getCurrentScreen**()
	Gets the current screen.
uint16_t	**getNumberOfRegisteredTimerWidgets**() const
	gets the number of timer widgets that has been registered.
uint16_t	**getTimerWidgetCountForDrawable**(const **Drawable** * w) const
	Gets the number of timer events registered to a widget, i.e.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Handle a click event.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Handle drag events.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Handle gestures.
virtual void	**handleKeyEvent**(uint8_t c)
	Handle an incoming character received by the HAL layer.
virtual void	**handlePendingScreenTransition**()
	Evaluates the pending Callback instances.
virtual void	**handleTickEvent**()
	Handle tick.
void	**registerTimerWidget**(**Drawable** * w)
	Adds a widget to the list of widgets receiving ticks every frame (typically 16.67ms)

virtual void	**requestRedraw**()
	An application specific function for requesting redraw of entire screen.
virtual void	**requestRedraw**(**Rect** & rect)
	An application specific function for requesting redraw of given Rect.
virtual void	**switchScreen**(**Screen** * newScreen)
	Switch to another Screen.
void	**unregisterTimerWidget**(const **Drawable** * w)
	Removes a widget from the list of widgets receiving ticks every frame (typically 16.67ms) milliseconds.

Protected Functions

	Application()
	Protected constructor.
void	**invalidateArea**(**Rect** area)
	Invalidates this area.

Public Attributes

const uint8_t	**MAX_TIMER_WIDGETS**
	Maximum number of widgets receiving ticks.
const uint16_t	**TICK_INTERVAL_MS**
	Deprecated, do not use this constant. Tick interval depends on VSYNC of your target platform.

Protected Attributes

RectVector_t	**cachedDirtyAreas**
	When draw caching is enabled, these rects keeps track of the dirty screen area.
bool	**drawCacheEnabled**

	True when draw caching is active.
RectVector_t	**lastRects**
	The dirty areas from last frame that needs to be redrawn because we have swapped frame buffers.
Rect	**redraw**
	Rect describing application requested invalidate area.
uint8_t	**timerWidgetCounter**
	A counter for each potentially registered timer widget. Increase when registering for timer events, decrease when unregistering.
Vector< Drawable *, MAX_TIMER_WIDGETS >	**timerWidgets**
	List of widgets that receive timer ticks.
bool	**transitionHandled**
	True if the transition is done and Screen::afterTransition has been called.
Screen *	**currentScreen**
	Pointer to currently displayed Screen.
Transition *	**currentTransition**
	Pointer to current transition.
DebugPrinter *	**debugPrinter**
	Pointer to the DebugPrinter instance.
Rect	**debugRegionInvalidRect**
	Invalidated Debug Region.
Application *	**instance**
	Pointer to the instance of the Application-derived subclass.

Additional inherited members

Public Functions inherited from UIEventListener

virtual	**~UIEventListener**()
	Finalizes an instance of the **UIEventListener** class.

Protected Types Documentation

RectVector_t

typedef Vector< Rect, 8 > RectVector_t

Type to ensure the same number of rects are in the Vector.

Public Functions Documentation

getDebugPrinter

static DebugPrinter * getDebugPrinter ( )

Returns the DebugPrinter object associated with the application.

Returns:

DebugPrinter The **DebugPrinter** object.

getInstance

static Application * getInstance ( )

Gets the single instance application.

Returns:

The instance of this application.

invalidateDebugRegion

static void invalidateDebugRegion ( )

Sets the debug string to be displayed onscreen on top of the framebuffer.

setDebugPrinter

static void setDebugPrinter (DebugPrinter * printer)

Sets the DebugPrinter object to be used by the application to print debug messages.

Parameters:

printer The debug printer to configure.

setDebugString

static void setDebugString (const char * string)

Sets the debug string to be displayed onscreen on top of the framebuffer.

Parameters:

string The debug string to display onscreen.

appSwitchScreen

virtual void appSwitchScreen (uint8_t screenId)

An application specific function for switching screen.

Overloading this can provide a means to switch screen from places that does not have access to a pointer to the new screen. Base implementation is empty.

Parameters:

screenId An id that maps to the desired screen.

cacheDrawOperations

virtual void cacheDrawOperations (bool enableCache)

This function allows for deferring draw operations to a later time.

If active, calls to draw will simply note that the specified area is dirty, but not perform any actual drawing. When disabling the draw cache, the dirty area will be flushed (drawn) immediately.

Parameters:

enableCache if true, all future draw operations will be cached. If false draw caching is disabled, and the current cache (if not empty) is drawn immediately.

clearAllTimerWidgets

void clearAllTimerWidgets ( )

Clears all currently registered timer widgets.

copyInvalidatedAreasFromTFTToClientBuffer

void copyInvalidatedAreasFromTFTToClientBuffer ( )

This function copies the parts that were updated in the previous frame (in the tft buffer) to the active framebuffer (client buffer).

This function only copies pixels in double buffering mode.

draw

virtual void draw ( )

Initiate a draw operation of the entire screen.

Standard implementation is to delegate draw request to the current **Screen**.

DEPRECATED

Use **draw(Rect&)**

draw

virtual void draw (Rect & rect)

Initiate a draw operation of the specified region of the screen.

Standard implementation is to delegate draw request to the current **Screen**.

Parameters:

rect The area to draw.

NOTE

Unlike Widget::draw this is safe to call from user code as it will properly traverse widgets in z-order. The coordinates given must be absolute coordinates.

getCurrentScreen

Screen * getCurrentScreen ( )

Gets the current screen.

Returns:

The current screen.

getNumberOfRegisteredTimerWidgets

uint16_t getNumberOfRegisteredTimerWidgets ( ) const

gets the number of timer widgets that has been registered.

Returns:

The size of timerWidgets.

getTimerWidgetCountForDrawable

uint16_t getTimerWidgetCountForDrawable (const Drawable * w)

Gets the number of timer events registered to a widget, i.e.

how many times a drawable must be unregistered until it no longer receives timer ticks.

Parameters:

w The widget to to get count from.

Returns:

0 if the drawable is not registered as a timer widget, otherwise returns how many times the drawable is currently registered.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Handle a click event.

Standard implementation is to delegate the event to the current screen. Called by the framework when a click is detected by some platform specific means.

Parameters:

evt The **ClickEvent**.

Reimplements: **touchgfx::UIEventListener::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Handle drag events.

Called by the framework when a drag is detected by some platform specific means. Standard implementation is to delegate drag event to current screen.

Parameters:

evt The drag event, expressed in absolute coordinates.

Reimplements: **touchgfx::UIEventListener::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Handle gestures.

Called by the framework when a gesture is detected by some platform specific means. Standard implementation is to delegate drag event to current screen.

Parameters:

evt The gesture event.

Reimplements: **touchgfx::UIEventListener::handleGestureEvent**

handleKeyEvent

virtual void handleKeyEvent (uint8_t c)

Handle an incoming character received by the HAL layer.

Standard implementation delegates to current screen (which, in turn, does nothing).

Parameters:

`c` The incomming character to handle.

Reimplements: **touchgfx::UIEventListener::handleKeyEvent**

handlePendingScreenTransition

virtual void handlePendingScreenTransition ( )

Evaluates the pending Callback instances.

If a callback is valid, it is executed and a **Screen** transition is executed. This base implementation is empty and does nothing.

Reimplements: **touchgfx::UIEventListener::handlePendingScreenTransition**

Reimplemented by: **touchgfx::MVPApplication::handlePendingScreenTransition**

handleTickEvent

virtual void handleTickEvent ( )

Handle tick.

Standard implementation is to delegate tick to the widgets that have registered to receive one. Called by some platform specific means.

Reimplements: **touchgfx::UIEventListener::handleTickEvent**

registerTimerWidget

void registerTimerWidget (Drawable * w)

Adds a widget to the list of widgets receiving ticks every frame (typically 16.67ms)

Parameters:

`w` The widget to add.

NOTE

The framework keeps track of the number of times a specific widget is registered.

See also:

unregisterTimerWidget

requestRedraw

virtual void requestRedraw ( )

An application specific function for requesting redraw of entire screen.

requestRedraw

virtual void requestRedraw (Rect & rect)

An application specific function for requesting redraw of given Rect.

Parameters:

rect The **Rect** that must be redrawn.

switchScreen

virtual void switchScreen (Screen * newScreen)

Switch to another Screen.

Will call tearDownScreen on current **Screen** before switching, and subsequently call setupScreen and draw automatically for the new **Screen**.

Parameters:

newScreen A pointer to the new screen.

unregisterTimerWidget

void unregisterTimerWidget (const Drawable * w)

Removes a widget from the list of widgets receiving ticks every frame (typically 16.67ms) milliseconds.

Parameters:

w The widget to remove.

NOTE

If widget has been registered multiple times, an equal number of calls to unregister are required to stop widget from receiving tick events.

Protected Functions Documentation

Application

Application ( )

Protected constructor.

invalidateArea

void invalidateArea (Rect area)

Invalidates this area.

Parameters:

area The area to invalidate.

Public Attributes Documentation

MAX_TIMER_WIDGETS

const uint8_t MAX_TIMER_WIDGETS = 32

Maximum number of widgets receiving ticks.

TICK_INTERVAL_MS

const uint16_t TICK_INTERVAL_MS = 10

Deprecated, do not use this constant. Tick interval depends on VSYNC of your target platform.

Protected Attributes Documentation

cachedDirtyAreas

RectVector_t cachedDirtyAreas

When draw caching is enabled, these rects keeps track of the dirty screen area.

drawCacheEnabled

bool drawCacheEnabled

True when draw caching is active.

lastRects

RectVector_t lastRects

The dirty areas from last frame that needs to be redrawn because we have swapped frame buffers.

redraw

Rect redraw

Rect describing application requested invalidate area.

timerWidgetCounter

uint8_t timerWidgetCounter

A counter for each potentially registered timer widget. Increase when registering for timer events, decrease when unregistering.

timerWidgets

Vector< Drawable *, MAX_TIMER_WIDGETS > timerWidgets

List of widgets that receive timer ticks.

transitionHandled

bool transitionHandled

True if the transition is done and Screen::afterTransition has been called.

currentScreen

Screen * currentScreen

Pointer to currently displayed Screen.

currentTransition

Transition * currentTransition

Pointer to current transition.

debugPrinter

DebugPrinter * debugPrinter

Pointer to the DebugPrinter instance.

debugRegionInvalidRect

Rect debugRegionInvalidRect

Invalidated Debug Region.

instance

Application * instance

Pointer to the instance of the Application-derived subclass.

Version: 4.16

Bitmap

This class provides a proxy object for a bitmap image stored in the application specific bitmap database. The proxy provides access to the raw bitmap data as well as metadata.

Public Classes

struct	**BitmapData**
	Data of a bitmap.
struct	**CacheTableEntry**
	Cache bookkeeping.
struct	**DynamicBitmapData**
	Data of a dynamic **Bitmap**.

Public Types

enum	**BitmapFormat** { RGB565, RGB888, ARGB8888, BW, BW_RLE, GRAY2, GRAY4, ARGB2222, ABGR2222, RGBA2222, BGRA2222, L8, A4, CUSTOM }
	Data of a bitmap can be stored in the following formats.
enum	**ClutFormat** { CLUT_FORMAT_L8_ARGB8888, CLUT_FORMAT_L8_RGB888, CLUT_FORMAT_L8_RGB565 }
	Color data of a clut can be stored in the following formats.

Public Functions

	Bitmap(const **BitmapId** id =**BITMAP_INVALID**)
	Creates and binds a **Bitmap** instance to the corresponding entry in the BitmapData array.
const uint8_t *	**getData**() const
	Gets a pointer to the **Bitmap** data.
const uint8_t *	**getExtraData**() const

	Gets a pointer to the extra (alpha) data, if present in the **Bitmap**.
BitmapFormat	**getFormat**() const
	Gets the format of how the **Bitmap** is stored.
uint16_t	**getHeight**() const
	Gets the height of the **Bitmap** in pixels.
BitmapId	**getId**() const
	Gets the id of this **Bitmap**.
Rect	**getRect**() const
	Gets the rectangle describing the dimensions of the **Bitmap**.
Rect	**getSolidRect**() const
	Gets the largest solid, i.e.
uint16_t	**getWidth**() const
	Gets the width of the **Bitmap** in pixels.
bool	**hasTransparentPixels**() const
	Query if this object has transparent pixels.
bool	**isAlphaPerPixel**() const
	Query if this object has an alpha channel.
	operator BitmapId() const
	Gets the id of this **Bitmap**.
bool	**cache**(**BitmapId** id)
	Cache this **Bitmap** into unused RAM in the bitmap cache.
bool	**cacheAll**()
	Cache all bitmaps from the **Bitmap** Database into RAM.
uint8_t *	**cacheGetAddress**(**BitmapId** id)
	Get address of cache buffer for this **Bitmap**.
bool	**cacheIsCached**(**BitmapId** id)
	Check if the **Bitmap** is cached.
bool	**cacheRemoveBitmap**(**BitmapId** id)
	Remove this **Bitmap** from the RAM cache.

bool	**cacheReplaceBitmap**(**BitmapId** out, **BitmapId** in)
	Replace a **Bitmap** in RAM with another **Bitmap**.
void	**clearCache**()
	Clears the cached bitmaps from RAM.
void	**compactCache**()
	Compact the bitmap cache to get continuous free memory on top.
bool	**dynamicBitmapAddSolidRect**(**BitmapId** id, const **Rect** & solidRect)
	Updates the solid rectangle of a dynamic **Bitmap** to include the given rectangle.
BitmapId	**dynamicBitmapCreate**(const uint16_t width, const uint16_t height, **BitmapFormat** format, **ClutFormat** clutFormat =**CLUT_FORMAT_L8_ARGB8888**)
	Create a dynamic **Bitmap**.
BitmapId	**dynamicBitmapCreateCustom**(const uint16_t width, const uint16_t height, uint8_t customSubformat, uint32_t size)
	Create a dynamic bitmap in custom format.
BitmapId	**dynamicBitmapCreateExternal**(const uint16_t width, const uint16_t height, const void * pixels, **BitmapFormat** format, uint8_t customSubformat =0)
	Create a dynamic bitmap without reserving memory in the dynamic bitmap cache.
bool	**dynamicBitmapDelete**(**BitmapId** id)
	Delete a dynamic bitmap.
uint8_t *	**dynamicBitmapGetAddress**(**BitmapId** id)
	Get the address of the dynamic **Bitmap** data.
bool	**dynamicBitmapSetSolidRect**(**BitmapId** id, const **Rect** & solidRect)
	Set the solid rectangle of a dynamic **Bitmap**.
uint8_t *	**getCacheTopAddress**()
	Gets the address of the first unused memory in the cache.
bool	**isDynamicBitmap**(**BitmapId** id)
	Check if a given bitmap id is the id of a dynamic bitmap.
void	**registerBitmapDatabase**(const **BitmapData** *data, const uint16_t n, uint16_t* cachep =0, uint32_t csize =0, uint32_t numberOfDynamicBitmaps =0)
	Registers an array of bitmaps.
void	**removeCache**()

	Removes the **Bitmap** cache.
void	**setCache**(uint16_t * cachep, uint32_t csize, uint32_t numberOfDynamicBitmaps =0)
	Register a memory region in which **Bitmap** data can be cached.

Public Types Documentation

BitmapFormat

enum BitmapFormat

Data of a bitmap can be stored in the following formats.

RGB565	16-bit, 5 bits for red, 6 bits for green, 5 bits for blue. No alpha channel
RGB888	24-bit, 8 bits for each of red, green and blue. No alpha channel
ARGB8888	32-bit, 8 bits for each of red, green, blue and alpha channel
BW	1-bit, black / white. No alpha channel
BW_RLE	1-bit, black / white. No alpha channel. Image is compressed with horizontal RLE
GRAY2	2-bit grayscale
GRAY4	4-bit grayscale
ARGB2222	8-bit color
ABGR2222	8-bit color
RGBA2222	8-bit color
BGRA2222	8-bit color
L8	8-bit indexed color
A4	4-bit alpha level
CUSTOM	Non-standard platform specific format.

ClutFormat

enum ClutFormat

Color data of a clut can be stored in the following formats.

CLUT_FORMAT_L8_ARGB8888	32-bit, 8 bits for each of red, green, blue and alpha

CLUT_FORMAT_L8_RGB888	24-bit, 8 bits for each of red, green and blue. No per pixel alpha channel
CLUT_FORMAT_L8_RGB565	16-bit, 5 bits for red, 6 bits for green, 5 bits for blue. No per pixel alpha channel

Public Functions Documentation

Bitmap

Bitmap (const BitmapId id =BITMAP_INVALID)

Creates and binds a **Bitmap** instance to the corresponding entry in the BitmapData array.

Parameters:

id (Optional) The unique bitmap identifier.

getData

const uint8_t * getData ( ) const

Gets a pointer to the **Bitmap** data.

Returns:

A pointer to the raw **Bitmap** data.

NOTE

If this Bitmap is cached, it will return the cached version of Bitmap data.

getExtraData

const uint8_t * getExtraData ( ) const

Gets a pointer to the extra (alpha) data, if present in the **Bitmap**.

For images stored in L8 format, a pointer to the CLUT will be returned. For non-opaque RGB565 images, a pointer to the alpha channel will be returned.

Returns:

A pointer to the raw alpha channel data or CLUT. If no alpha channel or CLUT exist for the given **Bitmap**, 0 is returned.

NOTE

If this Bitmap is cached, it will return the cached version of alpha data for this Bitmap.

getFormat

BitmapFormat getFormat ( ) const

Gets the format of how the **Bitmap** is stored.

Returns:

The format of how the **Bitmap** data is stored.

getHeight

uint16_t getHeight ( ) const

Gets the height of the **Bitmap** in pixels.

Returns:

The **Bitmap** height in pixels.

getId

BitmapId getId ( ) const

Gets the id of this **Bitmap**.

Returns:

The id of this **Bitmap**.

getRect

Rect getRect ( ) const

Gets the rectangle describing the dimensions of the **Bitmap**.

Returns:

a **Rect** describing the dimensions of this **Bitmap**.

getSolidRect

Rect getSolidRect ( ) const

Gets the largest solid, i.e.

not transparent, rectangle in the **Bitmap**.

Returns:

The maximum solid rectangle of the **Bitmap**.

getWidth

uint16_t getWidth ( ) const

Gets the width of the **Bitmap** in pixels.

Returns:

The **Bitmap** width in pixels.

hasTransparentPixels

bool hasTransparentPixels ( ) const

Query if this object has transparent pixels.

Returns:

True if this bitmap has transparent pixels.

isAlphaPerPixel

bool isAlphaPerPixel ( ) const

Query if this object has an alpha channel.

Returns:

True if the bitmap contains an alpha channel (an alpha value for each pixel)

operator BitmapId

operator BitmapId ( ) const

Gets the id of this **Bitmap**.

Returns:

The id of this **Bitmap**.

cache

static bool cache (BitmapId id)

Cache this **Bitmap** into unused RAM in the bitmap cache.

A memory region large enough to hold this bitmap must be configured and a large enough part of it must be available. Caching of a bitmap may involve a defragmentation of the bitmap cache.

Parameters:

id The id of the **Bitmap** to cache.

Returns:

true if caching went well, false otherwise.

See also:

registerBitmapDatabase, **compactCache**

cacheAll

static bool cacheAll ( )

Cache all bitmaps from the **Bitmap** Database into RAM.

A memory region large enough to hold all bitmaps must be configured.

Returns:

True if all bitmaps where cached.

See also:

cache

cacheGetAddress

static uint8_t * cacheGetAddress (BitmapId id)

Get address of cache buffer for this **Bitmap**.

Parameters:

id The id of the **Bitmap** in cache.

Returns:

Address if **Bitmap** was found, zero otherwise.

NOTE

The address is only valid until next Bitmap::cache() call.

cacheIsCached

static bool cacheIsCached (BitmapId id)

Check if the **Bitmap** is cached.

Parameters:

id The id of the **Bitmap**.

Returns:

true if **Bitmap** is cached.

cacheRemoveBitmap

static bool cacheRemoveBitmap (BitmapId id)

Remove this **Bitmap** from the RAM cache.

Unless the **Bitmap** is otherwise stored in (slower) memory it can not be drawn anymore and must be cached again before use. The RAM freed can be used for caching of another bitmap.

Parameters:

id The id of the **Bitmap** to cache.

Returns:

true if **Bitmap** was found and removed, false otherwise.

See also:

registerBitmapDatabase

cacheReplaceBitmap

```
static bool cacheReplaceBitmap ( BitmapId out ,
                                 BitmapId in
                               )
```

Replace a **Bitmap** in RAM with another **Bitmap**.

The Bitmaps must have same size.

Parameters:

out The id of the **Bitmap** to remove from the cache.
in The id of the **Bitmap** to cache.

Returns:

true if the replacement went well, false otherwise.

clearCache

```
static void clearCache ( )
```

Clears the cached bitmaps from RAM.

compactCache

```
static void compactCache ( )
```

Compact the bitmap cache to get continuous free memory on top.

This method is called by **Bitmap::cache** when required.

dynamicBitmapAddSolidRect

```
static bool dynamicBitmapAddSolidRect ( BitmapId       id ,
                                        const Rect & solidRect
                                      )
```

Updates the solid rectangle of a dynamic **Bitmap** to include the given rectangle.

Only relevant for ARGB8888 bitmap and 8bpp bitmap formats, as these formats include an alpha channel. The solid part of the **Bitmap** is drawn faster than the transparent parts.

Parameters:

id	The identifier.
solidRect	The solid rectangle.

Returns:

true if it succeeds, false if it fails.

dynamicBitmapCreate

```
static BitmapId dynamicBitmapCreate ( const uint16_t width ,
                                      const uint16_t height ,
                                      BitmapFormat format ,
                                      ClutFormat clutFormat =CLUT_FORMAT_L8_ARGB8888
                                      )
```

Create a dynamic **Bitmap**.

The clutFormat parameter is ignored for bitmaps not in L8 format. Creation of a new dynamic bitmap may cause existing dynamic bitmaps to be moved in memory. Do not rely on bitmap memory addresses of dynamic bitmaps obtained from **dynamicBitmapGetAddress()** to be valid across calls to **dynamicBitmapCreate()**.

Parameters:

width	Width of the **Bitmap**.
height	Height of the **Bitmap**.
format	**Bitmap** format of the **Bitmap**.
clutFormat	(Optional) **Color** lookup table format of the **Bitmap**.

Returns:

BitmapId of the new **Bitmap** or **BITMAP_INVALID** if cache memory is full.

See also:

DynamicBitmapData

dynamicBitmapCreateCustom

```
static BitmapId dynamicBitmapCreateCustom ( const uint16_t width ,
                                            const uint16_t height ,
                                            uint8_t customSubformat ,
                                            uint32_t size
                                            )
```

Create a dynamic bitmap in custom format.

```
Create a dynamic bitmap in custom format. size number of
bytes is reserved in the dynamic bitmap cache. A more
specific format can be given in the customSubformat
parameter for use when handling more than one CUSTOM
format. Set the solid rect if applicable.
```

Parameters:

width	Width of the bitmap.
height	Height of the bitmap.
customSubformat	Custom format specifier
size	Size in bytes of the dynamic bitmap

Returns:

BitmapId of the new bitmap or BITMAP_INVALID if cache memory is full.

NOTE

Creation of a new dynamic bitmap may cause existing dynamic bitmaps to be moved in memory. Do not rely on bitmap memory addresses of dynamic bitmaps obtained from dynamicBitmapGetAddress() to be valid across calls to dynamicBitmapCreateCustom() .

See also:

dynamicBitmapAddress, **dynamicBitmapCreate**, **dynamicBitmapSetSolidRect**

dynamicBitmapCreateExternal

```
static BitmapId dynamicBitmapCreateExternal ( const uint16_t width ,
                                              const uint16_t height ,
                                              const void *   pixels ,
                                              BitmapFormat format ,
                                              uint8_t        customSubformat =0
                                              )
```

Create a dynamic bitmap without reserving memory in the dynamic bitmap cache.

```
Create a dynamic bitmap without reserving memory in the
dynamic bitmap cache. The pixels must be already
available in the memory, e.g. in flash. No copying is
performed.
```

Parameters:

width	Width of the bitmap.
height	Height of the bitmap.
pixels	Pointer to the bitmap pixels.
format	**Bitmap** format of the bitmap.
customSubformat	Custom format specifier

Returns:

BitmapId of the new bitmap or BITMAP_INVALID if not possible.

See also:

dynamicBitmapAddress, **dynamicBitmapCreate**, **dynamicBitmapSetSolidRect**

dynamicBitmapDelete

static bool dynamicBitmapDelete (BitmapId id)

Delete a dynamic bitmap.

Parameters:

id The BitmapId of the dynamic **Bitmap**.

Returns:

true if it succeeds, false if it fails.

dynamicBitmapGetAddress

static uint8_t * dynamicBitmapGetAddress (BitmapId id)

Get the address of the dynamic **Bitmap** data.

It is important that the address of a dynamic **Bitmap** is not stored elsewhere as a dynamic **Bitmap** may be moved in memory when other bitmaps are added and removed. Only store the BitmapId and ask for the address of the **Bitmap** data when needed. The address of a dynamic bitmap may change when other dynamic bitmaps are added and removed.

Parameters:

id The BitmapId of the dynamic bitmap.

Returns:

null if it fails, else an uint8_t*.

NOTE

Never keep the address of dynamic images, only store the BitmapId as that will not change.

dynamicBitmapSetSolidRect

```
static bool dynamicBitmapSetSolidRect ( BitmapId      id ,
                                        const Rect & solidRect
                                        )
```

Set the solid rectangle of a dynamic **Bitmap**.

Only relevant for ARGB8888 **Bitmap** and 8bpp **Bitmap** formats, as these formats include an alpha channel. The solid part of the **Bitmap** is drawn faster than the transparent parts.

Parameters:

id	The identifier.
solidRect	The solid rectangle.

Returns:

true if it succeeds, false if it fails.

getCacheTopAddress

```
static uint8_t * getCacheTopAddress (  )
```

Gets the address of the first unused memory in the cache.

Can be used in advanced application to reduce power consumption of external RAM by turning off unused RAM.

Returns:

Returns the highest used address in the cache.

isDynamicBitmap

```
static bool isDynamicBitmap ( BitmapId id )
```

Check if a given bitmap id is the id of a dynamic bitmap.

Parameters:

id	The BitmapId of the dynamic **Bitmap**.

Returns:

true if the bitmap is dynamic, false otherwise.

registerBitmapDatabase

```
static void registerBitmapDatabase ( const BitmapData * data ,
                                     const uint16_t      n ,
                                     uint16_t *          cachep =0,
                                     uint32_t            csize =0,
                                     uint32_t            numberOfDynamicBitmaps =0
                                     )
```

Registers an array of bitmaps.

All **Bitmap** instances are bound to this database. This function is called automatically from HAL::touchgfx_generic_init().

Parameters:

data	A reference to the **BitmapData** storage array.
n	The number of bitmaps in the array.
cachep	(Optional) Pointer to memory region in which bitmap data can be cached.
csize	(Optional) Size of cache memory region in bytes (0 if unused)
numberOfDynamicBitmaps	(Optional) Number of dynamic bitmaps to be allowed in the cache.

removeCache

```
static void removeCache ( )
```

Removes the **Bitmap** cache.

The memory can hereafter be used for other purposes. All dynamic **Bitmap** IDs are invalid after this.

setCache

```
static void setCache ( uint16_t * cachep ,
                       uint32_t   csize ,
                       uint32_t   numberOfDynamicBitmaps =0
                       )
```

Register a memory region in which **Bitmap** data can be cached.

Parameters:

cachep	Pointer to memory region in which bitmap data can be cached.
csize	Size of cache memory region in bytes.
numberOfDynamicBitmaps	(Optional) Number of dynamic bitmaps to be allowed in the cache.

Version: 4.16

BitmapData

Data of a bitmap.

Public Functions

BitmapFormat	**getFormat**() const
	Gets the **Bitmap** format by combining the high and low parts (format_hi << 3)

Public Attributes

const uint8_t *const	**data**
	The data of this **Bitmap**.
const uint8_t *const	**extraData**
	The data of either the alpha channel (if present) or clut data in case of indexed color bitmap. 0 if not used.
const uint16_t	**format_hi**
	Determine the format of the data (high 3 bits)
const uint16_t	**format_lo**
	Determine the format of the data (low 3 bits)
const uint16_t	**height**
	The height of the **Bitmap**.
const uint16_t	**solidRect_height**
	The height of the maximum solid rectangle of the **Bitmap**.
const uint16_t	**solidRect_width**
	The width of the maximum solid rectangle of the **Bitmap**.
const uint16_t	**solidRect_x**
	The x coordinate of the maximum solid rectangle of the **Bitmap**.
const uint16_t	**solidRect_y**
	The y coordinate of the maximum solid rectangle of the **Bitmap**.

const uint16_t	**width**
	The width of the **Bitmap**.

Public Functions Documentation

getFormat

BitmapFormat getFormat ( ) const

Gets the **Bitmap** format by combining the high and low parts (format_hi << 3) | format_lo.

Returns:

The BitmapFormat

Public Attributes Documentation

data

const uint8_t *const data

The data of this **Bitmap**.

extraData

const uint8_t *const extraData

The data of either the alpha channel (if present) or clut data in case of indexed color bitmap. 0 if not used.

format_hi

const uint16_t format_hi

Determine the format of the data (high 3 bits)

format_lo

const uint16_t format_lo

Determine the format of the data (low 3 bits)

height

const uint16_t height

The height of the **Bitmap**.

solidRect_height

const uint16_t solidRect_height

The height of the maximum solid rectangle of the **Bitmap**.

solidRect_width

const uint16_t solidRect_width

The width of the maximum solid rectangle of the **Bitmap**.

solidRect_x

const uint16_t solidRect_x

The x coordinate of the maximum solid rectangle of the **Bitmap**.

solidRect_y

const uint16_t solidRect_y

The y coordinate of the maximum solid rectangle of the **Bitmap**.

width

const uint16_t width

The width of the **Bitmap**.

Version: 4.16

BlitOp

BlitOp instances carry the required information for performing operations on the LCD (framebuffer) using DMA.

Public Attributes

uint8_t	**alpha**
	The alpha to use.
colortype	**color**
	Color to fill.
uint8_t	**dstFormat**
	The destination format.
uint16_t	**dstLoopStride**
	The number of bytes to stride the destination after every loop.
uint16_t	**nLoops**
	The number of lines.
uint16_t	**nSteps**
	The number of pixels in a line.
uint32_t	**operation**
	The operation to perform.
const uint8_t *	**pClut**
	Pointer to the source CLUT entires.
uint16_t *	**pDst**
	Pointer to the destination.
const uint16_t *	**pSrc**
	Pointer to the source (pixels or indexes)
uint8_t	**srcFormat**
	The source format.

uint16_t	**srcLoopStride**
	The number of bytes to stride the source after every loop.

Public Attributes Documentation

alpha

uint8_t alpha

The alpha to use.

color

colortype color

Color to fill.

dstFormat

uint8_t dstFormat

The destination format.

dstLoopStride

uint16_t dstLoopStride

The number of bytes to stride the destination after every loop.

nLoops

uint16_t nLoops

The number of lines.

nSteps

uint16_t nSteps

The number of pixels in a line.

operation

uint32_t operation

The operation to perform.

pClut

const uint8_t * pClut

Pointer to the source CLUT entires.

pDst

uint16_t * pDst

Pointer to the destination.

pSrc

const uint16_t * pSrc

Pointer to the source (pixels or indexes)

srcFormat

uint8_t srcFormat

The source format.

srcLoopStride

uint16_t srcLoopStride

The number of bytes to stride the source after every loop.

Version: 4.16

BlockTransition

A Transition that draws two small blocks in every frame. It is therefore very usefull on MCUs with limited performance.

Inherits from: Transition

Public Functions

	BlockTransition()
	Initializes a new instance of the **BlockTransition** class.
virtual void	**handleTickEvent**()
	Handles the tick event when transitioning.
virtual void	**init**()
	Initializes the transition.
virtual void	**invalidate**()
	Block transition does not require an invalidation.
virtual void	**tearDown**()
	Tears down the Animation.

Additional inherited members

Public Functions inherited from Transition

bool	**isDone**() const
	Query if the transition is done transitioning.
virtual void	**setScreenContainer**(**Container** & cont)
	Sets the **ScreenContainer**.
	Transition()
	Initializes a new instance of the **Transition** class.

virtual	**~Transition**()
	Finalizes an instance of the **Transition** class.

Protected Attributes inherited from Transition

bool	**done**
	Flag that indicates when the transition is done. This should be set by implementing classes.
Container *	**screenContainer**
	The screen **Container** of the **Screen** transitioning to.

Public Functions Documentation

BlockTransition

BlockTransition ( )

Initializes a new instance of the **BlockTransition** class.

Parameters:

transitionSteps (Optional) Number of steps in the transition animation.

handleTickEvent

virtual void handleTickEvent ( )

Handles the tick event when transitioning.

It uncovers and invalidates two blocks in every frame, for a total of 24 frames.

Reimplements: **touchgfx::Transition::handleTickEvent**

init

virtual void init ( )

Initializes the transition.

Called after the constructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::init**

invalidate

virtual void invalidate ( )

Block transition does not require an invalidation.

Invalidation is handled by the class. Do no invalidation initially.

Reimplements: **touchgfx::Transition::invalidate**

tearDown

virtual void tearDown ( )

Tears down the Animation.

Called before the destructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::tearDown**

Version: 4.16

Box

Simple widget capable of showing a rectangle of a specific color and an optional alpha.

Inherits from: Widget, Drawable

Inherited by: BoxWithBorder

Public Functions

	Box()
	Construct a new **Box** with a default alpha value of 255 (solid)
	Box(uint16_t width, uint16_t height, **colortype** color, uint8_t alpha =255)
	Construct a **Box** with the given size and color (and optionally alpha).
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
FORCE_INLINE_FUNCTION uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
FORCE_INLINE_FUNCTION **colortype**	**getColor**() const
	Gets the current color of the **Box**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setColor**(**colortype** color)
	Sets the color of the **Box**.

Protected Attributes

uint8_t	**alpha**
	The alpha value used for this **Box**.

colortype	**color**
	The fill color for this **Box**.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.

virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.

int16_t	**getHeight**() const
	Gets the height of this **Drawable**.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.

Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.

bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

Box

```
Box (  )
```

Construct a new **Box** with a default alpha value of 255 (solid)

Box

```
Box ( uint16_t   width ,
      uint16_t   height ,
      colortype color ,
      uint8_t    alpha =255
      )
```

Construct a **Box** with the given size and color (and optionally alpha).

Parameters:

width	The width of the box.
height	The height of the box.
color	The color of the box.
alpha	(Optional) The alpha of the box. Default is 255 (solid).

draw

```
virtual void draw ( const Rect & invalidatedArea )
```

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

Reimplemented by: **touchgfx::BoxWithBorder::draw**

getAlpha

FORCE_INLINE_FUNCTION uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getColor

FORCE_INLINE_FUNCTION colortype getColor ( ) const

Gets the current color of the **Box**.

Returns:

The current color of the box.

See also:

setColor, **Color::getRedColor**, **Color::getGreenColor**, **Color::getRedColor**

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setColor

void setColor (colortype color)

Sets the color of the **Box**.

Parameters:

color The color of the box.

See also:

getColor, **Color::getColorFrom24BitRGB**

Protected Attributes Documentation

alpha

uint8_t alpha

The alpha value used for this **Box**.

color

colortype color

The fill color for this **Box**.

Version: 4.16

BoxProgress

A BoxProgress which shows the current progress using a simple Box. It is possible to set the color and the alpha of the box. It is also possible to control in what direction the box will progress (up, down, to the left or to the right).

Inherits from: AbstractDirectionProgress, AbstractProgressIndicator, Container, Drawable

Public Functions

	BoxProgress()
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **colortype**	**getColor**() const
	Gets the color of the **Box**.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setColor**(**colortype** color)
	Sets the color of the **Box**.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the actual progress indicator relative to the background image.
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.

Protected Attributes

Box	**box**
	The box.

Additional inherited members

Public Types inherited from AbstractDirectionProgress

enum	**DirectionType** { RIGHT, LEFT, DOWN, UP }
	Values that represent directions.

Public Functions inherited from AbstractDirectionProgress

	AbstractDirectionProgress()
virtual **DirectionType**	**getDirection**() const
	Gets the current direction for the progress indicator.
virtual void	**setDirection**(**DirectionType** direction)
	Sets a direction for the progress indicator.

Protected Attributes inherited from AbstractDirectionProgress

DirectionType	**progressDirection**
	The progress direction.

Public Functions inherited from AbstractProgressIndicator

	AbstractProgressIndicator()
	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.
virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.

virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.
void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.
int	**rangeMin**
	The range minimum.
uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const AbstractProgressIndicator & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const AbstractProgressIndicator & > *	**valueUpdatedCallback**

	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

BoxProgress

BoxProgress ( )

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getColor

virtual colortype getColor ( ) const

Gets the color of the **Box**.

Returns:

The color.

See also:

setColor

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setColor

virtual void setColor (colortype color)

Sets the color of the **Box**.

Parameters:

color The color.

See also:

getColor

setProgressIndicatorPosition

virtual void setProgressIndicatorPosition (int16_t x ,
int16_t y ,
int16_t width ,
int16_t height
)

Sets the position and dimensions of the actual progress indicator relative to the background image.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the box progress indicator.
height	The height of the box progress indicator.

See also:

getProgressIndicatorX, **getProgressIndicatorY**, **getProgressIndicatorWidth**, **getProgressIndicatorHeight**

Reimplements: **touchgfx::AbstractProgressIndicator::setProgressIndicatorPosition**

setValue

virtual void setValue (int value)

Sets the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. If a callback function has been set using setValueSetAction, that callback will be called (unless the new value is the same as the current value).

Parameters:

value The value.

NOTE

if value is equal to the current value, nothing happens, and the callback will not be called.

See also:

getValue, **updateValue**, **setValueSetAction**

Reimplements: **touchgfx::AbstractProgressIndicator::setValue**

Protected Attributes Documentation

box

Box box

The box.

Version: 4.16

BoxWithBorder

The BoxWithBorder class is used to create objects that can draw a box with a border on the display. The width of the border can be specified. If the border width is 0 the BoxWithBorder will function just like a Box.

Inherits from: Box, Widget, Drawable

Public Functions

	BoxWithBorder()
	BoxWithBorder(uint16_t width, uint16_t height, **colortype** color, **colortype** borderColor, uint16_t borderSize, uint8_t alpha =255)
	Constructor that allows specification of dimensions and colors of the **BoxWithBorder**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
FORCE_INLINE_FUNCTION **colortype**	**getBorderColor**() const
	Gets the color of the border drawn along the edge of the **BoxWithBorder**.
FORCE_INLINE_FUNCTION uint16_t	**getBorderSize**() const
	Gets the width of the border.
void	**setBorderColor**(**colortype** color)
	Sets the color of the border drawn along the edge of the **BoxWithBorder**.
void	**setBorderSize**(uint16_t size)
	Sets the width of the border.

Protected Attributes

colortype	**borderColor**

	The color of the border along the edge.
uint16_t	**borderSize**
	Width of the border along the edge.

Additional inherited members

Public Functions inherited from Box

	Box()
	Construct a new **Box** with a default alpha value of 255 (solid)
	Box(uint16_t width, uint16_t height, **colortype** color, uint8_t alpha =255)
	Construct a **Box** with the given size and color (and optionally alpha).
FORCE_INLINE_FUNCTION uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
FORCE_INLINE_FUNCTION **colortype**	**getColor**() const
	Gets the current color of the **Box**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setColor**(**colortype** color)
	Sets the color of the **Box**.

Protected Attributes inherited from Box

uint8_t	**alpha**
	The alpha value used for this **Box**.
colortype	**color**
	The fill color for this **Box**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.

virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.

virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)

	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

BoxWithBorder

BoxWithBorder ( )

BoxWithBorder

```
BoxWithBorder ( uint16_t   width ,
                uint16_t   height ,
                colortype  color ,
                colortype  borderColor ,
                uint16_t   borderSize ,
                uint8_t    alpha =255
                )
```

Constructor that allows specification of dimensions and colors of the **BoxWithBorder**.

Parameters:

width	The width.
height	The height.
color	The color.
borderColor	The border color.
borderSize	Size of the border.
alpha	(Optional) The alpha.

draw

```
virtual void draw ( const Rect & invalidatedArea )
```

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Box::draw**

getBorderColor

```
FORCE_INLINE_FUNCTION colortype getBorderColor (  ) const
```

Gets the color of the border drawn along the edge of the **BoxWithBorder**.

Returns:

The color of the border.

See also:

setBorderColor, **getColor**, **Color::getRedColor**, **Color::getGreenColor**, **Color::getRedColor**

getBorderSize

FORCE_INLINE_FUNCTION uint16_t getBorderSize ( ) const

Gets the width of the border.

Returns:

The width of the border.

See also:

setBorderSize

setBorderColor

void setBorderColor (colortype color)

Sets the color of the border drawn along the edge of the **BoxWithBorder**.

Parameters:

color The color of the border.

See also:

setColor, **getBorderColor**, **Color::getColorFrom24BitRGB**

setBorderSize

void setBorderSize (uint16_t size)

Sets the width of the border.

If the width is set to 0, the **BoxWithBorder** will look exactly like a **Box**.

Parameters:

size The width of the border.

See also:

getBorderSize

Protected Attributes Documentation

borderColor

colortype borderColor

The color of the border along the edge.

borderSize

uint16_t borderSize

Width of the border along the edge.

Version: 4.16

BoxWithBorderButtonStyle

A box with border button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show a box with a border in different colors depending on the state of the button (pressed or released).

An image button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show one of two images depending on the state of the button (pressed or released).

Template Parameters:

- **T** Generic type parameter. Typically a AbstractButtonContainer subclass.

See: AbstractButtonContainer, BoxWithBorder

Inherits from: T

Public Functions

	BoxWithBorderButtonStyle()
void	**setBorderSize**(uint8_t size)
	Sets border size.
void	**setBoxWithBorderColors**(const **colortype** colorReleased, const **colortype** colorPressed, const **colortype** borderColorReleased, const **colortype** borderColorPressed)
	Sets the colors.
void	**setBoxWithBorderHeight**(int16_t height)
	Sets the height.
void	**setBoxWithBorderPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **BoxWithBorderButtonStyle**, relative to its parent.
void	**setBoxWithBorderWidth**(int16_t width)
	Sets the width.

Protected Functions

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

BoxWithBorder	**borderBox**
	The border box.
colortype	**borderDown**
	The border down.
colortype	**borderUp**
	The border up.
colortype	**down**
	The down.
colortype	**up**
	The up.

Public Functions Documentation

BoxWithBorderButtonStyle

BoxWithBorderButtonStyle ( )

setBorderSize

void setBorderSize (uint8_t size)

Sets border size.

Parameters:

size The size.

setBoxWithBorderColors

```
void setBoxWithBorderColors ( const colortype colorReleased ,
                              const colortype colorPressed ,
                              const colortype borderColorReleased ,
                              const colortype borderColorPressed
                              )
```

Sets the colors.

Parameters:

colorReleased	The color released.
colorPressed	The color pressed.
borderColorReleased	The border color released.
borderColorPressed	The border color pressed.

setBoxWithBorderHeight

```
void setBoxWithBorderHeight ( int16_t height )
```

Sets the height.

Parameters:

height The height.

setBoxWithBorderPosition

```
void setBoxWithBorderPosition ( int16_t x ,
                                int16_t y ,
                                int16_t width ,
                                int16_t height
                                )
```

Sets the size and position of this **BoxWithBorderButtonStyle**, relative to its parent.

Parameters:

x	The x coordinate of this **BoxWithBorderButtonStyle**.
y	The y coordinate of this **BoxWithBorderButtonStyle**.
width	The width of this **BoxWithBorderButtonStyle**.
height	The height of this **BoxWithBorderButtonStyle**.

NOTE

Changing this does not automatically yield a redraw.

setBoxWithBorderWidth

void setBoxWithBorderWidth (int16_t width)

Sets the width.

Parameters:

width The width.

Protected Functions Documentation

handleAlphaUpdated

virtual void handleAlphaUpdated ( )

Handles what should happen when the alpha is updated.

handlePressedUpdated

virtual void handlePressedUpdated ( )

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

borderBox

BoxWithBorder borderBox

The border box.

borderDown

colortype borderDown

The border down.

borderUp

colortype borderUp

The border up.

down

colortype down

The down.

up

colortype up

The up.

Version: 4.16

Button

A button with two images. One image showing the unpressed button and one image showing the pressed state.

Inherits from: AbstractButton, Widget, Drawable

Inherited by: ButtonWithIcon, ButtonWithLabel, RepeatButton, ToggleButton

Public Functions

	Button()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBitmaps**(const **Bitmap** & bitmapReleased, const **Bitmap** & bitmapPressed)
	Sets the two bitmaps used by this button.

Protected Attributes

uint8_t	**alpha**
	The current alpha value. 255=solid, 0=invisible.
Bitmap	**down**

	The image to display when button is pressed.
Bitmap	**up**
	The image to display when button is released (normal state).

Additional inherited members

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.

int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.

void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.

virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

Button

Button ( )

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: touchgfx::Drawable::draw

Reimplemented by: touchgfx::ButtonWithLabel::draw, touchgfx::ButtonWithIcon::draw

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getCurrentlyDisplayedBitmap

Bitmap getCurrentlyDisplayedBitmap ( ) const

Gets currently displayed bitmap.

This depends on the current state of the button, released (normal) or pressed.

Returns:

The bitmap currently displayed.

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the **Drawable**, meaning that a completely solid widget should return the full size **Rect**(0, 0, **getWidth()**, **getHeight()**). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setBitmaps

virtual void setBitmaps (const Bitmap & bitmapReleased ,
const Bitmap & bitmapPressed
)

Sets the two bitmaps used by this button.

One bitmap for the released (normal) state and one bitmap for the pressed state. The images are expected to be of the same dimensions, and the **Button** is resized to the dimensions of the pressed **Bitmap**.

Parameters:

bitmapReleased **Bitmap** to use when button is released.
bitmapPressed **Bitmap** to use when button is pressed.

NOTE

It is assumed that the dimensions of the bitmaps are the same. Unexpected (visual) behavior may be observed if the bitmaps are of different sizes. The user code must call **invalidate()** in order to update the button on the display.

Reimplemented by: touchgfx::ToggleButton::setBitmaps

Protected Attributes Documentation

alpha

uint8_t alpha

The current alpha value. 255=solid, 0=invisible.

down

Bitmap down

The image to display when button is pressed.

up

Bitmap up

The image to display when button is released (normal state).

Version: 4.16

ButtonController

Interface for sampling external key events.

Public Functions

virtual void	**init**() =0
	Initializes button controller.
virtual void	**reset**()
	Resets button controller.
virtual bool	**sample**(uint8_t & key) =0
	Sample external key events.
virtual	**~ButtonController**()
	Finalizes an instance of the **ButtonController** class.

Public Functions Documentation

init

virtual void init ( ) =0

Initializes button controller.

reset

virtual void reset ( )

Resets button controller.

Does nothing in the default implementation.

sample

virtual bool sample (uint8_t & key)

Sample external key events.

Parameters:

key Output parameter that will be set to the key value if a keypress was detected.

Returns:

True if a keypress was detected and the "key" parameter is set to a value.

~ButtonController

virtual ~ButtonController ( )

Finalizes an instance of the **ButtonController** class.

Version: 4.16

Buttons

A class for accessing a physical button.

Public Functions

void	**init**()
	Perform configuration of IO pins.
unsigned int	**sample**()
	Sample button states.

Public Functions Documentation

init

static void init ( )

Perform configuration of IO pins.

sample

static unsigned int sample ( )

Sample button states.

Returns:

the sampled state of the buttons.

Version: 4.16

ButtonWithIcon

A Button that has a bitmap with an icon on top of it. It is possible to have two different icons depending on the current state of the button (released or pressed).

Typical use case could be a blue button with a released and a pressed image. Those images can be reused across several buttons. The ButtonWithIcon will then allow an image to superimposed on top of the blue button.

Inherits from: Button, AbstractButton, Widget, Drawable

Public Functions

	ButtonWithIcon()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
Bitmap	**getCurrentlyDisplayedIcon**() const
	Gets currently displayed icon.
int16_t	**getIconX**() const
	Gets the x coordinate of the icon bitmap.
int16_t	**getIconY**() const
	Gets the y coordinate of the icon bitmap.
virtual void	**setBitmaps**(const **Bitmap** & newBackgroundReleased, const **Bitmap** & newBackgroundPressed, const **Bitmap** & newIconReleased, const **Bitmap** & newIconPressed)
	Sets the four bitmaps used by this button.
void	**setIconX**(int16_t x)
	Sets the x coordinate of the icon bitmaps.
void	**setIconXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of the icon bitmap.
void	**setIconY**(int16_t y)
	Sets the y coordinate of the icon bitmaps.

Protected Attributes

Bitmap	**iconPressed**
	Icon to display when button is pressed.
Bitmap	**iconReleased**
	Icon to display when button is not pressed.
int16_t	**iconX**
	x coordinate offset for icon.
int16_t	**iconY**
	y coordinate offset for icon.

Additional inherited members

Public Functions inherited from Button

	Button()
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
virtual Rect	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

Protected Attributes inherited from Button

uint8_t	**alpha**
	The current alpha value. 255=solid, 0=invisible.

Bitmap	**down**
	The image to display when button is pressed.
Bitmap	**up**
	The image to display when button is released (normal state).

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const

	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

virtual void **translateRectToAbsolute**(**Rect** & r) const

	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

ButtonWithIcon

ButtonWithIcon ( )

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Button::draw**

getCurrentlyDisplayedIcon

Bitmap getCurrentlyDisplayedIcon ( ) const

Gets currently displayed icon.

This depends on the current state of the button, released (normal) or pressed.

Returns:

The icon currently displayed.

getIconX

int16_t getIconX ( ) const

Gets the x coordinate of the icon bitmap.

Returns:

The x coordinate of the icon bitmap.

getIconY

int16_t getIconY ( ) const

Gets the y coordinate of the icon bitmap.

Returns:

The y coordinate of the icon bitmap.

setBitmaps

```
virtual void setBitmaps ( const Bitmap & newBackgroundReleased ,
                          const Bitmap & newBackgroundPressed ,
                          const Bitmap & newIconReleased ,
                          const Bitmap & newIconPressed
```

)

Sets the four bitmaps used by this button.

The last two bitmaps are drawn on top of the first two, again depending on the current state of the **Button**. This means that when the button state is released (normal), the newIconReleased is drawn on top of the newBackgroundReleased, and when the button state is pressed, the newIconPressed is drawn on top of the newBackgroundPressed.

The default position if the icons is set to the center of the bitmaps. The two icons are assumed to have the same dimensions. The size of the released icon is used to position the icons centered on the **Button**.

Parameters:

newBackgroundReleased	**Bitmap** to use when button is released.
newBackgroundPressed	**Bitmap** to use when button is pressed.
newIconReleased	The bitmap for the icon in the released (normal) button state.
newIconPressed	The bitmap for the icon in the pressed button state.

NOTE

The user code must call invalidate() in order to update the button on the display.

setIconX

void setIconX (int16_t x)

Sets the x coordinate of the icon bitmaps.

The same x coordinate is used for both icons (released and pressed).

Parameters:

x	The new x value, relative to the background bitmap. A negative value is allowed.

NOTE

The user code must call invalidate() in order to update the button on the display. The value set is overwritten on a subsequent call to setBitmaps.

setIconXY

void setIconXY (int16_t x ,
int16_t y

)

Sets the x and y coordinates of the icon bitmap.

Same as calling setIconX and setIconY.

Parameters:

x The new x value, relative to the background bitmap. A negative value is allowed.
y The new y value, relative to the background bitmap. A negative value is allowed.

NOTE

The user code must call invalidate() in order to update the button on the display. The values set are overwritten on a subsequent call to setBitmaps.

setIconY

void setIconY (int16_t y)

Sets the y coordinate of the icon bitmaps.

The same y coordinate is used for both icons (released and pressed).

Parameters:

y The new y value, relative to the background bitmap. A negative value is allowed.

NOTE

The user code must call invalidate() in order to update the button on the display. The value set is overwritten on a subsequent call to setBitmaps.

Protected Attributes Documentation

iconPressed

Bitmap iconPressed

Icon to display when button is pressed.

iconReleased

Bitmap iconReleased

Icon to display when button is not pressed.

iconX

int16_t iconX

x coordinate offset for icon.

iconY

int16_t iconY

y coordinate offset for icon.

Version: 4.16

ButtonWithLabel

A Button that has a bitmap with a text on top of it. It is possible to have two different colors for the text depending on the current state of the button (released or pressed).

Typical use case could be a red button with a normal and a pressed image. Those images can be reused across several buttons. The ButtonWithLabel will then allow a text to be superimposed on top of the red button.

See: Button

Inherits from: Button, AbstractButton, Widget, Drawable

Public Functions

	ButtonWithLabel()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
TextRotation	**getLabelRotation**()
	Gets the current rotation of the text on the label.
TypedText	**getLabelText**() const
	Gets the text used for the label.
void	**setLabelColor**(**colortype** col)
	Sets label color for the text when the button is in the normal, released state.
void	**setLabelColorPressed**(**colortype** col)
	Sets label color for the text when the button is in the pressed state.
void	**setLabelRotation**(**TextRotation** rotation)
	Sets the rotation of the text on the label.
void	**setLabelText**(**TypedText** t)
	Sets the text to display on the button.
void	**updateTextPosition**()

Positions the label text horizontally centered.

Protected Attributes

colortype	**color**
	The color used for the label when the button is in the released, normal state.
colortype	**colorPressed**
	The color used for the label when the button is in the pressed state.
TextRotation	**rotation**
	The rotation used for the label.
uint8_t	**textHeightIncludingSpacing**
	Total height of the label (text height + spacing).
TypedText	**typedText**
	The TypedText used for the button label.

Additional inherited members

Public Functions inherited from Button

	Button()
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
virtual Rect	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBitmaps**(const Bitmap & bitmapReleased, const Bitmap & bitmapPressed)
	Sets the two bitmaps used by this button.

Protected Attributes inherited from Button

uint8_t	**alpha**
	The current alpha value. 255=solid, 0=invisible.
Bitmap	**down**
	The image to display when button is pressed.
Bitmap	**up**
	The image to display when button is released (normal state).

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()

	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)

	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

ButtonWithLabel

ButtonWithLabel ( )

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Button::draw**

getLabelRotation

TextRotation getLabelRotation ( )

Gets the current rotation of the text on the label.

Returns:

The current rotation of the text.

getLabelText

TypedText getLabelText ( ) const

Gets the text used for the label.

Returns:

The text used for the label.

setLabelColor

void setLabelColor (colortype col)

Sets label color for the text when the button is in the normal, released state.

Parameters:

col	The color with which the text label should be drawn.

NOTE

If the button is currently in the normal, released state, the button should be forced to redraw itself. This is done by calling invalidate() on the ButtonWithLabel. The user code must call invalidate() in order to update the button on the display.

See also:

setLabelColorPressed

setLabelColorPressed

void setLabelColorPressed (colortype col)

Sets label color for the text when the button is in the pressed state.

Parameters:

col The color with which the text label should be drawn when the button is pressed.

NOTE

If the button is currently in the pressed state, the button should be forced to redraw itself. This is done by calling invalidate() on the ButtonWithLabel. The user code must call invalidate() in order to update the button on the display.

See also:

setLabelColor

setLabelRotation

void setLabelRotation (TextRotation rotation)

Sets the rotation of the text on the label.

The text can be rotated in steps of 90 degrees.

Parameters:

rotation The rotation of the text. Default is TEXT_ROTATE_0.

NOTE

that this will not rotate the bitmap of the label, only the text. The user code must call invalidate() in order to update the button on the display.

See also:

TextArea::setRotation

setLabelText

void setLabelText (TypedText t)

Sets the text to display on the button.

Texts with wildcards are not supported.

Parameters:

t The text to display.

NOTE

The user code must call invalidate() in order to update the button on the display.

updateTextPosition

void updateTextPosition ( )

Positions the label text horizontally centered.

If the text changes due to a language change you may need to reposition the label text by calling this function to keep the text horizontally centered.

NOTE

The user code must call invalidate() in order to update the button on the display.

Protected Attributes Documentation

color

colortype color

The color used for the label when the button is in the released, normal state.

colorPressed

colortype colorPressed

The color used for the label when the button is in the pressed state.

rotation

TextRotation rotation

The rotation used for the label.

textHeightIncludingSpacing

uint8_t textHeightIncludingSpacing

Total height of the label (text height + spacing).

typedText

TypedText typedText

The TypedText used for the button label.

Version: 4.16

CacheableContainer

A CacheableContainer can be seen as a regular Container, i.e. a Drawable that can have child nodes. The z-order of children is determined by the order in which Drawables are added to the container - the Drawable added last will be front-most on the screen.

The important difference is that a CacheableContainer can also render its content to a dynamic bitmap which can then be used as a texture in subsequent drawing operations, either as a simple Image or in a TextureMapper. If the bitmap format of the dynamic bitmap differs from the format of the current LCD, the LCD from drawing the bitmap must be setup using HAL::setAuxiliaryLCD().

See: Container, Bitmap, Image, TextureMapper

Inherits from: Container, Drawable

Public Functions

	CacheableContainer()
void	**enableCachedMode**(bool enable)
	Toggle cached mode on and off.
BitmapId	**getCacheBitmap**() const
	Get the dynamic bitmap used by the **CacheableContainer**.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a subregion of this drawable is redrawn.
bool	**isChildInvalidated**() const
	Queries the **CacheableContainer** whether any child widget has been invalidated.
void	**setCacheBitmap**(**BitmapId** bitmapId)
	Set the dynamic bitmap into which the container content will be rendered.
bool	**setSolidRect**(const **Rect** & rect)
	Set the solid area on the dynamic bitmap assigned to the **CacheableContainer**.
void	**updateCache**()
	Render the container into the attached dynamic bitmap.

void	**updateCache**(const **Rect** & rect)
	Render the container into the attached dynamic bitmap.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.

virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const

	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()

	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

CacheableContainer

CacheableContainer ( )

enableCachedMode

void enableCachedMode (bool enable)

Toggle cached mode on and off.

The **CacheableContainer** behaves just like a regular **Container** when cached mode is turned off.

Parameters:

enable Enable or disable cached mode.

getCacheBitmap

BitmapId getCacheBitmap ( ) const

Get the dynamic bitmap used by the **CacheableContainer**.

Returns:

the id of the assigned bitmap or BITMAP_INVALID if no bitmap has been assigned.

See also:

setCacheBitmap

invalidateRect

virtual void invalidateRect (Rect & invalidatedArea)

Request that a subregion of this drawable is redrawn.

Will recursively traverse the children. When this function returns, the specified invalidated area has been redrawn for all appropriate Drawables covering the region.

Parameters:

invalidatedArea	The area of this drawable to redraw expressed in coordinates relative to its parent (e.g. to request a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::invalidateRect**

isChildInvalidated

bool isChildInvalidated ( ) const

Queries the **CacheableContainer** whether any child widget has been invalidated.

Returns:

True if a child widget has been invalidated and false otherwise.

setCacheBitmap

void setCacheBitmap (BitmapId bitmapId)

Set the dynamic bitmap into which the container content will be rendered.

The format of the bitmap must be the same as the current **LCD** or the same as the auxiliary **LCD** setup using **HAL::setAuxiliaryLCD**.

Parameters:

bitmapId	Id of the dynamic bitmap to serve as a render target.

See also:

updateCache, **getCacheBitmap**, **HAL::setAuxiliaryLCD**

setSolidRect

bool setSolidRect (const Rect & rect)

Set the solid area on the dynamic bitmap assigned to the **CacheableContainer**.

Parameters:

rect	The rectangle of th **CacheableContainer** that is solid.

Returns:

true if the operation succeeds, false otherwise.

updateCache

void updateCache ( )

Render the container into the attached dynamic bitmap.

See also:

setCacheBitmap

updateCache

void updateCache (const Rect & rect)

Render the container into the attached dynamic bitmap.

Only the specified **Rect** region is updated.

Parameters:

rect Region to update.

See also:

setCacheBitmap

Version: 4.16

CacheTableEntry

Cache bookkeeping.

Public Attributes

uint8_t *	**data**
	Pointer to location of image data for this **Bitmap** in the cache. 0 if bitmap not cached.

Public Attributes Documentation

data

uint8_t * data

Pointer to location of image data for this **Bitmap** in the cache. 0 if bitmap not cached.

Version: 4.16

Callback

A Callback is basically a wrapper of a pointer-to-member-function. It is used for registering callbacks between widgets. For instance, a Button can be configured to call a member function when it is clicked.

The class is templated in order to provide the class type of the object in which the member function resides, and the argument types of the function to call.

The Callback class exists in four versions, for supporting member functions with 0, 1, 2 or 3 arguments. The compiler will infer which type to use automatically.

Template Parameters:

- **dest_type** The type of the class in which the member function resides.
- **T1** The type of the first argument in the member function, or void if none.
- **T2** The type of the second argument in the member function, or void if none.
- **T3** The type of the third argument in the member function, or void if none.

Note: The member function to call must return void. The function can have zero, one, two or three arguments of any type.

Inherits from: GenericCallback< void, void, void >

Public Functions

	Callback()
	Initializes a new instance of the **Callback** class.
	Callback(dest_type *pobject, void(dest_type::*)(T1, T2, T3) pmemfun_3)
	Initializes a **Callback** with an object and a pointer to the member function in that object to call.
virtual void	**execute**(T1 t1, T2 t2, T3 t3)
	Calls the member function.
virtual bool	**isValid**() const
	Function to check whether the **Callback** has been initialized with values.

Additional inherited members

Public Functions inherited from GenericCallback< void, void, void >

virtual	~GenericCallback()
	Finalizes an instance of the GenericCallback class.

Public Functions Documentation

Callback

```
Callback ( )
```

Initializes a new instance of the **Callback** class.

Callback

```
Callback ( dest_type *                    pobject ,
           void(dest_type::*)(T1, T2, T3) pmemfun_3
           )
```

Initializes a **Callback** with an object and a pointer to the member function in that object to call.

Initializes a **Callback** with an object and a pointer to the member function in that object to call.

Parameters:

pobject	Pointer to the object on which the function should be called.
pmemfun_3	Address of member function. This is the version where function takes three arguments.

execute

```
virtual void execute ( T1 t1 ,
                       T2 t2 ,
                       T3 t3
                       )
```

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

Parameters:

t1 This value will be passed as the first argument in the function call.
t2 This value will be passed as the second argument in the function call.
t3 This value will be passed as the third argument in the function call.

isValid

virtual bool isValid ( ) const

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

Version: 4.16

Callback<dest_type,T1,T2,void>

A Callback is basically a wrapper of a pointer-to-member-function. It is used for registering callbacks between widgets. For instance, a Button can be configured to call a member function when it is clicked.

The class is templated in order to provide the class type of the object in which the member function resides, and the argument types of the function to call.

The Callback class exists in four versions, for supporting member functions with 0, 1, 2 or 3 arguments. The compiler will infer which type to use automatically.

Template Parameters:

- **dest_type** The type of the class in which the member function resides.
- **T1** The type of the first argument in the member function, or void if none.
- **T2** The type of the second argument in the member function, or void if none.

Note: The member function to call must return void. The function can have zero, one, two or three arguments of any type.

Inherits from: GenericCallback< T1, T2 >

Public Functions

	Callback()
	Initializes a new instance of the **Callback** class.
	Callback(dest_type *pobject, void(dest_type::)*(T1, T2) pmemfun_2)
	Initializes a **Callback** with an object and a pointer to the member function in that object to call.
virtual void	**execute**(T1 t1, T2 t2)
	Calls the member function.
virtual bool	**isValid**() const
	Function to check whether the **Callback** has been initialized with values.

Additional inherited members

Public Functions inherited from GenericCallback< T1, T2 >

virtual	**~GenericCallback**()
	Finalizes an instance of the **GenericCallback** class.

Public Functions Documentation

Callback

```
Callback ( )
```

Initializes a new instance of the **Callback** class.

Callback

```
Callback ( dest_type *                    pobject ,
           void(dest_type::*)(T1, T2) pmemfun_2
           )
```

Initializes a **Callback** with an object and a pointer to the member function in that object to call.

Parameters:

pobject	Pointer to the object on which the function should be called.
pmemfun_2	Address of member function. This is the version where function takes two arguments.

execute

```
virtual void execute ( T1 t1 ,
                       T2 t2
                       )
```

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

Parameters:

t1 This value will be passed as the first argument in the function call.
t2 This value will be passed as the second argument in the function call.

isValid

virtual bool isValid ( ) const

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

Version: 4.16

Callback<dest_type,T1,void,void>

A Callback is basically a wrapper of a pointer-to-member-function. It is used for registering callbacks between widgets. For instance, a Button can be configured to call a member function when it is clicked.

The class is templated in order to provide the class type of the object in which the member function resides, and the argument types of the function to call.

The Callback class exists in four versions, for supporting member functions with 0, 1, 2 or 3 arguments. The compiler will infer which type to use automatically.

Template Parameters:

- **dest_type** The type of the class in which the member function resides.
- **T1** The type of the first argument in the member function, or void if none.

Note: The member function to call must return void. The function can have zero, one, two or three arguments of any type.

Inherits from: GenericCallback< T1 >

Public Functions

	Callback()
	Initializes a new instance of the **Callback** class.
	Callback(dest_type *pobject, void(dest_type::*)(T1) pmemfun_1)
	Initializes a **Callback** with an object and a pointer to the member function in that object to call.
virtual void	**execute**(T1 t1)
	Calls the member function.
virtual bool	**isValid**() const
	Query if this object is valid.

Additional inherited members

Public Functions inherited from GenericCallback< T1 >

virtual	**~GenericCallback**()
	Finalizes an instance of the **GenericCallback** class.

Public Functions Documentation

Callback

Callback ( )

Initializes a new instance of the **Callback** class.

Callback

```
Callback ( dest_type *                pobject ,
           void(dest_type::*)(T1)     pmemfun_1
           )
```

Initializes a **Callback** with an object and a pointer to the member function in that object to call.

Parameters:

pobject	Pointer to the object on which the function should be called.
pmemfun_1	Address of member function. This is the version where function takes one argument.

execute

virtual void execute (T1 t1)

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

Parameters:

t1	This value will be passed as the first argument in the function call.

See also:

isValid

isValid

virtual bool isValid ( ) const

Query if this object is valid.

Returns:

true if valid, false if not.

Version: 4.16

Callback<dest_type,void,void,void>

A Callback is basically a wrapper of a pointer-to-member-function. It is used for registering callbacks between widgets. For instance, a Button can be configured to call a member function when it is clicked.

The class is templated in order to provide the class type of the object in which the member function resides, and the argument types of the function to call.

The Callback class exists in four versions, for supporting member functions with 0, 1, 2 or 3 arguments. The compiler will infer which type to use automatically.

Template Parameters:

- **dest_type** The type of the class in which the member function resides.

Note: The member function to call must return void. The function can have zero, one, two or three arguments of any type.

Inherits from: GenericCallback<>

Public Functions

	Callback()
	Initializes a new instance of the **Callback** class.
	Callback(dest_type *pobject, void(dest_type::)*() pmemfun_0)
	Initializes a **Callback** with an object and a pointer to the member function in that object to call.
virtual void	**execute**()
	Calls the member function.
virtual bool	**isValid**() const
	Function to check whether the **Callback** has been initialized with values.

Additional inherited members

Public Functions inherited from GenericCallback< >

virtual	**~GenericCallback**()
	Finalizes an instance of the **GenericCallback** class.

Public Functions Documentation

Callback

Callback ( )

Initializes a new instance of the **Callback** class.

Callback

Callback (dest_type * pobject ,
void(dest_type::*)() pmemfun_0
)

Initializes a **Callback** with an object and a pointer to the member function in that object to call.

Parameters:

pobject	Pointer to the object on which the function should be called.
pmemfun_0	Address of member function. This is the version where function takes zero arguments.

execute

virtual void execute ( )

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

isValid

virtual bool isValid ( ) const

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

Version: 4.16

CallbackArea

Mapping from rectangle to a callback method to execute.

Public Attributes

GenericCallback *	callback
	The callback to execute, when the area is "pressed". The callback should be a Callback<YourClass> member in the class using the keyboard.
BitmapId	highlightBitmapId
	A bitmap to show when the area is "pressed".
Rect	keyArea
	The area occupied by a key.

Public Attributes Documentation

callback

GenericCallback * callback

The callback to execute, when the area is "pressed". The callback should be a Callback<YourClass> member in the class using the keyboard.

highlightBitmapId

BitmapId highlightBitmapId

A bitmap to show when the area is "pressed".

keyArea

Rect keyArea

The area occupied by a key.

Version: 4.16

Canvas

Class for easy rendering using CanvasWidgetRenderer. The Canvas class will make implementation of a new CanvasWidget very easy. The few simple primitives allows moving a "pen" and drawing the outline of a shape which can then be rendered.

The Canvas class has been optimized to eliminate drawing unnecessary lines outside the currently invalidated rectangle.

Public Functions

	Canvas(const **CanvasWidget** * _widget, const **Rect** & invalidatedArea)
	Canvas Constructor.
void	**lineTo**(**CWRUtil::Q5** x, **CWRUtil::Q5** y)
	Draw line from the current (x, y) to the new (x, y) as part of the shape being drawn.
template \<typename T \> void	**lineTo**(T x, T y)
	Draw line from the current (x, y) to the new (x, y) as part of the shape being drawn.
void	**moveTo**(**CWRUtil::Q5** x, **CWRUtil::Q5** y)
	Move the current pen position to (x, y).
template \<typename T \> void	**moveTo**(T x, T y)
	Move the current pen position to (x, y).
bool	**render**(uint8_t customAlpha =255)
	Render the graphical shape drawn using **moveTo()** and **lineTo()** with the given Painter.
virtual	**~Canvas**()
	Finalizes an instance of the **Canvas** class.

Public Functions Documentation

Canvas

```
Canvas ( const CanvasWidget * _widget ,
         const Rect &         invalidatedArea
       )
```

Canvas Constructor.

Locks the framebuffer and prepares for drawing only in the allowed area which has been invalidated. The color depth of the **LCD** is taken into account.

Parameters:

_widget	a pointer to the **CanvasWidget** using this **Canvas**. Used for getting the canvas dimensions.
invalidatedArea	the are which should be updated.

NOTE

Locks the framebuffer.

lineTo

```
void lineTo ( CWRUtil::Q5 x ,
              CWRUtil::Q5 y
            )
```

Draw line from the current (x, y) to the new (x, y) as part of the shape being drawn.

As for moveTo, lineTo commands completely outside the drawing are are discarded.

Parameters:

x	The x coordinate for the pen position in **CWRUtil::Q5** format.
y	The y coordinate for the pen position in **CWRUtil::Q5** format.

See also:

CWRUtil::Q5, **moveTo**

lineTo

```
void lineTo ( T x ,
              T y
            )
```

Draw line from the current (x, y) to the new (x, y) as part of the shape being drawn.

As for moveTo, lineTo commands completely outside the drawing are are discarded.

Template Parameters:

T either int or float.

Parameters:

x The x coordinate for the pen position.
y The y coordinate for the pen position.

moveTo

```
void moveTo ( CWRUtil::Q5 x ,
              CWRUtil::Q5 y
              )
```

Move the current pen position to (x, y).

If the pen is outside the drawing area, nothing is done, but the coordinates are saved in case the next operation is lineTo a coordinate which is inside (or on the opposite side of) the drawing area.

Parameters:

x The x coordinate for the pen position in **CWRUtil::Q5** format.
y The y coordinate for the pen position in **CWRUtil::Q5** format.

See also:

CWRUtil::Q5, **lineTo**

moveTo

```
void moveTo ( T x ,
              T y
              )
```

Move the current pen position to (x, y).

If the pen is outside (above or below) the drawing area, nothing is done, but the coordinates are saved in case the next operation is lineTo a coordinate which is inside (or on the opposite side of) the drawing area.

Template Parameters:

T Either int or float.

Parameters:

x The x coordinate for the pen position.
y The y coordinate for the pen position.

render

bool render (uint8_t customAlpha =255)

Render the graphical shape drawn using **moveTo()** and **lineTo()** with the given Painter.

The shape is automatically closed, i.e. a **lineTo()** is automatically inserted connecting the current pen position with the initial pen position given in the first **moveTo()** command.

Parameters:

customAlpha (Optional) Alpha to apply to the entire canvas. Useful if the canvas is part of a more complex container setup that needs to be faded. Default is solid.

Returns:

true if the widget was rendered, false if insufficient memory was available to render the widget.

~Canvas

virtual ~Canvas ( )

Finalizes an instance of the **Canvas** class.

NOTE

Unlocks the framebuffer.

Version: 4.16

CanvasWidget

Class for drawing complex polygons on the display using CanvasWidgetRenderer. The CanvasWidget is used by passing it to a Canvas object, drawing the outline of the object and then having CanvasWidget render the outline on the display using the assigned painter.

Inherits from: Widget, Drawable

Inherited by: AbstractGraphElement, AbstractShape, Circle, Line

Public Functions

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)

	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()

	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const

	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

CanvasWidget

CanvasWidget ( )

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::Drawable::draw**

Reimplemented by: **touchgfx::GraphElementGridX::draw**, **touchgfx::GraphElementGridY::draw**, **touchgfx::GraphElementVerticalGapLine::draw**, **touchgfx::GraphElementHistogram::draw**, **touchgfx::GraphElementBoxes::draw**, **touchgfx::GraphLabelsX::draw**, **touchgfx::GraphLabelsY::draw**, **touchgfx::GraphTitle::draw**

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplemented by: **touchgfx::AbstractShape::drawCanvasWidget**, **touchgfx::Circle::drawCanvasWidget**, **touchgfx::Line::drawCanvasWidget**, **touchgfx::AbstractGraphElementNoCWR::drawCanvasWidget**, **touchgfx::GraphElementArea::drawCanvasWidget**, **touchgfx::GraphElementLine::drawCanvasWidget**, **touchgfx::GraphElementDots::drawCanvasWidget**, **touchgfx::GraphElementDiamonds::drawCanvasWidget**, **touchgfx::GraphTitle::drawCanvasWidget**

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getMinimalRect

virtual Rect getMinimalRect ( ) const

Gets minimal rectangle containing the shape drawn by this widget.

Default implementation returns the size of the entire widget, but this function should be overwritten in subclasses and return the minimal rectangle containing the shape. See classes such as **Circle** for example implementations.

Returns:

The minimal rectangle containing the shape drawn.

Reimplemented by: **touchgfx::AbstractShape::getMinimalRect**, **touchgfx::Circle::getMinimalRect**, **touchgfx::Line::getMinimalRect**

getPainter

virtual AbstractPainter & getPainter ( ) const

Gets the current painter for the **CanvasWidget**.

Returns:

The painter.

See also:

setPainter

getSolidRect

virtual Rect getSolidRect ( ) const

Gets the largest solid (non-transparent) rectangle.

Since canvas widgets typically do not have a solid rect, it is recommended to return an empty rectangle.

Returns:

The largest solid (non-transparent) rectangle.

NOTE

Function draw() might fail for some horizontal lines due to memory constraints. These lines will not be drawn and may cause strange display artifacts.

See also:

draw

Reimplements: **touchgfx::Drawable::getSolidRect**

invalidate

virtual void invalidate ( ) const

Invalidates the area covered by this **CanvasWidget**.

Since many widgets are a lot smaller than the actual size of the canvas widget, each widget must be able to tell the smallest rectangle completely containing the shape drawn by the widget. For example a circle arc is typically much smaller than the widget containing the circle.

See also:

getMinimalRect

Reimplements: **touchgfx::Drawable::invalidate**

resetMaxRenderLines

void resetMaxRenderLines ( )

Resets the maximum render lines.

The maximum render lines is decreates if the rendering buffer is found to be too small to render a complex outline. This is done to speed up subsequent draws by not having to draw the outline in vain (as was done previously) to force the outline to be drawn in smaller blocks. The **resetMaxRenderLines()** will try to render the entire outline in one go on the next call to **draw()**.

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setPainter

virtual void setPainter (AbstractPainter & painter)

Sets a painter for the **CanvasWidget**.

Parameters:

painter The painter for the **CanvasWidget**.

NOTE

If **setPainter()** is used to change the painter to a different painter, the area containing the **CanvasWidget** is not automatically invalidated.

See also:

getPainter

Reimplemented by: **touchgfx::AbstractGraphElementNoCWR::setPainter**

Version: 4.16

CanvasWidgetRenderer

Class for supporting drawing of figures. This class holds the memory which is used by the underlying algorithms. CanvasWidget will not allocate memory dynamically, but will use memory from the buffer passed to CanvasWidgetRenderer. When using the TouchGFX simulator, it is also possible to get a report on the actual amount of memory used for drawing with CanvasWidgetRenderer to help adjusting the buffer size.

See: Widget, setWriteMemoryUsageReport, getWriteMemoryUsageReport

Public Functions

unsigned	**getMissingBufferSize**()
	Calculate how much memory was required by CanvasWidgets, but was unavailable.
unsigned	**getUsedBufferSize**()
	Calculate how much memory has been required by CanvasWidgets.
bool	**getWriteMemoryUsageReport**()
	Gets write memory usage report flag.
void	**setupBuffer**(uint8_t * buffer, unsigned bufferSize)
	Setup the buffers used by **CanvasWidget**.
void	**setWriteMemoryUsageReport**(bool writeUsageReport)
	Memory reporting.

Public Functions Documentation

getMissingBufferSize

static unsigned getMissingBufferSize ( )

Calculate how much memory was required by CanvasWidgets, but was unavailable.

If the value returned is greater than 0 it means the This can be used to fine tune the size of the buffer passed to **CanvasWidgetRenderer** upon initialization.

Returns:

The number of bytes required.

getUsedBufferSize

static unsigned getUsedBufferSize ( )

Calculate how much memory has been required by CanvasWidgets.

This can be used to fine tune the size of the buffer passed to **CanvasWidgetRenderer** upon initialization.

Returns:

The number of bytes required.

getWriteMemoryUsageReport

static bool getWriteMemoryUsageReport ( )

Gets write memory usage report flag.

Returns:

true if it CWR writes memory reports, false if not.

setupBuffer

static void setupBuffer (uint8_t * buffer ,
unsigned bufferSize
)

Setup the buffers used by **CanvasWidget**.

Parameters:

buffer	Buffer reserved for **CanvasWidget**.
bufferSize	The size of the buffer.

setWriteMemoryUsageReport

static void setWriteMemoryUsageReport (bool writeUsageReport)

Memory reporting.

Memory reporting can be turned on (and off) using this method. CWR will try to work with the given amount of memory passed when calling **setupBuffer()**. If the outline of the figure is too complex, this will be reported.

"CWR requires X bytes" means that X bytes is the highest number of bytes required by CWR so far, but since the size of the invalidated area and the shape of things draw can influence this, this may be reported several times with a higher and higher number. Leave your app running for a long time to find out what the memory requirements are.

"CWR requires X bytes (Y bytes missing)" means the same as the report above, but there as was not enough memory to render the entire shape. To get around this, CWR will split the shape into two separate drawings of half size. This means that less memory is required, but drawing will be (somewhat) slower. After you see this message all future draws will be split into smaller chunks, so memory requirements might not get as high. This is followed by:

"CWR will split draw into multiple draws due to limited memory." actually just means that CWR will try to work with a smaller amount of memory.

In general, if there is enough memory available to run the simulation and never see the message "CWR will split draw ...", this is preferred. The size of the buffer required will be the highest number X reported as "CWR requires X bytes". Good numbers can also be around half of X.

Parameters:

`writeUsageReport` true to write report.

See also:

setupBuffer

Version: 4.16

Circle

Simple widget capable of drawing a circle, or part of a circle (an arc). The Circle can be filled or be drawn as a simple line along the circumference of the circle. Several parameters of the circle can be changed: Center, radius, line width, line cap, start angle and end angle.

Note:

- Since the underlying CanwasWidgetRenderer only supports straight lines, the circle is drawn using many small straight lines segments. The granularity can be adjusted to match the requirements - large circles need more line segments, small circles need fewer line segments, to look smooth and round.
- All circle parameters are internally handled as CWRUtil::Q5 which means that floating point values are rounded down to a fixed number of binary digits, for example:

```
Circle circle;
circle.setCircle(1.1f, 1.1f, 0.9); // Will use (35/32, 35/32, 28/32) = (1.09375f, 1.09375
int x, y, r;
circle.getCenter(&amp;x, &amp;y); // Will return (1, 1)
circle.getRadius(&amp;r); // Will return 0
```

.

Inherits from: CanvasWidget, Widget, Drawable

Public Functions

	Circle()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
template \<typename T \> void	**getArc**(T & startAngle, T & endAngle) const
	Gets the start and end angles in degrees for the circle arc.
int16_t	**getArcEnd**() const
	Gets the end angle in degrees for the arc.

template \<typename T \> void	**getArcEnd**(T & angle) const
	Gets the end angle in degrees for the arc.
int16_t	**getArcStart**() const
	Gets the start angle in degrees for the arc.
template \<typename T \> void	**getArcStart**(T & angle) const
	Gets the start angle in degrees for the arc.
int	**getCapPrecision**() const
	Gets the precision of the ends of the **Circle** arc.
template \<typename T \> void	**getCenter**(T & x, T & y) const
	Gets the center coordinates of the **Circle**.
template \<typename T \> void	**getLineWidth**(T & width) const
	Gets line width.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
Rect	**getMinimalRect**(**CWRUtil::Q5** arcStart, **CWRUtil::Q5** arcEnd) const
	Gets minimal rectangle containing a given circle arc using the set line width.
Rect	**getMinimalRect**(int16_t arcStart, int16_t arcEnd) const
	Gets minimal rectangle containing a given circle arc using the set line width.
int	**getPrecision**() const
	Gets the precision of the circle drawing function.
template \<typename T \> void	**getRadius**(T & r) const
	Gets the radius of the **Circle**.
void	**setArc**(const int16_t startAngle, const int16_t endAngle)
	Sets the start and end angles in degrees of the **Circle** arc.

template \<typename T \> void	**setArc**(const T startAngle, const T endAngle)
	Sets the start and end angles in degrees of the **Circle** arc.
void	**setCapPrecision**(const int precision)
	Sets the precision of the ends of the **Circle** arc.
void	**setCenter**(const int16_t x, const int16_t y)
	Sets the center of the **Circle**.
template \<typename T \> void	**setCenter**(const T x, const T y)
	Sets the center of the **Circle**.
void	**setCircle**(const int16_t x, const int16_t y, const int16_t r)
	Sets the center and radius of the **Circle**.
template \<typename T \> void	**setCircle**(const T x, const T y, const T r)
	Sets the center and radius of the **Circle**.
template \<typename T \> void	**setLineWidth**(const T width)
	Sets the line width for this **Circle**.
void	**setPixelCenter**(int x, int y)
	Sets the center of the circle / arc in the middle of a pixel.
void	**setPrecision**(const int precision)
	Sets precision of the **Circle** drawing function.
template \<typename T \> void	**setRadius**(const T r)
	Sets the radius of the **Circle**.
template \<typename T \> void	**updateArc**(const T startAngle, const T endAngle)
	Updates the start and end angle in degrees for this **Circle** arc.
template \<typename T \> void	**updateArcEnd**(const T endAngle)
	Updates the end angle in degrees for this **Circle** arc.

template \<typename T \> void	**updateArcStart**(const T startAngle)
	Updates the start angle in degrees for this **Circle** arc.

Protected Functions

void	**updateArc**(const **CWRUtil::Q5** setStartAngleQ5, const **CWRUtil::Q5** setEndAngleQ5)
	Updates the start and end angle in degrees for this **Circle** arc.

Additional inherited members

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const

	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const

	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

Circle

Circle ( )

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::CanvasWidget::drawCanvasWidget**

getArc

void getArc (T & startAngle , const
T & endAngle const
) const

Gets the start and end angles in degrees for the circle arc.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

startAngle The start angle rounded down to the precision of T.
endAngle The end angle rounded down to the precision of T.

See also:

setArc

getArcEnd

int16_t getArcEnd ( ) const

Gets the end angle in degrees for the arc.

Returns:

The end angle for the arc rounded down to an integer.

See also:

getArc, **setArc**

getArcEnd

void getArcEnd (T & angle)

Gets the end angle in degrees for the arc.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

angle The end angle rounded down to the precision of T.

getArcStart

int16_t getArcStart ( ) const

Gets the start angle in degrees for the arc.

Returns:

The starting angle for the arc rounded down to an integer.

See also:

getArc, **setArc**

getArcStart

void getArcStart (T & angle)

Gets the start angle in degrees for the arc.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

angle The starting angle rounded down to the precision of T.

See also:

getArc, **setArc**

getCapPrecision

int getCapPrecision ( ) const

Gets the precision of the ends of the **Circle** arc.

Returns:

The cap precision in degrees.

See also:

getCapPrecision

getCenter

void getCenter (T & x , const
T & y const
) const

Gets the center coordinates of the **Circle**.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of the center rounded down to the precision of T.
y The y coordinate of the center rounded down to the precision of T.

See also:

setCenter

getLineWidth

void getLineWidth (T & width)

Gets line width.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

width The line width rounded down to the precision of T.

See also:

setLineWidth

getMinimalRect

virtual Rect getMinimalRect ( ) const

Gets minimal rectangle containing the shape drawn by this widget.

Default implementation returns the size of the entire widget, but this function should be overwritten in subclasses and return the minimal rectangle containing the shape. See classes such as **Circle** for example implementations.

Returns:

The minimal rectangle containing the shape drawn.

Reimplements: **touchgfx::CanvasWidget::getMinimalRect**

getMinimalRect

Rect getMinimalRect (CWRUtil::Q5 arcStart , const
CWRUtil::Q5 arcEnd const
) const

Gets minimal rectangle containing a given circle arc using the set line width.

Parameters:

arcStart The arc start.
arcEnd The arc end.

Returns:

The minimal rectangle.

getMinimalRect

Rect getMinimalRect (int16_t arcStart , const
int16_t arcEnd const
) const

Gets minimal rectangle containing a given circle arc using the set line width.

Parameters:

arcStart The arc start.
arcEnd The arc end.

Returns:

The minimal rectangle.

getPrecision

```
int getPrecision (  ) const
```

Gets the precision of the circle drawing function.

The precision is the number of degrees used as step counter when drawing smaller line fragments around the circumference of the circle, the default being 5.

Returns:

The precision.

See also:

setPrecision

getRadius

```
void getRadius ( T & r )
```

Gets the radius of the **Circle**.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

r The radius rounded down to the precision of T.

setArc

```
void setArc ( const int16_t startAngle ,
              const int16_t endAngle
              )
```

Sets the start and end angles in degrees of the **Circle** arc.

0 degrees is straight up (12 o'clock) and 90 degrees is to the left (3 o'clock). Any positive or negative degrees can be used to specify the part of the **Circle** to draw.

Parameters:

startAngle The start degrees.
endAngle The end degrees.

NOTE

The area containing the Circle is not invalidated.

See also:

getArc, **updateArcStart**, **updateArcEnd**, **updateArc**

setArc

```
void setArc ( const T startAngle ,
              const T endAngle
              )
```

Sets the start and end angles in degrees of the **Circle** arc.

0 degrees is straight up (12 o'clock) and 90 degrees is to the left (3 o'clock). Any positive or negative degrees can be used to specify the part of the **Circle** to draw.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

startAngle The start degrees.
endAngle The end degrees.

NOTE

The area containing the Circle is not invalidated.

See also:

getArc, **updateArcStart**, **updateArcEnd**, **updateArc**

setCapPrecision

void setCapPrecision (const int precision)

Sets the precision of the ends of the **Circle** arc.

The precision is given in degrees where 180 is the default which results in a square ended arc (aka "butt cap"). 90 will draw "an arrow head" and smaller values gives a round cap. Larger values of precision results in faster rendering of the circle.

Parameters:

precision The new cap precision.

NOTE

The circle is not invalidated. The cap precision is not used if the circle is filled (if line width is zero) or when a full circle is drawn.

setCenter

void setCenter (const int16_t x ,
const int16_t y
)

Sets the center of the **Circle**.

Parameters:

x The x coordinate of center.
y The y coordinate of center.

NOTE

The area containing the Circle is not invalidated.

See also:

setRadius, **setCircle**, **getCenter**

setCenter

void setCenter (const T x ,
const T y
)

Sets the center of the **Circle**.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of center.
y The y coordinate of center.

NOTE

The area containing the Circle is not invalidated.

See also:

setRadius, **setCircle**, **getCenter**

setCircle

```
void setCircle ( const int16_t x ,
                 const int16_t y ,
                 const int16_t r
                 )
```

Sets the center and radius of the **Circle**.

Parameters:

x The x coordinate of center.
y The y coordinate of center.
r The radius.

NOTE

The area containing the Circle is not invalidated.

See also:

setCenter, **setRadius**

setCircle

```
void setCircle ( const T x ,
                 const T y ,
                 const T r
                 )
```

Sets the center and radius of the **Circle**.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of center.
y The y coordinate of center.
r The radius.

NOTE

The area containing the Circle is not invalidated.

See also:

setCenter, **setRadius**

setLineWidth

void setLineWidth (const T width)

Sets the line width for this **Circle**.

If the line width is set to zero, the circle will be filled.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

width The width of the line measured in pixels.

NOTE

The area containing the Circle is not invalidated. if the new line with is smaller than the old width, the circle should be invalidated before updating the width to ensure that the old circle is completely erased.

setPixelCenter

```
void setPixelCenter ( int x ,
                      int y
                      )
```

Sets the center of the circle / arc in the middle of a pixel.

Normally the coordinate is between pixel number x and x+1 horizontally and between pixel y and y+1 vertically. This function will set the center in the middle of the pixel by adding 0.5 to both x and y.

Parameters:

x The x coordinate of the center of the circle.
y The y coordinate of the center of the circle.

setPrecision

void setPrecision (const int precision)

Sets precision of the **Circle** drawing function.

The number given as precision is the number of degrees used as step counter when drawing the line fragments around the circumference of the circle, five being a reasonable value. Higher values results in less nice circles but faster rendering and possibly sufficient for very small circles. Large circles might require a precision smaller than five to make the edge of the circle look nice and smooth.

Parameters:

precision The precision measured in degrees.

NOTE

The circle is not invalidated.

setRadius

void setRadius (const T r)

Sets the radius of the **Circle**.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

r The radius.

NOTE

The area containing the **Circle** is not invalidated.

See also:

setCircle, **setCenter**, **getRadius**

updateArc

```
void updateArc ( const T startAngle ,
                 const T endAngle
                 )
```

Updates the start and end angle in degrees for this **Circle** arc.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

startAngle The new start angle in degrees.
endAngle The new end angle in degrees.

NOTE

The areas containing the updated **Circle** arcs are invalidated. As little as possible will be invalidated for best performance.

See also:

setArc, **getArc**, **updateArcStart**, **updateArcEnd**

updateArcEnd

```
void updateArcEnd ( const T endAngle )
```

Updates the end angle in degrees for this **Circle** arc.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

endAngle The end angle in degrees.

NOTE

The area containing the updated **Circle** arc is invalidated.

See also:

setArc, **updateArcStart**, **updateArc**

updateArcStart

void updateArcStart (const T startAngle)

Updates the start angle in degrees for this **Circle** arc.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

startAngle The start angle in degrees.

NOTE

The area containing the updated **Circle** arc is invalidated.

See also:

setArc, **updateArcEnd**, **updateArc**

Protected Functions Documentation

updateArc

void updateArc (const CWRUtil::Q5 setStartAngleQ5 ,
const CWRUtil::Q5 setEndAngleQ5
)

Updates the start and end angle in degrees for this **Circle** arc.

Parameters:

setStartAngleQ5 The new start angle in degrees.
setEndAngleQ5 The new end angle in degrees.

NOTE

The areas containing the updated **Circle** arcs are invalidated. As little as possible will be invalidated for best performance.

See also:

setArc, **getArc**, **updateArcStart**, **updateArcEnd**

Version: 4.16

CircleProgress

A circle progress indicator uses CanvasWidgetRenderer for drawing the arc of a Circle to show progress. This means that the user must create a painter for painting the circle. The circle progress is defined by setting the minimum and maximum angle of the arc.

Note: As CircleProgress uses CanvasWidgetRenderer, it is important that a buffer is set up by calling CanvasWidgetRendere::setBuffer().

Inherits from: AbstractProgressIndicator, Container, Drawable

Public Functions

	CircleProgress()
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual int	**getCapPrecision**() const
	Gets the cap precision.
virtual void	**getCenter**(int & x, int & y) const
	Gets the circle center coordinates.
virtual int	**getEndAngle**() const
	Gets end angle.
virtual int	**getLineWidth**() const
	Gets line width.
virtual int	**getRadius**() const
	Gets the radius of the circle.
virtual int	**getStartAngle**() const
	Gets start angle.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setCapPrecision**(int precision)

	Sets the cap precision of end of the circle arc.
virtual void	**setCenter**(int x, int y)
	Sets the center of the circle / arc.
virtual void	**setLineWidth**(int width)
	Sets line width of the circle.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets the painter to use for drawing the circle progress.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the actual progress indicator relative to the background image.
virtual void	**setRadius**(int r)
	Sets the radius of the circle.
virtual void	**setStartEndAngle**(int startAngle, int endAngle)
	Sets start and end angle.
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.

Protected Attributes

Circle	**circle**
	The circle.
int	**circleEndAngle**
	The end angle.

Additional inherited members

Public Functions inherited from AbstractProgressIndicator

	AbstractProgressIndicator()
	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.

virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.
void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)

	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.
int	**rangeMin**
	The range minimum.

uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)

	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.

virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)

	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**

	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

CircleProgress

CircleProgress ( )

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getCapPrecision

virtual int getCapPrecision ( ) const

Gets the cap precision.

Returns:

The cap precision.

See also:

setCapPrecision

getCenter

```
virtual void getCenter ( int & x ,      const
                         int & y        const
                         )       const
```

Gets the circle center coordinates.

Parameters:

x The x coordinate of the center of the circle.
y The y coordinate of the center of the circle.

getEndAngle

```
virtual int getEndAngle (  ) const
```

Gets end angle.

Beware that the value returned is not related to the current progress of the circle but rather the end point of the circle when it is at 100%.

Returns:

The end angle.

See also:

setStartEndAngle

getLineWidth

```
virtual int getLineWidth (  ) const
```

Gets line width.

Returns:

The line width.

See also:

setLineWidth

getRadius

virtual int getRadius ( ) const

Gets the radius of the circle.

Returns:

The radius.

getStartAngle

virtual int getStartAngle ( ) const

Gets start angle.

Returns:

The start angle.

See also:

setStartEndAngle, **getEndAngle**

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.The alpha can also be set on the Painter, but this can be controlled directly from the user app, setting alpha for the CircleProgress will set the alpha of the actual circle.

See also:

getAlpha

setCapPrecision

virtual void setCapPrecision (int precision)

Sets the cap precision of end of the circle arc.

This is not used if line width is zero.

Parameters:

precision The cap precision.

See also:

Circle::setCapPrecision, **getCapPrecision**

setCenter

virtual void setCenter (int x ,
int y
)

Sets the center of the circle / arc.

Parameters:

x The x coordinate of the center of the circle.
y The y coordinate of the center of the circle.

setLineWidth

virtual void setLineWidth (int width)

Sets line width of the circle.

If a line width of zero is specified, it has a special meaning of drawing a filled circle (with the set radius) instead of just the circle arc.

Parameters:

width The width of the line (0 produces a filled circle with the given radius).

See also:

Circle::setLineWidth, **setRadius**

setPainter

virtual void setPainter (AbstractPainter & painter)

Sets the painter to use for drawing the circle progress.

Parameters:

painter The painter.

See also:

Circle::setPainter, **AbstractPainter**

setProgressIndicatorPosition

virtual void setProgressIndicatorPosition (int16_t x ,
int16_t y ,
int16_t width ,
int16_t height
)

Sets the position and dimensions of the actual progress indicator relative to the background image.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the box progress indicator.
height	The height of the box progress indicator.

See also:

getProgressIndicatorX, **getProgressIndicatorY**, **getProgressIndicatorWidth**, **getProgressIndicatorHeight**

Reimplements: **touchgfx::AbstractProgressIndicator::setProgressIndicatorPosition**

setRadius

virtual void setRadius (int r)

Sets the radius of the circle.

Parameters:

r The radius.

See also:

Circle::setRadius

setStartEndAngle

```
virtual void setStartEndAngle ( int startAngle ,
                                int endAngle
                                )
```

Sets start and end angle.

By swapping end and start angles, circles can progress backwards.

Parameters:

startAngle The start angle.
endAngle The end angle.

setValue

```
virtual void setValue ( int value )
```

Sets the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. If a callback function has been set using setValueSetAction, that callback will be called (unless the new value is the same as the current value).

Parameters:

value The value.

NOTE

if value is equal to the current value, nothing happens, and the callback will not be called.

See also:

getValue, **updateValue**, **setValueSetAction**

Reimplements: **touchgfx::AbstractProgressIndicator::setValue**

Protected Attributes Documentation

circle

Circle circle

The circle.

circleEndAngle

int circleEndAngle

The end angle.

Version: 4.16

ClickButtonTrigger

A click button trigger. This trigger will create a button that reacts on clicks. This means it will call the set action when it gets a touch released event. The ClickButtonTrigger can be combined with one or more of the ButtonStyle classes to create a fully functional button.

See: TouchButtonTrigger

Inherits from: AbstractButtonContainer, Container, Drawable

Public Functions

virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Handles a ClickAvent.

Additional inherited members

Public Functions inherited from AbstractButtonContainer

	AbstractButtonContainer()
virtual void	**executeAction**()
	Executes the previously set action.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
bool	**getPressed**()
	Gets the pressed state.
void	**setAction**(**GenericCallback**< const **AbstractButtonContainer** & > & callback)
	Sets an action callback to be executed by the subclass of AbstractContainerButton.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setPressed**(bool isPressed)

	Sets the pressed state to the given state.

Protected Functions inherited from AbstractButtonContainer

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes inherited from AbstractButtonContainer

GenericCallback< const **AbstractButtonContainer** & > *	**action**
	The action to be executed.
uint8_t	**alpha**
	The current alpha value. 255 denotes solid, 0 denotes completely invisible.
bool	**pressed**
	True if pressed.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()

	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()

	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.

int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.

void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.

virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Handles a ClickAvent.

The action callback is called when the **ClickButtonTrigger** receives a **ClickEvent::RELEASED** event in PRESSED state. Function **setPressed()** will be called with the new button state.

Parameters:

event The click event.

See also:

setAction, **setPressed**, **getPressed**

Reimplements: **touchgfx::Drawable::handleClickEvent**

Version: 4.16

ClickEvent

A click event. The semantics of this event is slightly depending on hardware platform. ClickEvents are generated by the HAL layer.

See: Event

Inherits from: Event

Public Types

enum	**ClickEventType** { PRESSED, RELEASED, CANCEL }
	Values that represent click event types.

Public Functions

	ClickEvent(**ClickEventType** type, int16_t x, int16_t y, int16_t force =0)
	Initializes a new instance of the **ClickEvent** class.
virtual **Event::EventType**	**getEventType**()
	Gets event type.
int16_t	**getForce**() const
	Gets the force of the click.
ClickEventType	**getType**() const
	Gets the click type of this event.
int16_t	**getX**() const
	Gets the x coordinate of this event.
int16_t	**getY**() const
	Gets the y coordinate of this event.
void	**setType**(**ClickEventType** type)
	Sets the click type of this event.
void	**setX**(int16_t x)

	Sets the x coordinate of this event.
void	**setY**(int16_t y)
	Sets the y coordinate of this event.

Additional inherited members

Public Types inherited from Event

enum	**EventType** { EVENT_CLICK, EVENT_DRAG, EVENT_GESTURE }
	The event types.

Public Functions inherited from Event

virtual	**~Event**()
	Finalizes an instance of the **Event** class.

Public Types Documentation

ClickEventType

enum ClickEventType

Values that represent click event types.

PRESSED	An enum constant representing the pressed option.
RELEASED	An enum constant representing the released option.
CANCEL	An enum constant representing the cancel option.

Public Functions Documentation

ClickEvent

ClickEvent (ClickEventType type ,

```
int16_t x ,
int16_t y ,
int16_t force =0
)
```

Initializes a new instance of the **ClickEvent** class.

Parameters:

type	The type of the click event.
x	The x coordinate of the click event.
y	The y coordinate of the click event.
force	(Optional) The force of the click. On touch displays this usually means how hard the user pressed on the display. On the windows platform, this will always be zero.

getEventType

virtual Event::EventType getEventType ( )

Gets event type.

Returns:

The type of this event.

Reimplements: **touchgfx::Event::getEventType**

getForce

int16_t getForce ( ) const

Gets the force of the click.

On touch displays this usually means how hard the user pressed on the display. On the windows platform, this will always be zero.

Returns:

The force of the click.

getType

ClickEventType getType ( ) const

Gets the click type of this event.

Returns:

The click type of this event.

getX

int16_t getX ( ) const

Gets the x coordinate of this event.

Returns:

The x coordinate of this event.

getY

int16_t getY ( ) const

Gets the y coordinate of this event.

Returns:

The y coordinate of this event.

setType

void setType (ClickEventType type)

Sets the click type of this event.

Parameters:

type The type to set.

setX

void setX (int16_t x)

Sets the x coordinate of this event.

Parameters:

x The x coordinate of this event.

setY

void setY (int16_t y)

Sets the y coordinate of this event.

Parameters:

y The y coordinate of this event.

Version: 4.16

ClickListener

Mix-in class that extends a class with a click action event that is called when the class receives a click event.

Template Parameters:

- **T** specifies the type to extend with the ClickListener behavior.

Inherits from: T

Public Functions

	ClickListener()
	Initializes a new instance of the **ClickListener** class.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Ensures that the clickEvent is propagated to the super class T and to the clickAction listener.
void	**setClickAction**(**GenericCallback**< const T <, const **ClickEvent** & > & callback)
	Associates an action to be performed when the class T is clicked.

Protected Attributes

GenericCallback< const T <, const **ClickEvent** & > *	**clickAction**
	The callback to be executed when T is clicked.

Public Functions Documentation

ClickListener

ClickListener ( )

Initializes a new instance of the **ClickListener** class.

Make the object touchable.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Ensures that the clickEvent is propagated to the super class T and to the clickAction listener.

Parameters:

event Information about the click.

setClickAction

void setClickAction (GenericCallback< const T <, const ClickEvent & > & callback)

Associates an action to be performed when the class T is clicked.

Parameters:

callback The callback to be executed. The callback will be given a reference to T.

Protected Attributes Documentation

clickAction

GenericCallback< const T <, const ClickEvent & > * clickAction

The callback to be executed when T is clicked.

Version: 4.16

Color

Contains functionality for color conversion.

Public Functions

FORCE_INLINE_FUNCTION uint8_t	**getBlueColor**(**colortype** color)
	Gets the blue color part of a color.
colortype	**getColorFrom24BitHSL**(uint8_t hue, uint8_t saturation, uint8_t luminance)
	Convert a given color from HSV (Hue, Saturation, Value) to colortype.
colortype	**getColorFrom24BitHSV**(uint8_t hue, uint8_t saturation, uint8_t value)
	Convert a given color from HSV (Hue, Saturation, Value) to colortype.
colortype	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Generates a color representation to be used on the LCD, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint8_t	**getGreenColor**(**colortype** color)
	Gets the green color part of a color.
void	**getHSLFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue, uint8_t & hue, uint8_t & saturation, uint8_t & luminance)
	Convert a given color from RGB (Red, Green, Blue) to HSV (Hue, Saturation, Value).
void	**getHSLFromColor**(**colortype** color, uint8_t & hue, uint8_t & saturation, uint8_t & luminance)
	Convert a given colortype color to HSV (Hue, Saturation, Value).
void	**getHSLFromHSV**(uint8_t hue, uint8_t & saturation, uint8_t value, uint8_t & luminance)
	Convert HSV (Hue, Saturation, Value) to HSL (Hue, Saturation, Luminance).

void	**getHSVFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue, uint8_t & hue, uint8_t & saturation, uint8_t & value)
	Convert a given color from RGB (Red, Green, Blue) to HSV (Hue, Saturation, Value).
void	**getHSVFromColor**(**colortype** color, uint8_t & hue, uint8_t & saturation, uint8_t & value)
	Convert a given colortype color to HSV (Hue, Saturation, Value).
void	**getHSVFromHSL**(uint8_t hue, uint8_t & saturation, uint8_t luminance, uint8_t & value)
	Convert HSL (Hue, Saturation, Luminance) to HSV (Hue, Saturation, Value).
FORCE_INLINE_FUNCTION uint8_t	**getRedColor**(**colortype** color)
	Gets the red color part of a color.
void	**getRGBFrom24BitHSL**(uint8_t hue, uint8_t saturation, uint8_t luminance, uint8_t & red, uint8_t & green, uint8_t & blue)
	Convert a given color from HSV (Hue, Saturation, Value) to RGB (Red, Green, Blue).
void	**getRGBFrom24BitHSV**(uint8_t hue, uint8_t saturation, uint8_t value, uint8_t & red, uint8_t & green, uint8_t & blue)
	Convert a given color from HSV (Hue, Saturation, Value) to RGB (Red, Green, Blue).

Public Functions Documentation

getBlueColor

static FORCE_INLINE_FUNCTION uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible. If the color depth of the display is known, consider using function getBlueFromColor() from the current **LCD**.

Parameters:

color The color value.

Returns:

The blue part of the color.

See also:

LCD16bpp::getBlueFromColor

getColorFrom24BitHSL

```
static colortype getColorFrom24BitHSL ( uint8_t hue ,
                                        uint8_t saturation ,
                                        uint8_t luminance
                                        )
```

Convert a given color from HSV (Hue, Saturation, Value) to colortype.

Parameters:

hue	The input Hue (0 to 255).
saturation	The input Saturation (0 to 255).
luminance	The input Value (0 to 255).

Returns:

The colortype color.

NOTE

The conversion is an approximation.

getColorFrom24BitHSV

```
static colortype getColorFrom24BitHSV ( uint8_t hue ,
                                        uint8_t saturation ,
                                        uint8_t value
                                        )
```

Convert a given color from HSV (Hue, Saturation, Value) to colortype.

Parameters:

hue	The input Hue (0 to 255).
saturation	The input Saturation (0 to 255).
value	The input Value (0 to 255).

Returns:

The colortype color.

NOTE

The conversion is an approximation.

getColorFrom24BitRGB

```
static colortype getColorFrom24BitRGB ( uint8_t red ,
                                        uint8_t green ,
                                        uint8_t blue
                                        )
```

Generates a color representation to be used on the LCD, based on 24 bit RGB values.

Depending on your display color bit depth, the color might be interpreted internally as fewer than 24 bits with a loss of color precision.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

NOTE

This function is not available to call before the LCD has been setup, because the color depth is required. Consider using the function getColorFromRGB for a specific class, e.g. LCD16::getColorFromRGB().

See also:

LCD::getColorFrom24BitRGB, **LCD16bpp::getColorFromRGB**

getGreenColor

```
static FORCE_INLINE_FUNCTION uint8_t getGreenColor ( colortype color )
```

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible. If the color depth of the display is known, consider using function getGreenFromColor() from the current **LCD**.

Parameters:

color The color value.

Returns:

The green part of the color.

See also:

LCD16bpp::getGreenFromColor

getHSLFrom24BitRGB

```
static void getHSLFrom24BitRGB ( uint8_t    red ,
                                 uint8_t    green ,
                                 uint8_t    blue ,
                                 uint8_t & hue ,
                                 uint8_t & saturation ,
                                 uint8_t & luminance
                                 )
```

Convert a given color from RGB (Red, Green, Blue) to HSV (Hue, Saturation, Value).

Parameters:

red	The input Red (0 to 255).
green	The input Green (0 to 255).
blue	The input Blue (0 to 255).
hue	The output Hue (0 to 255).
saturation	The output Saturation (0 to 255).
luminance	The output Value (0 to 255).

NOTE

The conversion is an approximation.

getHSLFromColor

```
static void getHSLFromColor ( colortype color ,
                              uint8_t & hue ,
                              uint8_t & saturation ,
                              uint8_t & luminance
                              )
```

Convert a given colortype color to HSV (Hue, Saturation, Value).

Parameters:

color	The input color.
hue	The output Hue (0 to 255).
saturation	The output Saturation (0 to 255).
luminance	The output Value (0 to 255).

NOTE

The conversion is an approximation.

getHSLFromHSV

```
static void getHSLFromHSV ( uint8_t     hue ,
                            uint8_t & saturation ,
                            uint8_t     value ,
                            uint8_t & luminance
                            )
```

Convert HSV (Hue, Saturation, Value) to HSL (Hue, Saturation, Luminance).

The Hue is unaltered, the Saturation is changed and the Luminance is calculated.

Parameters:

hue	The hue (0 to 255).
saturation	The saturation (0 to 255).
value	The value (0 to 255).
luminance	The luminance (0 to 255).

getHSVFrom24BitRGB

```
static void getHSVFrom24BitRGB ( uint8_t     red ,
                                 uint8_t     green ,
                                 uint8_t     blue ,
                                 uint8_t & hue ,
                                 uint8_t & saturation ,
                                 uint8_t & value
                                 )
```

Convert a given color from RGB (Red, Green, Blue) to HSV (Hue, Saturation, Value).

Parameters:

red	The input Red.
green	The input Green.
blue	The input Blue.

hue	The output Hue.
saturation	The output Saturation.
value	The output Value.

NOTE

The conversion is an approximation.

getHSVFromColor

```
static void getHSVFromColor ( colortype color ,
                              uint8_t & hue ,
                              uint8_t & saturation ,
                              uint8_t & value
                              )
```

Convert a given colortype color to HSV (Hue, Saturation, Value).

Parameters:

color	The input color.
hue	The output Hue (0 to 255).
saturation	The output Saturation (0 to 255).
value	The output Value (0 to 255).

NOTE

The conversion is an approximation.

getHSVFromHSL

```
static void getHSVFromHSL ( uint8_t    hue ,
                            uint8_t & saturation ,
                            uint8_t    luminance ,
                            uint8_t & value
                            )
```

Convert HSL (Hue, Saturation, Luminance) to HSV (Hue, Saturation, Value).

The Hue is unaltered, the Saturation is changed and the Value is calculated.

Parameters:

hue	The hue (0 to 255).
saturation	The saturation (0 to 255).

luminance	The luminance (0 to 255).
value	The value (0 to 255).

getRedColor

static FORCE_INLINE_FUNCTION uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible. If the color depth of the display is known, consider using function getRedFromColor() from the current **LCD**.

Parameters:

color	The color value.

Returns:

The red part of the color.

See also:

LCD16bpp::getRedFromColor

getRGBFrom24BitHSL

```
static void getRGBFrom24BitHSL ( uint8_t    hue ,
                                 uint8_t    saturation ,
                                 uint8_t    luminance ,
                                 uint8_t &  red ,
                                 uint8_t &  green ,
                                 uint8_t &  blue
                                 )
```

Convert a given color from HSV (Hue, Saturation, Value) to RGB (Red, Green, Blue).

Parameters:

hue	The input Hue (0 to 255).
saturation	The input Saturation (0 to 255).
luminance	The input Value (0 to 255).
red	The output Red (0 to 255).
green	The output Green (0 to 255).
blue	The output Blue (0 to 255).

NOTE

The conversion is an approximation.

getRGBFrom24BitHSV

```
static void getRGBFrom24BitHSV ( uint8_t    hue ,
                                 uint8_t    saturation ,
                                 uint8_t    value ,
                                 uint8_t & red ,
                                 uint8_t & green ,
                                 uint8_t & blue
                                 )
```

Convert a given color from HSV (Hue, Saturation, Value) to RGB (Red, Green, Blue).

Parameters:

hue	The input Hue (0 to 255).
saturation	The input Saturation (0 to 255).
value	The input Value (0 to 255).
red	The output Red (0 to 255).
green	The output Green (0 to 255).
blue	The output Blue (0 to 255).

NOTE

The conversion is an approximation.

Version: 4.16

colortype

This type can contain a color value. Note that in order to maintain backwards compatibility, casting this type to an integral value will yield a 16-bit value. To extract a 24/32-bit color from this type, use the getColor32 function.

Public Functions

	colortype()
	Default constructor.
	colortype(uint32_t col)
	Constructor which creates a colortype with the given color.
FORCE_INLINE_FUNCTION uint32_t	**getColor32**() const
	Gets color as a 32bit value suitable for passing to **Color::getRedColor()**, **Color::getGreenColor()** and **Color::getBlueColor()** which will handle all bitdeptchs.
	operator uint32_t() const
	Cast that converts the given colortype to an uint32_t.

Public Attributes

uint32_t	**color**
	The color.

Public Functions Documentation

colortype

colortype ( )

Default constructor.

Creates a black (0) color.

colortype

colortype (uint32_t col)

Constructor which creates a colortype with the given color.

Use **Color::getColorFrom24BitRGB()** to create a color that will work on your selected **LCD** type.

Parameters:

col The color.

See also:

Color::getColorFrom24BitRGB

getColor32

FORCE_INLINE_FUNCTION uint32_t getColor32 ( ) const

Gets color as a 32bit value suitable for passing to **Color::getRedColor()**, **Color::getGreenColor()** and **Color::getBlueColor()** which will handle all bitdeptchs.

Returns:

The color 32.

See also:

Color::getRedColor, **Color::getGreenColor**, **Color::getBlueColor**

operator uint32_t

operator uint32_t ( ) const

Cast that converts the given colortype to an uint32_t.

Returns:

The result of the operation.

Public Attributes Documentation

color

uint32_t color

The color.

Version: 4.16

ConstFont

A ConstFont is a Font implementation that has its contents defined at compile-time and usually placed in read-only memory.

See: Font

Note: Pure virtual class. Create an application-specific implementation of getPixelData().

Inherits from: Font

Inherited by: InternalFlashFont

Public Functions

	ConstFont(const **GlyphNode** * list, uint16_t size, uint16_t height, uint8_t pixBelowBase, uint8_t bitsPerPixel, uint8_t byteAlignRow, uint8_t maxLeft, uint8_t maxRight, const **Unicode::UnicodeChar** fallbackChar, const **Unicode::UnicodeChar** ellipsisChar)
	Initializes a new instance of the **ConstFont** class.
const **GlyphNode** *	**find**(**Unicode::UnicodeChar** unicode) const
	Finds the glyph data associated with the specified unicode.
const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode)
	Gets the glyph data associated with the specified Unicode.
virtual const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode, const uint8_t *& pixelData, uint8_t & bitsPerPixel) const
	Gets the glyph data associated with the specified Unicode.
const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode, const uint8_t *& pixelData, uint8_t & bitsPerPixel)
	Gets the glyph data associated with the specified Unicode.
virtual int8_t	**getKerning**(**Unicode::UnicodeChar** prevChar, const **GlyphNode** * glyph) const =0
	Gets the kerning distance between two characters.
virtual const uint8_t *	**getPixelData**(const **GlyphNode** * glyph) const =0
	Gets the pixel date associated with this glyph.

Protected Attributes

const **GlyphNode** *	**glyphList**
	The list of glyphs.
uint16_t	**listSize**
	The size of the list of glyphs.

Additional inherited members

Public Functions inherited from Font

virtual FORCE_INLINE_FUNCTION uint8_t	**getBitsPerPixel**() const
	Gets bits per pixel for this font.
virtual FORCE_INLINE_FUNCTION uint8_t	**getByteAlignRow**() const
	Are the glyphs saved with each glyph row byte aligned?
virtual uint16_t	**getCharWidth**(const **Unicode::UnicodeChar** c) const
	Gets the width in pixels of the specified character.
virtual **Unicode::UnicodeChar**	**getEllipsisChar**() const
	Gets ellipsis character for the given font.
virtual **Unicode::UnicodeChar**	**getFallbackChar**() const
	Gets fallback character for the given font.
virtual FORCE_INLINE_FUNCTION uint16_t	**getFontHeight**() const
	Returns the height in pixels of this font.
virtual const uint16_t *	**getGSUBTable**() const
	Gets GSUB table.
FORCE_INLINE_FUNCTION uint8_t	**getMaxPixelsLeft**() const
	Gets maximum pixels left of any glyph in the font.
FORCE_INLINE_FUNCTION uint8_t	**getMaxPixelsRight**() const
	Gets maximum pixels right of any glyph in the font.

virtual uint16_t	**getMaxTextHeight**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the height of the highest character in a given string.
virtual FORCE_INLINE_FUNCTION uint16_t	**getMinimumTextHeight**() const
	Returns the minimum height needed for a text field that uses this font.
virtual uint16_t	**getNumberOfLines**(const **Unicode::UnicodeChar** * text, ...) const
	Count the number of lines in a given text.
virtual uint8_t	**getSpacingAbove**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the number of blank pixels at the top of the given text.
virtual uint16_t	**getStringWidth**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the width in pixels of the specified string.
virtual uint16_t	**getStringWidth**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, ...) const
	Gets the width in pixels of the specified string.
virtual	**~Font**()
	Finalizes an instance of the **Font** class.
FORCE_INLINE_FUNCTION bool	**isInvisibleZeroWidth**(**Unicode::UnicodeChar** character)
	Query if 'character' is invisible, zero width.

Protected Functions inherited from Font

	Font(uint16_t height, uint8_t pixBelowBase, uint8_t bitsPerPixel, uint8_t byteAlignRow, uint8_t maxLeft, uint8_t maxRight, const **Unicode::UnicodeChar** fallbackChar, const **Unicode::UnicodeChar** ellipsisChar)
	Initializes a new instance of the **Font** class.
uint16_t	**getStringWidthLTR**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, va_list pArg) const
	Gets the width in pixels of the specified string.
uint16_t	**getStringWidthRTL**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, va_list pArg) const

Gets the width in pixels of the specified string.

Protected Attributes inherited from Font

uint8_t	**bAlignRow**
	The glyphs are saved with each row byte aligned.
uint8_t	**bPerPixel**
	The number of bits per pixel.
Unicode::UnicodeChar	**ellipsisCharacter**
	The ellipsis character used for truncating long texts.
Unicode::UnicodeChar	**fallbackCharacter**
	The fallback character to use when no glyph exists for the wanted character.
uint16_t	**fontHeight**
	The font height in pixels.
uint8_t	**maxPixelsLeft**
	The maximum number of pixels a glyph extends to the left.
uint8_t	**maxPixelsRight**
	The maximum number of pixels a glyph extends to the right.
uint8_t	**pixelsBelowBaseline**
	The number of pixels below the base line.

Public Functions Documentation

ConstFont

```
ConstFont ( const GlyphNode *  list ,
            uint16_t           size ,
            uint16_t           height ,
            uint8_t            pixBelowBase ,
            uint8_t            bitsPerPixel ,
            uint8_t            byteAlignRow ,
            uint8_t            maxLeft ,
            uint8_t            maxRight ,
```

```
        const Unicode::UnicodeChar fallbackChar ,
        const Unicode::UnicodeChar ellipsisChar
        )
```

Initializes a new instance of the **ConstFont** class.

Parameters:

list	The array of glyphs known to this font.
size	The number of glyphs in list.
height	The height in pixels of the highest character in this font.
pixBelowBase	The maximum number of pixels that can be drawn below the baseline in this font.
bitsPerPixel	The number of bits per pixel in this font.
byteAlignRow	The glyphs are saved with each row byte aligned.
maxLeft	The maximum a character extends to the left.
maxRight	The maximum a character extends to the right.
fallbackChar	The fallback character for the typography in case no glyph is available.
ellipsisChar	The ellipsis character used for truncating long texts.

find

```
const GlyphNode * find ( Unicode::UnicodeChar unicode )
```

Finds the glyph data associated with the specified unicode.

Parameters:

unicode	The character to look up.

Returns:

A pointer to the glyph node or null if the glyph was not found.

getGlyph

```
const GlyphNode * getGlyph ( Unicode::UnicodeChar unicode )
```

Gets the glyph data associated with the specified Unicode.

Please note that in case of Thai letters and Arabic letters where diacritics can be placed relative to the previous character(s), please use **TextProvider::getNextLigature()** instead as it will create a temporary **GlyphNode** that will be adjusted with respect to X/Y position.

Parameters:

unicode	The character to look up.

Returns:

A pointer to the glyph node or null if the glyph was not found.

See also:

TextProvider::getNextLigature

getGlyph

```
virtual const GlyphNode * getGlyph ( Unicode::UnicodeChar unicode ,
                                     const uint8_t *& pixelData ,
                                     uint8_t & bitsPerPixel
                                     ) const
```

Gets the glyph data associated with the specified Unicode.

Please note that in case of Thai letters and Arabic letters where diacritics can be placed relative to the previous character(s), please use **TextProvider::getNextLigature()** instead as it will create a temporary **GlyphNode** that will be adjusted with respect to X/Y position.

Parameters:

unicode	The character to look up.
pixelData	Pointer to the pixel data for the glyph if the glyph is found. This is set by this method.
bitsPerPixel	Reference where to place the number of bits per pixel.

Returns:

A pointer to the glyph node or null if the glyph was not found.

Reimplements: **touchgfx::Font::getGlyph**

getGlyph

```
const GlyphNode * getGlyph ( Unicode::UnicodeChar unicode ,
                             const uint8_t *& pixelData ,
                             uint8_t & bitsPerPixel
                             )
```

Gets the glyph data associated with the specified Unicode.

Please note that in case of Thai letters and Arabic letters where diacritics can be placed relative to the previous character(s), please use **TextProvider::getNextLigature()** instead as it will create a temporary **GlyphNode** that will be adjusted with respect to X/Y position.

Parameters:

unicode	The character to look up.
pixelData	Pointer to the pixel data for the glyph if the glyph is found. This is set by this method.
bitsPerPixel	Reference where to place the number of bits per pixel.

Returns:

A pointer to the glyph node or null if the glyph was not found.

getKerning

```
virtual int8_t getKerning ( Unicode::UnicodeChar prevChar ,  const =0
                            const GlyphNode *     glyph      const =0
                            )                     const =0
```

Gets the kerning distance between two characters.

Parameters:

prevChar	The **Unicode** value of the previous character.
glyph	the glyph object for the current character.

Returns:

The kerning distance between prevChar and glyph char.

Reimplements: **touchgfx::Font::getKerning**

Reimplemented by: **touchgfx::InternalFlashFont::getKerning**

getPixelData

```
virtual const uint8_t * getPixelData ( const GlyphNode * glyph )
```

Gets the pixel date associated with this glyph.

Parameters:

glyph	The glyph to get the pixels data from.

Returns:

Pointer to the pixel data of this glyph.

Reimplemented by: **touchgfx::InternalFlashFont::getPixelData**

Protected Attributes Documentation

glyphList

const GlyphNode * glyphList

The list of glyphs.

listSize

uint16_t listSize

The size of the list of glyphs.

Version: 4.16

Container

A Container is a Drawable that can have child nodes. The z-order of children is determined by the order in which Drawables are added to the container - the Drawable added last will be front-most on the screen.

This class overrides a few functions in Drawable in order to traverse child nodes.

Note that containers act as view ports - that is, only the parts of children that intersect with the geometry of the container will be visible (e.g. setting a container's width to 0 will render all children invisible).

See: Drawable

Inherits from: Drawable

Inherited by: MoveAnimator< touchgfx::Container >, AbstractButtonContainer, AbstractClock, AbstractDataGraph, AbstractProgressIndicator, CacheableContainer, DrawableList, Keyboard, ListLayout, ModalWindow, ScrollableContainer, ScrollBase, SlideMenu, Slider, SwipeContainer, ZoomAnimationImage

Public Functions

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Additional inherited members

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.

Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

add

virtual void add (Drawable & d)

Adds a Drawable instance as child to this **Container**.

The **Drawable** added will be placed as the element to be drawn last, and thus appear on top of all previously added drawables in the **Container**.

Parameters:

d The **Drawable** to add.

NOTE

Never add a drawable more than once!

Reimplemented by: **touchgfx::ListLayout::add**, **touchgfx::ModalWindow::add**, **touchgfx::ScrollableContainer::add**, **touchgfx::SlideMenu::add**, **touchgfx::SwipeContainer::add**

Container

Container ( )

contains

virtual bool contains (const Drawable & d)

Query if a given Drawable has been added directly to this **Container**.

The search is not done recursively.

Parameters:

d The **Drawable** to look for.

Returns:

True if the specified **Drawable** instance is direct child of this container, false otherwise.

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

Reimplemented by: **touchgfx::Keyboard::draw**

forEachChild

virtual void forEachChild (GenericCallback< Drawable & > * function)

Executes the specified callback function for each child in the **Container**.

The callback to execute must have the following prototype: void T::func(**Drawable**&)

Parameters:

function The function to be executed for each child.

getFirstChild

virtual Drawable * getFirstChild ( )

Obtain a pointer to the first child of this container.

The first child is the **Drawable** drawn first, and therefore the **Drawable***behind* all other children of this **Container**. Useful if you want to manually iterate the children added to this container.

Returns:

Pointer to the first drawable added to this container. If nothing has been added return zero.

See also:

getNextSibling

Reimplements: **touchgfx::Drawable::getFirstChild**

getLastChild

```
virtual void getLastChild ( int16_t      x ,
                            int16_t      y ,
                            Drawable ** last
                            )
```

Gets the last child in the list of children in this **Container**.

If this **Container** is touchable (**isTouchable()**), it will be passed back as the result. Otherwise all *visible* children are traversed recursively to find the **Drawable** that intersects with the given coordinate.

Parameters:

x	The x coordinate of the intersection.
y	The y coordinate of the intersection.
last	out parameter in which the result is placed.

See also:

isVisible, **isTouchable**

Reimplements: **touchgfx::Drawable::getLastChild**

Reimplemented by: **touchgfx::ScrollableContainer::getLastChild**

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

insert

```
virtual void insert ( Drawable *  previous ,
                      Drawable & d
                      )
```

Inserts a Drawable after a specific child node.

If previous child node is 0, the drawable will be inserted as the first element in the list. The first element in the list of children is the element drawn first, so this makes it possible to insert a **Drawable***behind* all previously added children.

Parameters:

previous	The **Drawable** to insert after. If null, insert as header.
d	The **Drawable** to insert.

NOTE

As with add, do not add the same drawable twice.

Reimplemented by: **touchgfx::ListLayout::insert**

remove

```
virtual void remove ( Drawable & d )
```

Removes a Drawable from the container by removing it from the linked list of children.

If the **Drawable** is not in the list of children, nothing happens. It is possible to remove an element from whichever **Container** it is a member of using:

```
if (d.getParent()) d.getParent()->remove(d);
```

The **Drawable** will have the parent and next sibling cleared, but is otherwise left unaltered.

Parameters:

d The **Drawable** to remove.

NOTE

This is safe to call even if d is not a child of this Container (in which case nothing happens).

Reimplemented by: **touchgfx::ListLayout::remove**, **touchgfx::ModalWindow::remove**, **touchgfx::SlideMenu::remove**, **touchgfx::SwipeContainer::remove**

removeAll

virtual void removeAll ( )

Removes all children in the **Container** by resetting their parent and sibling pointers.

Please note that this is not done recursively, so any child which is itself a **Container** is not emptied.

Reimplemented by: **touchgfx::ListLayout::removeAll**

unlink

virtual void unlink ( )

Removes all children by unlinking the first child.

The parent and sibling pointers of the children are not reset.

See also:

getFirstChild

Protected Functions Documentation

getContainedArea

virtual Rect getContainedArea ( ) const

Gets a rectangle describing the total area covered by the children of this container.

Returns:

Rectangle covering all children.

Reimplemented by: touchgfx::ScrollableContainer::getContainedArea

moveChildrenRelative

virtual void moveChildrenRelative (int16_t deltaX ,
int16_t deltaY
)

Calls moveRelative on all children.

Parameters:

deltaX Horizontal displacement.
deltaY Vertical displacement.

Reimplemented by: touchgfx::ScrollableContainer::moveChildrenRelative

Protected Attributes Documentation

firstChild

Drawable * firstChild

Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Version: 4.16

CoverTransition

A Transition that slides the new screen over the previous *from* the given direction.

Inherits from: Transition

Public Functions

	CoverTransition(const uint8_t transitionSteps =20)
	Initializes a new instance of the **CoverTransition** class.
virtual void	**handleTickEvent**()
	Handles the tick event when transitioning.
virtual void	**init**()
	Initializes the transition.
virtual void	**tearDown**()
	Tears down the Animation.

Protected Functions

virtual void	**initMoveDrawable**(**Drawable** & d)
	Moves the Drawable to its initial position just outside of the visible area of the display.
virtual void	**tickMoveDrawable**(**Drawable** & d)
	Moves the Drawable to the new position as calculated in **handleTickEvent()**.

Additional inherited members

Public Functions inherited from Transition

virtual void	**invalidate**()
	Invalidates the screen when starting the **Transition**.

bool	**isDone**() const
	Query if the transition is done transitioning.
virtual void	**setScreenContainer**(**Container** & cont)
	Sets the **ScreenContainer**.
	Transition()
	Initializes a new instance of the **Transition** class.
virtual	**~Transition**()
	Finalizes an instance of the **Transition** class.

Protected Attributes inherited from Transition

bool	**done**
	Flag that indicates when the transition is done. This should be set by implementing classes.
Container *	**screenContainer**
	The screen **Container** of the **Screen** transitioning to.

Public Functions Documentation

CoverTransition

CoverTransition (const uint8_t transitionSteps =20)

Initializes a new instance of the **CoverTransition** class.

Parameters:

transitionSteps (Optional) Number of steps in the transition animation.

handleTickEvent

virtual void handleTickEvent ( )

Handles the tick event when transitioning.

It moves the contents of the **Screen**'s container. The direction of the transition determines the direction the contents of the container moves.

Reimplements: **touchgfx::Transition::handleTickEvent**

init

virtual void init ( )

Initializes the transition.

Called after the constructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::init**

tearDown

virtual void tearDown ( )

Tears down the Animation.

Called before the destructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::tearDown**

Protected Functions Documentation

initMoveDrawable

virtual void initMoveDrawable (Drawable & d)

Moves the Drawable to its initial position just outside of the visible area of the display.

Parameters:

d The **Drawable** to move.

tickMoveDrawable

virtual void tickMoveDrawable (Drawable & d)

Moves the Drawable to the new position as calculated in **handleTickEvent()**.

Parameters:

d The **Drawable** to move.

Version: 4.16

CWRUtil

Helper classes and functions for CanvasWidget. A handful of utility functions can be found here. These include helper functions for converting between float, int and Q5/Q10/Q15 formats. There are also functions for calculating sin() and cos() in integers with a high number of bits (15) reserved for fraction. Having sin() and cos() pre-calculated in this way allows very fast drawing of circles without the need for floating point arithmetic.

Using Q5, which uses 32 bit value internally, numbers from -67108865 to +67108864.96875 with a precision of 1/32 = 0.03125 can be represented, as described in http://en.wikipedia.org/wiki/Q_%28number_format%29.

Doing arithmetic operations on Q5, Q10 and Q15 numbers is described in detail on http://en.wikipedia.org/wiki/Fixed-point_arithmetic.

Public Classes

class	**Q10**
	Defines a "floating point number" with 10 bits reserved for the fractional part of the decimal number.
class	**Q15**
	Defines a "floating point number" with 15 bits reserved for the fractional part of the decimal number.
class	**Q5**
	Defines a "floating point number" with 5 bits reserved for the fractional part of the decimal number.

Public Functions

int	**angle**(**Q5** x, **Q5** y)
	Find angle of a coordinate relative to (0,0).
int	**angle**(**Q5** x, **Q5** y, **Q5** & d)
	Find the angle of the coordinate (x, y) relative to (0, 0).

template \<typename T \> int	**angle**(T x, T y)
	Find angle of a coordinate relative to (0,0).
template \<typename T \> int	**angle**(T x, T y, T & d)
	Find angle of a coordinate relative to (0,0).
int8_t	**arcsine**(**Q10** q10)
	Gets the arcsine of the given fraction (given as Q10).
Q15	**cosine**(int i)
	Find the value of cos(i) with 15 bits precision using the fact that cos(i)=sin(90-i).
Q15	**cosine**(**Q5** i)
	Find the value of cos(i) with 15 bits precision using the fact that cos(i)=sin(90-i).
Q5	**muldiv_toQ5**(int32_t factor1, int32_t factor2, int32_t divisor)
	Multiply two integers and divide by an integer without overflowing the multiplication.
Q5	**muldivQ10**(**Q10** factor1, **Q10** factor2, **Q10** divisor)
	Multiply two Q5's and divide by a Q5 without overflowing the multiplication (assuming that the final result can be stored in a Q5).
Q5	**muldivQ5**(**Q5** factor1, **Q5** factor2, **Q5** divisor)
	Multiply two Q5's and divide by a Q5 without overflowing the multiplication (assuming that the final result can be stored in a Q5).
Q5	**mulQ5**(**Q5** factor1, **Q10** factor2)
	Multiply one Q5 by a Q10 returning a new Q5 without overflowing.
Q5	**mulQ5**(**Q5** factor1, **Q5** factor2)
	Multiply two Q5's returning a new Q5 without overflowing.
Q15	**sine**(int i)
	Find the value of sin(i) with 15 bits precision.
Q15	**sine**(**Q5** i)
	Find the value of sin(i) with 15 bits precision.
Q5	**sqrtQ10**(**Q10** value)

	Find the square root of the given value.
template \<typename T \> FORCE_INLINE_FUNCTION **Q10**	**toQ10**(T value)
	Convert an integer to a fixed point number.
FORCE_INLINE_FUNCTION **Q5**	**toQ5**(**Q5** value)
	Convert a Q5 to itself.
template \<typename T \> FORCE_INLINE_FUNCTION **Q5**	**toQ5**(T value)
	Convert an integer to a fixed point number.

Public Functions Documentation

angle

```
static int angle ( Q5 x ,
                   Q5 y
                   )
```

Find angle of a coordinate relative to (0,0).

Parameters:

x The x coordinate.
y The y coordinate.

Returns:

The angle of the coordinate.

angle

```
static int angle ( Q5    x ,
                   Q5    y ,
                   Q5 & d
                   )
```

Find the angle of the coordinate (x, y) relative to (0, 0).

Parameters:

x The x coordinate.
y The y coordinate.

d The distance from (0,0) to (x,y).

Returns:

The angle.

angle

```
static int angle ( T x ,
                   T y
                   )
```

Find angle of a coordinate relative to (0,0).

Template Parameters:

T Generic type parameter (int or float).

Parameters:

x The x coordinate.
y The y coordinate.

Returns:

The angle of the coordinate.

angle

```
static int angle ( T    x ,
                   T    y ,
                   T & d
                   )
```

Find angle of a coordinate relative to (0,0).

Template Parameters:

T Generic type parameter (int or float).

Parameters:

x The x coordinate.
y The y coordinate.
d The distance from (0,0) to (x,y).

Returns:

The angle of the coordinate.

arcsine

static int8_t arcsine (Q10 q10)

Gets the arcsine of the given fraction (given as Q10).

The function is most precise for angles 0-45. To limit memory requirements, values above sqrt(1/2) is calculated as 90-arcsine(sqrt(1-q10^2)). Internally.

Parameters:

q10 The 10.

Returns:

An int8_t.

cosine

static Q15 cosine (int i)

Find the value of cos(i) with 15 bits precision using the fact that cos(i)=sin(90-i).

Parameters:

i the angle in degrees. The angle follows the angles of the clock, 0 being straight up and 90 being 3 o'clock.

Returns:

the value of cos(i) with 15 bits precision on the fractional part.

See also:

sine

cosine

static Q15 cosine (Q5 i)

Find the value of cos(i) with 15 bits precision using the fact that cos(i)=sin(90-i).

Parameters:

i the angle in degrees. The angle follows the angles of the clock, 0 being straight up and 90 being 3 o'clock.

Returns:

the value of cos(i) with 15 bits precision on the fractional part.

See also:

sine

muldiv_toQ5

```
static Q5 muldiv_toQ5 ( int32_t factor1 ,
                        int32_t factor2 ,
                        int32_t divisor
                        )
```

Multiply two integers and divide by an integer without overflowing the multiplication.

The result is returned in a **Q5** thus allowing a more precise calculation to be performed.

Parameters:

factor1 The first factor.
factor2 The second factor.
divisor The divisor.

Returns:

factor1 * factor2 / divisor as a **Q5**

muldivQ10

```
static Q5 muldivQ10 ( Q10 factor1 ,
                      Q10 factor2 ,
                      Q10 divisor
                      )
```

Multiply two Q5's and divide by a Q5 without overflowing the multiplication (assuming that the final result can be stored in a Q5).

Parameters:

factor1 The first factor.
factor2 The second factor.
divisor The divisor.

Returns:

factor1 * factor2 / divisor.

muldivQ5

```
static Q5 muldivQ5 ( Q5 factor1 ,
                     Q5 factor2 ,
                     Q5 divisor
                     )
```

Multiply two Q5's and divide by a Q5 without overflowing the multiplication (assuming that the final result can be stored in a Q5).

Parameters:

factor1 The first factor.
factor2 The second factor.
divisor The divisor.

Returns:

factor1 * factor2 / divisor.

mulQ5

```
static Q5 mulQ5 ( Q5   factor1 ,
                  Q10 factor2
                  )
```

Multiply one Q5 by a Q10 returning a new Q5 without overflowing.

Parameters:

factor1 The first factor.
factor2 The second factor.

Returns:

factor1 * factor2.

mulQ5

```
static Q5 mulQ5 ( Q5 factor1 ,
                  Q5 factor2
                  )
```

Multiply two Q5's returning a new Q5 without overflowing.

Parameters:

factor1 The first factor.
factor2 The second factor.

Returns:

factor1 * factor2.

sine

static Q15 sine (int i)

Find the value of sin(i) with 15 bits precision.

The returned value can be converted to a floating point number and divided by (1<<15) to get the rounded value of sin(i). By using this function, a complete circle can be drawn without the need for using floating point math.

Parameters:

i the angle in degrees. The angle follows the angles of the clock, 0 being straight up and 90 being 3 o'clock.

Returns:

the value of sin(i) with 15 bits precision on the fractional part.

sine

static Q15 sine (Q5 i)

Find the value of sin(i) with 15 bits precision.

The returned value can be converted to a floating point number and divided by (1<<15) to get the rounded value of sin(i). By using this function, a complete circle can be drawn without the need for using floating point math.

If the given degree is not an integer, the value is approximated by interpolation between sin(floor(i)) and sin(ceil(i)).

Parameters:

i the angle in degrees. The angle follows the angles of the clock, 0 being straight up and 90 being 3 o'clock.

Returns:

the value of sin(i) with 15 bits precision on the fractional part.

sqrtQ10

static Q5 sqrtQ10 (Q10 value)

Find the square root of the given value.

Parameters:

value The value to find the square root of.

Returns:

The square root of the given value.

toQ10

static FORCE_INLINE_FUNCTION Q10 toQ10 (T value)

Convert an integer to a fixed point number.

This is done by multiplying the floating point value by (1 << 10).

Template Parameters:

T Should be either int or float.

Parameters:

value the integer to convert.

Returns:

the converted integer.

toQ5

static FORCE_INLINE_FUNCTION Q5 toQ5 (Q5 value)

Convert a Q5 to itself.

Allows toQ5 to be called with a variable that is already **Q5**.

Parameters:

value the **Q5**.

Returns:

the value passed.

toQ5

static FORCE_INLINE_FUNCTION Q5 toQ5 (T value)

Convert an integer to a fixed point number.

This is done by multiplying the floating point value by (1 << 5)

Template Parameters:

T Should be either int or float.

Parameters:

value the integer to convert.

Returns:

the converted integer.

Version: 4.16

DataGraphScroll

DataGraphScroll is used to display a graph that continuously scrolls to the left every time a new value is added to the graph. Because the graph is scrolled every time a new value is added, the graph has to be re-drawn which can be quite demanding for the hardware depending on the graph elements used in the graph.

Inherits from: AbstractDataGraphWithY, AbstractDataGraph, Container, Drawable

Inherited by: GraphScroll< CAPACITY >

Public Functions

virtual void	**clear**()
	Clears the graph to its blank/initial state.
	DataGraphScroll(int16_t capacity, int * values)
	Initializes a new instance of the **DataGraphScroll** class.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.

Protected Functions

virtual int16_t	**addValue**(int value)
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.

Protected Attributes

int16_t	**current**
	The current position used for inserting new elements.

Additional inherited members

Public Functions inherited from AbstractDataGraphWithY

	AbstractDataGraphWithY(int16_t capacity, int * values)
	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)
	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.
virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const

	Get x coordinate axis scale value.
virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.
virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.
void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions inherited from AbstractDataGraphWithY

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.
virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.

virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)
	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraphWithY

uint32_t **dataCounter**

The data counter of how many times addDataPoint() has been called.

int **xOffset**

The x axis offset (real value of data point at index 0)

int **xScale**

The x axis scale (increment between two data points)

int * **yValues**

The values of the graph.

Public Classes inherited from AbstractDataGraph

class **GraphClickEvent**

An object of this type is passed with each callback that is sent when the graph is clicked.

class **GraphDragEvent**

An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

AbstractDataGraph(int16_t capacity)

Initializes a new instance of the **AbstractDataGraph** class.

void **addBottomElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area below the graph.

void **addGraphElement**(**AbstractGraphElement** & d)

Adds a graph element which will display the graph.

void **addLeftElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the left of the graph.

void **addRightElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the right of the graph.

void **addTopElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area above the graph.

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

int16_t **getGapBeforeIndex**() const

Gets gap before index as set using setGapBeforeIndex().

int16_t **getGraphAreaHeight**() const

Gets graph area height.

int16_t **getGraphAreaHeightIncludingPadding**() const

Gets graph area height including padding (but not margin).

int16_t **getGraphAreaMarginBottom**() const

Gets graph margin bottom.

int16_t **getGraphAreaMarginLeft**() const

Gets graph margin left.

int16_t **getGraphAreaMarginRight**() const

Gets graph margin right.

int16_t **getGraphAreaMarginTop**() const

Gets graph margin top.

int16_t **getGraphAreaPaddingBottom**() const

Gets graph area padding bottom.

int16_t **getGraphAreaPaddingLeft**() const

Gets graph area padding left.

int16_t **getGraphAreaPaddingRight**() const

Gets graph area padding right.

int16_t **getGraphAreaPaddingTop**() const

Gets graph area padding top.

int16_t **getGraphAreaWidth**() const

Gets graph area width.

int16_t **getGraphAreaWidthIncludingPadding**() const

Gets graph area width including padding (but not margin).

virtual int **getGraphRangeXMax**() const =0

Gets the maximum x coordinate for the graph.

virtual int	**getGraphRangeXMin**() const =0
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const =0
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const =0
	Gets minimum y coordinate for the graph.
int16_t	**getMaxCapacity**() const
	Gets the capacity (max number of points) of the graph.
virtual bool	**getNearestIndexForScreenX**(int16_t x, int16_t & index) const
	Gets graph index nearest to the given screen x coordinate.
virtual bool	**getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)
	Gets graph index nearest to the given screen position.
int	**getScale**() const
	Gets the scaling factor previously set using setScale().
int16_t	**getUsedCapacity**() const
	Gets the number of point used by the graph.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.
int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const

Get the data point y value for the given graph point index.

int **indexToDataPointYAsInt**(int16_t index) const

Get the data point y value for the given graph point index.

int16_t **indexToScreenX**(int16_t index) const

Get the screen x coordinate for the given graph point index.

int16_t **indexToScreenY**(int16_t index) const

Get the screen y coordinate for the given graph point index.

void **setAlpha**(uint8_t newAlpha)

Sets the opacity (alpha value).

void **setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)

Sets an action to be executed when the graph is clicked.

void **setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)

Sets an action to be executed when the graph is dragged.

void **setGapBeforeIndex**(int16_t index)

Makes gap before the specified index.

void **setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)

Sets graph position inside the widget by reserving a margin around the graph.

void **setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)

Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.

void **setGraphRange**(int xMin, int xMax, float yMin, float yMax)

Sets minimum and maximum x and y coordinate ranges for the graph.

void **setGraphRange**(int xMin, int xMax, int yMin, int yMax)

Sets minimum and maximum x and y coordinate ranges for the graph.

virtual void **setGraphRangeX**(int min, int max) =0

Sets minimum and maximum x coordinates for the graph.

virtual void **setGraphRangeY**(float min, float max) =0

Sets minimum and maximum y coordinates for the graph.

virtual void	**setGraphRangeY**(int min, int max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.

virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.
void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)
	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0

	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**
	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**

	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**
	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**
	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**
	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**
	The graph area top padding.
int16_t	**usedCapacity**

	The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

clear

virtual void clear ( )

Clears the graph to its blank/initial state.

Reimplements: **touchgfx::AbstractDataGraph::clear**

DataGraphScroll

DataGraphScroll (int16_t capacity ,
int * values
)

Initializes a new instance of the **DataGraphScroll** class.

Parameters:

capacity The capacity.
values Pointer to memory with room for capacity elements of type T.

indexToGlobalIndex

virtual int32_t indexToGlobalIndex (int16_t index)

Convert an index to global index.

The index is the index of any data point, The global index is a value that keeps growing whenever a new data point is added the the graph.

Parameters:

index Zero-based index of the point.

Returns:

The global index.

Reimplements: **touchgfx::AbstractDataGraph::indexToGlobalIndex**

Protected Functions Documentation

addValue

virtual int16_t addValue (int value)

Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).

Parameters:

value The value to add to the array.

Returns:

The index of the newly added value.

Reimplements: **touchgfx::AbstractDataGraphWithY::addValue**

beforeAddValue

virtual void beforeAddValue ( )

This function is called before a new value (data point) is added.

This allows for invalidation to be calculated based on the global data counter before it is increased as a result of adding the new point.

Reimplements: **touchgfx::AbstractDataGraphWithY::beforeAddValue**

realIndex

virtual int16_t realIndex (int16_t index)

Get the real index in the yValues array of the given index.

Normally this is just the 'i' but e.g. **DataGraphScroll** does not, for performance reasons.

Parameters:

index Zero-based index.

Returns:

The index in the yValues array.

Reimplements: **touchgfx::AbstractDataGraphWithY::realIndex**

Protected Attributes Documentation

current

int16_t current

The current position used for inserting new elements.

Version: 4.16

DataGraphWrapAndClear

The DataGraphWrapAndClear will show new points progressing across the graph. Once the graph is filled, the next point added will cause the graph to be cleared and a new graph will slowly be created as new values are added.

Inherits from: AbstractDataGraphWithY, AbstractDataGraph, Container, Drawable

Inherited by: GraphWrapAndClear< CAPACITY >

Public Functions

	DataGraphWrapAndClear(int16_t capacity, int * values)
	Initializes a new instance of the DataGraphWrapAndOverwrite class.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.

Protected Functions

virtual int16_t	**addValue**(int value)
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.

Additional inherited members

Public Functions inherited from AbstractDataGraphWithY

	AbstractDataGraphWithY(int16_t capacity, int * values)
	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)

	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.
virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const
	Get x coordinate axis scale value.
virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.
virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.

void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions inherited from AbstractDataGraphWithY

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.
virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.

virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)
	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraphWithY

uint32_t	**dataCounter**
	The data counter of how many times addDataPoint() has been called.
int	**xOffset**
	The x axis offset (real value of data point at index 0)
int	**xScale**
	The x axis scale (increment between two data points)
int *	**yValues**

The values of the graph.

Public Classes inherited from AbstractDataGraph

class **GraphClickEvent**

An object of this type is passed with each callback that is sent when the graph is clicked.

class **GraphDragEvent**

An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

AbstractDataGraph(int16_t capacity)

Initializes a new instance of the **AbstractDataGraph** class.

void **addBottomElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area below the graph.

void **addGraphElement**(**AbstractGraphElement** & d)

Adds a graph element which will display the graph.

void **addLeftElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the left of the graph.

void **addRightElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the right of the graph.

void **addTopElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area above the graph.

virtual void **clear**()

Clears the graph to its blank/initial state.

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

int16_t **getGapBeforeIndex**() const

Gets gap before index as set using setGapBeforeIndex().

int16_t **getGraphAreaHeight**() const

Gets graph area height.

int16_t	**getGraphAreaHeightIncludingPadding**() const
	Gets graph area height including padding (but not margin).
int16_t	**getGraphAreaMarginBottom**() const
	Gets graph margin bottom.
int16_t	**getGraphAreaMarginLeft**() const
	Gets graph margin left.
int16_t	**getGraphAreaMarginRight**() const
	Gets graph margin right.
int16_t	**getGraphAreaMarginTop**() const
	Gets graph margin top.
int16_t	**getGraphAreaPaddingBottom**() const
	Gets graph area padding bottom.
int16_t	**getGraphAreaPaddingLeft**() const
	Gets graph area padding left.
int16_t	**getGraphAreaPaddingRight**() const
	Gets graph area padding right.
int16_t	**getGraphAreaPaddingTop**() const
	Gets graph area padding top.
int16_t	**getGraphAreaWidth**() const
	Gets graph area width.
int16_t	**getGraphAreaWidthIncludingPadding**() const
	Gets graph area width including padding (but not margin).
virtual int	**getGraphRangeXMax**() const =0
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const =0
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const =0

	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const =0
	Gets minimum y coordinate for the graph.
int16_t	**getMaxCapacity**() const
	Gets the capacity (max number of points) of the graph.
virtual bool	**getNearestIndexForScreenX**(int16_t x, int16_t & index) const
	Gets graph index nearest to the given screen x coordinate.
virtual bool	**getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)
	Gets graph index nearest to the given screen position.
int	**getScale**() const
	Gets the scaling factor previously set using setScale().
int16_t	**getUsedCapacity**() const
	Gets the number of point used by the graph.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.
int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const
	Get the data point y value for the given graph point index.
int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.

int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setGraphRangeX**(int min, int max) =0
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)

	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0

	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.
void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)
	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.

virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**
	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**
	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**

	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**
	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**
	The graph area top padding.
int16_t	**usedCapacity**
	The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)

	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0

	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

DataGraphWrapAndClear

```
DataGraphWrapAndClear ( int16_t capacity ,
                        int *   values
                        )
```

Initializes a new instance of the DataGraphWrapAndOverwrite class.

Parameters:

capacity The capacity.
values Pointer to memory with room for capacity elements of type T.

indexToGlobalIndex

virtual int32_t indexToGlobalIndex (int16_t index)

Convert an index to global index.

The index is the index of any data point, The global index is a value that keeps growing whenever a new data point is added the the graph.

Parameters:

index Zero-based index of the point.

Returns:

The global index.

Reimplements: **touchgfx::AbstractDataGraph::indexToGlobalIndex**

Protected Functions Documentation

addValue

virtual int16_t addValue (int value)

Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).

Parameters:

value The value to add to the array.

Returns:

The index of the newly added value.

Reimplements: **touchgfx::AbstractDataGraphWithY::addValue**

beforeAddValue

virtual void beforeAddValue ( )

This function is called before a new value (data point) is added.

This allows for invalidation to be calculated based on the global data counter before it is increased as a result of adding the new point.

Reimplements: **touchgfx::AbstractDataGraphWithY::beforeAddValue**

Version: 4.16

DataGraphWrapAndOverwrite

A continuous data graph which will fill the graph with elements, and overwrite the first elements with new values after the graph has filled. There will be a gap between the newly inserted element and the element after. This similar behavior to a heart beat monitor.

Inherits from: AbstractDataGraphWithY, AbstractDataGraph, Container, Drawable

Inherited by: GraphWrapAndOverwrite< CAPACITY >

Public Functions

virtual void	**clear**()
	Clears the graph to its blank/initial state.
	DataGraphWrapAndOverwrite(int16_t capacity, int * values)
	Initializes a new instance of the **DataGraphWrapAndOverwrite** class.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.

Protected Functions

virtual int16_t	**addValue**(int value)
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.

Protected Attributes

int16_t	**current**
	The current index (used to keep track of where to insert new data point in value array)

Additional inherited members

Public Functions inherited from AbstractDataGraphWithY

	AbstractDataGraphWithY(int16_t capacity, int * values)
	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)
	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.
virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const
	Get x coordinate axis scale value.
virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.

virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.
void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions inherited from AbstractDataGraphWithY

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.
virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const

	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)
	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraphWithY

uint32_t	**dataCounter**

The data counter of how many times addDataPoint() has been called.

int **xOffset**

The x axis offset (real value of data point at index 0)

int **xScale**

The x axis scale (increment between two data points)

int * **yValues**

The values of the graph.

Public Classes inherited from AbstractDataGraph

class **GraphClickEvent**

An object of this type is passed with each callback that is sent when the graph is clicked.

class **GraphDragEvent**

An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

AbstractDataGraph(int16_t capacity)

Initializes a new instance of the **AbstractDataGraph** class.

void **addBottomElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area below the graph.

void **addGraphElement**(**AbstractGraphElement** & d)

Adds a graph element which will display the graph.

void **addLeftElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the left of the graph.

void **addRightElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the right of the graph.

void **addTopElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area above the graph.

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

int16_t **getGapBeforeIndex**() const

Gets gap before index as set using setGapBeforeIndex().

int16_t **getGraphAreaHeight**() const

Gets graph area height.

int16_t **getGraphAreaHeightIncludingPadding**() const

Gets graph area height including padding (but not margin).

int16_t **getGraphAreaMarginBottom**() const

Gets graph margin bottom.

int16_t **getGraphAreaMarginLeft**() const

Gets graph margin left.

int16_t **getGraphAreaMarginRight**() const

Gets graph margin right.

int16_t **getGraphAreaMarginTop**() const

Gets graph margin top.

int16_t **getGraphAreaPaddingBottom**() const

Gets graph area padding bottom.

int16_t **getGraphAreaPaddingLeft**() const

Gets graph area padding left.

int16_t **getGraphAreaPaddingRight**() const

Gets graph area padding right.

int16_t **getGraphAreaPaddingTop**() const

Gets graph area padding top.

int16_t **getGraphAreaWidth**() const

Gets graph area width.

int16_t **getGraphAreaWidthIncludingPadding**() const

Gets graph area width including padding (but not margin).

virtual int **getGraphRangeXMax**() const =0

Gets the maximum x coordinate for the graph.

virtual int **getGraphRangeXMin**() const =0

	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const =0
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const =0
	Gets minimum y coordinate for the graph.
int16_t	**getMaxCapacity**() const
	Gets the capacity (max number of points) of the graph.
virtual bool	**getNearestIndexForScreenX**(int16_t x, int16_t & index) const
	Gets graph index nearest to the given screen x coordinate.
virtual bool	**getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)
	Gets graph index nearest to the given screen position.
int	**getScale**() const
	Gets the scaling factor previously set using setScale().
int16_t	**getUsedCapacity**() const
	Gets the number of point used by the graph.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.
int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const
	Get the data point y value for the given graph point index.

int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.
int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setGraphRangeX**(int min, int max) =0
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max) =0

	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const

	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.
void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)
	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.

void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**
	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.

int16_t	**gapBeforeIndex** The graph is disconnected (there is a gap) before this element index.
Container	**graphArea** The graph area (the center area)
Container	**leftArea** The area to the left of the graph.
int16_t	**leftPadding** The graph area left padding.
int16_t	**maxCapacity** Maximum number of points in the graph.
Container	**rightArea** The area to the right of the graph.
int16_t	**rightPadding** The graph area right padding.
Container	**topArea** The area above the graph.
int16_t	**topPadding** The graph area top padding.
int16_t	**usedCapacity** The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.

virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.

Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.

bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

clear

virtual void clear ( )

Clears the graph to its blank/initial state.

Reimplements: **touchgfx::AbstractDataGraph::clear**

DataGraphWrapAndOverwrite

DataGraphWrapAndOverwrite (int16_t capacity ,
int * values
)

Initializes a new instance of the **DataGraphWrapAndOverwrite** class.

Parameters:

capacity The capacity.
values Pointer to memory with room for capacity elements of type T.

indexToGlobalIndex

virtual int32_t indexToGlobalIndex (int16_t index)

Convert an index to global index.

The index is the index of any data point, The global index is a value that keeps growing whenever a new data point is added the the graph.

Parameters:

index Zero-based index of the point.

Returns:

The global index.

Reimplements: **touchgfx::AbstractDataGraph::indexToGlobalIndex**

Protected Functions Documentation

addValue

virtual int16_t addValue (int value)

Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).

Parameters:

value The value to add to the array.

Returns:

The index of the newly added value.

Reimplements: **touchgfx::AbstractDataGraphWithY::addValue**

beforeAddValue

virtual void beforeAddValue ( )

This function is called before a new value (data point) is added.

This allows for invalidation to be calculated based on the global data counter before it is increased as a result of adding the new point.

Reimplements: **touchgfx::AbstractDataGraphWithY::beforeAddValue**

Protected Attributes Documentation

current

int16_t current

The current index (used to keep track of where to insert new data point in value array)

Version: 4.16

DebugPrinter

The class DebugPrinter defines the interface for printing debug messages on top of the framebuffer.

Inherited by: LCD16DebugPrinter, LCD1DebugPrinter, LCD24DebugPrinter, LCD2DebugPrinter, LCD32DebugPrinter, LCD4DebugPrinter, LCD8ABGR2222DebugPrinter, LCD8ARGB2222DebugPrinter, LCD8BGRA2222DebugPrinter, LCD8RGBA2222DebugPrinter

Public Functions

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
virtual void	**draw**(const **Rect** & rect) const =0
	Draws the debug string on top of the framebuffer content.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()
	Finalizes an instance of the **DebugPrinter** class.

Protected Functions

uint16_t	**getGlyph**(uint8_t c) const

Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

DebugPrinter

DebugPrinter ( )

Initializes a new instance of the **DebugPrinter** class.

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplemented by: **touchgfx::LCD16DebugPrinter::draw**, **touchgfx::LCD1DebugPrinter::draw**, **touchgfx::LCD24DebugPrinter::draw**, **touchgfx::LCD2DebugPrinter::draw**, **touchgfx::LCD32DebugPrinter::draw**, **touchgfx::LCD4DebugPrinter::draw**, **touchgfx::LCD8ABGR2222DebugPrinter::draw**, **touchgfx::LCD8ARGB2222DebugPrinter::draw**, **touchgfx::LCD8BGRA2222DebugPrinter::draw**, **touchgfx::LCD8RGBA2222DebugPrinter::draw**

getRegion

const Rect & getRegion ( ) const

Returns the region where the debug string is displayed.

Returns:

Rect The debug string region.

setColor

void setColor (colortype fg)

Sets the foreground color of the debug string.

Parameters:

fg The foreground color of the debug string.

setPosition

void setPosition (uint16_t x ,
uint16_t y ,
uint16_t w ,
uint16_t h
)

Sets the position onscreen where the debug string will be displayed.

Parameters:

x The coordinate of the region where the debug string is displayed.
y The coordinate of the region where the debug string is displayed.
w The width of the region where the debug string is displayed.
h The height of the region where the debug string is displayed.

setScale

void setScale (uint8_t scale)

Sets the font scale of the debug string.

Parameters:

scale The font scale of the debug string.

setString

void setString (const char * string)

Sets the debug string to be displayed on top of the framebuffer.

Parameters:

string The string to be displayed.

~DebugPrinter

virtual ~DebugPrinter ( )

Finalizes an instance of the **DebugPrinter** class.

Protected Functions Documentation

getGlyph

uint16_t getGlyph (uint8_t c)

Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Parameters:

c The character to get a glyph for.

Returns:

The glyph.

Protected Attributes Documentation

debugForegroundColor

colortype debugForegroundColor

Font color to use when displaying the debug string.

debugRegion

Rect debugRegion

Region onscreen where the debug message is displayed.

debugScale

uint8_t debugScale

Font scaling factor to use when displaying the debug string.

debugString

const char * debugString

Debug string to be displayed onscreen.

Version: 4.16

DigitalClock

A digital clock. Can be set in either 12 or 24 hour mode. Seconds are optional. Width and height must be set manually to match the typography and alignment specified in the text database. The Digital Clock requires a typedText with one wildcard and uses the following characters (not including quotes) "AMP :0123456789" These must be present in the text database with the same typography as the wildcard text. Leading zero for the hour indicator can be enabled/disable by the displayLeadingZeroForHourIndicator method.

Inherits from: AbstractClock, Container, Drawable

Public Types

enum	**DisplayMode** { DISPLAY_12_HOUR_NO_SECONDS, DISPLAY_24_HOUR_NO_SECONDS, DISPLAY_12_HOUR, DISPLAY_24_HOUR }
	Values that represent different display modes.

Public Functions

	DigitalClock()
void	**displayLeadingZeroForHourIndicator**(bool displayLeadingZero)
	Sets whether to display a leading zero for the hour indicator or not, when the hour value only has one digit.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **colortype**	**getColor**() const
	Gets the color of the text.
virtual **DisplayMode**	**getDisplayMode**() const
	Gets the current display mode.
virtual uint16_t	**getTextWidth**() const
	Gets text width of the currently displayed **DigitalClock**.
virtual void	**setAlpha**(uint8_t newAlpha)

	Sets the opacity (alpha value).
virtual void	**setBaselineY**(int16_t baselineY)
	Adjusts the **DigitalClock** y coordinate so the text will have its baseline at the specified value.
virtual void	**setColor**(**colortype** color)
	Sets the color of the text.
virtual void	**setDisplayMode**(**DisplayMode** dm)
	Sets the display mode to 12/24 hour clock with or without seconds.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setTypedText**(**TypedText** typedText)
	Sets the typed text of the **DigitalClock**.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.

Protected Functions

virtual void	**updateClock**()
	Update the visual representation of the clock on the display.

Protected Attributes

const int	**BUFFER_SIZE**
	Buffer size of the wild card, worst case is "12:59:59 AM" (12 chars)
Unicode::UnicodeChar	**buffer**
	Wild card buffer for the clock text.
DisplayMode	**displayMode**
	The current display mode.
TextAreaWithOneWildcard	**text**
	The clock text.

bool	**useLeadingZeroForHourIndicator**
	Print a leading zero if the hour is less than 10.

Additional inherited members

Public Functions inherited from AbstractClock

	AbstractClock()
bool	**getCurrentAM**() const
	Is the current time a.m.
uint8_t	**getCurrentHour**() const
	Gets the current hour.
uint8_t	**getCurrentHour12**() const
	Gets current hour 12, i.e.
uint8_t	**getCurrentHour24**() const
	Gets current hour 24, i.e.
uint8_t	**getCurrentMinute**() const
	Gets the current minute.
uint8_t	**getCurrentSecond**() const
	Gets the current second.
virtual void	**setTime12Hour**(uint8_t hour, uint8_t minute, uint8_t second, bool am)
	Sets the time with input format as 12H.
virtual void	**setTime24Hour**(uint8_t hour, uint8_t minute, uint8_t second)
	Sets the time with input format as 24H.

Protected Attributes inherited from AbstractClock

uint8_t	**currentHour**
	Local copy of the current hour.
uint8_t	**currentMinute**

	Local copy of the current minute.
uint8_t	**currentSecond**
	Local copy of the current second.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()

	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()

	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

DisplayMode

enum DisplayMode

Values that represent different display modes.

DISPLAY_12_HOUR_NO_SECONDS	12 Hour clock. Seconds are not displayed
DISPLAY_24_HOUR_NO_SECONDS	24 Hour clock. Seconds are not displayed
DISPLAY_12_HOUR	12 Hour clock. Seconds are displayed
DISPLAY_24_HOUR	24 Hour clock. Seconds are displayed

Public Functions Documentation

DigitalClock

DigitalClock ( )

displayLeadingZeroForHourIndicator

void displayLeadingZeroForHourIndicator (bool displayLeadingZero)

Sets whether to display a leading zero for the hour indicator or not, when the hour value only has one digit.

For example 8 can be displayed as "8:" (displayLeadingZero=false) or "08:" (displayLeadingZero=true).

Default value for this setting is false.

Parameters:

displayLeadingZero true = show leading zero. false = do not show leading zero.

NOTE

This does not affect the display of minutes or seconds.

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getColor

virtual colortype getColor ( ) const

Gets the color of the text.

Returns:

The color.

getDisplayMode

virtual DisplayMode getDisplayMode ( ) const

Gets the current display mode.

Returns:

The display mode.

See also:

DisplayMode, **setDisplayMode**

getTextWidth

virtual uint16_t getTextWidth ( ) const

Gets text width of the currently displayed **DigitalClock**.

Returns:

The text width of the currently displayed **DigitalClock**.

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setBaselineY

virtual void setBaselineY (int16_t baselineY)

Adjusts the **DigitalClock** y coordinate so the text will have its baseline at the specified value.

The placements is relative to the specified **TypedText** so if the **TypedText** is changed, you have to set the baseline again.

Parameters:

baselineY The y coordinate of the baseline of the text.

NOTE

that setTypedText must be called prior to setting the baseline.

setColor

virtual void setColor (colortype color)

Sets the color of the text.

Parameters:

color The new text color.

NOTE

Automatically invalidates the **DigitalClock**.

setDisplayMode

virtual void setDisplayMode (DisplayMode dm)

Sets the display mode to 12/24 hour clock with or without seconds.

Parameters:

dm The new display mode.

See also:

DisplayMode, **getDisplayMode**

setHeight

virtual void setHeight (int16_t height)

Sets the height of this drawable.

Parameters:

height The new height.

NOTE

For most **Drawable** widgets, changing this does normally not automatically yield a redraw.

Reimplements: **touchgfx::Drawable::setHeight**

setTypedText

virtual void setTypedText (TypedText typedText)

Sets the typed text of the **DigitalClock**.

Expects a **TypedText** with one wildcard and that the following characters are defined for the typography of the **TypedText**:

- 12 hour clock: "AMP :0123456789"
- 24 hour clock: ":0123456789"

Parameters:

typedText Describes the typed text to use.

NOTE

Automatically invalidates the DigitalClock.

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

Reimplements: **touchgfx::Drawable::setWidth**

Protected Functions Documentation

updateClock

virtual void updateClock ( )

Update the visual representation of the clock on the display.

Reimplements: **touchgfx::AbstractClock::updateClock**

Protected Attributes Documentation

BUFFER_SIZE

const int BUFFER_SIZE = 12

Buffer size of the wild card, worst case is "12:59:59 AM" (12 chars)

buffer

Unicode::UnicodeChar buffer

Wild card buffer for the clock text.

displayMode

DisplayMode displayMode

The current display mode.

text

TextAreaWithOneWildcard text

The clock text.

useLeadingZeroForHourIndicator

bool useLeadingZeroForHourIndicator

Print a leading zero if the hour is less than 10.

Version: 4.16

DisplayTransformation

Defines transformations from display space to framebuffer space. The display might be (considered) in portrait mode from 0,0 to 272,480, while the actual framebuffer is from 0,0 to 480,272. This class handles the transformations.

Public Functions

void	**transformDisplayToFrameBuffer**(float & x, float & y) Transform x,y from display to framebuffer coordinates.
void	**transformDisplayToFrameBuffer**(float & x, float & y, const **Rect** & in) Transform x,y from coordinates relative to the in rect to framebuffer coordinates.
void	**transformDisplayToFrameBuffer**(int16_t & x, int16_t & y) Transform x,y from display to framebuffer coordinates.
void	**transformDisplayToFrameBuffer**(int16_t & x, int16_t & y, const **Rect** & in) Transform x,y from coordinates relative to the in rect to framebuffer coordinates.
void	**transformDisplayToFrameBuffer**(**Rect** & r) Transform rectangle from display to framebuffer coordinates.
void	**transformDisplayToFrameBuffer**(**Rect** & r, const **Rect** & in) Transform rectangle r from coordinates relative to the in rect to framebuffer coordinates.
void	**transformFrameBufferToDisplay**(int16_t & x, int16_t & y) Transform x,y from framebuffer to display coordinates.
void	**transformFrameBufferToDisplay**(**Rect** & r) Transform rectangle from framebuffer to display coordinates.

Public Functions Documentation

transformDisplayToFrameBuffer

```
static void transformDisplayToFrameBuffer ( float & x ,
```

```
                                        float & y
                                        )
```

Transform x,y from display to framebuffer coordinates.

Parameters:

x the x part to translate.
y the y part to translate.

transformDisplayToFrameBuffer

```
static void transformDisplayToFrameBuffer ( float &       x ,
                                            float &       y ,
                                            const Rect & in
                                            )
```

Transform x,y from coordinates relative to the in rect to framebuffer coordinates.

Transform x,y from coordinates relative to the in rect to framebuffer coordinates.

Parameters:

x the x part to translate.
y the y part to translate.
in the rectangle defining the coordinate space.

transformDisplayToFrameBuffer

```
static void transformDisplayToFrameBuffer ( int16_t & x ,
                                            int16_t & y
                                            )
```

Transform x,y from display to framebuffer coordinates.

Parameters:

x the x part to translate.
y the y part to translate.

transformDisplayToFrameBuffer

```
static void transformDisplayToFrameBuffer ( int16_t &       x ,
                                            int16_t &       y ,
                                            const Rect & in
```

)

Transform x,y from coordinates relative to the in rect to framebuffer coordinates.

Transform x,y from coordinates relative to the in rect to framebuffer coordinates.

Parameters:

x the x part to translate.
y the y part to translate.
in the rectangle defining the coordinate space.

transformDisplayToFrameBuffer

static void transformDisplayToFrameBuffer (Rect & r)

Transform rectangle from display to framebuffer coordinates.

Parameters:

r the rectangle to translate.

transformDisplayToFrameBuffer

static void transformDisplayToFrameBuffer (Rect & r ,
const Rect & in
)

Transform rectangle r from coordinates relative to the in rect to framebuffer coordinates.

Transform rectangle r from coordinates relative to the in rect to framebuffer coordinates.

Parameters:

r the rectangle to translate.
in the rectangle defining the coordinate space.

transformFrameBufferToDisplay

static void transformFrameBufferToDisplay (int16_t & x ,
int16_t & y
)

Transform x,y from framebuffer to display coordinates.

Parameters:

x the x part to translate.
y the y part to translate.

transformFrameBufferToDisplay

static void transformFrameBufferToDisplay (Rect & r)

Transform rectangle from framebuffer to display coordinates.

Parameters:

r the rectangle to translate.

Version: 4.16

DMA_Interface

DMA_Interface provides basic functionality and structure for processing "blit" operations using DMA.

Inherited by: NoDMA

Public Functions

virtual void	**addToQueue**(const **BlitOp** & op)
	Inserts a **BlitOp** for processing.
virtual void	**flush**()
	This function blocks until all DMA transfers in the queue have been completed.
bool	**getAllowed**() const
	Gets whether a DMA operation is allowed to begin.
virtual **BlitOperations**	**getBlitCaps**() =0
	Gets the blit capabilities of this DMA.
virtual **DMAType**	**getDMAType**(void)
	Function for obtaining the DMA type of the concrete **DMA_Interface** implementation.
virtual void	**initialize**()
	Perform initialization.
uint8_t	**isDmaQueueEmpty**()
	Query if the DMA queue is empty.
uint8_t	**isDmaQueueFull**()
	Query if the DMA queue is full.
bool	**isDMARunning**()
	Query if the DMA is running.
void	**setAllowed**(bool allowed)
	Sets whether or not a DMA operation is allowed to begin.

virtual void	**signalDMAInterrupt**() =0
	This function is called automatically by the framework when a DMA interrupt has been received.
virtual void	**start**()
	Signals that DMA transfers can start.
virtual	**~DMA_Interface**()
	Finalizes an instance of the **DMA_Interface** class.

Protected Functions

virtual void	**disableAlpha**()
	Configures blit-op hardware for solid operation (no alpha-blending)
	DMA_Interface(**DMA_Queue** & dmaQueue)
	Constructs a DMA Interface object.
virtual void	**enableAlpha**(uint8_t alpha)
	Configures blit-op hardware for alpha-blending.
virtual void	**enableCopyWithTransparentPixels**(uint8_t alpha)
	Configures blit-op hardware for alpha-blending while simultaneously skipping transparent pixels.
virtual void	**execute**()
	Performs a queued blit-op.
virtual void	**executeCompleted**()
	To be called when blit-op has been performed.
virtual void	**seedExecution**()
	Called when elements are added to the DMA-queue.
virtual void	**setupDataCopy**(const **BlitOp** & blitOp) =0
	Configures blit-op hardware for a 2D copy as specified by blitOp.
virtual void	**setupDataFill**(const **BlitOp** & blitOp) =0
	Configures blit-op hardware for a 2D fill as specified by blitOp.
virtual void	**waitForFrameBufferSemaphore**()

Waits until framebuffer semaphore is available (i.e.

Protected Attributes

bool	**isAllowed**
	true if DMA transfers are currently allowed.
bool	**isRunning**
	true if a DMA transfer is currently ongoing.
DMA_Queue &	**queue**
	Reference to the DMA queue.

Public Functions Documentation

addToQueue

virtual void addToQueue (const BlitOp & op)

Inserts a **BlitOp** for processing.

This also potentially starts the DMA controller, if not already running.

Parameters:

op The operation to add.

flush

virtual void flush ( )

This function blocks until all DMA transfers in the queue have been completed.

Reimplemented by: **touchgfx::NoDMA::flush**

getAllowed

bool getAllowed ( ) const

Gets whether a DMA operation is allowed to begin.

Used in single-buffering to avoid changing the framebuffer while display is being updated.

Returns:

true if DMA is allowed to start, false if not.

getBlitCaps

virtual BlitOperations getBlitCaps ( ) =0

Gets the blit capabilities of this DMA.

Returns:

The blit operations supported by this DMA implementation.

Reimplemented by: **touchgfx::NoDMA::getBlitCaps**

getDMAType

virtual DMAType getDMAType (void)

Function for obtaining the DMA type of the concrete **DMA_Interface** implementation.

As default, will return DMA_TYPE_GENERIC type value.

Returns:

a DMAType value of the concrete **DMA_Interface** implementation.

initialize

virtual void initialize ( )

Perform initialization.

Does nothing in this base class.

isDmaQueueEmpty

uint8_t isDmaQueueEmpty ( )

Query if the DMA queue is empty.

Returns:

1 if DMA queue is empty, else 0.

isDmaQueueFull

uint8_t isDmaQueueFull ( )

Query if the DMA queue is full.

Returns:

1 if DMA queue is full, else 0.

isDMARunning

bool isDMARunning ( )

Query if the DMA is running.

Returns:

true if a DMA operation is currently in progress.

setAllowed

void setAllowed (bool allowed)

Sets whether or not a DMA operation is allowed to begin.

Used in single-buffering to avoid changing the framebuffer while display is being updated.

Parameters:

allowed true if DMA transfers are allowed.

signalDMAInterrupt

virtual void signalDMAInterrupt ( ) =0

This function is called automatically by the framework when a DMA interrupt has been received.

This function is called automatically by the framework when a DMA interrupt has been received.

Reimplemented by: **touchgfx::NoDMA::signalDMAInterrupt**

start

virtual void start ( )

Signals that DMA transfers can start.

If any elements are in the queue, start it.

~DMA_Interface

virtual ~DMA_Interface ( )

Finalizes an instance of the **DMA_Interface** class.

Protected Functions Documentation

disableAlpha

virtual void disableAlpha ( )

Configures blit-op hardware for solid operation (no alpha-blending)

DMA_Interface

DMA_Interface (DMA_Queue & dmaQueue)

Constructs a DMA Interface object.

Parameters:

dmaQueue Reference to the queue of DMA operations.

enableAlpha

virtual void enableAlpha (uint8_t alpha)

Configures blit-op hardware for alpha-blending.

Parameters:

alpha The alpha-blending value to apply.

enableCopyWithTransparentPixels

virtual void enableCopyWithTransparentPixels (uint8_t alpha)

Configures blit-op hardware for alpha-blending while simultaneously skipping transparent pixels.

Parameters:

alpha The alpha-blending value to apply.

execute

virtual void execute ( )

Performs a queued blit-op.

executeCompleted

virtual void executeCompleted ( )

To be called when blit-op has been performed.

seedExecution

virtual void seedExecution ( )

Called when elements are added to the DMA-queue.

NOTE

The framebuffer must be locked before this method returns if the DMA-queue is non- empty.

setupDataCopy

virtual void setupDataCopy (const BlitOp & blitOp)

Configures blit-op hardware for a 2D copy as specified by blitOp.

Parameters:

blitOp The operation to execute.

Reimplemented by: **touchgfx::NoDMA::setupDataCopy**

setupDataFill

virtual void setupDataFill (const BlitOp & blitOp)

Configures blit-op hardware for a 2D fill as specified by blitOp.

Parameters:

blitOp The operation to execute.

Reimplemented by: **touchgfx::NoDMA::setupDataFill**

waitForFrameBufferSemaphore

virtual void waitForFrameBufferSemaphore ( )

Waits until framebuffer semaphore is available (i.e.

neither DMA or application is accessing the framebuffer).

Protected Attributes Documentation

isAllowed

bool isAllowed

true if DMA transfers are currently allowed.

isRunning

bool isRunning

true if a DMA transfer is currently ongoing.

queue

DMA_Queue & queue

Reference to the DMA queue.

Version: 4.16

DMA_Queue

This class provides an interface for a FIFO (circular) list used by DMA_Interface and descendants for storing BlitOp's.

Inherited by: LockFreeDMA_Queue

Public Functions

virtual bool	**isEmpty**() =0
	Query if this object is empty.
virtual bool	**isFull**() =0
	Query if this object is full.
virtual void	**pushCopyOf**(const **BlitOp** & op) =0
	Adds the specified blitop to the queue.
virtual	**~DMA_Queue**()
	Finalizes an instance of the **DMA_Queue** class.

Protected Functions

	DMA_Queue()
	Initializes a new instance of the **DMA_Queue** class.
virtual const **BlitOp** *	**first**() =0
	Gets the first element in the queue.
virtual void	**pop**() =0
	Pops an element from the queue.

Public Functions Documentation

isEmpty

virtual bool isEmpty ( ) =0

Query if this object is empty.

Returns:

true if the queue is empty.

Reimplemented by: **touchgfx::LockFreeDMA_Queue::isEmpty**

isFull

virtual bool isFull ( ) =0

Query if this object is full.

Returns:

true if the queue is full.

Reimplemented by: **touchgfx::LockFreeDMA_Queue::isFull**

pushCopyOf

virtual void pushCopyOf (const BlitOp & op)

Adds the specified blitop to the queue.

Parameters:

op The blitop to add.

Reimplemented by: **touchgfx::LockFreeDMA_Queue::pushCopyOf**

~DMA_Queue

virtual ~DMA_Queue ( )

Finalizes an instance of the **DMA_Queue** class.

Protected Functions Documentation

DMA_Queue

DMA_Queue ( )

Initializes a new instance of the **DMA_Queue** class.

first

virtual const BlitOp * first ( ) =0

Gets the first element in the queue.

Returns:

The first element in the queue.

Reimplemented by: **touchgfx::LockFreeDMA_Queue::first**

pop

virtual void pop ( ) =0

Pops an element from the queue.

Reimplemented by: **touchgfx::LockFreeDMA_Queue::pop**

Version: 4.16

DragEvent

A drag event. The only drag event currently supported is DRAGGED, which will be issued every time the input system detects a drag.

See: Event

Inherits from: Event

Public Types

enum	**DragEventType** { DRAGGED }
	Values that represent drag event types.

Public Functions

	DragEvent(**DragEventType** type, int16_t fromX, int16_t fromY, int16_t toX, int16_t toY)
	Initializes a new instance of the **DragEvent** class.
int16_t	**getDeltaX**() const
	Gets the distance in x coordinates (how long was the drag).
int16_t	**getDeltaY**() const
	Gets the distance in y coordinates (how long was the drag).
virtual **Event::EventType**	**getEventType**()
	Gets event type.
int16_t	**getNewX**() const
	Gets the new x coordinate (dragged to).
int16_t	**getNewY**() const
	Gets the new x coordinate (dragged to).
int16_t	**getOldX**() const
	Gets the x coordinate where the drag operation was started (dragged from).

int16_t	**getOldY**() const
	Gets the y coordinate where the drag operation was started (dragged from).

DragEventType	**getType**() const
	Gets the type of this drag event.

Additional inherited members

Public Types inherited from Event

enum	**EventType** { EVENT_CLICK, EVENT_DRAG, EVENT_GESTURE }
	The event types.

Public Functions inherited from Event

virtual	**~Event**()
	Finalizes an instance of the **Event** class.

Public Types Documentation

DragEventType

enum DragEventType

Values that represent drag event types.

DRAGGED	An enum constant representing the dragged option.

Public Functions Documentation

DragEvent

DragEvent (DragEventType type ,

```
int16_t fromX ,
int16_t fromY ,
int16_t toX ,
int16_t toY
)
```

Initializes a new instance of the **DragEvent** class.

Parameters:

type	The type of the drag event.
fromX	The x coordinate of the drag start position (dragged from)
fromY	The y coordinate of the drag start position (dragged from)
toX	The x coordinate of the new position (dragged to)
toY	The y coordinate of the new position (dragged to)

getDeltaX

int16_t getDeltaX ( ) const

Gets the distance in x coordinates (how long was the drag).

Returns:

The distance of this drag event.

getDeltaY

int16_t getDeltaY ( ) const

Gets the distance in y coordinates (how long was the drag).

Returns:

The distance of this drag event.

getEventType

virtual Event::EventType getEventType ( )

Gets event type.

Returns:

The type of this event.

Reimplements: touchgfx::Event::getEventType

getNewX

int16_t getNewX ( ) const

Gets the new x coordinate (dragged to).

Returns:

The new x coordinate (dragged to).

getNewY

int16_t getNewY ( ) const

Gets the new x coordinate (dragged to).

Returns:

The new y coordinate (dragged to).

getOldX

int16_t getOldX ( ) const

Gets the x coordinate where the drag operation was started (dragged from).

Returns:

The x coordinate where the drag operation was started (dragged from).

getOldY

int16_t getOldY ( ) const

Gets the y coordinate where the drag operation was started (dragged from).

Returns:

The y coordinate where the drag operation was started (dragged from).

getType

DragEventType getType ( ) const

Gets the type of this drag event.

Returns:

The type of this drag event.

Version: 4.16

Draggable

Mix-in class that extends a class to become Draggable, which means that the object on screen can be freely moved around using the touch screen.

Template Parameters:

- **T** specifies the type to extend with the Draggable behavior.

Inherits from: T

Inherited by: Snapper< T >

Public Functions

	Draggable()
	Initializes a new instance of the **Draggable** class.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Called when dragging the **Draggable** object.

Public Functions Documentation

Draggable

Draggable ( )

Initializes a new instance of the **Draggable** class.

Make the object touchable.

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Called when dragging the **Draggable** object.

The object is moved according to the drag event.

Parameters:

evt The drag event.

Reimplemented by: **touchgfx::Snapper::handleDragEvent**

Version: 4.16

Drawable

The Drawable class is an abstract definition of something that can be drawn. In the composite design pattern, the Drawable is the component interface. Drawables can be added to a screen as a tree structure through the leaf node class Widget and the Container class. A Drawable contains a pointer to its next sibling and a pointer to its parent node. These are maintained by the Container to which the Drawable is added.

The Drawable interface contains two pure virtual functions which must be implemented by widgets, namely draw() and getSolidRect(). In addition it contains general functionality for receiving events and navigating the tree structure.

The coordinates of a Drawable are always relative to its parent node.

See: Widget, Container

Inherited by: Container, Widget

Public Functions

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.

int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.

virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.

bool	**touchable**
	True if this drawable should receive touch events.

bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

childGeometryChanged

virtual void childGeometryChanged ( )

This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

Currently only used in **ScrollableContainer** to redraw scrollbars when the size of the scrolling contents changes.

Reimplemented by: **touchgfx::ScrollableContainer::childGeometryChanged**

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplemented by: **touchgfx::WipeTransition::FullSolidRect::draw**, **touchgfx::Box::draw**, **touchgfx::BoxWithBorder::draw**, **touchgfx::ButtonWithLabel::draw**, **touchgfx::TextArea::draw**, **touchgfx::TextAreaWithOneWildcard::draw**, **touchgfx::TextAreaWithTwoWildcards::draw**, **touchgfx::Container::draw**, **touchgfx::Button::draw**, **touchgfx::ButtonWithIcon::draw**, **touchgfx::CanvasWidget::draw**, **touchgfx::GraphElementGridX::draw**, **touchgfx::GraphElementGridY::draw**, **touchgfx::GraphElementVerticalGapLine::draw**, **touchgfx::GraphElementHistogram::draw**, **touchgfx::GraphElementBoxes::draw**,

touchgfx::GraphLabelsX::draw, **touchgfx::GraphLabelsY::draw**, **touchgfx::GraphTitle::draw**, **touchgfx::Image::draw**, **touchgfx::Keyboard::draw**, **touchgfx::PixelDataWidget::draw**, **touchgfx::RadioButton::draw**, **touchgfx::ScalableImage::draw**, **touchgfx::SnapshotWidget::draw**, **touchgfx::TextureMapper::draw**, **touchgfx::TiledImage::draw**, **touchgfx::TouchArea::draw**

Drawable

Drawable ( )

Initializes a new instance of the **Drawable** class.

drawToDynamicBitmap

void drawToDynamicBitmap (BitmapId id)

Render the **Drawable** object into a dynamic bitmap.

Parameters:

id The target dynamic bitmap to use for rendering.

getAbsoluteRect

Rect getAbsoluteRect ( ) const

Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.

Returns:

The rectangle this **Drawable** covers expressed in absolute coordinates.

See also:

getRect, **translateRectToAbsolute**

getFirstChild

virtual Drawable * getFirstChild ( )

Function for obtaining the first child of this drawable if any.

Returns:

A pointer on the first child drawable if any.

See also:

Container::getFirstChild

Reimplemented by: **touchgfx::Container::getFirstChild**

getHeight

int16_t getHeight ( ) const

Gets the height of this **Drawable**.

Returns:

The height of this **Drawable**.

getLastChild

```
virtual void getLastChild ( int16_t      x ,   =0
                            int16_t      y ,   =0
                            Drawable ** last   =0
                            )             =0
```

Function for obtaining the the last child of this drawable that intersects with the specified point.

The last child is the **Drawable** that is drawn last and therefore the topmost child. Used in input event handling for obtaining the appropriate drawable that should receive the event.

Parameters:

x	The point of intersection expressed in coordinates relative to the parent.
y	The point of intersection expressed in coordinates relative to the parent.
last	Last (topmost) **Drawable** on the given coordinate.

NOTE

Input events must be delegated to the last drawable of the tree (meaning highest z- order / front-most drawable).

Reimplemented by: **touchgfx::Container::getLastChild**, **touchgfx::ScrollableContainer::getLastChild**, **touchgfx::Widget::getLastChild**

getNextSibling

Drawable * getNextSibling ( )

Gets the next sibling node.

This will be the next **Drawable** that has been added to the same **Container** as this **Drawable**.

Returns:

The next sibling. If there are no more siblings, the return value is 0.

getParent

Drawable * getParent ( ) const

Returns the parent node.

For the root container, the return value is 0.

Returns:

The parent node. For the root container, the return value is 0.

NOTE

A disconnected Drawable also has parent 0 which may cause strange side effects.

getRect

const Rect & getRect ( ) const

Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.

Returns:

The rectangle this **Drawable** covers expressed in coordinates relative to its parent.

See also:

getAbsoluteRect

getSolidRect

virtual Rect getSolidRect ( ) const =0

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplemented by: **touchgfx::Container::getSolidRect**, **touchgfx::WipeTransition::FullSolidRect::getSolidRect**, **touchgfx::Box::getSolidRect**, **touchgfx::Button::getSolidRect**, **touchgfx::CanvasWidget::getSolidRect**, **touchgfx::Image::getSolidRect**, **touchgfx::PixelDataWidget::getSolidRect**, **touchgfx::RadioButton::getSolidRect**, **touchgfx::ScalableImage::getSolidRect**, **touchgfx::SnapshotWidget::getSolidRect**, **touchgfx::TextArea::getSolidRect**, **touchgfx::TextureMapper::getSolidRect**, **touchgfx::TiledImage::getSolidRect**, **touchgfx::TouchArea::getSolidRect**

getSolidRectAbsolute

virtual Rect getSolidRectAbsolute ( )

Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.

Will recursively traverse to the root of the tree to find the proper location of the rectangle on the display.

Returns:

The (largest) solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.

getVisibleRect

virtual void getVisibleRect (Rect & rect)

Function for finding the visible part of this drawable.

If the parent node has a smaller area than this **Drawable**, or if the **Drawable** is placed "over the edge" of the parent, the parent will act as a view port, cutting off the parts of this **Drawable** that are outside the region. Traverses the tree and yields a result expressed in absolute coordinates.

Parameters:

rect The region of the **Drawable** that is visible.

getWidth

int16_t getWidth ( ) const

Gets the width of this **Drawable**.

Returns:

The width of this **Drawable**.

getX

int16_t getX ( ) const

Gets the x coordinate of this **Drawable**, relative to its parent.

Returns:

The x value, relative to the parent.

getY

int16_t getY ( ) const

Gets the y coordinate of this **Drawable**, relative to its parent.

Returns:

The y value, relative to the parent.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplemented by: **touchgfx::ClickButtonTrigger::handleClickEvent**, **touchgfx::RepeatButtonTrigger::handleClickEvent**, **touchgfx::ToggleButtonTrigger::handleClickEvent**, **touchgfx::TouchButtonTrigger::handleClickEvent**, **touchgfx::Slider::handleClickEvent**, **touchgfx::AbstractButton::handleClickEvent**, **touchgfx::AbstractDataGraph::handleClickEvent**, **touchgfx::RadioButton::handleClickEvent**, **touchgfx::RepeatButton::handleClickEvent**, **touchgfx::ToggleButton::handleClickEvent**, **touchgfx::TouchArea::handleClickEvent**, **touchgfx::ScrollableContainer::handleClickEvent**, **touchgfx::ScrollList::handleClickEvent**, **touchgfx::ScrollWheelBase::handleClickEvent**, **touchgfx::SwipeContainer::handleClickEvent**, **touchgfx::Keyboard::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplemented by: **touchgfx::Slider::handleDragEvent**, **touchgfx::AbstractDataGraph::handleDragEvent**, **touchgfx::ScrollableContainer::handleDragEvent**, **touchgfx::ScrollBase::handleDragEvent**, **touchgfx::ScrollWheelBase::handleDragEvent**, **touchgfx::SwipeContainer::handleDragEvent**, **touchgfx::Keyboard::handleDragEvent**, **touchgfx::TouchArea::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Defines the event handler interface for GestureEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **GestureEvent** received from the **HAL**.

Reimplemented by: **touchgfx::ScrollableContainer::handleGestureEvent**, **touchgfx::ScrollBase::handleGestureEvent**, **touchgfx::ScrollWheelBase::handleGestureEvent**, **touchgfx::SwipeContainer::handleGestureEvent**

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplemented by: **touchgfx::RepeatButtonTrigger::handleTickEvent**, **touchgfx::AbstractProgressIndicator::handleTickEvent**, **touchgfx::ScrollableContainer::handleTickEvent**, **touchgfx::ScrollBase::handleTickEvent**, **touchgfx::SlideMenu::handleTickEvent**, **touchgfx::SwipeContainer::handleTickEvent**, **touchgfx::ZoomAnimationImage::handleTickEvent**, **touchgfx::MoveAnimator::handleTickEvent**, **touchgfx::AnimatedImage::handleTickEvent**, **touchgfx::AnimationTextureMapper::handleTickEvent**, **touchgfx::RepeatButton::handleTickEvent**

invalidate

virtual void invalidate ( ) const

Tell the framework that this entire **Drawable** needs to be redrawn.

It is the same as calling **invalidateRect()** with **Rect**(0, 0, **getWidth()**, **getHeight()**).

See also:

invalidateRect

Reimplemented by: **touchgfx::CanvasWidget::invalidate**

invalidateRect

virtual void invalidateRect (Rect & invalidatedArea)

Request that a region of this drawable is redrawn.

Will recursively traverse the tree towards the root, and once reached, issue a draw operation. When this function returns, the specified invalidated area has been redrawn for all appropriate Drawables covering the region.

To invalidate the entire **Drawable**, use **invalidate()**

Parameters:

invalidatedArea The area of this drawable to redraw expressed in relative coordinates.

See also:

invalidate

Reimplemented by: **touchgfx::CacheableContainer::invalidateRect**

isTouchable

bool isTouchable ( ) const

Gets whether this **Drawable** receives touch events or not.

Returns:

True if touch events are received.

See also:

setTouchable

isVisible

bool isVisible ( ) const

Gets whether this **Drawable** is visible.

Returns:

true if the **Drawable** is visible.

See also:

setVisible

moveRelative

```
virtual void moveRelative ( int16_t x ,
                            int16_t y
                            )
```

Moves the drawable.

Parameters:

x The relative position to move to.
y The relative position to move to.

NOTE

Will redraw the appropriate areas of the screen.

See also:

moveTo, **setXY**

moveTo

```
virtual void moveTo ( int16_t x ,
                      int16_t y
                      )
```

Moves the drawable.

Parameters:

x The absolute position to move to.
y The absolute position to move to.

NOTE

Will redraw the appropriate areas of the screen.

See also:

moveRelative, **setXY**

setHeight

```
virtual void setHeight ( int16_t height )
```

Sets the height of this drawable.

Parameters:

height The new height.

NOTE

For most **Drawable** widgets, changing this does normally not automatically yield a redraw.

Reimplemented by: **touchgfx::DigitalClock::setHeight**, **touchgfx::DrawableList::setHeight**, **touchgfx::ScrollBase::setHeight**, **touchgfx::ScrollWheelWithSelectionStyle::setHeight**, **touchgfx::ZoomAnimationImage::setHeight**, **touchgfx::Gauge::setHeight**, **touchgfx::AbstractDataGraph::setHeight**

setPosition

```
void setPosition ( const Drawable & drawable )
```

Sets the position of the **Drawable** to the same as the given **Drawable**.

This will copy the x, y, width and height.

Parameters:

drawable The drawable.

See also:

setPosition(int16_t,int16_t,int16_t,int16_t)

setPosition

```
void setPosition ( int16_t x ,
                   int16_t y ,
                   int16_t width ,
                   int16_t height
                   )
```

Sets the size and position of this **Drawable**, relative to its parent.

The same as calling **setXY()**, **setWidth()** and **setHeight()** in that order.

Parameters:

x The x coordinate of this **Drawable** relative to its parent.
y The y coordinate of this **Drawable** relative to its parent.
width The width of this **Drawable**.

height The height of this **Drawable**.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

setTouchable

void setTouchable (bool touch)

Controls whether this **Drawable** receives touch events or not.

Parameters:

touch If true it will receive touch events, if false it will not.

setVisible

void setVisible (bool vis)

Controls whether this **Drawable** should be visible.

Only visible Drawables will have their draw function called. Additionally, invisible drawables will not receive input events.

Parameters:

vis true if this **Drawable** should be visible. By default, drawables are visible unless this function has been called with false as argument.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most **Drawable** widgets, changing this does normally not automatically yield a redraw.

Reimplemented by: **touchgfx::DigitalClock::setWidth**, **touchgfx::DrawableList::setWidth**, **touchgfx::ScrollBase::setWidth**, **touchgfx::ScrollWheelWithSelectionStyle::setWidth**, **touchgfx::ZoomAnimationImage::setWidth**, **touchgfx::Gauge::setWidth**, **touchgfx::AbstractDataGraph::setWidth**

setWidthHeight

void setWidthHeight (const Bitmap & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

Parameters:

bitmap The **Bitmap** to copy the width and height from.

See also:

setWidthHeight(int16_t,int16_t)

setWidthHeight

void setWidthHeight (const Drawable & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

Parameters:

drawable The **Drawable** to copy the width and height from.

See also:

setWidthHeight(int16_t,int16_t)

setWidthHeight

void setWidthHeight (const Rect & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

Parameters:

rect The **Rect** to copy the width and height from.

See also:

setWidthHeight(int16_t,int16_t)

setWidthHeight

```
void setWidthHeight ( int16_t width ,
                      int16_t height
                      )
```

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

Parameters:

width The width.
height The height.

setX

```
virtual void setX ( int16_t x )
```

Sets the x coordinate of this **Drawable**, relative to its parent.

Parameters:

x The new x value, relative to the parent. A negative value is allowed.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

setXY

```
void setXY ( const Drawable & drawable )
```

Sets the x and y coordinates of this **Drawable**.

Parameters:

drawable The **Drawable** to copy the x and y coordinates from.

See also:

setXY(int16_t,int16_t)

setXY

void setXY (int16_t x ,
int16_t y
)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

The same as calling **setX()** followed by calling **setY()**.

Parameters:

x The new x value, relative to the parent. A negative value is allowed.
y The new y value, relative to the parent. A negative value is allowed.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

See also:

moveTo

setY

virtual void setY (int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

Parameters:

y The new y value, relative to the parent. A negative value is allowed.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

translateRectToAbsolute

virtual void translateRectToAbsolute (Rect & r)

Helper function for converting a region of this **Drawable** to absolute coordinates.

Parameters:

r The **Rect** to translate.

~Drawable

virtual ~Drawable ( )

Finalizes an instance of the **Drawable** class.

Protected Attributes Documentation

nextSibling

Drawable * nextSibling

Pointer to the next **Drawable**.

parent

Drawable * parent

Pointer to this drawable's parent.

rect

Rect rect

The coordinates of this **Drawable**, relative to its parent.

touchable

bool touchable

True if this drawable should receive touch events.

visible

bool visible

True if this drawable should be drawn.

Version: 4.16

DrawableList

A container able to display many items using only a few drawables. This is done by only having drawables for visible items, and populating these drawables with new content when each of these become visible.

This means that all drawables must have an identical structure in some way, for example an Image or a Container with a button and a text.

Inherits from: Container, Drawable

Public Functions

	DrawableList()
virtual bool	**getCircular**() const
	Gets the circular setting, previously set using setCircular().
int16_t	**getDrawableIndex**(int16_t itemIndex, int16_t prevDrawableIndex =-1) const
	Gets the drawable index of an item.
int16_t	**getDrawableIndices**(int16_t itemIndex, int16_t * drawableIndexArray, int16_t arraySize) const
	Gets drawable indices.
virtual int16_t	**getDrawableMargin**() const
	Gets drawable margin.
virtual int16_t	**getDrawableSize**() const
	Gets drawable size as set by setDrawables.
virtual bool	**getHorizontal**() const
	Gets the orientation of the drawables, previously set using setHorizontal().
int16_t	**getItemIndex**(int16_t drawableIndex) const
	Gets item stored in a given **Drawable**.
virtual int16_t	**getItemSize**() const
	Gets size of each item.

int16_t	**getNumberOfDrawables**() const
	Gets number of drawables based on the size of each item and the size of the widget.
int16_t	**getNumberOfItems**() const
	Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().
int32_t	**getOffset**() const
	Gets offset, as previously set using setOffset().
int16_t	**getRequiredNumberOfDrawables**() const
	Gets required number of drawables.
void	**itemChanged**(int16_t itemIndex)
	Item changed.
void	**refreshDrawables**()
	Refresh drawables.
virtual void	**setCircular**(bool circular)
	Sets whether the list is circular (infinite) or not.
virtual void	**setDrawables**(**DrawableListItemsInterface** & drawableListItems, int16_t drawableItemIndexOffset, **GenericCallback**< **DrawableListItemsInterface** *, int16_t, int16_t > & updateDrawableCallback)
	Sets the drawables parameters.
void	**setDrawableSize**(int16_t drawableSize, int16_t drawableMargin)
	Sets drawables size.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setHorizontal**(bool horizontal)
	Sets a horizontal or vertical layout.
void	**setNumberOfItems**(int16_t numberOfItems)
	Sets number of items in the list.
void	**setOffset**(int32_t ofs)
	Sets virtual coordinate.
virtual void	**setWidth**(int16_t width)

	Sets the width of this drawable.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()

	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()

	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

DrawableList

```
DrawableList ( )
```

getCircular

```
virtual bool getCircular ( ) const
```

Gets the circular setting, previously set using setCircular().

Returns:

True if the list is circular (infinite), false if the list is not circular (finite).

See also:

ScrollBase::getCircular, **setCircular**

getDrawableIndex

```
int16_t getDrawableIndex ( int16_t itemIndex ,                const
                           int16_t prevDrawableIndex =-1      const
                           )       const
```

Gets the drawable index of an item.

If the number of items is smaller than the number of drawables and the **DrawableList** is circular, the same item can be in more than one drawable. In that case, calling this function again with the previously returned index as second parameter, the index of the next drawable containing the item will be returned.

Parameters:

itemIndex	Index of the item.
prevDrawableIndex	(Optional) Index of the previous drawable. If given, search starts after this index.

Returns:

The first drawable index with the given item. Returns -1 if the item is not in a drawable.

See also:

getDrawableIndices

getDrawableIndices

```
int16_t getDrawableIndices ( int16_t itemIndex , const
                             int16_t * drawableIndexArray , const
                             int16_t arraySize const
                             ) const
```

Gets drawable indices.

Useful when the number of items is smaller than the number of drawables as the same item might be in more than one drawable on the screen (if the **DrawableList** is circular). The passed array will be filled with the drawable indices and the number of indices found is returned.

Parameters:

itemIndex	Zero-based index of the item.
drawableIndexArray	Array where the drawable indices are stored.
arraySize	Size of drawable array.

Returns:

The number of drawable indices found.

See also:

getItemIndex, **setCircular**, **getDrawableIndex**

getDrawableMargin

```
virtual int16_t getDrawableMargin ( ) const
```

Gets drawable margin.

Gets drawable margin as set by setDrawables.

Returns:

The drawable margin.

getDrawableSize

```
virtual int16_t getDrawableSize ( ) const
```

Gets drawable size as set by setDrawables.

Returns:

The drawable size.

See also:

setDrawables

getHorizontal

virtual bool getHorizontal ( ) const

Gets the orientation of the drawables, previously set using setHorizontal().

Returns:

True if it horizontal, false if it is vertical.

See also:

ScrollBase::getHorizontal, **setHorizontal**

getItemIndex

int16_t getItemIndex (int16_t drawableIndex)

Gets item stored in a given **Drawable**.

Parameters:

drawableIndex Zero-based index of the drawable.

Returns:

The item index.

getItemSize

virtual int16_t getItemSize ( ) const

Gets size of each item.

This equals the drawable size plus the drawable margin as set in **setDrawables()**. Equals **getDrawableSize()** + 2 * **getDrawableMargin()**.

Returns:

The item size.

NOTE

Not the same as **getDrawableSize()**.

See also:

setDrawables, **setDrawableSize**, **getDrawableMargin**

getNumberOfDrawables

int16_t getNumberOfDrawables ( ) const

Gets number of drawables based on the size of each item and the size of the widget.

Returns:

The number of drawables.

See also:

setDrawables

getNumberOfItems

int16_t getNumberOfItems ( ) const

Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().

Returns:

The number of items.

See also:

setNumberOfItems

getOffset

int32_t getOffset ( ) const

Gets offset, as previously set using setOffset().

Returns:

The virtual offset.

See also:

setOffset

getRequiredNumberOfDrawables

int16_t getRequiredNumberOfDrawables ( ) const

Gets required number of drawables.

After setting up the **DrawableList** it is possible to request how many drawables are needed to ensure that the list can always be drawn properly. If the **DrawableList** has been setup with fewer Drawables than the required number of drawables, part of the lower part of the **DrawableList** will look wrong.

The number of required drawables depend on the size of the widget and the size of the drawables and the margin around drawables. If there are fewer drawables than required, the widget will not display correctly. If there are more drawables than required, some will be left unused.

Returns:

The required number of drawables.

See also:

setDrawables

itemChanged

void itemChanged (int16_t itemIndex)

Item changed.

Item changed and drawables containing this item must be updated. This function can be called when an item has changed and needs to be updated on screen. If the given item is displayed on screen, possible more than once for cyclic lists, each drawable is request to refresh its content to reflect the new value.

Parameters:

itemIndex Zero-based index of the item.

refreshDrawables

void refreshDrawables ( )

Refresh drawables.

Useful to call if the number or items, their size or other properties have changed.

setCircular

virtual void setCircular (bool circular)

Sets whether the list is circular (infinite) or not.

A circular list is a list where the first drawable re-appears after the last item in the list - and the last item in the list appears before the first item in the list.

Parameters:

circular True if the list should be circular, false if the list should not be circular.

See also:

ScrollBase::setCircular, **getCircular**

setDrawables

virtual void setDrawables (DrawableListItemsInterface & drawableListItems ,
int16_t drawableItemIndexOffset ,
GenericCallback< DrawableListItemsInterface *, int16_t, int16_t > & updateDrawableCallback
)

Sets the drawables parameters.

These parameters are

- The access class to the array of drawables
- The offset in the drawableListItems array to start using drawable and
- **Callback** to update the contents of a drawable.

Parameters:

drawableListItems	Number of drawables allocated.
drawableItemIndexOffset	The offset of the drawable item.
updateDrawableCallback	A callback to update the contents of a drawable.

See also:

getRequiredNumberOfDrawables

setDrawableSize

```
void setDrawableSize ( int16_t drawableSize ,
                       int16_t drawableMargin
                       )
```

Sets drawables size.

The drawable is is the size of each drawable in the list in the set direction of the list (this is enforced by the **DrawableList** class). The specified margin is added above and below each item for spacing. The entire size of an item is thus size + 2 * spacing.

For a horizontal list each element will be *drawableSize* high and have the same width as set using **setWidth()**. For a vertical list each element will be *drawableSize* wide and have the same height as set using **setHeight()**.

Parameters:

drawableSize	The size of the drawable.
drawableMargin	The margin around drawables (margin before and margin after).

See also:

setWidth, **setHeight**, **setHorizontal**

setHeight

```
virtual void setHeight ( int16_t height )
```

Sets the height of this drawable.

Parameters:

height	The new height.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.If the list is horizontal, the height is also propagated to all drawables in the list.

Reimplements: **touchgfx::Drawable::setHeight**

setHorizontal

```
virtual void setHorizontal ( bool horizontal )
```

Sets a horizontal or vertical layout.

If parameter horizontal is set true, all drawables are arranged side by side. If horizontal is set false, the drawables are arranged above and below each other (vertically).

Parameters:

horizontal True to align drawables horizontal, false to align drawables vertically.

NOTE

Default value is false, i.e. vertical layout.

See also:

ScrollBase::setHorizontal, **getHorizontal**

setNumberOfItems

void setNumberOfItems (int16_t numberOfItems)

Sets number of items in the list.

This forces all drawables to be updated to ensure that the content is correct.

Parameters:

numberOfItems Number of items.

NOTE

The DrawableList is refreshed to reflect the change.

setOffset

void setOffset (int32_t ofs)

Sets virtual coordinate.

Does not move to the given coordinate, but places the drawables and fill correct content into the drawables to give the impression that everything has been scrolled to the given coordinate.

Setting a value of 0 means that item 0 is at the start of the **DrawableList**. Setting a value of "-getItemSize()" places item 0 outside the start of the **DrawableList** and item 1 at the start of it.

Items that are completely outside of view, will be updated with new content using the provided callback from **setDrawables()**. Care is taken to not fill drawables more than strictly required.

Parameters:

ofs The virtual coordinate.

See also:

getOffset, **setDrawables**

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most **Drawable** widgets, changing this does normally not automatically yield a redraw.If the list is vertical, the width is also propagated to all drawables in the list.

Reimplements: **touchgfx::Drawable::setWidth**

Version: 4.16

DrawableListItems

An array of drawables used by DrawableList. This class is used to ease the setup of a callback function to get access to a specific drawable in the array.

Example usage:

```
static const int NUMBER_OF_DRAWABLES = 5;
DrawableListItems<TextAreaWithOneWildcardContainer, NUMBER_OF_DRAWABLES> menuItems;
```

Template Parameters:

- **TYPE** Type of the drawables. Can be a simple drawable, such as Image or a more complex container.
- **SIZE** Size of the array. This is the number of drawables to allocate and should be all visible drawables on the screen at any given time.

Inherits from: DrawableListItemsInterface

Public Functions

virtual **Drawable** *	**getDrawable**(int16_t index)
	Gets a drawable at a given index.
virtual int16_t	**getNumberOfDrawables**()
	Gets number of drawables.
TYPE &	**operator[]**(int index)
	Array indexer operator.

Public Attributes

TYPE	**element**
	The array of drawables.

Additional inherited members

Public Functions inherited from DrawableListItemsInterface

virtual	**~DrawableListItemsInterface**()
	Finalizes an instance of the **DrawableListItemsInterface** class.

Public Functions Documentation

getDrawable

virtual Drawable * getDrawable (int16_t index)

Gets a drawable at a given index.

Parameters:

index Zero-based index of the drawable.

Returns:

Null if it fails, else the drawable.

Reimplements: **touchgfx::DrawableListItemsInterface::getDrawable**

getNumberOfDrawables

virtual int16_t getNumberOfDrawables ( )

Gets number of drawables.

Returns:

The number of drawables.

Reimplements: **touchgfx::DrawableListItemsInterface::getNumberOfDrawables**

operator[]

TYPE & operator[] (int index)

Array indexer operator.

Parameters:

index Zero-based index of elements to access.

Returns:

The indexed value.

Public Attributes Documentation

element

TYPE element

The array of drawables.

Version: 4.16

DrawableListItemsInterface

A drawable list items interface. Used to pass the allocated array of drawable elements to ScrollList::setDrawables(), ScrollWheel::setDrawables() or ScrollWheelWithSelectionStyle::setDrawables(). Provides easy access to each element in the array as well as the size of the array.

See: ScrollList::setDrawables, ScrollWheel::setDrawables, ScrollWheelWithSelectionStyle::setDrawables

Inherited by: DrawableListItems< TYPE, SIZE >

Public Functions

virtual **Drawable** *	**getDrawable**(int16_t index) =0
	Gets a drawable at a given index.
virtual int16_t	**getNumberOfDrawables**() =0
	Gets number of drawables.
virtual	**~DrawableListItemsInterface**()
	Finalizes an instance of the **DrawableListItemsInterface** class.

Public Functions Documentation

getDrawable

virtual Drawable * getDrawable (int16_t index)

Gets a drawable at a given index.

Parameters:

index Zero-based index of the drawable.

Returns:

Null if it fails, else the drawable.

Reimplemented by: **touchgfx::DrawableListItems::getDrawable**

getNumberOfDrawables

virtual int16_t getNumberOfDrawables ( ) =0

Gets number of drawables.

Returns:

The number of drawables.

Reimplemented by: **touchgfx::DrawableListItems::getNumberOfDrawables**

~DrawableListItemsInterface

virtual ~DrawableListItemsInterface ( )

Finalizes an instance of the **DrawableListItemsInterface** class.

Version: 4.16

DrawingSurface

The destination of a draw operation. Contains a pointer to where to draw and the stride of the drawing surface.

Public Attributes

uint16_t *	**address**
	The bits.
int32_t	**stride**
	The stride.

Public Attributes Documentation

address

uint16_t * address

The bits.

stride

int32_t stride

The stride.

Version: 4.16

DrawTextureMapScanLineBase

Base class for drawing scanline by the texture mapper.

Public Functions

virtual void	**drawTextureMapScanLineSubdivisions**(int subdivisions, const int widthModLength, int pixelsToDraw, const int affineLength, float oneOverZRight, float UOverZRight, float VOverZRight, **fixed16_16** U, **fixed16_16** V, **fixed16_16** deltaU, **fixed16_16** deltaV, float ULeft, float VLeft, float URight, float VRight, float ZRight, const **DrawingSurface** & dest, const int destX, const int destY, const **TextureSurface** & texture, uint8_t alpha, const float dOneOverZdXAff, const float dUOverZdXAff, const float dVOverZdXAff) =0
	Draw texture map scan line subdivisions.
virtual	**~DrawTextureMapScanLineBase**()
	Finalizes an instance of the **DrawTextureMapScanLineBase** class.

Protected Functions

FORCE_INLINE_FUNCTION void	**drawTextureMapNextSubdivision**(float & ULeft, float & VLeft, float & ZRight, float & URight, float & VRight, float & oneOverZRight, const float dOneOverZdXAff, float & UOverZRight, const float dUOverZdXAff, float & VOverZRight, const float dVOverZdXAff, const int affineLength, **fixed16_16** & U, **fixed16_16** & V, **fixed16_16** & deltaU, **fixed16_16** & deltaV)
	Draw texture map next subdivision.
FORCE_INLINE_FUNCTION bool	**is1Inside**(int value, int limit)
	Check if value is inside [0..limit[.
FORCE_INLINE_FUNCTION bool	**is1x1Inside**(int x, int y, int width, int height)
	Check if (x,y) is inside ([0..width[, [0..height[)
FORCE_INLINE_FUNCTION bool	**is2Inside**(int value, int limit)
	Check if both value and value+1 are inside [0..limit[.
FORCE_INLINE_FUNCTION bool	**is2PartiallyInside**(int value, int limit)
	Check if either value or value+1 is inside [0..limit[.

FORCE_INLINE_FUNCTION bool	**is2x2Inside**(int x, int y, int width, int height)
	Check if both (x,y) and (x+1,y+1) are inside ([0..width[,[0..height[)

FORCE_INLINE_FUNCTION bool	**is2x2PartiallyInside**(int x, int y, int width, int height)
	Check if either (x,y) or (x+1,y+1) is inside ([0..width[,[0..height[)

Protected Attributes

const **fixed16_16**	**half**
	1/2 in fixed16_16 format

Public Functions Documentation

drawTextureMapScanLineSubdivisions

virtual void drawTextureMapScanLineSubdivisions (	int	subdivisions ,	=0
	const int	widthModLength ,	=0
	int	pixelsToDraw ,	=0
	const int	affineLength ,	=0
	float	oneOverZRight ,	=0
	float	UOverZRight ,	=0
	float	VOverZRight ,	=0
	fixed16_16	U ,	=0
	fixed16_16	V ,	=0
	fixed16_16	deltaU ,	=0
	fixed16_16	deltaV ,	=0
	float	ULeft ,	=0
	float	VLeft ,	=0
	float	URight ,	=0
	float	VRight ,	=0
	float	ZRight ,	=0
	const DrawingSurface &	dest ,	=0
	const int	destX ,	=0
	const int	destY ,	=0
	const TextureSurface &	texture ,	=0
	uint8_t	alpha ,	=0
	const float	dOneOverZdXAff ,	=0
	const float	dUOverZdXAff ,	=0

	const float	dVOverZdXAff	=0
	)	=0	

Draw texture map scan line subdivisions.

Parameters:

subdivisions	The number of subdivisions.
widthModLength	Remainder of length (after subdivisions).
pixelsToDraw	The pixels to draw.
affineLength	Length of one subdivision.
oneOverZRight	1/Z right.
UOverZRight	U/Z right.
VOverZRight	V/Z right.
U	U Coordinate in fixed16_16 notation.
V	V Coordinate in fixed16_16 notation.
deltaU	U delta to get to next pixel coordinate.
deltaV	V delta to get to next pixel coordinate.
ULeft	The left U.
VLeft	The left V.
URight	The right U.
VRight	The right V.
ZRight	The right Z.
dest	Destination drawing surface.
destX	Destination x coordinate.
destY	Destination y coordinate.
texture	The texture.
alpha	The global alpha.
dOneOverZdXAff	1/ZdX affine.
dUOverZdXAff	U/ZdX affine.
dVOverZdXAff	V/ZdX affine.

~DrawTextureMapScanLineBase

virtual ~DrawTextureMapScanLineBase ( )

Finalizes an instance of the **DrawTextureMapScanLineBase** class.

Protected Functions Documentation

drawTextureMapNextSubdivision

FORCE_INLINE_FUNCTION void drawTextureMapNextSubdivision	(float &	ULeft ,
	float &	VLeft ,
	float &	ZRight ,
	float &	URight ,
	float &	VRight ,
	float &	oneOverZRight ,
	const float	dOneOverZdXAff ,
	float &	UOverZRight ,
	const float	dUOverZdXAff ,
	float &	VOverZRight ,
	const float	dVOverZdXAff ,
	const int	affineLength ,
	fixed16_16 &	U ,
	fixed16_16 &	V ,
	fixed16_16 &	deltaU ,
	fixed16_16 &	deltaV
	)	

Draw texture map next subdivision.

Parameters:

ULeft	U left.
VLeft	V left.
ZRight	Z right.
URight	U right.
VRight	V right.
oneOverZRight	1/Z right.
dOneOverZdXAff	d1/ZdX affine.
UOverZRight	U/Z right.
dUOverZdXAff	dU/ZdX affine.
VOverZRight	V/Z right.
dVOverZdXAff	dV/ZdX affine.
affineLength	Length of the affine.
U	**Bitmap** X in fixed16_16.
V	**Bitmap** Y in fixed16_16.
deltaU	U delta.
deltaV	V delta.

is1Inside

```
FORCE_INLINE_FUNCTION bool is1Inside ( int value ,
                                       int limit
                                       )
```

Check if value is inside [0..limit[.

Parameters:

value	Value to check.
limit	Upper limit.

Returns:

true if value is inside given limit.

is1x1Inside

```
FORCE_INLINE_FUNCTION bool is1x1Inside ( int x ,
                                         int y ,
                                         int width ,
                                         int height
                                         )
```

Check if (x,y) is inside ([0..width[, [0..height[)

Parameters:

x	X coordinate.
y	Y coordinate.
width	X limit.
height	Y limit.

Returns:

true if (x,y) is inside given limits.

is2Inside

```
FORCE_INLINE_FUNCTION bool is2Inside ( int value ,
                                       int limit
                                       )
```

Check if both value and value+1 are inside [0..limit[.

Parameters:

value Value to check.
limit Upper limit.

Returns:

true if value and value+1 are inside given limit.

is2PartiallyInside

```
FORCE_INLINE_FUNCTION bool is2PartiallyInside ( int value ,
                                                int limit
                                                )
```

Check if either value or value+1 is inside [0..limit[.

Parameters:

value Value to check.
limit Upper limit.

Returns:

true if either value or value+1 is inside given limit.

is2x2Inside

```
FORCE_INLINE_FUNCTION bool is2x2Inside ( int x ,
                                         int y ,
                                         int width ,
                                         int height
                                         )
```

Check if both (x,y) and (x+1,y+1) are inside ([0..width[,[0..height[)

Parameters:

x X coordinate.
y Y coordinate.
width X limit.
height Y limit.

Returns:

true if (x,y) and (x+1,y+1) are inside given limits.

is2x2PartiallyInside

```
FORCE_INLINE_FUNCTION bool is2x2PartiallyInside ( int x ,
                                                  int y ,
                                                  int width ,
                                                  int height
                                                  )
```

Check if either (x,y) or (x+1,y+1) is inside ([0..width[,[0..height[)

Parameters:

x	X coordinate.
y	Y coordinate.
width	X limit.
height	Y limit.

Returns:

true if either (x,y) or (x+1,y+1) is inside given limits.

Protected Attributes Documentation

half

const fixed16_16 half = 0x8000

1/2 in fixed16_16 format

Version: 4.16

DynamicBitmapData

Data of a dynamic Bitmap.

Public Attributes

uint8_t	**customSubformat**
	Custom format specifier.
uint8_t	**extra**
	Extra data field, dependent on format.
uint8_t	**format**
	Determine the format of the data.
uint16_t	**height**
	The height of the **Bitmap**.
uint8_t	**inuse**
	Zero if not in use.
Rect	**solid**
	The solidRect of this **Bitmap**.
uint16_t	**width**
	The width of the **Bitmap**.

Public Attributes Documentation

customSubformat

uint8_t customSubformat

Custom format specifier.

extra

uint8_t extra

Extra data field, dependent on format.

format

uint8_t format

Determine the format of the data.

height

uint16_t height

The height of the **Bitmap**.

inuse

uint8_t inuse

Zero if not in use.

solid

Rect solid

The solidRect of this **Bitmap**.

width

uint16_t width

The width of the **Bitmap**.

Version: 4.16

EasingEquations

Defines the "Penner easing functions", which are a de facto standard for computing aesthetically pleasing motion animations. See http://easings.net/ for visual illustrations of the easing equations.

Public Functions

int16_t **backEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Back easing in: Overshooting cubic easing: (s+1)*t^3 - s*t^2.

int16_t **backEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Back easing in/out: Overshooting cubic easing: (s+1)*t^3 - s*t^2.

int16_t **backEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Back easing out: Overshooting cubic easing: (s+1)*t^3 - s*t^2.

int16_t **bounceEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Bounce easing in - exponentially decaying parabolic bounce.

int16_t **bounceEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Bounce easing in/out - exponentially decaying parabolic bounce.

int16_t **bounceEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Bounce easing out - exponentially decaying parabolic bounce.

int16_t **circEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Circular easing in: sqrt(1-t^2).

int16_t **circEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Circular easing in/out: sqrt(1-t^2).

int16_t **circEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Circular easing out: sqrt(1-t^2).

int16_t **cubicEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Cubic easing in: t^3.

int16_t **cubicEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Cubic easing in/out: t^3.

int16_t **cubicEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Cubic easing out: t^3.

int16_t **elasticEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Elastic easing in - exponentially decaying sine wave.

int16_t **elasticEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Elastic easing in/out - exponentially decaying sine wave.

int16_t **elasticEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Elastic easing out - exponentially decaying sine wave.

int16_t **expoEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Exponential easing in: 2^t.

int16_t **expoEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Exponential easing in/out: 2^t.

int16_t **expoEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Exponential easing out: 2^t.

int16_t **linearEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Simple linear tweening - no easing.

int16_t **linearEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Simple linear tweening - no easing.

int16_t **linearEaseNone**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Simple linear tweening - no easing.

int16_t **linearEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Simple linear tweening - no easing.

int16_t **quadEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Quadratic easing in: t^2.

int16_t **quadEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Quadratic easing in/out: t^2.

int16_t **quadEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)

Quadratic easing out: t^2.

int16_t **quartEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)

	Quartic easing in: t^4.
int16_t	**quartEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Quartic easing in/out: t^4.
int16_t	**quartEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Quartic easing out: t^4.
int16_t	**quintEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Quintic/strong easing in: t^5.
int16_t	**quintEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Quintic/strong easing in/out: t^5.
int16_t	**quintEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Quintic/strong easing out: t^5.
int16_t	**sineEaseIn**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Sinusoidal easing in: sin(t).
int16_t	**sineEaseInOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Sinusoidal easing in/out: sin(t).
int16_t	**sineEaseOut**(uint16_t t, int16_t b, int16_t c, uint16_t d)
	Sinusoidal easing out: sin(t).

Public Functions Documentation

backEaseIn

```
static int16_t backEaseIn ( uint16_t t ,
                            int16_t  b ,
                            int16_t  c ,
                            uint16_t d
                            )
```

Back easing in: Overshooting cubic easing: (s+1)*t^3* - *s*t^2.

Backtracking slightly, then reversing direction and moving to target.

Parameters:

- **t** Time. The current time or step.

b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

backEaseInOut

```
static int16_t backEaseInOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
                               )
```

Back easing in/out: Overshooting cubic easing: (s+1)*t*^3 - *s*t^2.

Backtracking slightly, then reversing direction and moving to target, then overshooting target, reversing, and finally coming back to target.

Parameters:

t Time. The current time or step.
b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

backEaseOut

```
static int16_t backEaseOut ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Back easing out: Overshooting cubic easing: (s+1)*t*^3 - *s*t^2.

Moving towards target, overshooting it slightly, then reversing and coming back to target.

Parameters:

t Time. The current time or step.

b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

bounceEaseIn

```
static int16_t bounceEaseIn ( uint16_t t ,
                              int16_t  b ,
                              int16_t  c ,
                              uint16_t d
                              )
```

Bounce easing in - exponentially decaying parabolic bounce.

Parameters:

t Time. The current time or step.
b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

bounceEaseInOut

```
static int16_t bounceEaseInOut ( uint16_t t ,
                                 int16_t  b ,
                                 int16_t  c ,
                                 uint16_t d
                                 )
```

Bounce easing in/out - exponentially decaying parabolic bounce.

Parameters:

t Time. The current time or step.
b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

bounceEaseOut

```
static int16_t bounceEaseOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
                               )
```

Bounce easing out - exponentially decaying parabolic bounce.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

circEaseIn

```
static int16_t circEaseIn ( uint16_t t ,
                            int16_t  b ,
                            int16_t  c ,
                            uint16_t d
                            )
```

Circular easing in: sqrt(1-t^2).

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

circEaseInOut

```
static int16_t circEaseInOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
                               )
```

Circular easing in/out: sqrt(1-t^2).

Acceleration until halfway, then deceleration.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

circEaseOut

```
static int16_t circEaseOut ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Circular easing out: sqrt(1-t^2).

Decelerating to zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

cubicEaseIn

```
static int16_t cubicEaseIn ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Cubic easing in: t^3.

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

cubicEaseInOut

```
static int16_t cubicEaseInOut ( uint16_t t ,
                                int16_t  b ,
                                int16_t  c ,
                                uint16_t d
                                )
```

Cubic easing in/out: t^3.

Acceleration until halfway, then deceleration.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

cubicEaseOut

```
static int16_t cubicEaseOut ( uint16_t t ,
```

```
                                int16_t  b,
                                int16_t  c,
                                uint16_t d
                                )
```

Cubic easing out: t^3.

Decelerating to zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

elasticEaseIn

```
static int16_t elasticEaseIn ( uint16_t t,
                               int16_t  b,
                               int16_t  c,
                               uint16_t d
                               )
```

Elastic easing in - exponentially decaying sine wave.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

elasticEaseInOut

```
static int16_t elasticEaseInOut ( uint16_t t,
                                  int16_t  b,
                                  int16_t  c,
                                  uint16_t d
```

```
                                    )
```

Elastic easing in/out - exponentially decaying sine wave.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

elasticEaseOut

```
static int16_t elasticEaseOut ( uint16_t t ,
                                int16_t  b ,
                                int16_t  c ,
                                uint16_t d
                                )
```

Elastic easing out - exponentially decaying sine wave.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

expoEaseIn

```
static int16_t expoEaseIn ( uint16_t t ,
                            int16_t  b ,
                            int16_t  c ,
                            uint16_t d
                            )
```

Exponential easing in: 2^t.

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

expoEaseInOut

```
static int16_t expoEaseInOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
                               )
```

Exponential easing in/out: 2^t.

Accelerating until halfway, then decelerating.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

expoEaseOut

```
static int16_t expoEaseOut ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Exponential easing out: 2^t.

Deceleration to zero velocity.

Parameters:

t Time. The current time or step.
b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

linearEaseIn

```
static int16_t linearEaseIn ( uint16_t t ,
                              int16_t  b ,
                              int16_t  c ,
                              uint16_t d
                              )
```

Simple linear tweening - no easing.

Parameters:

t Time. The current time or step.
b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

linearEaseInOut

```
static int16_t linearEaseInOut ( uint16_t t ,
                                 int16_t  b ,
                                 int16_t  c ,
                                 uint16_t d
                                 )
```

Simple linear tweening - no easing.

Parameters:

t Time. The current time or step.
b Beginning. The beginning value.
c Change. The change between the beginning value and the destination value.
d Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

linearEaseNone

```
static int16_t linearEaseNone ( uint16_t t ,
                                int16_t  b ,
                                int16_t  c ,
                                uint16_t d
                                )
```

Simple linear tweening - no easing.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

linearEaseOut

```
static int16_t linearEaseOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
                               )
```

Simple linear tweening - no easing.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quadEaseIn

```
static int16_t quadEaseIn ( uint16_t t ,
                            int16_t  b ,
                            int16_t  c ,
                            uint16_t d
                            )
```

Quadratic easing in: t^2.

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quadEaseInOut

```
static int16_t quadEaseInOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
                               )
```

Quadratic easing in/out: t^2.

Acceleration until halfway, then deceleration.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quadEaseOut

```
static int16_t quadEaseOut ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Quadratic easing out: t^2.

Decelerating to zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quartEaseIn

```
static int16_t quartEaseIn ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Quartic easing in: t^4.

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quartEaseInOut

```
static int16_t quartEaseInOut ( uint16_t t ,
```

```
                                int16_t  b,
                                int16_t  c,
                                uint16_t d
                                )
```

Quartic easing in/out: t^4.

Acceleration until halfway, then deceleration.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quartEaseOut

```
static int16_t quartEaseOut ( uint16_t t,
                              int16_t  b,
                              int16_t  c,
                              uint16_t d
                              )
```

Quartic easing out: t^4.

Decelerating to zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quintEaseIn

```
static int16_t quintEaseIn ( uint16_t t,
                             int16_t  b,
```

```
                                int16_t   c,
                                uint16_t d
                                )
```

Quintic/strong easing in: t^5.

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quintEaseInOut

```
static int16_t quintEaseInOut ( uint16_t t,
                                int16_t   b,
                                int16_t   c,
                                uint16_t d
                                )
```

Quintic/strong easing in/out: t^5.

Acceleration until halfway, then deceleration.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

quintEaseOut

```
static int16_t quintEaseOut ( uint16_t t,
                              int16_t   b,
                              int16_t   c,
```

```
                                uint16_t d
                                )
```

Quintic/strong easing out: t^5.

Decelerating to zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

sineEaseIn

```
static int16_t sineEaseIn ( uint16_t t ,
                            int16_t  b ,
                            int16_t  c ,
                            uint16_t d
                            )
```

Sinusoidal easing in: sin(t).

Accelerating from zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

sineEaseInOut

```
static int16_t sineEaseInOut ( uint16_t t ,
                               int16_t  b ,
                               int16_t  c ,
                               uint16_t d
```

```
                                    )
```

Sinusoidal easing in/out: sin(t).

Acceleration until halfway, then deceleration.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

sineEaseOut

```
static int16_t sineEaseOut ( uint16_t t ,
                             int16_t  b ,
                             int16_t  c ,
                             uint16_t d
                             )
```

Sinusoidal easing out: sin(t).

Decelerating to zero velocity.

Parameters:

- **t** Time. The current time or step.
- **b** Beginning. The beginning value.
- **c** Change. The change between the beginning value and the destination value.
- **d** Duration. The total time or total number of steps.

Returns:

The current value as a function of the current time or step.

Version: 4.16

Edge

An edge contains information about one edge, between two points, of a triangle, as well as information about how to interpolate values when moving in the vertical direction.

Public Functions

	Edge(const **Gradients** & gradients, const **Point3D** * vertices, int top, int bottom)
	Initializes a new instance of the TextureMapTypes class.
FORCE_INLINE_FUNCTION int	**step**()
	Perform a step along the edge.
FORCE_INLINE_FUNCTION int	**step**(int steps)
	Performs a number of steps along the edge.

Public Attributes

int32_t	**denominator**
	The denominator.
int32_t	**errorTerm**
	The error term.
int	**height**
	The height.
int32_t	**numerator**
	The numerator.
float	**oneOverZ**
	The one over z coordinate.
float	**oneOverZStep**
	The one over z coordinate step.
float	**oneOverZStepExtra**

	The one over z coordinate step extra.
float	**UOverZ**
	The over z coordinate.
float	**UOverZStep**
	The over z coordinate step.
float	**UOverZStepExtra**
	The over z coordinate step extra.
float	**VOverZ**
	The over z coordinate.
float	**VOverZStep**
	The over z coordinate step.
float	**VOverZStepExtra**
	The over z coordinate step extra.
int32_t	**X**
	The X coordinate.
int32_t	**XStep**
	Amount to increment x.
int	**Y**
	The Y coordinate.

Public Functions Documentation

Edge

```
Edge ( const Gradients & gradients ,
       const Point3D *   vertices ,
       int               top ,
       int               bottom
       )
```

Initializes a new instance of the TextureMapTypes class.

Construct the edge between two vertices and uses the gradients for calculating the interpolation values.

Parameters:

gradients	The gradients for the triangle.
vertices	The vertices for the triangle.
top	The index in the vertices array of the top vertex of this edge.
bottom	The index in the vertices array of the bottom vertex of this edge.

step

```
FORCE_INLINE_FUNCTION int step ( )
```

Perform a step along the edge.

Increase the Y and decrease the height.

Returns:

The remaining height.

step

```
FORCE_INLINE_FUNCTION int step ( int steps )
```

Performs a number of steps along the edge.

Parameters:

steps	The number of steps the perform.

Returns:

The remaining height.

Public Attributes Documentation

denominator

int32_t denominator

The denominator.

errorTerm

int32_t errorTerm

The error term.

height

int height

The height.

numerator

int32_t numerator

The numerator.

oneOverZ

float oneOverZ

The one over z coordinate.

oneOverZStep

float oneOverZStep

The one over z coordinate step.

oneOverZStepExtra

float oneOverZStepExtra

The one over z coordinate step extra.

UOverZ

float UOverZ

The over z coordinate.

UOverZStep

float UOverZStep

The over z coordinate step.

UOverZStepExtra

float UOverZStepExtra

The over z coordinate step extra.

VOverZ

float VOverZ

The over z coordinate.

VOverZStep

float VOverZStep

The over z coordinate step.

VOverZStepExtra

float VOverZStepExtra

The over z coordinate step extra.

X

int32_t X

The X coordinate.

XStep

int32_t XStep

Amount to increment x.

Y

int Y

The Y coordinate.

Version: 4.16

Event

Simple base class for events.

Inherited by: ClickEvent, DragEvent, GestureEvent

Public Types

enum	**EventType** { EVENT_CLICK, EVENT_DRAG, EVENT_GESTURE }
	The event types.

Public Functions

virtual **EventType**	**getEventType**() =0
	Gets event type.
virtual	**~Event**()
	Finalizes an instance of the **Event** class.

Public Types Documentation

EventType

enum EventType

The event types.

EVENT_CLICK	A click.
EVENT_DRAG	A drag.
EVENT_GESTURE	A gesture.

Public Functions Documentation

getEventType

virtual EventType getEventType ( ) =0

Gets event type.

Returns:

The type of this event.

Reimplemented by: **touchgfx::ClickEvent::getEventType**, **touchgfx::DragEvent::getEventType**, **touchgfx::GestureEvent::getEventType**

~Event

virtual ~Event ( )

Finalizes an instance of the **Event** class.

Version: 4.16

FadeAnimator

A FadeAnimator makes the template class T able to animate the alpha value from its current value to a specified end value. It is possible to use a fade in effect as well as fade out effect using FadeAnimator. The alpha progression can be described by supplying an EasingEquation. The FadeAnimator performs a callback when the animation has finished.

This mixin can be used on any Drawable that has a 'void setAlpha(uint8_t)' and a 'uint8_t getAlpha()' method.

Template Parameters:

- **T** specifies the type to extend with the FadeAnimator behavior.

Inherits from: T

Public Functions

void	**cancelFadeAnimation**()
	Cancel fade animation.
void	**clearFadeAnimationEndedAction**()
	Clears the fade animation ended action previously set by setFadeAnimationEndedAction.
	FadeAnimator()
virtual uint16_t	**getFadeAnimationDelay**() const
	Gets the current animation delay.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
bool	**isFadeAnimationRunning**() const
	Gets whether or not the fade animation is running.
virtual void	**setFadeAnimationDelay**(uint16_t delay)
	Sets a delay before the actual animation starts for the animation done by the **FadeAnimator**.

void	**setFadeAnimationEndedAction**(**GenericCallback**< const **FadeAnimator**< T > & > & callback)
	Associates an action to be performed when the animation ends.

void	**startFadeAnimation**(uint8_t endAlpha, uint16_t duration, **EasingEquation** alphaProgressionEquation =&**EasingEquations::linearEaseNone**)
	Starts the fade animation from the current alpha value to the specified end alpha value.

Protected Functions

void	**nextFadeAnimationStep**()
	Execute next step in fade animation and stop the timer if necessary.

Protected Attributes

EasingEquation	**fadeAnimationAlphaEquation**
	EasingEquation expressing the progression of the alpha value during the animation.

uint16_t	**fadeAnimationCounter**
	To the current step in the animation.

uint16_t	**fadeAnimationDelay**
	A delay that is applied before animation start. Expressed in ticks.

uint16_t	**fadeAnimationDuration**
	The complete duration of the animation. Expressed in ticks.

int16_t	**fadeAnimationEndAlpha**
	The alpha value at the end of the animation.

GenericCallback< const **FadeAnimator**< T > & > *	**fadeAnimationEndedCallback**
	Animation ended **Callback**.

bool	**fadeAnimationRunning**
	True if the animation is running.

int16_t	**fadeAnimationStartAlpha** The alpha value at the beginning of the animation.

Public Functions Documentation

cancelFadeAnimation

void cancelFadeAnimation ( )

Cancel fade animation.

The animation is stopped and the alpha value is left where it currently is.

clearFadeAnimationEndedAction

void clearFadeAnimationEndedAction ( )

Clears the fade animation ended action previously set by setFadeAnimationEndedAction.

Clears the fade animation ended action previously set by setFadeAnimationEndedAction.

See also:

setFadeAnimationEndedAction

FadeAnimator

FadeAnimator ( )

getFadeAnimationDelay

virtual uint16_t getFadeAnimationDelay ( ) const

Gets the current animation delay.

Returns:

The current animation delay.

See also:

setFadeAnimationDelay

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

isFadeAnimationRunning

bool isFadeAnimationRunning ( ) const

Gets whether or not the fade animation is running.

Returns:

true if the fade animation is running.

setFadeAnimationDelay

virtual void setFadeAnimationDelay (uint16_t delay)

Sets a delay before the actual animation starts for the animation done by the **FadeAnimator**.

Parameters:

delay The delay in ticks.

See also:

getFadeAnimationDelay

setFadeAnimationEndedAction

void setFadeAnimationEndedAction (GenericCallback< const FadeAnimator< T > & > & callback)

Associates an action to be performed when the animation ends.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **FadeAnimator**.

startFadeAnimation

```
void startFadeAnimation ( uint8_t endAlpha ,
                          uint16_t duration ,
                          EasingEquation alphaProgressionEquation =&EasingEquations::linearEaseNone
                          )
```

Starts the fade animation from the current alpha value to the specified end alpha value.

The progression of the alpha value during the animation is described by the supplied EasingEquation.

Parameters:

endAlpha	The alpha value of T at animation end.
duration	The duration of the animation measured in ticks.
alphaProgressionEquation	(Optional) The equation that describes the development of the alpha value during the animation. Default is **EasingEquations::linearEaseNone**.

Protected Functions Documentation

nextFadeAnimationStep

```
void nextFadeAnimationStep ( )
```

Execute next step in fade animation and stop the timer if necessary.

Protected Attributes Documentation

fadeAnimationAlphaEquation

EasingEquation fadeAnimationAlphaEquation

EasingEquation expressing the progression of the alpha value during the animation.

fadeAnimationCounter

uint16_t fadeAnimationCounter

To the current step in the animation.

fadeAnimationDelay

uint16_t fadeAnimationDelay

A delay that is applied before animation start. Expressed in ticks.

fadeAnimationDuration

uint16_t fadeAnimationDuration

The complete duration of the animation. Expressed in ticks.

fadeAnimationEndAlpha

int16_t fadeAnimationEndAlpha

The alpha value at the end of the animation.

fadeAnimationEndedCallback

GenericCallback< const FadeAnimator< T > & > * fadeAnimationEndedCallback

Animation ended **Callback**.

fadeAnimationRunning

bool fadeAnimationRunning

True if the animation is running.

fadeAnimationStartAlpha

int16_t fadeAnimationStartAlpha

The alpha value at the beginning of the animation.

Version: 4.16

FlashDataReader

This class is an abstract interface for a class reading data from a flash. The flash can be any type, but is mostly used for flashes that are not memory mapped. Applications must implement access to the flash through this interface.

Public Functions

virtual bool	**addressIsAddressable**(const void * address) =0
	Compute if an address is directly addressable by the MCU.
virtual void	**copyData**(const void *src, void* dst, uint32_t bytes) =0
	Copy data from flash to a buffer.
virtual void	**startFlashLineRead**(const void * src, uint32_t bytes) =0
	Initiate a read operation from flash to a buffer.
virtual const uint8_t *	**waitFlashReadComplete**() =0
	Waits until the previous startFlashLineRead operation is complete.
virtual	**~FlashDataReader**()
	Finalizes an instance of the **FlashDataReader** class.

Public Functions Documentation

addressIsAddressable

virtual bool addressIsAddressable (const void * address)

Compute if an address is directly addressable by the MCU.

Compute if an address is directly addressable by the MCU. The data is addressable it should be read direct through a pointer and not through this interface.

Parameters:

address The address in the flash.

Returns:

True if the address is addressable by the MCU.

copyData

```
virtual void copyData ( const void * src ,    =0
                        void *       dst ,    =0
                        uint32_t     bytes    =0
                        )                     =0
```

Copy data from flash to a buffer.

This must be a synchrony method that does not return until the copy is done.

Parameters:

src	Address of source data in the flash.
dst	Address of destination buffer in RAM.
bytes	Number of bytes to copy.

startFlashLineRead

```
virtual void startFlashLineRead ( const void * src ,    =0
                                  uint32_t     bytes    =0
                                  )                     =0
```

Initiate a read operation from flash to a buffer.

This can be an asynchrony operation that is still running after this function returns. Buffers must be handled by the subclass. **LCD16bppSerialFlash** will at most copy 4 bytes times the width of the display.

Parameters:

src	Address of source data in the flash.
bytes	Number of bytes to copy.

waitFlashReadComplete

```
virtual const uint8_t * waitFlashReadComplete (  ) =0
```

Waits until the previous startFlashLineRead operation is complete.

Waits until the previous startFlashLineRead operation is complete. If the startFlashLineRead method is asynchrony, this method must wait until the previous operation has completed.

Returns:

The address of a buffer containing the read data.

~FlashDataReader

virtual ~FlashDataReader ( )

Finalizes an instance of the **FlashDataReader** class.

Version: 4.16

Font

The font base class. This class is abstract and requires the implementation of getGlyph. It provides utility functions such as obtaining string width and font height.

Inherited by: ConstFont

Public Functions

virtual FORCE_INLINE_FUNCTION uint8_t	**getBitsPerPixel**() const
	Gets bits per pixel for this font.
virtual FORCE_INLINE_FUNCTION uint8_t	**getByteAlignRow**() const
	Are the glyphs saved with each glyph row byte aligned?
virtual uint16_t	**getCharWidth**(const **Unicode::UnicodeChar** c) const
	Gets the width in pixels of the specified character.
virtual **Unicode::UnicodeChar**	**getEllipsisChar**() const
	Gets ellipsis character for the given font.
virtual **Unicode::UnicodeChar**	**getFallbackChar**() const
	Gets fallback character for the given font.
virtual FORCE_INLINE_FUNCTION uint16_t	**getFontHeight**() const
	Returns the height in pixels of this font.
virtual const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode) const
	Gets the glyph data associated with the specified Unicode.
virtual const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode, const uint8_t *& pixelData, uint8_t & bitsPerPixel) const =0
	Gets the glyph data associated with the specified Unicode.
virtual const uint16_t *	**getGSUBTable**() const
	Gets GSUB table.

virtual int8_t	**getKerning**(**Unicode::UnicodeChar** prevChar, const **GlyphNode** * glyph) const
	Gets the kerning distance between two characters.
FORCE_INLINE_FUNCTION uint8_t	**getMaxPixelsLeft**() const
	Gets maximum pixels left of any glyph in the font.
FORCE_INLINE_FUNCTION uint8_t	**getMaxPixelsRight**() const
	Gets maximum pixels right of any glyph in the font.
virtual uint16_t	**getMaxTextHeight**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the height of the highest character in a given string.
virtual FORCE_INLINE_FUNCTION uint16_t	**getMinimumTextHeight**() const
	Returns the minimum height needed for a text field that uses this font.
virtual uint16_t	**getNumberOfLines**(const **Unicode::UnicodeChar** * text, ...) const
	Count the number of lines in a given text.
virtual uint8_t	**getSpacingAbove**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the number of blank pixels at the top of the given text.
virtual uint16_t	**getStringWidth**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the width in pixels of the specified string.
virtual uint16_t	**getStringWidth**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, ...) const
	Gets the width in pixels of the specified string.
virtual	**~Font**()
	Finalizes an instance of the **Font** class.
FORCE_INLINE_FUNCTION bool	**isInvisibleZeroWidth**(**Unicode::UnicodeChar** character)
	Query if 'character' is invisible, zero width.

Protected Functions

	Font(uint16_t height, uint8_t pixBelowBase, uint8_t bitsPerPixel, uint8_t byteAlignRow, uint8_t maxLeft, uint8_t maxRight, const **Unicode::UnicodeChar** fallbackChar, const **Unicode::UnicodeChar** ellipsisChar)
	Initializes a new instance of the **Font** class.
uint16_t	**getStringWidthLTR**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, va_list pArg) const
	Gets the width in pixels of the specified string.
uint16_t	**getStringWidthRTL**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, va_list pArg) const
	Gets the width in pixels of the specified string.

Protected Attributes

uint8_t	**bAlignRow**
	The glyphs are saved with each row byte aligned.
uint8_t	**bPerPixel**
	The number of bits per pixel.
Unicode::UnicodeChar	**ellipsisCharacter**
	The ellipsis character used for truncating long texts.
Unicode::UnicodeChar	**fallbackCharacter**
	The fallback character to use when no glyph exists for the wanted character.
uint16_t	**fontHeight**
	The font height in pixels.
uint8_t	**maxPixelsLeft**
	The maximum number of pixels a glyph extends to the left.
uint8_t	**maxPixelsRight**
	The maximum number of pixels a glyph extends to the right.
uint8_t	**pixelsBelowBaseline**
	The number of pixels below the base line.

Public Functions Documentation

getBitsPerPixel

virtual FORCE_INLINE_FUNCTION uint8_t getBitsPerPixel ( ) const

Gets bits per pixel for this font.

Returns:

The number of bits used per pixel in this font.

getByteAlignRow

virtual FORCE_INLINE_FUNCTION uint8_t getByteAlignRow ( ) const

Are the glyphs saved with each glyph row byte aligned?

Returns:

True if each glyph row is stored byte aligned, false otherwise.

getCharWidth

virtual uint16_t getCharWidth (const Unicode::UnicodeChar c)

Gets the width in pixels of the specified character.

Parameters:

c The **Unicode** character.

Returns:

The width in pixels of the specified character.

getEllipsisChar

virtual Unicode::UnicodeChar getEllipsisChar ( ) const

Gets ellipsis character for the given font.

This is the character which is used when truncating long lines.

Returns:

The ellipsis character for the typography.

See also:

TextArea::setWideTextAction

getFallbackChar

virtual Unicode::UnicodeChar getFallbackChar ( ) const

Gets fallback character for the given font.

The fallback character is the character used when no glyph is available for some character. If 0 (zero) is returned, there is no default character.

Returns:

The default character for the typography in case no glyph is available.

getFontHeight

virtual FORCE_INLINE_FUNCTION uint16_t getFontHeight ( ) const

Returns the height in pixels of this font.

The returned value corresponds to the maximum height occupied by a character in the font.

Returns:

The height in pixels of this font.

NOTE

It is not sufficient to allocate text areas with this height. Use getMinimumTextHeight for this.

getGlyph

virtual const GlyphNode * getGlyph (Unicode::UnicodeChar unicode)

Gets the glyph data associated with the specified Unicode.

Please note that in case of Thai letters and Arabic letters where diacritics can be placed relative to the previous character(s), please use **TextProvider::getNextLigature()** instead as it will create a temporary **GlyphNode** that will be adjusted with respect to X/Y position.

Parameters:

unicode The character to look up.

Returns:

A pointer to the glyph node or null if the glyph was not found.

See also:

TextProvider::getNextLigature

getGlyph

virtual const GlyphNode * getGlyph (Unicode::UnicodeChar unicode , const =0
const uint8_t *& pixelData , const =0
uint8_t & bitsPerPixel const =0
) const =0

Gets the glyph data associated with the specified Unicode.

Please note that in case of Thai letters and Arabic letters where diacritics can be placed relative to the previous character(s), please use **TextProvider::getNextLigature()** instead as it will create a temporary **GlyphNode** that will be adjusted with respect to X/Y position.

Parameters:

unicode	The character to look up.
pixelData	Pointer to the pixel data for the glyph if the glyph is found. This is set by this method.
bitsPerPixel	Reference where to place the number of bits per pixel.

Returns:

A pointer to the glyph node or null if the glyph was not found.

Reimplemented by: **touchgfx::ConstFont::getGlyph**

getGSUBTable

virtual const uint16_t * getGSUBTable ( ) const

Gets GSUB table.

Currently only used for Devanagari fonts.

Returns:

The GSUB table or null if font has GSUB no table.

getKerning

virtual int8_t getKerning (Unicode::UnicodeChar prevChar , const
const GlyphNode * glyph const
) const

Gets the kerning distance between two characters.

Parameters:

prevChar The **Unicode** value of the previous character.
glyph the glyph object for the current character.

Returns:

The kerning distance between prevChar and glyph char.

Reimplemented by: **touchgfx::InternalFlashFont::getKerning**, **touchgfx::ConstFont::getKerning**

getMaxPixelsLeft

FORCE_INLINE_FUNCTION uint8_t getMaxPixelsLeft ( ) const

Gets maximum pixels left of any glyph in the font.

This is the max value of "left" for all glyphs. The value is negated so if a "g" has left=-6 maxPixelsLeft is 6. This value is calculated by the font converter.

Returns:

The maximum pixels left.

getMaxPixelsRight

FORCE_INLINE_FUNCTION uint8_t getMaxPixelsRight ( ) const

Gets maximum pixels right of any glyph in the font.

This is the max value of "width+left-advance" for all glyphs. The is the number of pixels a glyph reaches to the right of its normal area. This value is calculated by the font converter.

Returns:

The maximum pixels right.

getMaxTextHeight

```
virtual uint16_t getMaxTextHeight ( const Unicode::UnicodeChar * text ,  const
                                    ...                                  const
                                    )                                    const
```

Gets the height of the highest character in a given string.

The height includes the spacing above the text which is included in the font.

Parameters:

text A null-terminated **Unicode** string.
... Variable arguments providing additional information inserted at wildcard placeholders.

Returns:

The height if the given text.

getMinimumTextHeight

```
virtual FORCE_INLINE_FUNCTION uint16_t getMinimumTextHeight (  ) const
```

Returns the minimum height needed for a text field that uses this font.

Takes into account that certain characters (eg 'g') have pixels below the baseline, thus making the text height larger than the font height.

Returns:

The minimum height needed for a text field that uses this font.

getNumberOfLines

```
virtual uint16_t getNumberOfLines ( const Unicode::UnicodeChar * text ,  const
                                    ...                                  const
                                    )                                    const
```

Count the number of lines in a given text.

Parameters:

text The text.
... Variable arguments providing additional information.

Returns:

The number of lines.

getSpacingAbove

```
virtual uint8_t getSpacingAbove ( const Unicode::UnicodeChar * text , const
                                  ...                                 const
                                  )                                   const
```

Gets the number of blank pixels at the top of the given text.

Parameters:

text A null-terminated **Unicode** string.
... Variable arguments providing additional information inserted at wildcard placeholders.

Returns:

The number of blank pixels above the text.

getStringWidth

```
virtual uint16_t getStringWidth ( const Unicode::UnicodeChar * text , const
                                  ...                                 const
                                  )                                   const
```

Gets the width in pixels of the specified string.

If the string contains multiple lines, the width of the widest line is found. Please note that the correct number of arguments must be given if the text contains wildcards.

It is recommended to use the **getStringWidth()** implementation with the TextDirection parameter to ensure correct calculation of the width. Kerning could result in different results depending on the TextDirection. This method assumes TextDirection to be TEXT_DIRECTION_LTR.

Parameters:

text A null-terminated **Unicode** string with arguments to insert if the text contains wildcards.
... Variable arguments providing additional information inserted at wildcard placeholders.

Returns:

The width in pixels of the longest line of the specified string.

getStringWidth

```
virtual uint16_t getStringWidth ( TextDirection                textDirection , const
                                  const Unicode::UnicodeChar * text ,          const
```

...		const
)	const	

Gets the width in pixels of the specified string.

If the string contains multiple lines, the width of the widest line is found. Please note that the correct number of arguments must be given if the text contains wildcards.

The TextDirection should be set correctly for the text supplied. For example the string "10 20 30" will be calculated differently depending on the TextDirection. If TextDirection is TEXT_DIRECTION_LTR the width is calculated as the with of "10 20 30" (with kerning between all characters) but for TEXT_DIRECTION_RTL it is calculated as "10"+" "+"20"+" "+"30" (with kerning only between characters in the substrings and not between substrings). For most fonts there might not be a difference between the two calculations, but some fonts might cause slightly different results.

Parameters:

textDirection	The text direction.
text	A null-terminated **Unicode** string with arguments to insert if the text contains wildcards.
...	Variable arguments providing additional information inserted at wildcard placeholders.

Returns:

The width in pixels of the longest line of the specified string.

~Font

virtual ~Font ( )

Finalizes an instance of the **Font** class.

isInvisibleZeroWidth

static FORCE_INLINE_FUNCTION bool isInvisibleZeroWidth (Unicode::UnicodeChar character)

Query if 'character' is invisible, zero width.

Parameters:

character	The character.

Returns:

True if invisible, zero width, false if not.

Protected Functions Documentation

Font

```
Font ( uint16_t                        height ,
       uint8_t                         pixBelowBase ,
       uint8_t                         bitsPerPixel ,
       uint8_t                         byteAlignRow ,
       uint8_t                         maxLeft ,
       uint8_t                         maxRight ,
       const Unicode::UnicodeChar      fallbackChar ,
       const Unicode::UnicodeChar      ellipsisChar
       )
```

Initializes a new instance of the **Font** class.

The protected constructor of a **Font**.

Parameters:

height	The font height in pixels.
pixBelowBase	The number of pixels below the base line.
bitsPerPixel	The number of bits per pixel.
byteAlignRow	The glyphs are saved with each row byte aligned.
maxLeft	The maximum left extend for a glyph in the font.
maxRight	The maximum right extend for a glyph in the font.
fallbackChar	The fallback character for the typography in case no glyph is available.
ellipsisChar	The ellipsis character used for truncating long texts.

getStringWidthLTR

```
uint16_t getStringWidthLTR ( TextDirection                  textDirection ,  const
                             const Unicode::UnicodeChar *   text ,           const
                             va_list                        pArg             const
                             )                              const
```

Gets the width in pixels of the specified string.

If the string contains multiple lines, the width of the widest line is found. Please note that the correct number of arguments must be given if the text contains wildcards.

Parameters:

textDirection	The text direction.

text	A null-terminated **Unicode** string with arguments to insert if the text contains wildcards.
pArg	Variable arguments providing additional information inserted at wildcard placeholders.

Returns:

The width in pixels of the longest line of the specified string.

NOTE

The string is assumed to be purely left-to-right.

getStringWidthRTL

```
uint16_t getStringWidthRTL ( TextDirection textDirection , const
                             const Unicode::UnicodeChar * text , const
                             va_list pArg const
                             ) const
```

Gets the width in pixels of the specified string.

If the string contains multiple lines, the width of the widest line is found. Please note that the correct number of arguments must be given if the text contains wildcards.

The string is handled as a right-to-left string and subdivided into smaller text strings to correctly handle mixing of left-to-right and right-to-left strings.

Parameters:

textDirection	The text direction.
text	A null-terminated **Unicode** string with arguments to insert if the text contains wildcards.
pArg	Variable arguments providing additional information inserted at wildcard placeholders.

Returns:

The string width RTL.

Protected Attributes Documentation

bAlignRow

uint8_t bAlignRow

The glyphs are saved with each row byte aligned.

bPerPixel

uint8_t bPerPixel

The number of bits per pixel.

ellipsisCharacter

Unicode::UnicodeChar ellipsisCharacter

The ellipsis character used for truncating long texts.

fallbackCharacter

Unicode::UnicodeChar fallbackCharacter

The fallback character to use when no glyph exists for the wanted character.

fontHeight

uint16_t fontHeight

The font height in pixels.

maxPixelsLeft

uint8_t maxPixelsLeft

The maximum number of pixels a glyph extends to the left.

maxPixelsRight

uint8_t maxPixelsRight

The maximum number of pixels a glyph extends to the right.

pixelsBelowBaseline

uint8_t pixelsBelowBaseline

The number of pixels below the base line.

Version: 4.16

FontManager

This class is the entry point for looking up a font based on a font id. Must be initialized with the appropriate FontProvider by the application.

Public Functions

Font *	**getFont**(**FontId** fontId)
	Gets a font.
void	**setFontProvider**(**FontProvider** * fontProvider)
	Sets the font provider.

Public Functions Documentation

getFont

static Font * getFont (FontId fontId)

Gets a font.

Parameters:

fontId The FontId of the font to get.

Returns:

The font with a FontId of fontId.

setFontProvider

static void setFontProvider (FontProvider * fontProvider)

Sets the font provider.

Must be initialized with the appropriate **FontProvider** by the application.

Parameters:

fontProvider Sets the font provider. Must be initialized with the appropriate **FontProvider** by the application.

Version: 4.16

FontProvider

A generic pure virtual definition of a FontProvider, which is a class capable of returning a Font based on a FontId. An application-specific derivation of this class must be implemented.

Public Functions

virtual **Font** *	**getFont**(**FontId** fontId) =0
	Gets a **Font**.
virtual	**~FontProvider**()
	Finalizes an instance of the **FontProvider** class.

Public Functions Documentation

getFont

virtual Font * getFont (FontId fontId)

Gets a **Font**.

Parameters:

fontId The FontId of the font to get.

Returns:

The font with a font id of fontId.

~FontProvider

virtual ~FontProvider ( )

Finalizes an instance of the **FontProvider** class.

Version: 4.16

FrameBufferAllocator

This class is an abstract interface for a class allocating partial framebuffer blocks. The interface must be implemented by a subclass.

See: ManyBlockAllocator

Inherited by: ManyBlockAllocator< block_size, blocks, bytes_pr_pixel >

Protected Types

enum	**BlockState** { EMPTY, ALLOCATED, DRAWN, SENDING }
	BlockState is used for internal state of each block.

Public Functions

virtual uint16_t	**allocateBlock**(const uint16_t x, const uint16_t y, const uint16_t width, const uint16_t height, uint8_t ** block) =0
	Allocates a framebuffer block.
virtual void	**freeBlockAfterTransfer**() =0
	Free a block after transfer to the LCD.
virtual const uint8_t *	**getBlockForTransfer**(**Rect** & rect) =0
	Get the block ready for transfer.
virtual bool	**hasBlockReadyForTransfer**() =0
	Check if a block is ready for transfer to the LCD.
virtual bool	**hasEmptyBlock**() =0
	Check if a block is ready for drawing (the block is empty).
virtual void	**markBlockReadyForTransfer**() =0
	Marks a previously allocated block as ready to be transferred to the LCD.
virtual const **Rect** &	**peekBlockForTransfer**() =0
	Get the Rect of the next block to transfer.

virtual	**~FrameBufferAllocator**()
	Finalizes an instance of the **FrameBufferAllocator** class.

Protected Types Documentation

BlockState

enum BlockState

BlockState is used for internal state of each block.

EMPTY	Block is empty, can be allocated.
ALLOCATED	Block is allocated for drawing.
DRAWN	Block has been drawn to, can be send.
SENDING	Block is being transmitted to the display.

Public Functions Documentation

allocateBlock

```
virtual uint16_t allocateBlock ( const uint16_t x ,      =0
                                 const uint16_t y ,      =0
                                 const uint16_t width ,  =0
                                 const uint16_t height , =0
                                 uint8_t **      block   =0
                                 )               =0
```

Allocates a framebuffer block.

The block will have at least the width requested. The height of the allocated block can be lower than requested if not enough memory is available.

Parameters:

x	The absolute x coordinate of the block on the screen.
y	The absolute y coordinate of the block on the screen.
width	The width of the block.
height	The height of the block.
block	Pointer to pointer to return the block address in.

Returns:

The height of the allocated block.

Reimplemented by: **touchgfx::ManyBlockAllocator::allocateBlock**

freeBlockAfterTransfer

virtual void freeBlockAfterTransfer ( ) =0

Free a block after transfer to the LCD.

Marks a previously allocated block as transferred and ready to reuse.

Reimplemented by: **touchgfx::ManyBlockAllocator::freeBlockAfterTransfer**

getBlockForTransfer

virtual const uint8_t * getBlockForTransfer (Rect & rect)

Get the block ready for transfer.

Parameters:

rect Reference to rect to write block x, y, width, and height.

Returns:

Returns the address of the block ready for transfer.

Reimplemented by: **touchgfx::ManyBlockAllocator::getBlockForTransfer**

hasBlockReadyForTransfer

virtual bool hasBlockReadyForTransfer ( ) =0

Check if a block is ready for transfer to the LCD.

Returns:

True if a block is ready for transfer.

Reimplemented by: **touchgfx::ManyBlockAllocator::hasBlockReadyForTransfer**

hasEmptyBlock

virtual bool hasEmptyBlock ( ) =0

Check if a block is ready for drawing (the block is empty).

Returns:

True if a block is empty.

Reimplemented by: **touchgfx::ManyBlockAllocator::hasEmptyBlock**

markBlockReadyForTransfer

virtual void markBlockReadyForTransfer ( ) =0

Marks a previously allocated block as ready to be transferred to the LCD.

Reimplemented by: **touchgfx::ManyBlockAllocator::markBlockReadyForTransfer**

peekBlockForTransfer

virtual const Rect & peekBlockForTransfer ( ) =0

Get the Rect of the next block to transfer.

Returns:

Rect ready for transfer.

NOTE

This function should only be called when the allocator has a block ready for transfer.

See also:

hasBlockReadyForTransfer

Reimplemented by: **touchgfx::ManyBlockAllocator::peekBlockForTransfer**

~FrameBufferAllocator

virtual ~FrameBufferAllocator ( )

Finalizes an instance of the **FrameBufferAllocator** class.

Version: 4.16

FullSolidRect

A Widget that reports solid and but does not draw anything.

Inherits from: Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.

virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**

	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to

be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

Version: 4.16

Gauge

A gauge is a graphical element that shows a needle on a dial, often a speedometer or similar. Much like a progress indicator, the minimum and maximum value of the Gauge, as well as steps can be set. For more information on this, consult the documentation on ProgressIndicators.

A Gauge has a needle and optionally an arc that follows the needle.

Inherits from: AbstractProgressIndicator, Container, Drawable

Public Functions

	Gauge()
	Initializes a new instance of the **Gauge** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Circle &	**getArc**()
	Gets a reference to the arc (**Circle**).
virtual void	**getCenter**(int & x, int & y) const
	Gets the texture mapper center coordinates.
virtual int	**getEndAngle**() const
	Gets end angle.
virtual int	**getStartAngle**() const
	Gets start angle for the needle (and arc).
void	**putArcOnTop**(bool arcOnTop =true)
	Shows the arc on top of the needle.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setArcPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets arc position inside the **Gauge**.
void	**setArcVisible**(bool show =true)

	Allow the arc to be shown or hidden.
void	**setBackgroundOffset**(int16_t offsetX, int16_t offsetY)
	Sets background offset inside the **Gauge**.
virtual void	**setCenter**(int x, int y)
	Sets the center of the texture mapper and the arc inside the **Gauge**.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setMovingNeedleRenderingAlgorithm**(**TextureMapper::RenderingAlgorithm** algorithm)
	Sets rendering algorithm used when the needle is moving during an animation.
void	**setNeedle**(const **BitmapId** bitmapId, int16_t rotationCenterX, int16_t rotationCenterY)
	Sets a bitmap for the needle and the rotation point in the needle bitmap.
virtual void	**setStartEndAngle**(int startAngle, int endAngle)
	Sets start and end angle for the needle and arc.
void	**setSteadyNeedleRenderingAlgorithm**(**TextureMapper::RenderingAlgorithm** algorithm)
	Sets rendering algorithm used when the needle is steady (after an animation).
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.

Protected Functions

virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	This function has no effect on a **Gauge**.
void	**setupNeedleTextureMapper**()
	Sets up the needle texture mapper.

Protected Attributes

TextureMapper::RenderingAlgorithm	**algorithmMoving**
	The algorithm used when the needle is moving.
TextureMapper::RenderingAlgorithm	**algorithmSteady**
	The algorithm used when the needle is steady.
Circle	**arc**
	The arc.
int16_t	**gaugeCenterX**
	The x coordinate of the rotation point of the hands.
int16_t	**gaugeCenterY**
	The y coordinate of the rotation point of the hands.
TextureMapper	**needle**
	The textureMapper.
int16_t	**needleCenterX**
	The x coordinate of the rotation point of the hands.
int16_t	**needleCenterY**
	The y coordinate of the rotation point of the hands.
int	**needleEndAngle**
	The end angle.
int	**needleStartAngle**
	The start angle.

Additional inherited members

Public Functions inherited from AbstractProgressIndicator

AbstractProgressIndicator()
Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.

virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.
void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)

	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.
int	**rangeMin**
	The range minimum.

uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)

	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.

virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**

	True if this drawable should receive touch events.
bool **visible**	
	True if this drawable should be drawn.

Public Functions Documentation

Gauge

Gauge ( )

Initializes a new instance of the **Gauge** class.

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getArc

Circle & getArc ( )

Gets a reference to the arc (**Circle**).

This allows for setting radius, line width, painter, etc. on the arc (**Circle**).

Returns:

The arc (**Circle**).

getCenter

```
virtual void getCenter ( int & x ,      const
                         int & y        const
                         )      const
```

Gets the texture mapper center coordinates.

Parameters:

x The x coordinate of the center of the texture mapper.
y The y coordinate of the center of the texture mapper.

See also:

setCenter

getEndAngle

```
virtual int getEndAngle (  ) const
```

Gets end angle.

Beware that the value returned is not related to the current progress of the texture mapper but rather the end point of the **Gauge** when it is at max value.

Returns:

The end angle.

See also:

setStartEndAngle

getStartAngle

```
virtual int getStartAngle (  ) const
```

Gets start angle for the needle (and arc).

Returns:

The start angle.

See also:

setStartEndAngle, **getEndAngle**

putArcOnTop

void putArcOnTop (bool arcOnTop =true)

Shows the arc on top of the needle.

By default the needle is drawn on top of the arc.

Parameters:

arcOnTop (Optional) True to put the arc on top of the needle (default), false to put the needle on top of the arc.

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setArcPosition

void setArcPosition (int16_t x ,
int16_t y ,
int16_t width ,
int16_t height
)

Sets arc position inside the **Gauge**.

This is especially useful if the arc is using a bitmap painter. If the center has previously been set, the arc center will be updated to be at the same offset relative to the top left corner of the **Gauge**.

Parameters:

x The x coordinate.

y	The y coordinate.
width	The width.
height	The height.

See also:

setCenter, **getArc**

setArcVisible

```
void setArcVisible ( bool show =true )
```

Allow the arc to be shown or hidden.

Parameters:

show	(Optional) True to show, false to hide. Default is to show the arc.

setBackgroundOffset

```
void setBackgroundOffset ( int16_t offsetX ,
                           int16_t offsetY
                           )
```

Sets background offset inside the **Gauge**.

If the dial is smaller than the size needed for the **Gauge** to show the needle, the background image can be moved inside the **Gauge**.

Parameters:

offsetX	The offset x coordinate.
offsetY	The offset y coordinate.

See also:

setBackground

setCenter

```
virtual void setCenter ( int x ,
                         int y
                         )
```

Sets the center of the texture mapper and the arc inside the **Gauge**.

Parameters:

x The x coordinate of the center of the texture mapper.
y The y coordinate of the center of the texture mapper.

See also:

getCenter

setHeight

virtual void setHeight (int16_t height)

Sets the height of this drawable.

Parameters:

height The new height.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.

Reimplements: **touchgfx::Drawable::setHeight**

setMovingNeedleRenderingAlgorithm

void setMovingNeedleRenderingAlgorithm (TextureMapper::RenderingAlgorithm algorithm)

Sets rendering algorithm used when the needle is moving during an animation.

For better performance, this can be set to **TextureMapper::NEAREST_NEIGHBOR**. For nicer graphics, it should be set to **TextureMapper::BILINEAR_INTERPOLATION** (this is the default behavior).

Parameters:

algorithm The algorithm.

See also:

updateValue, **setSteadyNeedleRenderingAlgorithm**

setNeedle

void setNeedle (const BitmapId bitmapId ,

```
                int16_t         rotationCenterX ,
                int16_t         rotationCenterY
                )
```

Sets a bitmap for the needle and the rotation point in the needle bitmap.

Parameters:

bitmapId	Identifier for the bitmap.
rotationCenterX	The rotation center x coordinate.
rotationCenterY	The rotation center y coordinate.

setStartEndAngle

```
virtual void setStartEndAngle ( int startAngle ,
                                int endAngle
                                )
```

Sets start and end angle for the needle and arc.

By swapping end and start angles, these can progress backwards.

Parameters:

startAngle	The start angle.
endAngle	The end angle.

setSteadyNeedleRenderingAlgorithm

```
void setSteadyNeedleRenderingAlgorithm ( TextureMapper::RenderingAlgorithm algorithm )
```

Sets rendering algorithm used when the needle is steady (after an animation).

For better performance, this can be set to **TextureMapper::NEAREST_NEIGHBOR**. For nicer graphics, it should be set to **TextureMapper::BILINEAR_INTERPOLATION** (this is the default behavior).

Parameters:

algorithm	The algorithm.

See also:

updateValue, **setMovingNeedleRenderingAlgorithm**

setValue

virtual void setValue (int value)

Sets the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. If a callback function has been set using setValueSetAction, that callback will be called (unless the new value is the same as the current value).

Parameters:

value The value.

NOTE

if value is equal to the current value, nothing happens, and the callback will not be called.

See also:

getValue, **updateValue**, **setValueSetAction**

Reimplements: **touchgfx::AbstractProgressIndicator::setValue**

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most **Drawable** widgets, changing this does normally not automatically yield a redraw.

Reimplements: **touchgfx::Drawable::setWidth**

Protected Functions Documentation

setProgressIndicatorPosition

virtual void setProgressIndicatorPosition (int16_t x ,
int16_t y ,

```
                                    int16_t width ,
                                    int16_t height
                                    )
```

This function has no effect on a **Gauge**.

Parameters:

x	unused
y	unused
width	unused
height	unused

Reimplements: **touchgfx::AbstractProgressIndicator::setProgressIndicatorPosition**

setupNeedleTextureMapper

```
void setupNeedleTextureMapper ( )
```

Sets up the needle texture mapper.

Protected Attributes Documentation

algorithmMoving

TextureMapper::RenderingAlgorithm algorithmMoving

The algorithm used when the needle is moving.

algorithmSteady

TextureMapper::RenderingAlgorithm algorithmSteady

The algorithm used when the needle is steady.

arc

Circle arc

The arc.

gaugeCenterX

int16_t gaugeCenterX

The x coordinate of the rotation point of the hands.

gaugeCenterY

int16_t gaugeCenterY

The y coordinate of the rotation point of the hands.

needle

TextureMapper needle

The textureMapper.

needleCenterX

int16_t needleCenterX

The x coordinate of the rotation point of the hands.

needleCenterY

int16_t needleCenterY

The y coordinate of the rotation point of the hands.

needleEndAngle

int needleEndAngle

The end angle.

needleStartAngle

int needleStartAngle

The start angle.

Version: 4.16

GenericCallback

GenericCallback is the base class for callbacks. The reason this base class exists, is that a normal Callback requires the class type where the callback function resides to be known. This is problematic for ie. framework widgets like AbstractButton, on which it should be possible to register a callback on object types that are user-specific and thus unknown to AbstractButton. This is solved by having AbstractButton contain a pointer to a GenericCallback instead. This pointer must then be initialized to point on an instance of Callback, created by the user, which is initialized with the appropriate object type.

Template Parameters:

- **T1** The type of the first argument in the member function, or void if none.
- **T2** The type of the second argument in the member function, or void if none.
- **T3** The type of the third argument in the member function, or void if none.

See: Callback

Note: As with Callback, this class exists in four versions to support callback functions taking zero, one, two or three arguments.

Public Functions

virtual void	**execute**(T1 val1, T2 val2, T3 val3) =0
	Calls the member function.
virtual bool	**isValid**() const =0
	Function to check whether the **Callback** has been initialized with values.
virtual	**~GenericCallback**()
	Finalizes an instance of the **GenericCallback** class.

Public Functions Documentation

execute

```
virtual void execute ( T1 val1 ,  =0
```

```
                T2 val2 ,  =0
                T3 val3    =0
                )    =0
```

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

Parameters:

val1	This value will be passed as the first argument in the function call.
val2	This value will be passed as the second argument in the function call.
val3	This value will be passed as the third argument in the function call.

Reimplemented by: **touchgfx::Callback::execute**, **touchgfx::Callback::execute**, **touchgfx::Callback::execute**, **touchgfx::Callback::execute**, **touchgfx::Callback::execute**

isValid

```
virtual bool isValid (  ) const =0
```

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

Reimplemented by: **touchgfx::Callback::isValid**, **touchgfx::Callback::isValid**, **touchgfx::Callback::isValid**, **touchgfx::Callback::isValid**, **touchgfx::Callback::isValid**

~GenericCallback

```
virtual ~GenericCallback (  )
```

Finalizes an instance of the **GenericCallback** class.

Version: 4.16

GenericCallback<T1,T2,void>

GenericCallback is the base class for callbacks. The reason this base class exists, is that a normal Callback requires the class type where the callback function resides to be known. This is problematic for ie. framework widgets like AbstractButton, on which it should be possible to register a callback on object types that are user-specific and thus unknown to AbstractButton. This is solved by having AbstractButton contain a pointer to a GenericCallback instead. This pointer must then be initialized to point on an instance of Callback, created by the user, which is initialized with the appropriate object type.

Template Parameters:

- **T1** The type of the first argument in the member function, or void if none.
- **T2** The type of the second argument in the member function, or void if none.

See: Callback

Note: As with Callback, this class exists in four versions to support callback functions taking zero, one, two or three arguments.

Public Functions

virtual void	**execute**(T1 val1, T2 val2) =0
	Calls the member function.
virtual bool	**isValid**() const =0
	Function to check whether the **Callback** has been initialized with values.
virtual	**~GenericCallback**()
	Finalizes an instance of the void> class.

Public Functions Documentation

execute

```
virtual void execute ( T1 val1 ,  =0
                       T2 val2    =0
```

) =0

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

Parameters:

val1 This value will be passed as the first argument in the function call.
val2 This value will be passed as the second argument in the function call.

isValid

virtual bool isValid ( ) const =0

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

~GenericCallback

virtual ~GenericCallback ( )

Finalizes an instance of the void> class.

Version: 4.16

GenericCallback<T1,void,void>

GenericCallback is the base class for callbacks. The reason this base class exists, is that a normal Callback requires the class type where the callback function resides to be known. This is problematic for ie. framework widgets like AbstractButton, on which it should be possible to register a callback on object types that are user-specific and thus unknown to AbstractButton. This is solved by having AbstractButton contain a pointer to a GenericCallback instead. This pointer must then be initialized to point on an instance of Callback, created by the user, which is initialized with the appropriate object type.

Template Parameters:

- **T1** The type of the first argument in the member function, or void if none.

See: Callback

Note: As with Callback, this class exists in four versions to support callback functions taking zero, one, two or three arguments.

Public Functions

virtual void	**execute**(T1 val1) =0
	Calls the member function.
virtual bool	**isValid**() const =0
	Function to check whether the **Callback** has been initialized with values.
virtual	**~GenericCallback**()
	Finalizes an instance of the void> class.

Public Functions Documentation

execute

virtual void execute (T1 val1)

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

Parameters:

val1 This value will be passed as the first argument in the function call.

isValid

virtual bool isValid ( ) const =0

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

~GenericCallback

virtual ~GenericCallback ( )

Finalizes an instance of the void> class.

Version: 4.16

GenericCallback<void>

GenericCallback is the base class for callbacks. The reason this base class exists, is that a normal Callback requires the class type where the callback function resides to be known. This is problematic for ie. framework widgets like AbstractButton, on which it should be possible to register a callback on object types that are user-specific and thus unknown to AbstractButton. This is solved by having AbstractButton contain a pointer to a GenericCallback instead. This pointer must then be initialized to point on an instance of Callback, created by the user, which is initialized with the appropriate object type.

See: Callback

Note: As with Callback, this class exists in four versions to support callback functions taking zero, one, two or three arguments.

Public Functions

virtual void	**execute**() =0
	Calls the member function.
virtual bool	**isValid**() const =0
	Function to check whether the **Callback** has been initialized with values.
virtual	**~GenericCallback**()
	Finalizes an instance of the **GenericCallback<void>** class.

Public Functions Documentation

execute

virtual void execute ( ) =0

Calls the member function.

Do not call execute unless **isValid()** returns true (ie. a pointer to the object and the function has been set).

isValid

virtual bool isValid ( ) const =0

Function to check whether the **Callback** has been initialized with values.

Returns:

true If the callback is valid (i.e. safe to call execute).

~GenericCallback

virtual ~GenericCallback ( )

Finalizes an instance of the **GenericCallback<void>** class.

Version: 4.16

GestureEvent

A gesture event. The only gesture events currently supported is SWIPE_HORIZONTAL and SWIPE_VERTICAL, which will be issued every time the input system detects a swipe.

See: Event

Inherits from: Event

Public Types

enum	**GestureEventType** { SWIPE_HORIZONTAL, SWIPE_VERTICAL }
	Values that represent gesture types.

Public Functions

	GestureEvent(**GestureEventType** type, int16_t velocity, int16_t x, int16_t y)
	Constructor.
virtual **Event::EventType**	**getEventType**()
	Gets event type.
GestureEventType	**getType**() const
	Gets the type of this gesture event.
int16_t	**getVelocity**() const
	Gets the velocity of this gesture event.
int16_t	**getX**() const
	Gets the x coordinate of this gesture event.
int16_t	**getY**() const
	Gets the y coordinate of this gesture event.

Additional inherited members

Public Types inherited from Event

enum	**EventType** { EVENT_CLICK, EVENT_DRAG, EVENT_GESTURE }
	The event types.

Public Functions inherited from Event

virtual	**~Event**()
	Finalizes an instance of the **Event** class.

Public Types Documentation

GestureEventType

enum GestureEventType

Values that represent gesture types.

SWIPE_HORIZONTAL	An enum constant representing a horizontal swipe.
SWIPE_VERTICAL	An enum constant representing a vertical swipe.

Public Functions Documentation

GestureEvent

```
GestureEvent ( GestureEventType type ,
               int16_t          velocity ,
               int16_t          x ,
               int16_t          y
               )
```

Constructor.

Create a gesture event of the specified type with the specified coordinates.

Parameters:

type	The type of the gesture event.
velocity	The velocity of this gesture (swipe)

x	The x coordinate of the gesture.
y	The y coordinate of the gesture.

getEventType

virtual Event::EventType getEventType ( )

Gets event type.

Returns:

The type of this event.

Reimplements: **touchgfx::Event::getEventType**

getType

GestureEventType getType ( ) const

Gets the type of this gesture event.

Returns:

The type of this gesture event.

getVelocity

int16_t getVelocity ( ) const

Gets the velocity of this gesture event.

Returns:

The velocity of this gesture event.

getX

int16_t getX ( ) const

Gets the x coordinate of this gesture event.

Returns:

The x coordinate of this gesture event.

getY

int16_t getY ( ) const

Gets the y coordinate of this gesture event.

Returns:

The y coordinate of this gesture event.

Version: 4.16

Gestures

This class implements the detection of gestures.

Public Functions

	Gestures()
	Default constructor.
void	**registerClickEvent**(**ClickEvent::ClickEventType** evt, uint16_t x, uint16_t y)
	Register a click event and figure out if this is a drag event, too.
bool	**registerDragEvent**(uint16_t oldX, uint16_t oldY, uint16_t newX, uint16_t newY)
	Register a drag event.
void	**registerEventListener**(**UIEventListener** & l)
	Register the event listener.
void	**setDragThreshold**(uint16_t val)
	Configure the threshold for reporting drag events.
void	**tick**()
	Has to be called during the timer tick.

Public Functions Documentation

Gestures

Gestures ( )

Default constructor.

Does nothing.

registerClickEvent

void registerClickEvent (ClickEvent::ClickEventType evt ,

```
                uint16_t        x ,
                uint16_t        y
                )
```

Register a click event and figure out if this is a drag event, too.

Parameters:

evt	The type of the click event.
x	The x coordinate of the click event.
y	The y coordinate of the click event.

registerDragEvent

```
bool registerDragEvent ( uint16_t oldX ,
                         uint16_t oldY ,
                         uint16_t newX ,
                         uint16_t newY
                         )
```

Register a drag event.

Parameters:

oldX	The x coordinate of the drag start position (dragged from)
oldY	The y coordinate of the drag start position (dragged from)
newX	The x coordinate of the new position (dragged to)
newY	The y coordinate of the new position (dragged to)

Returns:

True if the drag exceeds threshold value (and therefore was reported as a drag), or false if the drag did not exceed threshold (and therefore was discarded).

registerEventListener

```
void registerEventListener ( UIEventListener & l )
```

Register the event listener.

Parameters:

l	The EventListener to register.

setDragThreshold

void setDragThreshold (uint16_t val)

Configure the threshold for reporting drag events.

A touch input movement must exceed this value in either axis in order to report a drag. Default value is 0.

Parameters:

val New threshold value.

tick

void tick ( )

Has to be called during the timer tick.

Version: 4.16

GlyphNode

struct providing information about a glyph. Used by LCD when rendering.

Public Functions

FORCE_INLINE_FUNCTION uint16_t	**advance**() const
	Gets the "advance" value where the 9th bit is stored in flags.
FORCE_INLINE_FUNCTION uint16_t	**height**() const
	Gets the "height" value where the 9th bit is stored in flags.
FORCE_INLINE_FUNCTION uint16_t	**kerningTablePos**() const
	Gets the "kerningTablePos" value where the 8th and 9th bits are stored in flags.
FORCE_INLINE_FUNCTION void	**setTop**(int16_t newTop)
	Sets a new value for top.
FORCE_INLINE_FUNCTION int16_t	**top**() const
	Gets the "top" value where the 9th bit and the sign bit are stored in flags.
FORCE_INLINE_FUNCTION uint16_t	**width**() const
	Gets the "width" value where the 9th bit is stored in flags.

Public Attributes

uint8_t	**_advance**
	Width of the glyph (including space to the left and right)
uint8_t	**_height**
	Height of the actual glyph data.
uint8_t	**_kerningTablePos**
	Where are the kerning information for this glyph stored in the kerning table.
uint8_t	**_top**

	Vertical offset from baseline of the glyph.
uint8_t	**_width**
	Width of the actual glyph data.
uint32_t	**dataOffset**
	The index to the data of this glyph.
uint8_t	**flags**
	Additional glyph flags (font encoding and extra precision for width/height/top/advance)
uint8_t	**kerningTableSize**
	How many entries are there in the kerning table (following kerningTablePos) for this glyph.
int8_t	**left**
	Horizontal offset from the left of the glyph.
Unicode::UnicodeChar	**unicode**
	The Unicode of this glyph.

Public Functions Documentation

advance

FORCE_INLINE_FUNCTION uint16_t advance ( ) const

Gets the "advance" value where the 9th bit is stored in flags.

Returns:

the right value of "advance".

height

FORCE_INLINE_FUNCTION uint16_t height ( ) const

Gets the "height" value where the 9th bit is stored in flags.

Returns:

the right value of "height".

kerningTablePos

FORCE_INLINE_FUNCTION uint16_t kerningTablePos ( ) const

Gets the "kerningTablePos" value where the 8th and 9th bits are stored in flags.

Returns:

the right value of "kerningTablePos".

setTop

FORCE_INLINE_FUNCTION void setTop (int16_t newTop)

Sets a new value for top.

Used to adjust the vertical position of a glyph - this is used when positioning some Thai glyphs and some Arabic glyphs.

Parameters:

newTop The new top.

top

FORCE_INLINE_FUNCTION int16_t top ( ) const

Gets the "top" value where the 9th bit and the sign bit are stored in flags.

Returns:

the right value of "top".

width

FORCE_INLINE_FUNCTION uint16_t width ( ) const

Gets the "width" value where the 9th bit is stored in flags.

Returns:

the right value of "width".

Public Attributes Documentation

_advance

uint8_t _advance

Width of the glyph (including space to the left and right)

_height

uint8_t _height

Height of the actual glyph data.

_kerningTablePos

uint8_t _kerningTablePos

Where are the kerning information for this glyph stored in the kerning table.

_top

uint8_t _top

Vertical offset from baseline of the glyph.

_width

uint8_t _width

Width of the actual glyph data.

dataOffset

uint32_t dataOffset

The index to the data of this glyph.

flags

uint8_t flags

Additional glyph flags (font encoding and extra precision for width/height/top/advance)

kerningTableSize

uint8_t kerningTableSize

How many entries are there in the kerning table (following kerningTablePos) for this glyph.

left

int8_t left

Horizontal offset from the left of the glyph.

unicode

Unicode::UnicodeChar unicode

The Unicode of this glyph.

Version: 4.16

GPIO

Interface class for manipulating GPIOs in order to do performance measurements on target. Not used on the PC simulator.

Public Types

enum	**GPIO_ID** { VSYNC_FREQ, RENDER_TIME, FRAME_RATE, MCU_ACTIVE }
	Enum for the GPIOs used.

Public Functions

void	**clear**(**GPIO_ID** id)
	Sets a pin low.
bool	**get**(**GPIO_ID** id)
	Gets the state of a pin.
void	**init**()
	Perform configuration of IO pins.
void	**set**(**GPIO_ID** id)
	Sets a pin high.
void	**toggle**(**GPIO_ID** id)
	Toggles a pin.

Public Types Documentation

GPIO_ID

enum GPIO_ID

Enum for the GPIOs used.

VSYNC_FREQ	

RENDER_TIME	Pin is toggled at each VSYNC.
FRAME_RATE	Pin is high when frame rendering begins, low when finished.
MCU_ACTIVE	Pin is toggled when the framebuffers are swapped. Pin is high when the MCU is doing work (i.e. not in idle task).

Public Functions Documentation

clear

static void clear (GPIO_ID id)

Sets a pin low.

Parameters:

id the pin to set.

get

static bool get (GPIO_ID id)

Gets the state of a pin.

Parameters:

id the pin to get.

Returns:

true if the pin is high, false otherwise.

init

static void init ( )

Perform configuration of IO pins.

set

static void set (GPIO_ID id)

Sets a pin high.

Parameters:

id the pin to set.

toggle

static void toggle (GPIO_ID id)

Toggles a pin.

Parameters:

id the pin to toggle.

Version: 4.16

Gradients

Gradients contains all the data to interpolate u,v texture coordinates and z coordinates across a planar surface.

Public Functions

Gradients(const **Point3D** * vertices)

Initializes a new instance of the TextureMapTypes class.

Public Attributes

float **dOneOverZdX**

d(1/z)/dX

float **dOneOverZdY**

d(1/z)/dY

fixed16_16 **dUdXModifier**

The dUdX x coordinate modifier.

float **dUOverZdX**

d(u/z)/dX

float **dUOverZdY**

d(u/z)/dY

fixed16_16 **dVdXModifier**

The dVdX x coordinate modifier.

float **dVOverZdX**

d(v/z)/dX

float **dVOverZdY**

d(v/z)/dY

float **oneOverZ**

	1/z for each vertex
float	**UOverZ**
	u/z for each vertex
float	**VOverZ**
	v/z for each vertex

Public Functions Documentation

Gradients

Gradients (const Point3D * vertices)

Initializes a new instance of the TextureMapTypes class.

Construct the gradients using three 3D vertices.

Parameters:

vertices The vertices.

See also:

Point3D

Public Attributes Documentation

dOneOverZdX

float dOneOverZdX

d(1/z)/dX

dOneOverZdY

float dOneOverZdY

d(1/z)/dY

dUdXModifier

fixed16_16 dUdXModifier

The dUdX x coordinate modifier.

dUOverZdX

float dUOverZdX

d(u/z)/dX

dUOverZdY

float dUOverZdY

d(u/z)/dY

dVdXModifier

fixed16_16 dVdXModifier

The dVdX x coordinate modifier.

dVOverZdX

float dVOverZdX

d(v/z)/dX

dVOverZdY

float dVOverZdY

d(v/z)/dY

oneOverZ

float oneOverZ

1/z for each vertex

UOverZ

float UOverZ

u/z for each vertex

VOverZ

float VOverZ

v/z for each vertex

Version: 4.16

GraphClickEvent

An object of this type is passed with each callback that is sent when the graph is clicked. The object contains the data index that was pressed and the details of the click event, e.g. PRESSED, RELEASED and screen coordinates.

Public Functions

GraphClickEvent(int16_t i, const **ClickEvent** & event)
Initializes a new instance of the **GraphClickEvent** class.

Public Attributes

const **ClickEvent** &	**clickEvent**
	The ClickEvent that caused the callback to be executed.
int16_t	**index**
	The index of the item clicked.

Public Functions Documentation

GraphClickEvent

```
GraphClickEvent ( int16_t                i ,
                  const ClickEvent & event
                  )
```

Initializes a new instance of the **GraphClickEvent** class.

Parameters:

i	The index of the item clicked.
event	The **ClickEvent** that caused the callback to be executed.

See also:

setClickAction

Public Attributes Documentation

clickEvent

const ClickEvent & clickEvent

The ClickEvent that caused the callback to be executed.

index

int16_t index

The index of the item clicked.

Version: 4.16

GraphDragEvent

An object of this type is passed with each callback that is sent when the graph is dragged. The object contains the data index that was pressed and the details of the drag event, e.g. old and new screen coordinates.

Public Functions

GraphDragEvent(int16_t i, const **DragEvent** & event)
Initializes a new instance of the **GraphDragEvent** class.

Public Attributes

const **DragEvent** &	**dragEvent**
	The DragEvent that caused the callback to be executed.
int16_t	**index**
	The index of the item where the drag has ended.

Public Functions Documentation

GraphDragEvent

```
GraphDragEvent ( int16_t                i ,
                 const DragEvent & event
                 )
```

Initializes a new instance of the **GraphDragEvent** class.

Parameters:

i	The index of the item where the drag has ended.
event	The **DragEvent** that caused the callback to be executed.

See also:

setDragAction

Public Attributes Documentation

dragEvent

const DragEvent & dragEvent

The DragEvent that caused the callback to be executed.

index

int16_t index

The index of the item where the drag has ended.

Version: 4.16

GraphElementArea

GraphElementArea will fill the area below the line connecting the data points in the graph.

Note: The Area is drawn using CanvasWidget Renderer which is slower but produces much nicer graphics.

Inherits from: AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
float	**getBaselineAsFloat**() const
	Gets the base previously set using setBase.
int	**getBaselineAsInt**() const
	Gets the base previously set using setBase.
	GraphElementArea()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setBaseline**(float baseline)
	Sets the base of the area drawn.
void	**setBaseline**(int baseline)
	Sets the base of the area drawn.

Protected Functions

int	**getBaselineScaled**() const
	Gets the base previously set using setBase.
void	**setBaselineScaled**(int baseline)

Sets the base of the area drawn.

Protected Attributes

int **yBaseline**

The base value.

Additional inherited members

Public Functions inherited from AbstractGraphElement

AbstractGraphElement()

int **getScale**() const

Gets the scaling factor set using setScale.

void **setScale**(int scale)

Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int **convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const

Converts a number with one scale to a number that has the same scale as the graph.

AbstractDataGraph * **getGraph**() const

Gets a pointer to the the graph containing the GraphElement.

int **getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const

Gets maximum y coordinate for the graph.

int **getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const

Gets minimum y coordinate for the graph.

int **getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const

Get x axis offset as a scaled value.

int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const

	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

virtual void **translateRectToAbsolute**(**Rect** & r) const

	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: touchgfx::CanvasWidget::drawCanvasWidget

getBaselineAsFloat

float getBaselineAsFloat ( ) const

Gets the base previously set using setBase.

Returns:

The base value.

See also:

setBaseline

getBaselineAsInt

int getBaselineAsInt ( ) const

Gets the base previously set using setBase.

Returns:

The base value.

See also:

setBaseline

GraphElementArea

GraphElementArea ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and

the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setBaseline

void setBaseline (float baseline)

Sets the base of the area drawn.

Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values. Setting the base to a very high number will cause the area above the graph to be drawn. Setting the base to a very low number will cause the area below the graph to be drawn (even for negative numbers, which are higher than the base value).

Parameters:

baseline The baseline value.

See also:

getBaselineAsInt, **getBaselineAsFloat**

setBaseline

void setBaseline (int baseline)

Sets the base of the area drawn.

Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values. Setting the base to a very high number will cause the area above the graph to be drawn. Setting the base to a very low number will cause the area below the graph to be drawn (even for negative numbers, which are higher than the base value).

Parameters:

baseline The baseline value.

See also:

getBaselineAsInt, **getBaselineAsFloat**

Protected Functions Documentation

getBaselineScaled

int getBaselineScaled ( ) const

Gets the base previously set using setBase.

Returns:

The base value.

NOTE

The baseline returned here is left unscaled. For internal use.

See also:

setBaseline

setBaselineScaled

void setBaselineScaled (int baseline)

Sets the base of the area drawn.

Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values. Setting the base to a very high number will cause the area above the graph to be drawn. Setting the base to a very low number will cause the area below the graph to be drawn (even for negative numbers, which are higher than the base value).

Parameters:

baseline The baseline value.

NOTE

The baseline set here must already be scaled. For internal use.

See also:

getBaselineAsInt, **getBaselineAsFloat**

Protected Attributes Documentation

yBaseline

int yBaseline

The base value.

Version: 4.16

GraphElementBoxes

GraphElementBoxes will draw square box for every data point in graph.

Note: The boxes are drawn using LCD::fillRect for higher performance. This also means that boxes with an odd width will not align properly if combined with a GraphElementLine or any other GraphElement that uses CanvasWidget Renderer. Use an even number for box width in these cases.

Inherits from: AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
uint16_t	**getBoxWidth**() const
	Gets box width.
	GraphElementBoxes()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setBoxWidth**(uint16_t width)
	Sets box width.

Protected Attributes

uint16_t	**boxWidth**
	Width of the box.

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)

Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.

CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.

void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.

Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.

virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

getBoxWidth

uint16_t getBoxWidth ( ) const

Gets box width.

Returns:

The box width.

See also:

setBoxWidth

GraphElementBoxes

GraphElementBoxes ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setBoxWidth

void setBoxWidth (uint16_t width)

Sets box width.

Parameters:

width The width.

See also:

getBoxWidth

Protected Attributes Documentation

boxWidth

uint16_t boxWidth

Width of the box.

Version: 4.16

GraphElementDiamonds

GraphElementDiamonds will draw a diamond (a square with the corners up/down/left/right) for every data point in graph.

Note: The Diamonds are drawn using CanvasWidget Renderer which is slower but produces much nicer graphics.

Inherits from: AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
uint8_t	**getDiamondWidth**() const
	Gets diamond width.
	GraphElementDiamonds()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setDiamondWidth**(uint8_t width)
	Sets diamond width.

Protected Attributes

uint8_t	**diamondWidth**
	Width of the diamond.

Additional inherited members

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()

int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.

Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.

virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()

	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)

	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**

	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::CanvasWidget::drawCanvasWidget**

getDiamondWidth

uint8_t getDiamondWidth ( ) const

Gets diamond width.

Returns:

The diamond width.

See also:

setDiamondWidth

GraphElementDiamonds

GraphElementDiamonds ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setDiamondWidth

void setDiamondWidth (uint8_t width)

Sets diamond width.

Parameters:

width The width.

See also:

getDiamondWidth

Protected Attributes Documentation

diamondWidth

uint8_t diamondWidth

Width of the diamond.

Version: 4.16

GraphElementDots

GraphElementDots will draw a circular dot for every data point in graph.

Note: The Dots are drawn using CanvasWidget Renderer which is slower but produces much nicer graphics.

Inherits from: AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
uint8_t	**getDotWidth**() const
	Gets dot width.
	GraphElementDots()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setDotWidth**(uint8_t width)
	Sets dot width.

Protected Attributes

uint8_t	**dotWidth**
	Width of the dot.

Additional inherited members

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()

int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.

Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.

virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()

	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)

	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**

	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::CanvasWidget::drawCanvasWidget**

getDotWidth

uint8_t getDotWidth ( ) const

Gets dot width.

Returns:

The dot width.

See also:

setDotWidth

GraphElementDots

GraphElementDots ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setDotWidth

void setDotWidth (uint8_t width)

Sets dot width.

Parameters:

width The width.

See also:

getDotWidth

Protected Attributes Documentation

dotWidth

uint8_t dotWidth

Width of the dot.

Version: 4.16

GraphElementGridBase

GraphElementGridBase is a helper class used to implement classed to draw grid lines in the graph.

Inherits from: AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Inherited by: GraphElementGridX, GraphElementGridY

Public Functions

float	**getIntervalAsFloat**() const
	Gets the interval between each grid line.
int	**getIntervalAsInt**() const
	Gets the interval between each grid line.
uint8_t	**getLineWidth**() const
	Gets line width.
	GraphElementGridBase()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setInterval**(float interval)
	Sets the interval between each grid line.
void	**setInterval**(int interval)
	Sets the interval between each grid line.
void	**setLineWidth**(uint8_t width)
	Sets line width of the grid lines.
void	**setMajorGrid**(const **GraphElementGridBase** & major)
	Sets "major" grid that will be responsible for drawing major grid lines.

Protected Functions

int	**getCorrectlyScaledMajorInterval**(const **AbstractDataGraph** * graph) const
	Gets correctly scaled major interval, as the major grid may have a scale that differs the scale of the graph and this grid line.
int	**getIntervalScaled**() const
	Gets the interval between each grid line.
void	**setIntervalScaled**(int interval)
	Sets the interval between each grid line.

Protected Attributes

int	**gridInterval**
	The grid line interval.
uint8_t	**lineWidth**
	Width of the line.
const **GraphElementGridBase** *	**majorGrid**
	A pointer to a major grid, if any.

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.

int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.

virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)

	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getIntervalAsFloat

float getIntervalAsFloat ( ) const

Gets the interval between each grid line.

Returns:

The interval between each grid line.

See also:

setInterval

getIntervalAsInt

int getIntervalAsInt ( ) const

Gets the interval between each grid line.

Returns:

The interval between each grid line.

See also:

setInterval

getLineWidth

uint8_t getLineWidth ( ) const

Gets line width.

Returns:

The line width.

See also:

setLineWidth

GraphElementGridBase

GraphElementGridBase ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setInterval

void setInterval (float interval)

Sets the interval between each grid line.

Parameters:

interval The interval between each grid line.

NOTE

If interval is 0 only the axis is shown.

See also:

getIntervalAsInt, **getIntervalAsFloat**, **setMajorGrid**

setInterval

void setInterval (int interval)

Sets the interval between each grid line.

Parameters:

interval The interval between each grid line.

NOTE

If interval is 0 only the axis is shown.

See also:

getIntervalAsInt, **getIntervalAsFloat**, **setMajorGrid**

setLineWidth

void setLineWidth (uint8_t width)

Sets line width of the grid lines.

Parameters:

width The width of the grid lines.

See also:

getLineWidth

setMajorGrid

void setMajorGrid (const GraphElementGridBase & major)

Sets "major" grid that will be responsible for drawing major grid lines.

If a grid line would be drawn at the same position as the major grid line, the grid line will not be drawn.

Parameters:

major Reference to a major grid line object.

Protected Functions Documentation

getCorrectlyScaledMajorInterval

int getCorrectlyScaledMajorInterval (const AbstractDataGraph * graph)

Gets correctly scaled major interval, as the major grid may have a scale that differs the scale of the graph and this grid line.

Parameters:

graph The graph.

Returns:

The correctly scaled major interval.

getIntervalScaled

int getIntervalScaled ( ) const

Gets the interval between each grid line.

Returns:

The interval between each grid line.

NOTE

The interval returned here is left unscaled. For internal use.

See also:

setInterval

setIntervalScaled

void setIntervalScaled (int interval)

Sets the interval between each grid line.

Parameters:

interval The interval between each grid line.

NOTE

If interval is 0 only the axis is shown.The interval set here must already be scaled. For internal use.

See also:

getIntervalAsInt, **getIntervalAsFloat**, **setMajorGrid**

Protected Attributes Documentation

gridInterval

int gridInterval

The grid line interval.

lineWidth

uint8_t lineWidth

Width of the line.

majorGrid

const GraphElementGridBase * majorGrid

A pointer to a major grid, if any.

Version: 4.16

GraphElementGridX

GraphElementGridX draws vertical lines at selected intervals along the x axis. By combining two GraphElementGridX instances, it is possible to have minor and major grid lines.

Note: The grid lines are drawn using LCD::fillRect for higher performance.

Inherits from: GraphElementGridBase, AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.

Protected Functions

void	**drawLine**(const **Rect** & invalidatedArea, int16_t xMin, int16_t yMin, int16_t width, int16_t length, **colortype** color, uint8_t alpha) const
	Draw vertical line using LCD::fillRect and handles negative dimensions properly.

Additional inherited members

Public Functions inherited from GraphElementGridBase

float	**getIntervalAsFloat**() const
	Gets the interval between each grid line.
int	**getIntervalAsInt**() const
	Gets the interval between each grid line.
uint8_t	**getLineWidth**() const
	Gets line width.
	GraphElementGridBase()

virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setInterval**(float interval)
	Sets the interval between each grid line.
void	**setInterval**(int interval)
	Sets the interval between each grid line.
void	**setLineWidth**(uint8_t width)
	Sets line width of the grid lines.
void	**setMajorGrid**(const **GraphElementGridBase** & major)
	Sets "major" grid that will be responsible for drawing major grid lines.

Protected Functions inherited from GraphElementGridBase

int	**getCorrectlyScaledMajorInterval**(const **AbstractDataGraph** * graph) const
	Gets correctly scaled major interval, as the major grid may have a scale that differs the scale of the graph and this grid line.
int	**getIntervalScaled**() const
	Gets the interval between each grid line.
void	**setIntervalScaled**(int interval)
	Sets the interval between each grid line.

Protected Attributes inherited from GraphElementGridBase

int	**gridInterval**
	The grid line interval.
uint8_t	**lineWidth**
	Width of the line.
const **GraphElementGridBase** *	**majorGrid**
	A pointer to a major grid, if any.

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
virtual void	**invalidateGraphPointAt**(int16_t index) =0

	Invalidate the point at the given index.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const

	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.

virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

Protected Functions Documentation

drawLine

void drawLine (const Rect & invalidatedArea , const
int16_t xMin , const

	int16_t	yMin ,	const
	int16_t	width ,	const
	int16_t	length ,	const
	colortype	color ,	const
	uint8_t	alpha	const
	)	const	

Draw vertical line using LCD::fillRect and handles negative dimensions properly.

Parameters:

invalidatedArea	The invalidated area to intersect the line with.
xMin	The minimum x coordinate.
yMin	The minimum y coordinate.
width	The width of the line.
length	The length of the line.
color	The color of the line.
alpha	The alpha of the line.

Version: 4.16

GraphElementGridY

GraphElementGridY draws horizontal lines at selected intervals along the y axis. By combining two GraphElementGridY instances, it is possible to have minor and major grid lines.

Note: The grid lines are drawn using LCD::fillRect for higher performance.

Inherits from: GraphElementGridBase, AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.

Protected Functions

void	**drawLine**(const **Rect** & invalidatedArea, int16_t xMin, int16_t yMin, int16_t width, int16_t length, **colortype** color, uint8_t alpha) const
	Draw horizontal line using LCD::fillRect and handles negative dimensions properly.

Additional inherited members

Public Functions inherited from GraphElementGridBase

float	**getIntervalAsFloat**() const
	Gets the interval between each grid line.
int	**getIntervalAsInt**() const
	Gets the interval between each grid line.
uint8_t	**getLineWidth**() const
	Gets line width.
	GraphElementGridBase()

virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setInterval**(float interval)
	Sets the interval between each grid line.
void	**setInterval**(int interval)
	Sets the interval between each grid line.
void	**setLineWidth**(uint8_t width)
	Sets line width of the grid lines.
void	**setMajorGrid**(const **GraphElementGridBase** & major)
	Sets "major" grid that will be responsible for drawing major grid lines.

Protected Functions inherited from GraphElementGridBase

int	**getCorrectlyScaledMajorInterval**(const **AbstractDataGraph** * graph) const
	Gets correctly scaled major interval, as the major grid may have a scale that differs the scale of the graph and this grid line.
int	**getIntervalScaled**() const
	Gets the interval between each grid line.
void	**setIntervalScaled**(int interval)
	Sets the interval between each grid line.

Protected Attributes inherited from GraphElementGridBase

int	**gridInterval**
	The grid line interval.
uint8_t	**lineWidth**
	Width of the line.
const **GraphElementGridBase** *	**majorGrid**
	A pointer to a major grid, if any.

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
virtual void	**invalidateGraphPointAt**(int16_t index) =0

	Invalidate the point at the given index.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const

	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.

virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

Protected Functions Documentation

drawLine

void drawLine (const Rect & invalidatedArea , const
int16_t xMin , const

	int16_t	yMin ,	const
	int16_t	width ,	const
	int16_t	length ,	const
	colortype	color ,	const
	uint8_t	alpha	const
	)	const	

Draw horizontal line using LCD::fillRect and handles negative dimensions properly.

Parameters:

invalidatedArea	The invalidated area to intersect the line with.
xMin	The minimum x coordinate.
yMin	The minimum y coordinate.
width	The width of the line.
length	The length of the line.
color	The color of the line.
alpha	The alpha of the line.

Version: 4.16

GraphElementHistogram

The GraphElementHistogram is used to draw blocks from the x axis to the data point in the graph. If more graphs are placed on top of each other, the histogram can be moved slightly to the left/right.

Note: Historgram boxes are drawn using LCD::fillRect for higher performance.

Inherits from: AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
int16_t	**getBarOffset**() const
	Gets bar offset (horizontally).
uint16_t	**getBarWidth**() const
	Gets bar width of the histogram columns.
float	**getBaselineAsFloat**() const
	Gets the base previously set using setBaseline.
int	**getBaselineAsInt**() const
	Gets the base previously set using setBaseline.
	GraphElementHistogram()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setBarOffset**(int16_t offset)
	Sets bar offset (horizontally).
void	**setBarWidth**(uint16_t width)
	Sets bar width of each histogram column.
void	**setBaseline**(float baseline)

Sets the base of the area drawn.

void **setBaseline**(int baseline)

Sets the base of the area drawn.

Protected Functions

int **getBaselineScaled**() const

Gets the base previously set using setBaseline.

void **setBaselineScaled**(int baseline)

Sets the base of the area drawn.

Protected Attributes

int16_t **barOffset**

The horizontal bar offset.

uint16_t **barWidth**

Width of each bar.

int **yBaseline**

The baseline.

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

AbstractGraphElementNoCWR()

virtual bool **drawCanvasWidget**(const **Rect** & invalidatedArea) const

Draw canvas widget for the given invalidated area.

virtual **colortype** **getColor**() const

Gets the color of the graph element.

virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.

int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const

Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()

	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)

	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

getBarOffset

int16_t getBarOffset ( ) const

Gets bar offset (horizontally).

Bar offset can be used when there are two different histogram graphs on top of each other to prevent one histogram from covering the other.

Returns:

The bar offset.

See also:

setBarOffset

getBarWidth

uint16_t getBarWidth ( ) const

Gets bar width of the histogram columns.

Returns:

The bar width.

See also:

setBarWidth

getBaselineAsFloat

float getBaselineAsFloat ( ) const

Gets the base previously set using setBaseline.

Returns:

The base value.

See also:

setBaseline

getBaselineAsInt

int getBaselineAsInt ( ) const

Gets the base previously set using setBaseline.

Returns:

The base value.

See also:

setBaseline

GraphElementHistogram

GraphElementHistogram ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setBarOffset

void setBarOffset (int16_t offset)

Sets bar offset (horizontally).

This can be used when there are two different histogram graphs on top of each other to prevent one histogram from covering the other.

Parameters:

offset The offset.

See also:

getBarOffset

setBarWidth

void setBarWidth (uint16_t width)

Sets bar width of each histogram column.

Parameters:

width The width.

See also:

getBarWidth

setBaseline

void setBaseline (float baseline)

Sets the base of the area drawn.

Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values. Setting the base to a very high number will cause the area above the graph to be

drawn. Setting the base to a very low number will cause the area below the graph to be drawn (even for negative numbers, which are higher than the base value).

Parameters:

baseline The base value.

See also:

getBaselineAsInt, **getBaselineAsFloat**

setBaseline

void setBaseline (int baseline)

Sets the base of the area drawn.

Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values. Setting the base to a very high number will cause the area above the graph to be drawn. Setting the base to a very low number will cause the area below the graph to be drawn (even for negative numbers, which are higher than the base value).

Parameters:

baseline The base value.

See also:

getBaselineAsInt, **getBaselineAsFloat**

Protected Functions Documentation

getBaselineScaled

int getBaselineScaled ( ) const

Gets the base previously set using setBaseline.

Returns:

The base value.

NOTE

The baseline returned here is left unscaled. For internal use.

See also:

setBaseline

setBaselineScaled

void setBaselineScaled (int baseline)

Sets the base of the area drawn.

Normally, the base is 0 which means that the area is drawn below positive y values and above negative y values. Setting the base to a very high number will cause the area above the graph to be drawn. Setting the base to a very low number will cause the area below the graph to be drawn (even for negative numbers, which are higher than the base value).

Parameters:

baseline The base value.

NOTE

The baseline set here must already be scaled. For internal use.

See also:

getBaselineAsInt, **getBaselineAsFloat**

Protected Attributes Documentation

barOffset

int16_t barOffset

The horizontal bar offset.

barWidth

uint16_t barWidth

Width of each bar.

yBaseline

int yBaseline

The baseline.

Version: 4.16

GraphElementLine

GraphElementLine will draw a line with a given thickness through the data points in the graph.

Note: The Line is drawn using CanvasWidget Renderer which is slower but produces much nicer graphics.

Inherits from: AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
uint8_t	**getLineWidth**() const
	Gets line width.
	GraphElementLine()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setLineWidth**(uint8_t width)
	Sets line width.

Protected Functions

void	**drawIndexRange**(**Canvas** & canvas, const **AbstractDataGraph** * graph, int16_t indexMin, int16_t indexMax) const
	Draw a line between all indexes in the given range.

Protected Attributes

uint8_t	**lineWidth**
	Width of the line.

Additional inherited members

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const

	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const

	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**

	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::CanvasWidget::drawCanvasWidget**

getLineWidth

uint8_t getLineWidth ( ) const

Gets line width.

Returns:

The line width.

See also:

setLineWidth

GraphElementLine

GraphElementLine ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setLineWidth

void setLineWidth (uint8_t width)

Sets line width.

Parameters:

width The width.

See also:

getLineWidth

Protected Functions Documentation

drawIndexRange

void drawIndexRange (	Canvas &	canvas ,	const
	const AbstractDataGraph *	graph ,	const
	int16_t	indexMin ,	const
	int16_t	indexMax	const
	)	const	

Draw a line between all indexes in the given range.

This is used where there is a gap in the graph and the line has to be drawn as two separate lines.

Parameters:

canvas	The canvas.
graph	The graph.
indexMin	The minimum index.
indexMax	The maximum index.

Protected Attributes Documentation

lineWidth

uint8_t lineWidth

Width of the line.

Version: 4.16

GraphElementVerticalGapLine

The GraphElementVerticalGapLine is used to draw a vertical line where the gap in the graph is. This only makes sense to add to a GraphWrapAndOverwrite (or DataGraphWrapAndOverwrite).

Note: The vertical line is drawn using LCD::fillRect for higher performance.

Inherits from: AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
uint16_t	**getGapLineWidth**() const
	Gets the width of the gap line as set using setGapLineWidth().
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setGapLineWidth**(uint16_t width)
	Sets the width of the gap line in pixels.

Protected Attributes

uint16_t	**lineWidth**
	Width of the line.

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

AbstractGraphElementNoCWR()

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const

	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.

const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.

bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)

	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

getGapLineWidth

uint16_t getGapLineWidth ( ) const

Gets the width of the gap line as set using setGapLineWidth().

Returns:

The gap line width.

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and

the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setGapLineWidth

void setGapLineWidth (uint16_t width)

Sets the width of the gap line in pixels.

If the gap line is set to 0 the gap line will extend to the next point in the graph.

Parameters:

width The width.

See also:

getGapLineWidth

Protected Attributes Documentation

lineWidth

uint16_t lineWidth

Width of the line.

Version: 4.16

GraphLabelsBase

Helper class for adding labels on the side of a graph.

See: GraphLabelsX, GraphLabelsY

Inherits from: AbstractGraphDecoration, AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Inherited by: GraphLabelsX, GraphLabelsY

Public Functions

float	**getIntervalAsFloat**() const
	Gets the interval between each label.
int	**getIntervalAsInt**() const
	Gets the interval between each label.
Unicode::UnicodeChar	**getLabelDecimalPoint**() const
	Gets label decimal point previously set.
uint16_t	**getLabelDecimals**() const
	Gets number of decimals for labels.
TextRotation	**getLabelRotation**() const
	Gets label rotation.
TypedText	**getLabelTypedText**() const
	Gets TypedText label.
	GraphLabelsBase()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setInterval**(float interval)
	Sets the interval between each label.
void	**setInterval**(int interval)

	Sets the interval between each label.
void	**setLabelDecimalPoint**(**Unicode::UnicodeChar** decimalPoint)
	Sets label decimal point.
void	**setLabelDecimals**(uint16_t decimals)
	Sets number of decimals for labels, default is no decimals and no decimal point.
void	**setLabelRotation**(**TextRotation** rotation)
	Sets label rotation.
void	**setLabelTypedText**(const **TypedText** & typedText)
	Sets TypedText to use for the label.
void	**setMajorLabel**(const **GraphLabelsBase** & major)
	Sets "major" label that will be responsible for drawing major labels.

Protected Functions

void	**formatLabel**(**Unicode::UnicodeChar** * buffer, int16_t bufferSize, int label, int decimals, **Unicode::UnicodeChar** decimalPoint, int scale) const
	Format label according to the set number of decimals and the decimal point.
int	**getCorrectlyScaledMajorInterval**(const **AbstractDataGraph** * graph) const
	Gets correctly scaled major interval, as the major label may have a scale that differs the scale of the graph and this label.
int	**getIntervalScaled**() const
	Gets the interval between each label.
void	**setIntervalScaled**(int interval)
	Sets the interval between each label.

Protected Attributes

Unicode::UnicodeChar	**labelDecimalPoint**
	The label decimal point character.
uint16_t	**labelDecimals**

	The number of decimals on the label.
int	**labelInterval**
	The interval between each label.
TextRotation	**labelRotation**
	The TextRotation to use for the label.
TypedText	**labelTypedText**
	The TypedText to use for the label.
const **GraphLabelsBase** *	**majorLabel**
	A pointer to a major label, if any.

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	color
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	getScale() const
	Gets the scaling factor set using setScale.
void	setScale(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	convertToGraphScale(const AbstractDataGraph * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	getGraph() const
	Gets a pointer to the the graph containing the GraphElement.
int	getGraphRangeYMaxScaled(const AbstractDataGraph * graph) const
	Gets maximum y coordinate for the graph.
int	getGraphRangeYMinScaled(const AbstractDataGraph * graph) const
	Gets minimum y coordinate for the graph.
int	getGraphXAxisOffsetScaled(const AbstractDataGraph * graph) const
	Get x axis offset as a scaled value.
int	getGraphXAxisScaleScaled(const AbstractDataGraph * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	indexToScreenXQ5(const AbstractDataGraph * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.

CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.

	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const

	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()

	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getIntervalAsFloat

float getIntervalAsFloat ( ) const

Gets the interval between each label.

Returns:

The interval between each label.

See also:

setInterval

getIntervalAsInt

int getIntervalAsInt ( ) const

Gets the interval between each label.

Returns:

The interval between each label.

See also:

setInterval

getLabelDecimalPoint

Unicode::UnicodeChar getLabelDecimalPoint ( ) const

Gets label decimal point previously set.

Returns:

The label decimal point.

See also:

setLabelDecimalPoint

getLabelDecimals

uint16_t getLabelDecimals ( ) const

Gets number of decimals for labels.

Returns:

The number of label decimals.

getLabelRotation

TextRotation getLabelRotation ( ) const

Gets label rotation.

Returns:

The label rotation.

See also:

setLabelRotation

getLabelTypedText

TypedText getLabelTypedText ( ) const

Gets TypedText label.

Returns:

The label typed text.

See also:

setLabelTypedText

GraphLabelsBase

GraphLabelsBase ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

Reimplemented by: **touchgfx::GraphLabelsX::invalidateGraphPointAt**

setInterval

void setInterval (float interval)

Sets the interval between each label.

Parameters:

interval The interval between each label.

NOTE

If interval is 0 only the axis is shown.

See also:

getIntervalAsInt, **getIntervalAsFloat**, **setMajorLabel**

setInterval

void setInterval (int interval)

Sets the interval between each label.

Parameters:

interval The interval between each label.

NOTE

If interval is 0 only the axis is shown.

See also:

getIntervalAsInt, **getIntervalAsFloat**, **setMajorLabel**

setLabelDecimalPoint

void setLabelDecimalPoint (Unicode::UnicodeChar decimalPoint)

Sets label decimal point.

Default is to use '.' but this can be changed using this function.

Parameters:

decimalPoint The character to use for decimal point.

NOTE

The decimal point is only set if the label decimals > 0.

See also:

setLabelDecimals

setLabelDecimals

void setLabelDecimals (uint16_t decimals)

Sets number of decimals for labels, default is no decimals and no decimal point.

Parameters:

decimals The number of label decimals.

See also:

setLabelDecimalPoint

setLabelRotation

void setLabelRotation (TextRotation rotation)

Sets label rotation.

Parameters:

rotation The rotation or the text.

See also:

getLabelRotation

setLabelTypedText

void setLabelTypedText (const TypedText & typedText)

Sets TypedText to use for the label.

The **TypedText** should contain exactly one wildcard.

Parameters:

typedText The typed text.

See also:

getLabelTypedText

setMajorLabel

void setMajorLabel (const GraphLabelsBase & major)

Sets "major" label that will be responsible for drawing major labels.

If a label would be drawn at the same position as the major label, the label will not be drawn.

Parameters:

major	Reference to a major label object.

Protected Functions Documentation

formatLabel

```
void formatLabel ( Unicode::UnicodeChar * buffer ,                const
                   int16_t                bufferSize ,            const
                   int                    label ,                 const
                   int                    decimals ,              const
                   Unicode::UnicodeChar   decimalPoint ,          const
                   int                    scale                   const
                   )                      const
```

Format label according to the set number of decimals and the decimal point.

Parameters:

buffer	The buffer to fill with the formatted number.
bufferSize	Size of the buffer.
label	The label value.
decimals	The number of decimals.
decimalPoint	The decimal point.
scale	The scale of the label value.

getCorrectlyScaledMajorInterval

```
int getCorrectlyScaledMajorInterval ( const AbstractDataGraph * graph )
```

Gets correctly scaled major interval, as the major label may have a scale that differs the scale of the graph and this label.

Parameters:

graph	The graph.

Returns:

The correctly scaled major interval.

getIntervalScaled

int getIntervalScaled ( ) const

Gets the interval between each label.

Returns:

The interval between each label.

NOTE

The interval returned here is left unscaled. For internal use.

See also:

setInterval

setIntervalScaled

void setIntervalScaled (int interval)

Sets the interval between each label.

Parameters:

interval The interval between each label.

NOTE

If interval is 0 only the axis is shown.The interval set here must already be scaled. For internal use.

See also:

getIntervalAsInt, **getIntervalAsFloat**, **setMajorLabel**

Protected Attributes Documentation

labelDecimalPoint

Unicode::UnicodeChar labelDecimalPoint

The label decimal point character.

labelDecimals

uint16_t labelDecimals

The number of decimals on the label.

labelInterval

int labelInterval

The interval between each label.

labelRotation

TextRotation labelRotation

The TextRotation to use for the label.

labelTypedText

TypedText labelTypedText

The TypedText to use for the label.

majorLabel

const GraphLabelsBase * majorLabel

A pointer to a major label, if any.

Version: 4.16

GraphLabelsX

GraphLabelsX will draw labels along the X axis at given intervals. By combining two GraphLabelsX it is possible to have different appearance for major and minor y offsets.

Inherits from: GraphLabelsBase, AbstractGraphDecoration, AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.

Protected Functions

void	**drawIndexRange**(const **Rect** & invalidatedArea, const **Font** *fontToDraw, const* ***AbstractDataGraph*** graph, int indexLow, int indexHigh, const uint8_t alpha) const
	Draw labels for all indexes in the given range.
void	**drawString**(const **Rect** & invalidatedArea, const **Font** *fontToDraw, const* ***AbstractDataGraph*** graph, int index, const uint8_t alpha) const
	Draw string using rotation and clipping to make sure it is written properly.

Additional inherited members

Public Functions inherited from GraphLabelsBase

float	**getIntervalAsFloat**() const
	Gets the interval between each label.
int	**getIntervalAsInt**() const
	Gets the interval between each label.

Unicode::UnicodeChar	**getLabelDecimalPoint**() const
	Gets label decimal point previously set.
uint16_t	**getLabelDecimals**() const
	Gets number of decimals for labels.
TextRotation	**getLabelRotation**() const
	Gets label rotation.
TypedText	**getLabelTypedText**() const
	Gets TypedText label.
	GraphLabelsBase()
void	**setInterval**(float interval)
	Sets the interval between each label.
void	**setInterval**(int interval)
	Sets the interval between each label.
void	**setLabelDecimalPoint**(**Unicode::UnicodeChar** decimalPoint)
	Sets label decimal point.
void	**setLabelDecimals**(uint16_t decimals)
	Sets number of decimals for labels, default is no decimals and no decimal point.
void	**setLabelRotation**(**TextRotation** rotation)
	Sets label rotation.
void	**setLabelTypedText**(const **TypedText** & typedText)
	Sets TypedText to use for the label.
void	**setMajorLabel**(const **GraphLabelsBase** & major)
	Sets "major" label that will be responsible for drawing major labels.

Protected Functions inherited from GraphLabelsBase

void	**formatLabel**(**Unicode::UnicodeChar** * buffer, int16_t bufferSize, int label, int decimals, **Unicode::UnicodeChar** decimalPoint, int scale) const
	Format label according to the set number of decimals and the decimal point.

int	**getCorrectlyScaledMajorInterval**(const **AbstractDataGraph** * graph) const
	Gets correctly scaled major interval, as the major label may have a scale that differs the scale of the graph and this label.
int	**getIntervalScaled**() const
	Gets the interval between each label.
void	**setIntervalScaled**(int interval)
	Sets the interval between each label.

Protected Attributes inherited from GraphLabelsBase

Unicode::UnicodeChar	**labelDecimalPoint**
	The label decimal point character.
uint16_t	**labelDecimals**
	The number of decimals on the label.
int	**labelInterval**
	The interval between each label.
TextRotation	**labelRotation**
	The TextRotation to use for the label.
TypedText	**labelTypedText**
	The TypedText to use for the label.
const **GraphLabelsBase** *	**majorLabel**
	A pointer to a major label, if any.

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.

virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.

int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const

Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()

	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)

	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::GraphLabelsBase::invalidateGraphPointAt**

Protected Functions Documentation

drawIndexRange

void drawIndexRange (const Rect & invalidatedArea , const
const Font * fontToDraw , const
const AbstractDataGraph * graph , const

	int	indexLow ,	const
	int	indexHigh ,	const
	const uint8_t	alpha	const
	)	const	

Draw labels for all indexes in the given range.

This is used where there is a gap in the graph and the labels have to be drawn using different x scales.

Parameters:

invalidatedArea	The canvas.
fontToDraw	The font to draw.
graph	The graph.
indexLow	The minimum index.
indexHigh	The maximum index.
alpha	The alpha.

drawString

void drawString (	const Rect &	invalidatedArea ,	const
	const Font *	fontToDraw ,	const
	const AbstractDataGraph *	graph ,	const
	int	index ,	const
	const uint8_t	alpha	const
	)	const	

Draw string using rotation and clipping to make sure it is written properly.

Parameters:

invalidatedArea	The invalidated area.
fontToDraw	The font to draw.
graph	The graph.
index	index of the data point.
alpha	The alpha.

Version: 4.16

GraphLabelsY

GraphLabelsY will draw labels along the Y axis at given intervals. By combining two GraphLabelsY it is possible to have different appearance for major and minor y offsets.

Inherits from: GraphLabelsBase, AbstractGraphDecoration, AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.

Protected Functions

void	**drawString**(const **Rect** & invalidatedArea, const **Font** *fontToDraw, const* ***AbstractDataGraph*** graph, int valueScaled, int labelScaled, const uint8_t alpha) const
	Draw string using rotation and clipping to make sure it is written properly.

Additional inherited members

Public Functions inherited from GraphLabelsBase

float	**getIntervalAsFloat**() const
	Gets the interval between each label.
int	**getIntervalAsInt**() const
	Gets the interval between each label.
Unicode::UnicodeChar	**getLabelDecimalPoint**() const
	Gets label decimal point previously set.
uint16_t	**getLabelDecimals**() const
	Gets number of decimals for labels.

TextRotation	**getLabelRotation**() const
	Gets label rotation.
TypedText	**getLabelTypedText**() const
	Gets TypedText label.
	GraphLabelsBase()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setInterval**(float interval)
	Sets the interval between each label.
void	**setInterval**(int interval)
	Sets the interval between each label.
void	**setLabelDecimalPoint**(**Unicode::UnicodeChar** decimalPoint)
	Sets label decimal point.
void	**setLabelDecimals**(uint16_t decimals)
	Sets number of decimals for labels, default is no decimals and no decimal point.
void	**setLabelRotation**(**TextRotation** rotation)
	Sets label rotation.
void	**setLabelTypedText**(const **TypedText** & typedText)
	Sets TypedText to use for the label.
void	**setMajorLabel**(const **GraphLabelsBase** & major)
	Sets "major" label that will be responsible for drawing major labels.

Protected Functions inherited from GraphLabelsBase

void	**formatLabel**(**Unicode::UnicodeChar** * buffer, int16_t bufferSize, int label, int decimals, **Unicode::UnicodeChar** decimalPoint, int scale) const
	Format label according to the set number of decimals and the decimal point.
int	**getCorrectlyScaledMajorInterval**(const **AbstractDataGraph** * graph) const
	Gets correctly scaled major interval, as the major label may have a scale that differs the scale of the graph and this label.

int	**getIntervalScaled**() const
	Gets the interval between each label.
void	**setIntervalScaled**(int interval)
	Sets the interval between each label.

Protected Attributes inherited from GraphLabelsBase

Unicode::UnicodeChar	**labelDecimalPoint**
	The label decimal point character.
uint16_t	**labelDecimals**
	The number of decimals on the label.
int	**labelInterval**
	The interval between each label.
TextRotation	**labelRotation**
	The TextRotation to use for the label.
TypedText	**labelTypedText**
	The TypedText to use for the label.
const **GraphLabelsBase** *	**majorLabel**
	A pointer to a major label, if any.

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
virtual **colortype**	**getColor**() const
	Gets the color of the graph element.
virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()
int	**getScale**() const
	Gets the scaling factor set using setScale.
virtual void	**invalidateGraphPointAt**(int16_t index) =0
	Invalidate the point at the given index.
void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.

int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.
CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()

	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)

	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea	The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

Protected Functions Documentation

drawString

```
void drawString ( const Rect &                invalidatedArea ,  const
                  const Font *                fontToDraw ,       const
                  const AbstractDataGraph * graph ,              const
                  int                         valueScaled ,      const
                  int                         labelScaled ,      const
                  const uint8_t               alpha              const
                  )                           const
```

Draw string using rotation and clipping to make sure it is written properly.

Parameters:

invalidatedArea	The invalidated area.
fontToDraw	The font to draw.
graph	The graph.
valueScaled	The value (left scaled according to graph scale).
labelScaled	The label value (left scaled according to graph label scale).
alpha	The alpha.

Version: 4.16

GraphScroll

A Widget capable of drawing a graph with various visual styles and different appearances for the new values added to the graph.

Inherits from: DataGraphScroll, AbstractDataGraphWithY, AbstractDataGraph, Container, Drawable

Public Functions

	GraphScroll()

Additional inherited members

Public Functions inherited from DataGraphScroll

virtual void	**clear**()
	Clears the graph to its blank/initial state.
	DataGraphScroll(int16_t capacity, int * values)
	Initializes a new instance of the **DataGraphScroll** class.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.

Protected Functions inherited from DataGraphScroll

virtual int16_t	**addValue**(int value)
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.

Protected Attributes inherited from DataGraphScroll

int16_t	**current**
	The current position used for inserting new elements.

Public Functions inherited from AbstractDataGraphWithY

	AbstractDataGraphWithY(int16_t capacity, int * values)
	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)
	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.
virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const
	Get x coordinate axis scale value.

virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.
virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.
void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions inherited from AbstractDataGraphWithY

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.
virtual int16_t	**addValue**(int value) =0
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.

virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)
	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const

	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraphWithY

uint32_t	**dataCounter**
	The data counter of how many times addDataPoint() has been called.
int	**xOffset**
	The x axis offset (real value of data point at index 0)
int	**xScale**
	The x axis scale (increment between two data points)

Public Classes inherited from AbstractDataGraph

class	**GraphClickEvent**
	An object of this type is passed with each callback that is sent when the graph is clicked.
class	**GraphDragEvent**
	An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

	AbstractDataGraph(int16_t capacity)
	Initializes a new instance of the **AbstractDataGraph** class.
void	**addBottomElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area below the graph.
void	**addGraphElement**(**AbstractGraphElement** & d)
	Adds a graph element which will display the graph.
void	**addLeftElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area to the left of the graph.

void	**addRightElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area to the right of the graph.
void	**addTopElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area above the graph.
virtual void	**clear**()
	Clears the graph to its blank/initial state.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
int16_t	**getGapBeforeIndex**() const
	Gets gap before index as set using setGapBeforeIndex().
int16_t	**getGraphAreaHeight**() const
	Gets graph area height.
int16_t	**getGraphAreaHeightIncludingPadding**() const
	Gets graph area height including padding (but not margin).
int16_t	**getGraphAreaMarginBottom**() const
	Gets graph margin bottom.
int16_t	**getGraphAreaMarginLeft**() const
	Gets graph margin left.
int16_t	**getGraphAreaMarginRight**() const
	Gets graph margin right.
int16_t	**getGraphAreaMarginTop**() const
	Gets graph margin top.
int16_t	**getGraphAreaPaddingBottom**() const
	Gets graph area padding bottom.
int16_t	**getGraphAreaPaddingLeft**() const
	Gets graph area padding left.
int16_t	**getGraphAreaPaddingRight**() const
	Gets graph area padding right.
int16_t	**getGraphAreaPaddingTop**() const

Gets graph area padding top.

int16_t **getGraphAreaWidth**() const

Gets graph area width.

int16_t **getGraphAreaWidthIncludingPadding**() const

Gets graph area width including padding (but not margin).

virtual int **getGraphRangeXMax**() const =0

Gets the maximum x coordinate for the graph.

virtual int **getGraphRangeXMin**() const =0

Gets the minimum x coordinate for the graph.

virtual float **getGraphRangeYMaxAsFloat**() const =0

Gets maximum y coordinate for the graph.

virtual int **getGraphRangeYMaxAsInt**() const =0

Gets maximum y coordinate for the graph.

virtual float **getGraphRangeYMinAsFloat**() const =0

Gets minimum y coordinate for the graph.

virtual int **getGraphRangeYMinAsInt**() const =0

Gets minimum y coordinate for the graph.

int16_t **getMaxCapacity**() const

Gets the capacity (max number of points) of the graph.

virtual bool **getNearestIndexForScreenX**(int16_t x, int16_t & index) const

Gets graph index nearest to the given screen x coordinate.

virtual bool **getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)

Gets graph index nearest to the given screen position.

int **getScale**() const

Gets the scaling factor previously set using setScale().

int16_t **getUsedCapacity**() const

Gets the number of point used by the graph.

virtual void **handleClickEvent**(const **ClickEvent** & evt)

Defines the event handler interface for ClickEvents.

virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.
int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const
	Get the data point y value for the given graph point index.
int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.
int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)

	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setGraphRangeX**(int min, int max) =0
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const

	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.

void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)
	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**

	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**
	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**
	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**

	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**
	The graph area top padding.
int16_t	**usedCapacity**
	The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.

virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**

	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

GraphScroll

GraphScroll ( )

Version: 4.16

GraphTitle

The GraphTitle is just a simple text, but it is automatically moved with the graph. Also, the alpha value is combined with the alpha of the graph and so it will be faded if the graph is faded.

Inherits from: AbstractGraphDecoration, AbstractGraphElementNoCWR, AbstractGraphElement, CanvasWidget, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
TextRotation	**getTitleRotation**() const
	Gets title rotation.
TypedText	**getTitleTypedText**() const
	Gets title typed text.
	GraphTitle()
virtual void	**invalidateGraphPointAt**(int16_t index)
	Invalidate the point at the given index.
void	**setTitleRotation**(**TextRotation** rotation)
	Sets TextRotation of the title.
void	**setTitleTypedText**(const **TypedText** & typedText)
	Sets TypedText to use as a title.

Additional inherited members

Public Functions inherited from AbstractGraphElementNoCWR

	AbstractGraphElementNoCWR()

virtual **colortype**	**getColor**() const
	Gets the color of the graph element.

virtual void	**setColor**(**colortype** newColor)
	Sets the color of the graph element.

Protected Functions inherited from AbstractGraphElementNoCWR

void	**normalizeRect**(**Rect** & rect) const
	Normalize rectangle by changing a rectangle with negative width or height to a rectangle with positive width or height at the correct position.

virtual void	**setPainter**(**AbstractPainter** & painter)
	Protected function to prevent users from setting a painter.

Protected Attributes inherited from AbstractGraphElementNoCWR

colortype	**color**
	The currently assigned color.

Public Functions inherited from AbstractGraphElement

	AbstractGraphElement()

int	**getScale**() const
	Gets the scaling factor set using setScale.

void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.

Protected Functions inherited from AbstractGraphElement

int	**convertToGraphScale**(const **AbstractDataGraph** * graph, int value, int scale) const

	Converts a number with one scale to a number that has the same scale as the graph.
AbstractDataGraph *	**getGraph**() const
	Gets a pointer to the the graph containing the GraphElement.
int	**getGraphRangeYMaxScaled**(const **AbstractDataGraph** * graph) const
	Gets maximum y coordinate for the graph.
int	**getGraphRangeYMinScaled**(const **AbstractDataGraph** * graph) const
	Gets minimum y coordinate for the graph.
int	**getGraphXAxisOffsetScaled**(const **AbstractDataGraph** * graph) const
	Get x axis offset as a scaled value.
int	**getGraphXAxisScaleScaled**(const **AbstractDataGraph** * graph) const
	Get x axis scale as a scaled value.
CWRUtil::Q5	**indexToScreenXQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
CWRUtil::Q5	**indexToScreenYQ5**(const **AbstractDataGraph** * graph, int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
bool	**isCenterInvisible**(const **AbstractDataGraph** * graph, int16_t index) const
	Query if the center of a given data point index is visible inside the graph area.
Rect	**rectAround**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5, **CWRUtil::Q5** diameterQ5) const
	Find the screen rectangle around a given point with the specified diameter.
Rect	**rectFromQ5Coordinates**(**CWRUtil::Q5** screenXminQ5, **CWRUtil::Q5** screenYminQ5, **CWRUtil::Q5** screenXmaxQ5, **CWRUtil::Q5** screenYmaxQ5) const
	Find the screen rectangle containing the Q5 screen rectangle by rounding the coordinates up/down.
CWRUtil::Q5	**roundQ5**(**CWRUtil::Q5** q5) const
	Round the given CWRUtil::Q5 to the nearest integer and return it as a CWRUtil::Q5 instead of an integer.
CWRUtil::Q5	**valueToScreenXQ5**(const **AbstractDataGraph** * graph, int x) const
	Gets graph screen x for x value.

CWRUtil::Q5	**valueToScreenYQ5**(const **AbstractDataGraph** * graph, int y) const
	Gets graph screen y for y value.
bool	**xScreenRangeToIndexRange**(int16_t xLow, int16_t xHigh, int16_t & elementLow, int16_t & elementHigh) const
	Gets graph element range for screen x coordinate range.

Protected Attributes inherited from AbstractGraphElement

int	**dataScale**
	The scaling factor.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0

	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draws the given invalidated area.

If the underlying **CanvasWidgetRenderer** fail to render the widget (if the widget is too complex), the invalidated area is cut into smaller slices (horizontally) which are then drawn separately. If

drawing a single raster line fails, that line is considered too complex and skipped (it is left blank/transparent) and drawing continues on the next raster line.

If drawing has failed at least once, which means that the number of horizontal lines draw has been reduced, the number of successfully drawn horizontal lines is remembered for the next invocation of **draw()**. A future call to **draw()** would then start off with the reduced number of horizontal lines to prevent potentially drawing the canvas widget in vain, as happened previously in **draw()**.

Parameters:

invalidatedArea The invalidated area.

NOTE

Subclasses of **CanvasWidget** should implement **drawCanvasWidget()**, not **draw()**. The term "too complex" means that the size of the buffer (assigned to **CanvasWidgetRenderer** using **CanvasWidgetRenderer::setupBuffer()**) is too small.

See also:

drawCanvasWidget

Reimplements: **touchgfx::CanvasWidget::draw**

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::AbstractGraphElementNoCWR::drawCanvasWidget**

getTitleRotation

TextRotation getTitleRotation ( ) const

Gets title rotation.

Returns:

The title rotation.

See also:

setTitleRotation

getTitleTypedText

TypedText getTitleTypedText ( ) const

Gets title typed text.

Returns:

The title typed text.

See also:

setTitleTypedText

GraphTitle

GraphTitle ( )

invalidateGraphPointAt

virtual void invalidateGraphPointAt (int16_t index)

Invalidate the point at the given index.

This allows a graph element to only invalidate the minimum rectangle required for the given index. The Graph will call this function before and after changing a point to ensure that both the old and the new area are redrawn (invalidated).

Parameters:

index Zero-based index of the point.

Reimplements: **touchgfx::AbstractGraphElement::invalidateGraphPointAt**

setTitleRotation

void setTitleRotation (TextRotation rotation)

Sets TextRotation of the title.

Parameters:

rotation The rotation.

See also:

setTitleTypedText, **getTitleRotation**

setTitleTypedText

void setTitleTypedText (const TypedText & typedText)

Sets TypedText to use as a title.

It can be any static text which is just added as a title.

Parameters:

typedText The typed text.

See also:

getTitleTypedText

Version: 4.16

GraphWrapAndClear

The GraphWrapAndClear will show new points progressing across the graph. Once the graph is filled, the next point added will cause the graph to be cleared and a new graph will slowly be created as new values are added.

Inherits from: DataGraphWrapAndClear, AbstractDataGraphWithY, AbstractDataGraph, Container, Drawable

Public Functions

	GraphWrapAndClear()

Additional inherited members

Public Functions inherited from DataGraphWrapAndClear

	DataGraphWrapAndClear(int16_t capacity, int * values)
	Initializes a new instance of the DataGraphWrapAndOverwrite class.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.

Protected Functions inherited from DataGraphWrapAndClear

virtual int16_t	**addValue**(int value)
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.

Public Functions inherited from AbstractDataGraphWithY

	AbstractDataGraphWithY(int16_t capacity, int * values)

	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)
	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.
virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const
	Get x coordinate axis scale value.
virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.
virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.

virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.
void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions inherited from AbstractDataGraphWithY

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.
virtual int16_t	**addValue**(int value) =0
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.
virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.

virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)
	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraphWithY

uint32_t **dataCounter**

The data counter of how many times addDataPoint() has been called.

int **xOffset**

The x axis offset (real value of data point at index 0)

int **xScale**

The x axis scale (increment between two data points)

Public Classes inherited from AbstractDataGraph

class **GraphClickEvent**

An object of this type is passed with each callback that is sent when the graph is clicked.

class **GraphDragEvent**

An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

AbstractDataGraph(int16_t capacity)

Initializes a new instance of the **AbstractDataGraph** class.

void **addBottomElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area below the graph.

void **addGraphElement**(**AbstractGraphElement** & d)

Adds a graph element which will display the graph.

void **addLeftElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the left of the graph.

void **addRightElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the right of the graph.

void **addTopElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area above the graph.

virtual void **clear**()

Clears the graph to its blank/initial state.

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

int16_t **getGapBeforeIndex**() const

Gets gap before index as set using setGapBeforeIndex().

int16_t **getGraphAreaHeight**() const

Gets graph area height.

int16_t **getGraphAreaHeightIncludingPadding**() const

Gets graph area height including padding (but not margin).

int16_t **getGraphAreaMarginBottom**() const

Gets graph margin bottom.

int16_t **getGraphAreaMarginLeft**() const

Gets graph margin left.

int16_t **getGraphAreaMarginRight**() const

Gets graph margin right.

int16_t **getGraphAreaMarginTop**() const

Gets graph margin top.

int16_t **getGraphAreaPaddingBottom**() const

Gets graph area padding bottom.

int16_t **getGraphAreaPaddingLeft**() const

Gets graph area padding left.

int16_t **getGraphAreaPaddingRight**() const

Gets graph area padding right.

int16_t **getGraphAreaPaddingTop**() const

Gets graph area padding top.

int16_t **getGraphAreaWidth**() const

Gets graph area width.

int16_t **getGraphAreaWidthIncludingPadding**() const

Gets graph area width including padding (but not margin).

virtual int **getGraphRangeXMax**() const =0

Gets the maximum x coordinate for the graph.

virtual int	**getGraphRangeXMin**() const =0
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const =0
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const =0
	Gets minimum y coordinate for the graph.
int16_t	**getMaxCapacity**() const
	Gets the capacity (max number of points) of the graph.
virtual bool	**getNearestIndexForScreenX**(int16_t x, int16_t & index) const
	Gets graph index nearest to the given screen x coordinate.
virtual bool	**getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)
	Gets graph index nearest to the given screen position.
int	**getScale**() const
	Gets the scaling factor previously set using setScale().
int16_t	**getUsedCapacity**() const
	Gets the number of point used by the graph.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.
int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const

	Get the data point y value for the given graph point index.
int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.
int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setGraphRangeX**(int min, int max) =0
	Sets minimum and maximum x coordinates for the graph.

virtual void	**setGraphRangeY**(float min, float max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const
	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0
	Gets minimum y coordinate for the graph.

virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.
void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)

	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**
	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.

GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**
	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**
	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**
	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**
	The graph area top padding.
int16_t	**usedCapacity**

	The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

GraphWrapAndClear

```
GraphWrapAndClear ( )
```

Version: 4.16

GraphWrapAndOverwrite

A Continuous graph. A quick way to create a DataGraphWrapAndOverwrite.

Inherits from: DataGraphWrapAndOverwrite, AbstractDataGraphWithY, AbstractDataGraph, Container, Drawable

Public Functions

	GraphWrapAndOverwrite()

Additional inherited members

Public Functions inherited from DataGraphWrapAndOverwrite

virtual void	**clear**()
	Clears the graph to its blank/initial state.
	DataGraphWrapAndOverwrite(int16_t capacity, int * values)
	Initializes a new instance of the **DataGraphWrapAndOverwrite** class.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.

Protected Functions inherited from DataGraphWrapAndOverwrite

virtual int16_t	**addValue**(int value)
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()
	This function is called before a new value (data point) is added.

Protected Attributes inherited from DataGraphWrapAndOverwrite

int16_t	**current**
	The current index (used to keep track of where to insert new data point in value array)

Public Functions inherited from AbstractDataGraphWithY

	AbstractDataGraphWithY(int16_t capacity, int * values)
	Initializes a new instance of the **AbstractDataGraphWithY** class.
int16_t	**addDataPoint**(float y)
	Adds a new data point to the end of the graph.
int16_t	**addDataPoint**(int y)
	Adds a new data point to the end of the graph.
virtual int	**getGraphRangeXMax**() const
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const
	Gets minimum y coordinate for the graph.
virtual float	**getXAxisOffsetAsFloat**() const
	Get x coordinate axis offset value.
virtual int	**getXAxisOffsetAsInt**() const
	Get x coordinate axis offset value.
virtual float	**getXAxisScaleAsFloat**() const

	Get x coordinate axis scale value.
virtual int	**getXAxisScaleAsInt**() const
	Get x coordinate axis scale value.
virtual void	**setGraphRangeX**(int min, int max)
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max)
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max)
	Sets minimum and maximum y coordinates for the graph.
void	**setGraphRangeYAuto**(bool showXaxis =true, int margin =0)
	Automatic adjust min and max y coordinate to show entire graph.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setXAxisOffset**(float offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisOffset**(int offset)
	Set x coordinate axis offset value.
virtual void	**setXAxisScale**(float scale)
	Set x coordinate axis scale value.
virtual void	**setXAxisScale**(int scale)
	Set x coordinate axis scale value.

Protected Functions inherited from AbstractDataGraphWithY

int16_t	**addDataPointScaled**(int y)
	Same as **addDataPoint(int)** except the passed argument is assumed scaled.
virtual int16_t	**addValue**(int value) =0
	Adds a value to the internal data array and keeps track of when graph points, graph axis and the entire graph needs to be redrawn (invalidated).
virtual void	**beforeAddValue**()

	This function is called before a new value (data point) is added.
virtual int	**getGraphRangeYMaxScaled**() const
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const
	Gets screen y coordinate for a specific data point added to the graph.
virtual int16_t	**realIndex**(int16_t index) const
	Get the real index in the yValues array of the given index.
virtual void	**setGraphRangeYScaled**(int min, int max)
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
virtual void	**setXAxisOffsetScaled**(int offset)
	Set x coordinate axis offset value with a pre-scaled offset value.
virtual void	**setXAxisScaleScaled**(int scale)
	Set x coordinate axis scale value using a pre-scaled value.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const
	Gets screen x coordinate for an absolute value.

virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraphWithY

uint32_t	**dataCounter**
	The data counter of how many times addDataPoint() has been called.
int	**xOffset**
	The x axis offset (real value of data point at index 0)
int	**xScale**
	The x axis scale (increment between two data points)

Public Classes inherited from AbstractDataGraph

class	**GraphClickEvent**
	An object of this type is passed with each callback that is sent when the graph is clicked.
class	**GraphDragEvent**
	An object of this type is passed with each callback that is sent when the graph is dragged.

Public Functions inherited from AbstractDataGraph

	AbstractDataGraph(int16_t capacity)
	Initializes a new instance of the **AbstractDataGraph** class.
void	**addBottomElement**(**AbstractGraphDecoration** & d)
	Adds an element to be shown in the area below the graph.
void	**addGraphElement**(**AbstractGraphElement** & d)
	Adds a graph element which will display the graph.
void	**addLeftElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the left of the graph.

void **addRightElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area to the right of the graph.

void **addTopElement**(**AbstractGraphDecoration** & d)

Adds an element to be shown in the area above the graph.

virtual void **clear**()

Clears the graph to its blank/initial state.

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

int16_t **getGapBeforeIndex**() const

Gets gap before index as set using setGapBeforeIndex().

int16_t **getGraphAreaHeight**() const

Gets graph area height.

int16_t **getGraphAreaHeightIncludingPadding**() const

Gets graph area height including padding (but not margin).

int16_t **getGraphAreaMarginBottom**() const

Gets graph margin bottom.

int16_t **getGraphAreaMarginLeft**() const

Gets graph margin left.

int16_t **getGraphAreaMarginRight**() const

Gets graph margin right.

int16_t **getGraphAreaMarginTop**() const

Gets graph margin top.

int16_t **getGraphAreaPaddingBottom**() const

Gets graph area padding bottom.

int16_t **getGraphAreaPaddingLeft**() const

Gets graph area padding left.

int16_t **getGraphAreaPaddingRight**() const

Gets graph area padding right.

int16_t	**getGraphAreaPaddingTop**() const
	Gets graph area padding top.
int16_t	**getGraphAreaWidth**() const
	Gets graph area width.
int16_t	**getGraphAreaWidthIncludingPadding**() const
	Gets graph area width including padding (but not margin).
virtual int	**getGraphRangeXMax**() const =0
	Gets the maximum x coordinate for the graph.
virtual int	**getGraphRangeXMin**() const =0
	Gets the minimum x coordinate for the graph.
virtual float	**getGraphRangeYMaxAsFloat**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMaxAsInt**() const =0
	Gets maximum y coordinate for the graph.
virtual float	**getGraphRangeYMinAsFloat**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getGraphRangeYMinAsInt**() const =0
	Gets minimum y coordinate for the graph.
int16_t	**getMaxCapacity**() const
	Gets the capacity (max number of points) of the graph.
virtual bool	**getNearestIndexForScreenX**(int16_t x, int16_t & index) const
	Gets graph index nearest to the given screen x coordinate.
virtual bool	**getNearestIndexForScreenXY**(int16_t x, int16_t y, int16_t & index)
	Gets graph index nearest to the given screen position.
int	**getScale**() const
	Gets the scaling factor previously set using setScale().
int16_t	**getUsedCapacity**() const
	Gets the number of point used by the graph.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
float	**indexToDataPointXAsFloat**(int16_t index) const
	Get the data point x value for the given graph point index.
int	**indexToDataPointXAsInt**(int16_t index) const
	Get the data point x value for the given graph point index.
float	**indexToDataPointYAsFloat**(int16_t index) const
	Get the data point y value for the given graph point index.
int	**indexToDataPointYAsInt**(int16_t index) const
	Get the data point y value for the given graph point index.
virtual int32_t	**indexToGlobalIndex**(int16_t index) const
	Convert an index to global index.
int16_t	**indexToScreenX**(int16_t index) const
	Get the screen x coordinate for the given graph point index.
int16_t	**indexToScreenY**(int16_t index) const
	Get the screen y coordinate for the given graph point index.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setClickAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphClickEvent** & > & callback)
	Sets an action to be executed when the graph is clicked.
void	**setDragAction**(**GenericCallback**< const **AbstractDataGraph** &, const **GraphDragEvent** & > & callback)
	Sets an action to be executed when the graph is dragged.
void	**setGapBeforeIndex**(int16_t index)
	Makes gap before the specified index.
void	**setGraphAreaMargin**(int16_t top, int16_t left, int16_t right, int16_t bottom)
	Sets graph position inside the widget by reserving a margin around the graph.
void	**setGraphAreaPadding**(int16_t top, int16_t left, int16_t right, int16_t bottom)

	Adds some padding around the graph that will not be drawn in (apart from dots, boxes etc.
void	**setGraphRange**(int xMin, int xMax, float yMin, float yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
void	**setGraphRange**(int xMin, int xMax, int yMin, int yMax)
	Sets minimum and maximum x and y coordinate ranges for the graph.
virtual void	**setGraphRangeX**(int min, int max) =0
	Sets minimum and maximum x coordinates for the graph.
virtual void	**setGraphRangeY**(float min, float max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setGraphRangeY**(int min, int max) =0
	Sets minimum and maximum y coordinates for the graph.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScale**(int scale)
	Sets a scaling factor to be multiplied on each added element.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
int	**float2scaled**(float f, int scale)
	Multiply a floating point value with a constant and round the result.
int	**int2scaled**(int i, int scale)
	Multiply an integer value with a constant.
float	**scaled2float**(int i, int scale)
	Divide a floating point number with a constant.
int	**scaled2int**(int i, int scale)
	Divide an integer with a constant and round the result.

Protected Functions inherited from AbstractDataGraph

int	**convertToGraphScale**(int value, int scale) const

	Converts a number with one scale to a number that has the same scale as the graph.
int	**float2scaled**(float f) const
	Same as **float2scaled(float,int)** using the graph's scale.
virtual int	**getGraphRangeYMaxScaled**() const =0
	Gets maximum y coordinate for the graph.
virtual int	**getGraphRangeYMinScaled**() const =0
	Gets minimum y coordinate for the graph.
virtual int	**getXAxisOffsetScaled**() const
	Get x axis offset as a scaled value.
virtual int	**getXAxisScaleScaled**() const
	Get x axis scale as a scaled value.
virtual int	**indexToDataPointXScaled**(int16_t index) const =0
	Same as indexToDataPointXAsInt(int16_t) except the returned value is left scaled.
virtual int	**indexToDataPointYScaled**(int16_t index) const =0
	Same as indexToDataPointYAsInt(int16_t) except the returned value is left scaled.
virtual **CWRUtil::Q5**	**indexToScreenXQ5**(int16_t index) const =0
	Gets screen x coordinate for a specific data point added to the graph.
virtual **CWRUtil::Q5**	**indexToScreenYQ5**(int16_t index) const =0
	Gets screen y coordinate for a specific data point added to the graph.
int	**int2scaled**(int i) const
	Same as **int2scaled(int,int)** using the graph's scale.
void	**invalidateAllXAxisPoints**()
	Invalidate all x axis points.
void	**invalidateGraphArea**()
	Invalidate entire graph area (the center of the graph).
void	**invalidateGraphPointAt**(int16_t index)
	Invalidate point at a given index.

void	**invalidateXAxisPointAt**(int16_t index)
	Invalidate x axis point at the given index.
float	**scaled2float**(int i) const
	Same as **scaled2float(int,int)** using the graph's scale.
int	**scaled2int**(int i) const
	Same as **scaled2int(int,int)** using the graph's scale.
void	**setGraphRangeScaled**(int xMin, int xMax, int yMin, int yMax)
	Same as **setGraphRange(int,int,int,int)** except the passed arguments are assumed scaled.
virtual void	**setGraphRangeYScaled**(int min, int max) =0
	Same as **setGraphRangeY(int,int)** except the passed arguments are assumed scaled.
void	**updateAreasPosition**()
	Updates the position of all elements in all area after a change in size of the graph area and/or label padding.
virtual **CWRUtil::Q5**	**valueToScreenXQ5**(int x) const =0
	Gets screen x coordinate for an absolute value.
virtual **CWRUtil::Q5**	**valueToScreenYQ5**(int y) const =0
	Gets screen y coordinate for an absolute value.
virtual bool	**xScreenRangeToIndexRange**(int16_t xLo, int16_t xHi, int16_t & indexLow, int16_t & indexHigh) const =0
	Gets index range for screen x coordinate range taking the current graph range into account.

Protected Attributes inherited from AbstractDataGraph

uint8_t	**alpha**
	The alpha of the entire graph.
Container	**bottomArea**
	The area below the graph.
int16_t	**bottomPadding**

	The graph area bottom padding.
GenericCallback< const **AbstractDataGraph** &, const **GraphClickEvent** & > *	**clickAction**
	The callback to be executed when this Graph is clicked.
int	**dataScale**
	The data scale applied to all values.
GenericCallback< const **AbstractDataGraph** &, const **GraphDragEvent** & > *	**dragAction**
	The callback to be executed when this Graph is dragged.
int16_t	**gapBeforeIndex**
	The graph is disconnected (there is a gap) before this element index.
Container	**graphArea**
	The graph area (the center area)
Container	**leftArea**
	The area to the left of the graph.
int16_t	**leftPadding**
	The graph area left padding.
int16_t	**maxCapacity**
	Maximum number of points in the graph.
Container	**rightArea**
	The area to the right of the graph.
int16_t	**rightPadding**

	The graph area right padding.
Container	**topArea**
	The area above the graph.
int16_t	**topPadding**
	The graph area top padding.
int16_t	**usedCapacity**
	The number of used points in the graph.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.

virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**

	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

GraphWrapAndOverwrite

GraphWrapAndOverwrite ( )

Version: 4.16

HAL

Hardware Abstraction Layer. Contains functions that are specific to the hardware platform the code is running on.

Inherited by: HALSDL2

Public Types

enum	**FrameRefreshStrategy** { REFRESH_STRATEGY_DEFAULT, REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL, REFRESH_STRATEGY_PARTIAL_FRAMEBUFFER }
	A list of available frame refresh strategies.
enum	**RenderingVariant** { SOFTWARE, HARDWARE }
	A list of rendering variants.

Public Functions

virtual void	**allowDMATransfers**()
	Allow the DMA to start transfers.
virtual void	**backPorchExited**()
	Has to be called from within the LCD IRQ rutine when the Back Porch Exit is reached.
void	**blitCopy**(const uint16_t *pSrc, const uint8_t* pClut, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha, bool hasTransparentPixels, uint16_t dstWidth, **Bitmap::BitmapFormat** srcFormat, **Bitmap::BitmapFormat** dstFormat)
	Blits a 2D source-array to the framebuffer performing alpha-blending as specified.
virtual void	**blitCopy**(const uint16_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha, bool hasTransparentPixels)
	Blits a 2D source-array to the framebuffer performing alpha-blending as specified using the default lcd format.

virtual void	**blitCopy**(const uint16_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha, bool hasTransparentPixels, uint16_t dstWidth, **Bitmap::BitmapFormat** srcFormat, **Bitmap::BitmapFormat** dstFormat)
	Blits a 2D source-array to the framebuffer performing alpha-blending as specified.
virtual void	**blitCopyARGB8888**(const uint16_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing per-pixel alpha blending.
virtual void	**blitCopyGlyph**(const uint8_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, **colortype** color, uint8_t alpha, **BlitOperations** operation)
	Blits a 4bpp or 8bpp glyph - maybe use the same method and supply additional color mode arg.
virtual void	**blitFill**(**colortype** color, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t alpha)
	Blits a color value to the framebuffer performing alpha-blending as specified.
virtual void	**blitFill**(**colortype** color, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t alpha, uint16_t dstWidth, **Bitmap::BitmapFormat** dstFormat)
	Blits a color value to the framebuffer performing alpha-blending as specified.
virtual bool	**blockCopy**(void *RESTRICT dest, const void* RESTRICT src, uint32_t numBytes)
	This function performs a platform-specific memcpy, if supported by the hardware.
virtual void	**configureInterrupts**() =0
	Configures the interrupts relevant for TouchGFX.
virtual uint16_t	**configurePartialFrameBuffer**(const uint16_t x, const uint16_t y, const uint16_t width, const uint16_t height)
	Configures a partial framebuffer as current framebuffer.
virtual uint16_t *	**copyFBRegionToMemory**(**Rect** meAbs)
	Copies a region of the currently displayed framebuffer to memory.
virtual uint16_t *	**copyFBRegionToMemory**(**Rect** meAbs, uint16_t * dst, uint32_t stride)

	Copies a region of the currently displayed framebuffer to a buffer.
virtual void	**disableInterrupts**() =0
	Disables the DMA and LCD interrupts.
virtual void	**drawDrawableInDynamicBitmap**(**Drawable** & drawable, **BitmapId** bitmapId)
	Render a **Drawable** and its widgets into a dynamic bitmap.
virtual void	**drawDrawableInDynamicBitmap**(**Drawable** & drawable, **BitmapId** bitmapId, const **Rect** & rect)
	Render a **Drawable** and its widgets into a dynamic bitmap.
void	**enableDMAAcceleration**(const bool enable)
	Sets a flag to allow use of DMA operations to speed up drawing operations.
virtual void	**enableInterrupts**() =0
	Enables the DMA and LCD interrupts.
virtual void	**enableLCDControllerInterrupt**() =0
	Configure the LCD controller to fire interrupts at VSYNC.
void	**enableMCULoadCalculation**(bool enabled)
	This method sets a flag that determines if generic **HAL** should calculate MCU load based on concrete MCU instrumentation.
virtual void	**flushDMA**()
	This function blocks until the DMA queue (containing BlitOps) is empty.
virtual void	**flushFrameBuffer**()
	This function is called whenever the framework has performed a complete draw.
virtual void	**flushFrameBuffer**(const **Rect** & rect)
	This function is called whenever the framework has performed a partial draw.
void	**frontPorchEntered**()
	Has to be called from within the LCD IRQ routine when the Front Porch Entry is reached.
uint16_t *	**getAnimationStorage**() const
	Gets the optional framebuffer used for animation storage.

LCD *	**getAuxiliaryLCD**()
	Get the auxiliary LCD class attached to the **HAL** instance if any.
virtual **BlitOperations**	**getBlitCaps**()
	Function for obtaining the blit capabilities of the concrete **HAL** implementation.
ButtonController *	**getButtonController**() const
	Gets the associated **ButtonController**.
uint32_t	**getCPUCycles**()
	Gets the current cycle counter.
uint16_t	**getDisplayHeight**() const
	Gets display height.
DisplayOrientation	**getDisplayOrientation**() const
	Gets the current display orientation.
uint16_t	**getDisplayWidth**() const
	Gets display width.
virtual **DMAType**	**getDMAType**()
	Function for obtaining the DMA type of the concrete DMA implementation.
uint8_t	**getFingerSize**() const
	Gets the finger size in pixels.
virtual **FlashDataReader** *	**getFlashDataReader**() const
	Gets the flash data reader.
FrameBufferAllocator *	**getFrameBufferAllocator**()
	Gets the framebuffer allocator.
FrameRefreshStrategy	**getFrameRefreshStrategy**() const
	Used internally by TouchGFX core to manage the timing and process of drawing into the framebuffer.
Gestures *	**getGestures**()
	Get the Gesture class attached to the **HAL** instance.
uint32_t	**getLCDRefreshCount**()

	Returns the number of VSync interrupts between the current drawing operation and the last drawing operation, i.e.
uint8_t	**getMCULoadPct**() const
	Gets the current MCU load.
virtual uint16_t	**getTFTCurrentLine**()
	Get the current line (Y) of the TFT controller.
virtual uint16_t *	**getTFTFrameBuffer**() const =0
	Gets the framebuffer address used by the TFT controller.
int8_t	**getTouchSampleRate**() const
	Gets the number of ticks between each touch screen sample.
	HAL(**DMA_Interface** & dmaInterface, **LCD** & display, **TouchController** & touchCtrl, uint16_t width, uint16_t height)
	Initializes a new instance of the **HAL** class.
void	**initialize**()
	This function is responsible for initializing the entire framework.
void	**lockDMAToFrontPorch**(bool enableLock)
	Function to set whether the DMA transfers are locked to the TFT update cycle.
virtual uint16_t *	**lockFrameBuffer**()
	Waits for the framebuffer to become available for use (i.e.
virtual void	**registerEventListener**(**UIEventListener** & listener)
	Registers an event handler implementation with the underlying event system.
void	**registerTaskDelayFunction**(void(*)(uint16_t) delayF)
	Registers a function capable of delaying GUI task execution.
virtual bool	**sampleKey**(uint8_t & key)
	Sample external key event.
void	**setAuxiliaryLCD**(**LCD** * auxLCD)
	Set an auxiliary LCD class to be used for offscreen rendering.
void	**setButtonController**(**ButtonController** * btnCtrl)

	Stores a pointer to an instance of a specific implementation of a **ButtonController**.
virtual void	**setDisplayOrientation**(**DisplayOrientation** orientation)
	Sets the desired display orientation (landscape or portrait).
void	**setDragThreshold**(uint8_t value)
	Configure the threshold for reporting drag events.
void	**setFingerSize**(uint8_t size)
	Sets the finger size in pixels.
void	**setFrameBufferAllocator**(**FrameBufferAllocator** * allocator)
	Sets a framebuffer allocator.
virtual void	**setFrameBufferStartAddresses**(void *frameBuffer, void* doubleBuffer, void * animationStorage)
	Sets framebuffer start addresses.
void	**setFrameRateCompensation**(bool enabled)
	Enables or disables compensation for lost frames.
bool	**setFrameRefreshStrategy**(**FrameRefreshStrategy** s)
	Set a specific strategy for handling timing and mechanism of framebuffer drawing.
void	**setMCUActive**(bool active)
	Register if MCU is active by measuring cpu cycles.
void	**setMCUInstrumentation**(**MCUInstrumentation** * mcuInstr)
	Stores a pointer to an instance of an MCU specific instrumentation class.
void	**setRenderingVariant**(**RenderingVariant** variant)
	Set current rendering variant for cache maintenance.
void	**setTouchSampleRate**(int8_t sampleRateInTicks)
	Sets the number of ticks between each touch screen sample.
void	**signalDMAInterrupt**()
	Notify the framework that a DMA interrupt has occurred.
void	**swapFrameBuffers**()
	Swaps the two framebuffers.

virtual void	**taskDelay**(uint16_t ms)
	Delay GUI task execution by number of milliseconds.
virtual void	**taskEntry**()
	Main event loop.
virtual void	**unlockFrameBuffer**()
	Unlocks the framebuffer (MUST be called exactly once for each call to **lockFrameBuffer()**).
void	**vSync**()
	Called by the VSync interrupt.
virtual	**~HAL**()
	Finalizes an instance of the **HAL** class.
HAL *	**getInstance**()
	Gets the **HAL** instance.
LCD &	**lcd**()
	Gets a reference to the LCD.

Protected Functions

virtual bool	**beginFrame**()
	Called when beginning to rendering a frame.
virtual void	**endFrame**()
	Called when a rendering pass is completed.
virtual void	**FlushCache**()
	Flush D-Cache.
uint16_t *	**getClientFrameBuffer**()
	Gets client framebuffer.
virtual void	**InvalidateCache**()
	Invalidate D-Cache.
virtual void	**noTouch**()
	Called by the touch driver to indicate that no touch is currently detected.

virtual void	**performDisplayOrientationChange**()
	Perform the actual display orientation change.
virtual void	**setTFTFrameBuffer**(uint16_t * address) =0
	Sets the framebuffer address used by the TFT controller.
virtual void	**tick**()
	This function is called at each timer tick, depending on platform implementation.
virtual void	**touch**(int32_t x, int32_t y)
	Called by the touch driver to indicate a touch.

Public Attributes

uint16_t	**DISPLAY_HEIGHT**
	The height of the LCD display in pixels.
DisplayRotation	**DISPLAY_ROTATION**
	The rotation from display to framebuffer.
uint16_t	**DISPLAY_WIDTH**
	The width of the LCD display in pixels.
uint16_t	**FRAME_BUFFER_HEIGHT**
	The height of the framebuffer in pixels.
uint16_t	**FRAME_BUFFER_WIDTH**
	The width of the framebuffer in pixels.
bool	**USE_ANIMATION_STORAGE**
	Is animation storage enabled?
bool	**USE_DOUBLE_BUFFERING**
	Is double buffering enabled?

Protected Attributes

LCD *	**auxiliaryLCD**
	Auxiliary LCD class used to render Drawables into dynamic bitmaps.

ButtonController *	**buttonController**
	A reference to an optional **ButtonController**.
DMA_Interface &	**dma**
	A reference to the DMA interface.
uint8_t	**fingerSize**
	The radius of the finger in pixels.
uint16_t *	**frameBuffer0**
	Pointer to the first framebuffer.
uint16_t *	**frameBuffer1**
	Pointer to the second framebuffer.
uint16_t *	**frameBuffer2**
	Pointer to the optional third framebuffer used for animation storage.
FrameBufferAllocator *	**frameBufferAllocator**
	A reference to an optional **FrameBufferAllocator**.
bool	**frameBufferUpdatedThisFrame**
	True if something was drawn in the current frame.
Gestures	**gestures**
	Class for low-level interpretation of touch events.
LCD &	**lcdRef**
	A reference to the LCD.
bool	**lockDMAToPorch**
	Whether or not to lock DMA transfers with TFT porch signal.
MCUInstrumentation *	**mcuInstrumentation**
	A reference to an optional MCU instrumentation.
DisplayOrientation	**nativeDisplayOrientation**
	Contains the native display orientation. If desired orientation is different, apply rotation.
Rect	**partialFrameBufferRect**
	The region of the screen covered by the partial framebuffer.

FrameRefreshStrategy	**refreshStrategy**
	The selected display refresh strategy.
void(*	**taskDelayFunc**
	Pointer to a function that can delay GUI task for a number of milliseconds.
TouchController &	**touchController**
	A reference to the touch controller.
bool	**isDrawing**
	True if currently in the process of rendering a screen.

Public Types Documentation

FrameRefreshStrategy

enum FrameRefreshStrategy

A list of available frame refresh strategies.

REFRESH_STRATEGY_DEFAULT	If not explicitly set, this strategy is used.
REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL	Strategy optimized for single framebuffer on systems with TFT controller.
REFRESH_STRATEGY_PARTIAL_FRAMEBUFFER	Strategy using less than a full framebuffer.

RenderingVariant

enum RenderingVariant

A list of rendering variants.

SOFTWARE	
HARDWARE	

Public Functions Documentation

allowDMATransfers

virtual void allowDMATransfers ( )

Allow the DMA to start transfers.

Front Porch Entry is a good place to call this.

backPorchExited

virtual void backPorchExited ( )

Has to be called from within the LCD IRQ rutine when the Back Porch Exit is reached.

Has to be called from within the **LCD** IRQ rutine when the Back Porch Exit is reached.

blitCopy

```
void blitCopy ( const uint16_t *          pSrc ,
                const uint8_t *           pClut ,
                uint16_t                  x ,
                uint16_t                  y ,
                uint16_t                  width ,
                uint16_t                  height ,
                uint16_t                  srcWidth ,
                uint8_t                   alpha ,
                bool                      hasTransparentPixels ,
                uint16_t                  dstWidth ,
                Bitmap::BitmapFormat      srcFormat ,
                Bitmap::BitmapFormat      dstFormat
                )
```

Blits a 2D source-array to the framebuffer performing alpha-blending as specified.

Parameters:

pSrc	The source-array pointer (points to first value to copy)
pClut	The CLUT pointer (points to CLUT header data which include the type and size of this CLUT followed by colors data)
x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.

height	The height of desired area of the source 2D array.
srcWidth	The distance (in elements) from first value of first line, to first value of second line (the source 2D array width)
alpha	The alpha value to use for blending (255 = solid, no blending)
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.
dstWidth	The distance (in elements) from first value of first line, to first value of second line (the destination 2D array width)
srcFormat	The source buffer color format (default is the framebuffer format)
dstFormat	The destination buffer color format (default is the framebuffer format)

NOTE

Alpha=255 is assumed "solid" and shall be used if HAL does not support BLIT_OP_COPY_WITH_ALPHA.

blitCopy

```
virtual void blitCopy ( const uint16_t * pSrc ,
                        uint16_t         x ,
                        uint16_t         y ,
                        uint16_t         width ,
                        uint16_t         height ,
                        uint16_t         srcWidth ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits a 2D source-array to the framebuffer performing alpha-blending as specified using the default lcd format.

Parameters:

pSrc	The source-array pointer (points to first value to copy)
x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.
height	The height of desired area of the source 2D array.
srcWidth	The distance (in elements) from first value of first line, to first value of second line (the source 2D array width)
alpha	The alpha value to use for blending (255 = solid, no blending)
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

NOTE

Alpha=255 is assumed "solid" and shall be used if HAL does not support BLIT_OP_COPY_WITH_ALPHA.

blitCopy

```
virtual void blitCopy ( const uint16_t *        pSrc ,
                        uint16_t                x ,
                        uint16_t                y ,
                        uint16_t                width ,
                        uint16_t                height ,
                        uint16_t                srcWidth ,
                        uint8_t                 alpha ,
                        bool                    hasTransparentPixels ,
                        uint16_t                dstWidth ,
                        Bitmap::BitmapFormat    srcFormat ,
                        Bitmap::BitmapFormat    dstFormat
                        )
```

Blits a 2D source-array to the framebuffer performing alpha-blending as specified.

Parameters:

pSrc	The source-array pointer (points to first value to copy)
x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.
height	The height of desired area of the source 2D array.
srcWidth	The distance (in elements) from first value of first line, to first value of second line (the source 2D array width)
alpha	The alpha value to use for blending (255 = solid, no blending)
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.
dstWidth	The distance (in elements) from first value of first line, to first value of second line (the destination 2D array width)
srcFormat	The source buffer color format (default is the framebuffer format)
dstFormat	The destination buffer color format (default is the framebuffer format)

NOTE

Alpha=255 is assumed "solid" and shall be used if HAL does not support BLIT_OP_COPY_WITH_ALPHA.

blitCopyARGB8888

```
virtual void blitCopyARGB8888 ( const uint16_t * pSrc ,
                                uint16_t         x ,
```

```
                                uint16_t        y ,
                                uint16_t        width ,
                                uint16_t        height ,
                                uint16_t        srcWidth ,
                                uint8_t         alpha
                                )
```

Blits a 2D source-array to the framebuffer performing per-pixel alpha blending.

Parameters:

pSrc	The source-array pointer (points to first value to copy)
x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.
height	The height of desired area of the source 2D array.
srcWidth	The distance (in elements) from first value of first line, to first value of second line (the source 2D array width)
alpha	The alpha value to use for blending. This is applied on every pixel, in addition to the per-pixel alpha value (255 = solid, no blending)

blitCopyGlyph

```
virtual void blitCopyGlyph ( const uint8_t * pSrc ,
                             uint16_t        x ,
                             uint16_t        y ,
                             uint16_t        width ,
                             uint16_t        height ,
                             uint16_t        srcWidth ,
                             colortype       color ,
                             uint8_t         alpha ,
                             BlitOperations  operation
                             )
```

Blits a 4bpp or 8bpp glyph - maybe use the same method and supply additional color mode arg.

Parameters:

pSrc	The source-array pointer (points to first value to copy)
x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.
height	The height of desired area of the source 2D array.
srcWidth	The distance (in elements) from first value of first line, to first value of second line (the source 2D array width)
color	**Color** of the text.
alpha	The alpha value to use for blending (255 = solid, no blending)

operation The operation type to use for blit copy.

blitFill

```
virtual void blitFill ( colortype color ,
                        uint16_t  x ,
                        uint16_t  y ,
                        uint16_t  width ,
                        uint16_t  height ,
                        uint8_t   alpha
                        )
```

Blits a color value to the framebuffer performing alpha-blending as specified.

Parameters:

color	The desired fill-color.
x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.
height	The height of desired area of the source 2D array.
alpha	The alpha value to use for blending (255 = solid, no blending)

NOTE

Alpha=255 is assumed "solid" and shall be used if HAL does not support BLIT_OP_FILL_WITH_ALPHA.

blitFill

```
virtual void blitFill ( colortype              color ,
                        uint16_t               x ,
                        uint16_t               y ,
                        uint16_t               width ,
                        uint16_t               height ,
                        uint8_t                alpha ,
                        uint16_t               dstWidth ,
                        Bitmap::BitmapFormat   dstFormat
                        )
```

Blits a color value to the framebuffer performing alpha-blending as specified.

Parameters:

color	The desired fill-color.

x	The destination x coordinate on the framebuffer.
y	The destination y coordinate on the framebuffer.
width	The width desired area of the source 2D array.
height	The height of desired area of the source 2D array.
alpha	The alpha value to use for blending (255 = solid, no blending)
dstWidth	The distance (in elements) from first value of first line, to first value of second line (the destination 2D array width)
dstFormat	The destination buffer color format (default is the framebuffer format)

NOTE

Alpha=255 is assumed "solid" and shall be used if **HAL** does not support BLIT_OP_FILL_WITH_ALPHA.

blockCopy

```
virtual bool blockCopy ( void *RESTRICT       dest ,
                         const void *RESTRICT src ,
                         uint32_t             numBytes
                         )
```

This function performs a platform-specific memcpy, if supported by the hardware.

Parameters:

dest	Pointer to destination memory.
src	Pointer to source memory.
numBytes	Number of bytes to copy.

Returns:

true if the copy succeeded, false if copy was not performed.

Reimplemented by: **touchgfx::HALSDL2::blockCopy**

configureInterrupts

```
virtual void configureInterrupts ( ) =0
```

Configures the interrupts relevant for TouchGFX.

This primarily entails setting the interrupt priorities for the DMA and **LCD** interrupts.

Reimplemented by: **touchgfx::HALSDL2::configureInterrupts**

configurePartialFrameBuffer

```
virtual uint16_t configurePartialFrameBuffer ( const uint16_t x ,
                                               const uint16_t y ,
                                               const uint16_t width ,
                                               const uint16_t height
                                               )
```

Configures a partial framebuffer as current framebuffer.

This method uses the assigned **FrameBufferAllocator** to allocate block of compatible dimensions. The height of the allocated block is returned.

Parameters:

x	The absolute x coordinate of the block on the screen.
y	The absolute y coordinate of the block on the screen.
width	The width of the block.
height	The height of the block requested.

Returns:

The height of the block allocated.

copyFBRegionToMemory

```
virtual uint16_t * copyFBRegionToMemory ( Rect meAbs )
```

Copies a region of the currently displayed framebuffer to memory.

Used for e.g. **SlideTransition** and for displaying pre-rendered drawables e.g. in animations where redrawing the drawable is not necessary.

Parameters:

meAbs	The framebuffer region to copy.

Returns:

A pointer to the memory address containing the copy of the framebuffer.

NOTE

Requires animation storage to be present.

copyFBRegionToMemory

```
virtual uint16_t * copyFBRegionToMemory ( Rect       meAbs ,
                                          uint16_t * dst ,
                                          uint32_t   stride
                                          )
```

Copies a region of the currently displayed framebuffer to a buffer.

Used for e.g. **SlideTransition** and for displaying pre-rendered drawables e.g. in animations where redrawing the drawable is not necessary. The buffer can e.g. be a dynamic bitmap.

Parameters:

meAbs	The framebuffer region to copy.
dst	Address of the buffer to store the copy in.
stride	The width of the target buffer (row length).

Returns:

A pointer to the memory address containing the copy of the framebuffer.

NOTE

Requires animation storage to be present.

disableInterrupts

```
virtual void disableInterrupts (  ) =0
```

Disables the DMA and LCD interrupts.

Reimplemented by: **touchgfx::HALSDL2::disableInterrupts**

drawDrawableInDynamicBitmap

```
virtual void drawDrawableInDynamicBitmap ( Drawable & drawable ,
                                           BitmapId   bitmapId
                                           )
```

Render a **Drawable** and its widgets into a dynamic bitmap.

Parameters:

drawable	A reference on the **Drawable** object to render.
bitmapId	Dynamic bitmap to be used as a rendertarget.

drawDrawableInDynamicBitmap

```
virtual void drawDrawableInDynamicBitmap ( Drawable &  drawable ,
                                           BitmapId    bitmapId ,
                                           const Rect & rect
                                           )
```

Render a **Drawable** and its widgets into a dynamic bitmap.

Only the specified **Rect** region is updated.

Parameters:

drawable	A reference on the **Drawable** object to render.
bitmapId	Dynamic bitmap to be used as a rendertarget.
rect	Region to update.

enableDMAAcceleration

```
void enableDMAAcceleration ( const bool enable )
```

Sets a flag to allow use of DMA operations to speed up drawing operations.

Parameters:

enable	True to enable, false to disable.

See also:

getBlitCaps

enableInterrupts

```
virtual void enableInterrupts (  ) =0
```

Enables the DMA and LCD interrupts.

Reimplemented by: **touchgfx::HALSDL2::enableInterrupts**

enableLCDControllerInterrupt

```
virtual void enableLCDControllerInterrupt (  ) =0
```

Configure the LCD controller to fire interrupts at VSYNC.

Called automatically once TouchGFX initialization has completed.

Reimplemented by: **touchgfx::HALSDL2::enableLCDControllerInterrupt**

enableMCULoadCalculation

void enableMCULoadCalculation (bool enabled)

This method sets a flag that determines if generic **HAL** should calculate MCU load based on concrete MCU instrumentation.

Parameters:

enabled If true, set flag to update MCU load.

flushDMA

virtual void flushDMA ( )

This function blocks until the DMA queue (containing BlitOps) is empty.

flushFrameBuffer

virtual void flushFrameBuffer ( )

This function is called whenever the framework has performed a complete draw.

On some platforms, a local framebuffer needs to be pushed to the display through a SPI channel or similar. Implement that functionality here. This function is called whenever the framework has performed a complete draw.

Reimplemented by: **touchgfx::HALSDL2::flushFrameBuffer**

flushFrameBuffer

virtual void flushFrameBuffer (const Rect & rect)

This function is called whenever the framework has performed a partial draw.

Parameters:

rect The area of the screen that has been drawn, expressed in absolute coordinates.

See also:

flushFrameBuffer

Reimplemented by: **touchgfx::HALSDL2::flushFrameBuffer**

frontPorchEntered

void frontPorchEntered ( )

Has to be called from within the LCD IRQ routine when the Front Porch Entry is reached.

getAnimationStorage

uint16_t * getAnimationStorage ( ) const

Gets the optional framebuffer used for animation storage.

Returns:

The address or 0 if unused.

getAuxiliaryLCD

LCD * getAuxiliaryLCD ( )

Get the auxiliary LCD class attached to the **HAL** instance if any.

Returns:

A pointer on the axiliary **LCD** class attached to the **HAL** instance.

getBlitCaps

virtual BlitOperations getBlitCaps ( )

Function for obtaining the blit capabilities of the concrete **HAL** implementation.

As default, will return whatever blitcaps are reported by the associated DMA object.

DMA operations can be disabled by calling **enableDMAAcceleration(bool)**.

Returns:

a bitmask of the supported blitcaps.

See also:

enableDMAAcceleration

getButtonController

ButtonController * getButtonController ( ) const

Gets the associated **ButtonController**.

Returns:

A pointer to the **ButtonController**, or zero if no **ButtonController** has been set.

getCPUCycles

uint32_t getCPUCycles ( )

Gets the current cycle counter.

Returns:

the cycle counter.

getDisplayHeight

uint16_t getDisplayHeight ( ) const

Gets display height.

Returns:

The display height.

getDisplayOrientation

DisplayOrientation getDisplayOrientation ( ) const

Gets the current display orientation.

Will be equal to the native orientation of the display unless setDisplayOrientation has been explicitly called earlier.

Returns:

The current display orientation.

getDisplayWidth

uint16_t getDisplayWidth ( ) const

Gets display width.

Returns:

The display width.

getDMAType

virtual DMAType getDMAType ( )

Function for obtaining the DMA type of the concrete DMA implementation.

As default, will return DMA_TYPE_GENERIC type value.

Returns:

a DMAType value of the concrete DMA implementation.

getFingerSize

uint8_t getFingerSize ( ) const

Gets the finger size in pixels.

Returns:

The size of the finger in pixels, 1 is the default value.

getFlashDataReader

virtual FlashDataReader * getFlashDataReader ( ) const

Gets the flash data reader.

This method must be implemented in subclasses that uses a **FlashDataReader** object.

Returns:

the **FlashDataReader**.

getFrameBufferAllocator

FrameBufferAllocator * getFrameBufferAllocator ( )

Gets the framebuffer allocator.

Returns:

The framebuffer allocator.

getFrameRefreshStrategy

FrameRefreshStrategy getFrameRefreshStrategy ( ) const

Used internally by TouchGFX core to manage the timing and process of drawing into the framebuffer.

Returns:

Current frame refresh strategy.

See also:

setFrameRefreshStrategy

getGestures

Gestures * getGestures ( )

Get the Gesture class attached to the **HAL** instance.

Returns:

A pointer to the **Gestures** object.

getLCDRefreshCount

uint32_t getLCDRefreshCount ( )

Returns the number of VSync interrupts between the current drawing operation and the last drawing operation, i.e.

the number of lost frames.

Returns:

Number of VSync since previous draw.

getMCULoadPct

uint8_t getMCULoadPct ( ) const

Gets the current MCU load.

Returns:

mcuLoadPct the MCU Load in %.

getTFTCurrentLine

virtual uint16_t getTFTCurrentLine ( )

Get the current line (Y) of the TFT controller.

This function is used to obtain the progress of the TFT controller. More specifically, the line (or Y-value) currently being transferred.

Note: The value must be adjusted to account for vertical back porch before returning, such that the value is always within the range of [0; actual display height in pixels[

It is used for the REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL frame refresh strategy in order to synchronize framebuffer drawing with TFT controller progress. If this strategy is used, the concrete **HAL** subclass must provide an override of this function that returns correct line value. If this strategy is not used, then the getTFTCurrentLine function is never called and can be disregarded.

Returns:

In this default implementation, 0xFFFF is returned to signify "not implemented".

getTFTFrameBuffer

virtual uint16_t * getTFTFrameBuffer ( ) const =0

Gets the framebuffer address used by the TFT controller.

Returns:

The address of the framebuffer currently being displayed on the TFT.

Reimplemented by: **touchgfx::HALSDL2::getTFTFrameBuffer**

getTouchSampleRate

int8_t getTouchSampleRate ( ) const

Gets the number of ticks between each touch screen sample.

Returns:

Number of ticks between each touch screen sample.

HAL

```
HAL ( DMA_Interface &  dmaInterface ,
      LCD &            display ,
      TouchController & touchCtrl ,
      uint16_t         width ,
      uint16_t         height
      )
```

Initializes a new instance of the **HAL** class.

Parameters:

dmaInterface	Reference to the DMA interface.
display	Reference to the **LCD**.
touchCtrl	Reference to the touch controller.
width	The width of the **LCD** display, in pixels.
height	The height of the **LCD** display, in pixels.

initialize

void initialize ( )

This function is responsible for initializing the entire framework.

lockDMAToFrontPorch

void lockDMAToFrontPorch (bool enableLock)

Function to set whether the DMA transfers are locked to the TFT update cycle.

If locked, DMA transfer will not begin until the TFT controller has finished updating the display. If not locked, DMA transfers will begin as soon as possible. Default is true (DMA is locked with TFT).

Disabling the lock will in most cases significantly increase rendering performance. It is therefore strongly recommended to disable it. Depending on platform this may in rare cases cause rendering

problems (visible tearing on display). Please see the chapter "Optimizing DMA During TFT Controller Access" for details on this setting.

Parameters:

enableLock True to lock DMA transfers to the front porch signal. Conservative, default setting. False to disable, which will normally yield substantial performance improvement.

NOTE

This setting only has effect when using double buffering.

lockFrameBuffer

virtual uint16_t * lockFrameBuffer ( )

Waits for the framebuffer to become available for use (i.e.

not used by DMA transfers).

Returns:

A pointer to the beginning of the currently used framebuffer.

NOTE

Function blocks until framebuffer is available. Client code MUST call unlockFrameBuffer() when framebuffer operation has completed.

registerEventListener

virtual void registerEventListener (UIEventListener & listener)

Registers an event handler implementation with the underlying event system.

The actual **HAL** implementation decides whether or not multiple **UIEventListener** instances are allowed (including execution order).

Parameters:

listener The listener to register.

registerTaskDelayFunction

void registerTaskDelayFunction (void(*)(uint16_t) delayF)

Registers a function capable of delaying GUI task execution.

In order to make use of the **HAL::taskDelay** function, a delay function must be registered by calling this function. Usually the delay function would be **OSWrappers::taskDelay**.

Parameters:

delayF A pointer to a function returning void with an uint16_t parameter specifying number of milliseconds to delay.

NOTE

The task delay capability is only used when the frame refresh strategy REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL is selected. Otherwise it is not necessary to register a delay function.

sampleKey

virtual bool sampleKey (uint8_t & key)

Sample external key event.

Parameters:

key Output parameter that will be set to the key value if a keypress was detected.

Returns:

True if a keypress was detected and the "key" parameter is set to a value.

Reimplemented by: **touchgfx::HALSDL2::sampleKey**

setAuxiliaryLCD

void setAuxiliaryLCD (LCD * auxLCD)

Set an auxiliary LCD class to be used for offscreen rendering.

Parameters:

auxLCD A pointer on the axiliary **LCD** class to use for offscreen rendering.

setButtonController

void setButtonController (ButtonController * btnCtrl)

Stores a pointer to an instance of a specific implementation of a **ButtonController**.

Parameters:

btnCtrl pointer to button controller.

setDisplayOrientation

virtual void setDisplayOrientation (DisplayOrientation orientation)

Sets the desired display orientation (landscape or portrait).

If desired orientation is different from the native orientation of the display, a rotation is automatically applied. The rotation does not incur any performance cost.

Parameters:

orientation The desired display orientation.

NOTE

A screen transition must occur before this takes effect!

setDragThreshold

void setDragThreshold (uint8_t value)

Configure the threshold for reporting drag events.

A touch input movement must exceed this value in either axis in order to report a drag. Default value is 0.

Parameters:

value New threshold value.

NOTE

Use if touch controller is not completely accurate to avoid "false" drags.

setFingerSize

void setFingerSize (uint8_t size)

Sets the finger size in pixels.

Setting the finger size to a size of more than 1 pixel will emulate a finger of width and height of 2*(fingersize-1)+1. This can be especially useful when trying to interact with small elements on a high ppi display. The finger size will influence which element is chosen as the point of interaction, when clicking, dragging, ... the display. A number of samples will be drawn from within the finger area and a best matching drawable will be chosen. The best matching algorithm will consider the size of the drawable and the distance from the touch point.

Parameters:

size	the size of the finger.

setFrameBufferAllocator

```
void setFrameBufferAllocator ( FrameBufferAllocator * allocator )
```

Sets a framebuffer allocator.

The framebuffer allocator is only used in partial framebuffer mode.

Parameters:

allocator	pointer to a framebuffer allocator object.

setFrameBufferStartAddresses

```
virtual void setFrameBufferStartAddresses ( void * frameBuffer ,
                                            void * doubleBuffer ,
                                            void * animationStorage
                                            )
```

Sets framebuffer start addresses.

Sets individual framebuffer start addresses.

Parameters:

frameBuffer	Buffer for framebuffer data, must be non-null.
doubleBuffer	If non-null, buffer for double buffer data. If null double buffering is disabled.
animationStorage	If non-null, the animation storage. If null animation storage is disabled.

setFrameRateCompensation

void setFrameRateCompensation (bool enabled)

Enables or disables compensation for lost frames.

See knowledge base article.

Parameters:

enabled true to enable, false to disable.

setFrameRefreshStrategy

bool setFrameRefreshStrategy (FrameRefreshStrategy s)

Set a specific strategy for handling timing and mechanism of framebuffer drawing.

By setting a different frame refresh strategy, the internals of how TouchGFX interacts with the framebuffer can be modified.

Currently there are two strategies available. This will increase over time.

- REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL: this strategy is available on targets that use single buffering on a TFT controller based system. It requires an implementation of the **getTFTCurrentLine()** function as well as a task delay function being registered. The implementation of this strategy is that TouchGFX will carefully track the progress of the TFT controller, and draw parts of the framebuffer whenever possible. The effect is that the risk of tearing is much reduced compared to the default single buffer strategy of only drawing in porch areas. It does have a drawback of slightly increased MCU load. But in many cases employing this strategy will make it possible to avoid external RAM, by using just a single framebuffer in internal RAM and still avoid tearing.
- REFRESH_STRATEGY_DEFAULT: This is a general strategy that works for all target configurations. Recommendation: Try using REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL if you're on a TFT controller based system (ie. non-8080) and you have a desire to avoid external RAM. Otherwise stick to REFRESH_STRATEGY_DEFAULT.

Parameters:

s The desired strategy to use.

Returns:

true if the desired strategy will be used, false otherwise.

setMCUActive

void setMCUActive (bool active)

Register if MCU is active by measuring cpu cycles.

If user wishes to track MCU load, this method should be called whenever the OS Idle task is scheduled in or out. This method makes calls to a concrete implementation of **GPIO** functionality and a concrete implementation of cpu cycles.

Parameters:

active If true, MCU is registered as being active, inactive otherwise.

setMCUInstrumentation

void setMCUInstrumentation (MCUInstrumentation * mcuInstr)

Stores a pointer to an instance of an MCU specific instrumentation class.

Parameters:

mcuInstr pointer to MCU instrumentation.

setRenderingVariant

void setRenderingVariant (RenderingVariant variant)

Set current rendering variant for cache maintenance.

This function is used to keep track of previous rendering variant and will determine if cache should be flush or invalidated depending on transition state.

Parameters:

variant The rendering variant used.

setTouchSampleRate

void setTouchSampleRate (int8_t sampleRateInTicks)

Sets the number of ticks between each touch screen sample.

Parameters:

sampleRateInTicks Sample rate. Default is 1 (every tick).

signalDMAInterrupt

void signalDMAInterrupt ( )

Notify the framework that a DMA interrupt has occurred.

swapFrameBuffers

void swapFrameBuffers ( )

Swaps the two framebuffers.

taskDelay

virtual void taskDelay (uint16_t ms)

Delay GUI task execution by number of milliseconds.

This function requires the presence of a task delay function. If a task delay function has not been registered, it returns immediately. Otherwise it returns when number of milliseconds has passed.

Parameters:

ms Number of milliseconds to wait.

See also:

registerTaskDelayFunction

taskEntry

virtual void taskEntry ( )

Main event loop.

Will wait for VSYNC signal, and then process next frame. Call this function from your GUI task.

NOTE

This function never returns!

Reimplemented by: **touchgfx::HALSDL2::taskEntry**

unlockFrameBuffer

virtual void unlockFrameBuffer ( )

Unlocks the framebuffer (MUST be called exactly once for each call to **lockFrameBuffer()**).

vSync

void vSync ( )

Called by the VSync interrupt.

Called by the VSync interrupt for counting of **LCD** refreshes.

~HAL

virtual ~HAL ( )

Finalizes an instance of the **HAL** class.

getInstance

static HAL * getInstance ( )

Gets the **HAL** instance.

Returns:

The **HAL** instance.

lcd

static LCD & lcd ( )

Gets a reference to the LCD.

Returns:

A reference to the **LCD**.

Protected Functions Documentation

beginFrame

virtual bool beginFrame ( )

Called when beginning to rendering a frame.

Returns:

true if rendering can begin, false otherwise.

endFrame

virtual void endFrame ( )

Called when a rendering pass is completed.

FlushCache

virtual void FlushCache ( )

Flush D-Cache.

Called by setRenderingVariant when chaning rendering variant from Software to Hardware indicating the cache should be invalidated.

getClientFrameBuffer

uint16_t * getClientFrameBuffer ( )

Gets client framebuffer.

Returns:

The address of the framebuffer currently used by the framework to draw in.

InvalidateCache

virtual void InvalidateCache ( )

Invalidate D-Cache.

Called by setRenderingVariant when chaning rendering variant from Hardware to Software indicating the cache should be invalidated.

noTouch

virtual void noTouch ( )

Called by the touch driver to indicate that no touch is currently detected.

performDisplayOrientationChange

virtual void performDisplayOrientationChange ( )

Perform the actual display orientation change.

Reimplemented by: **touchgfx::HALSDL2::performDisplayOrientationChange**

setTFTFrameBuffer

virtual void setTFTFrameBuffer (uint16_t * address)

Sets the framebuffer address used by the TFT controller.

Parameters:

address New framebuffer address.

Reimplemented by: **touchgfx::HALSDL2::setTFTFrameBuffer**

tick

virtual void tick ( )

This function is called at each timer tick, depending on platform implementation.

touch

virtual void touch (int32_t x ,
int32_t y
)

Called by the touch driver to indicate a touch.

Parameters:

x The x coordinate of the touch.
y The y coordinate of the touch.

Public Attributes Documentation

DISPLAY_HEIGHT

uint16_t DISPLAY_HEIGHT

The height of the LCD display in pixels.

DISPLAY_ROTATION

DisplayRotation DISPLAY_ROTATION

The rotation from display to framebuffer.

DISPLAY_WIDTH

uint16_t DISPLAY_WIDTH

The width of the LCD display in pixels.

FRAME_BUFFER_HEIGHT

uint16_t FRAME_BUFFER_HEIGHT

The height of the framebuffer in pixels.

FRAME_BUFFER_WIDTH

uint16_t FRAME_BUFFER_WIDTH

The width of the framebuffer in pixels.

USE_ANIMATION_STORAGE

bool USE_ANIMATION_STORAGE

Is animation storage enabled?

USE_DOUBLE_BUFFERING

bool USE_DOUBLE_BUFFERING

Is double buffering enabled?

Protected Attributes Documentation

auxiliaryLCD

LCD * auxiliaryLCD

Auxiliary LCD class used to render Drawables into dynamic bitmaps.

buttonController

ButtonController * buttonController

A reference to an optional **ButtonController**.

dma

DMA_Interface & dma

A reference to the DMA interface.

fingerSize

uint8_t fingerSize

The radius of the finger in pixels.

frameBuffer0

uint16_t * frameBuffer0

Pointer to the first framebuffer.

frameBuffer1

uint16_t * frameBuffer1

Pointer to the second framebuffer.

frameBuffer2

uint16_t * frameBuffer2

Pointer to the optional third framebuffer used for animation storage.

frameBufferAllocator

FrameBufferAllocator * frameBufferAllocator

A reference to an optional **FrameBufferAllocator**.

frameBufferUpdatedThisFrame

bool frameBufferUpdatedThisFrame

True if something was drawn in the current frame.

gestures

Gestures gestures

Class for low-level interpretation of touch events.

lcdRef

LCD & lcdRef

A reference to the LCD.

lockDMAToPorch

bool lockDMAToPorch

Whether or not to lock DMA transfers with TFT porch signal.

mcuInstrumentation

MCUInstrumentation * mcuInstrumentation

A reference to an optional MCU instrumentation.

nativeDisplayOrientation

DisplayOrientation nativeDisplayOrientation

Contains the native display orientation. If desired orientation is different, apply rotation.

partialFrameBufferRect

Rect partialFrameBufferRect

The region of the screen covered by the partial framebuffer.

refreshStrategy

FrameRefreshStrategy refreshStrategy

The selected display refresh strategy.

taskDelayFunc

void(* taskDelayFunc

Pointer to a function that can delay GUI task for a number of milliseconds.

touchController

TouchController & touchController

A reference to the touch controller.

isDrawing

bool isDrawing

True if currently in the process of rendering a screen.

Version: 4.16

HALSDL2

HAL implementation for the TouchGFX simulator. This particular simulator HAL implementation uses SDL2 to show the content of the framebuffer in a window.

See: HAL

Inherits from: HAL

Public Functions

virtual bool	**blockCopy**(void *RESTRICT dest, const void* RESTRICT src, uint32_t numBytes)
	This function performs a platform-specific memcpy, if supported by the hardware.
virtual void	**copyScreenshotToClipboard**()
	Copies the screenshot to clipboard.
bool	**doSampleTouch**(int32_t & x, int32_t & y) const
	Samples the position of the mouse cursor.
virtual void	**flushFrameBuffer**()
	This function is called whenever the framework has performed a complete draw.
virtual void	**flushFrameBuffer**(const **Rect** & rect)
	This function is called whenever the framework has performed a partial draw.
bool	**getConsoleVisible**() const
	Is console window visible?
bool	**getWindowVisible**() const
	Is the window visible?
	HALSDL2(**DMA_Interface** & dma, **LCD** & lcd, **TouchController** & touchCtrl, uint16_t width, uint16_t height)
	Initializes a new instance of the **HALSDL2** class.
void	**loadSkin**(**DisplayOrientation** orientation, int x, int y)
	Loads a skin for a given display orientation that will be rendered in the simulator window with the the TouchGFX framebuffer placed inside the bitmap at the given coordinates.

virtual bool	**sampleKey**(uint8_t & key)
	Sample key event from keyboard.
virtual void	**saveNextScreenshots**(int n)
	Copy the next N screenshots to disk.
void	**saveScreenshot**()
	Saves a screenshot to the default folder and default filename.
virtual void	**saveScreenshot**(char *folder, char* filename)
	Saves a screenshot.
virtual bool	**sdl_init**(int argcount, char ** args)
	Initializes SDL.
void	**setConsoleVisible**(bool visible, bool redrawWindow =true)
	Change visibility of console window (hidden vs.
void	**setVsyncInterval**(float ms)
	Sets vsync interval for simulating same tick speed as the real hardware.
void	**setWindowVisible**(bool visible, bool redrawWindow =true)
	Change visibility of window (hidden vs.
virtual void	**taskEntry**()
	Main event loop.
uint8_t *	**doRotate**(uint8_t *dst, uint8_t* src)
	Rotates a framebuffer if the display is rotated.
char **	**getArgv**(int * argc)
	Gets the argc and argv for a Windows program.
const char *	**getWindowTitle**()
	Gets window title.
bool	**isSingleStepping**()
	Is single stepping.
uint8_t *	**scaleTo24bpp**(uint8_t *dst, uint16_t* src, **Bitmap::BitmapFormat** format)
	Scale framebuffer to 24bpp.
void	**setSingleStepping**(bool singleStepping =true)

	Single stepping enable/disable.
void	**setWindowTitle**(const char * title)
	Sets window title.
void	**singleStep**(uint16_t steps =1)
	Single step a number of steps.

Protected Functions

virtual void	**configureInterrupts**()
	Configures the interrupts relevant for TouchGFX.
virtual void	**configureLCDInterrupt**()
	Configures LCD interrupt.
virtual void	**disableInterrupts**()
	Disables the DMA and LCD interrupts.
virtual void	**enableInterrupts**()
	Enables the DMA and LCD interrupts.
virtual void	**enableLCDControllerInterrupt**()
	Enables the LCD interrupt.
virtual uint16_t *	**getTFTFrameBuffer**() const
	Gets TFT framebuffer.
virtual void	**performDisplayOrientationChange**()
	Perform the actual display orientation change.
virtual void	**renderLCD_FrameBufferToMemory**(const **Rect** & _rectToUpdate, uint8_t * frameBuffer)
	Update framebuffer using an SDL Surface.
virtual void	**setTFTFrameBuffer**(uint16_t * addr)
	Sets TFT framebuffer.

Additional inherited members

Public Types inherited from HAL

enum	**FrameRefreshStrategy** { REFRESH_STRATEGY_DEFAULT, REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL, REFRESH_STRATEGY_PARTIAL_FRAMEBUFFER }
	A list of available frame refresh strategies.

enum	**RenderingVariant** { SOFTWARE, HARDWARE }
	A list of rendering variants.

Public Functions inherited from HAL

virtual void	**allowDMATransfers**()
	Allow the DMA to start transfers.

virtual void	**backPorchExited**()
	Has to be called from within the LCD IRQ rutine when the Back Porch Exit is reached.

void	**blitCopy**(const uint16_t *pSrc, const uint8_t* pClut, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha, bool hasTransparentPixels, uint16_t dstWidth, **Bitmap::BitmapFormat** srcFormat, **Bitmap::BitmapFormat** dstFormat)
	Blits a 2D source-array to the framebuffer performing alpha-blending as specified.

virtual void	**blitCopy**(const uint16_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha, bool hasTransparentPixels)
	Blits a 2D source-array to the framebuffer performing alpha-blending as specified using the default lcd format.

virtual void	**blitCopy**(const uint16_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha, bool hasTransparentPixels, uint16_t dstWidth, **Bitmap::BitmapFormat** srcFormat, **Bitmap::BitmapFormat** dstFormat)
	Blits a 2D source-array to the framebuffer performing alpha-blending as specified.

virtual void	**blitCopyARGB8888**(const uint16_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing per-pixel alpha blending.

virtual void	**blitCopyGlyph**(const uint8_t * pSrc, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t srcWidth, **colortype** color, uint8_t alpha, **BlitOperations** operation)
	Blits a 4bpp or 8bpp glyph - maybe use the same method and supply additional color mode arg.
virtual void	**blitFill**(**colortype** color, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t alpha)
	Blits a color value to the framebuffer performing alpha-blending as specified.
virtual void	**blitFill**(**colortype** color, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t alpha, uint16_t dstWidth, **Bitmap::BitmapFormat** dstFormat)
	Blits a color value to the framebuffer performing alpha-blending as specified.
virtual uint16_t	**configurePartialFrameBuffer**(const uint16_t x, const uint16_t y, const uint16_t width, const uint16_t height)
	Configures a partial framebuffer as current framebuffer.
virtual uint16_t *	**copyFBRegionToMemory**(**Rect** meAbs)
	Copies a region of the currently displayed framebuffer to memory.
virtual uint16_t *	**copyFBRegionToMemory**(**Rect** meAbs, uint16_t * dst, uint32_t stride)
	Copies a region of the currently displayed framebuffer to a buffer.
virtual void	**drawDrawableInDynamicBitmap**(**Drawable** & drawable, **BitmapId** bitmapId)
	Render a **Drawable** and its widgets into a dynamic bitmap.
virtual void	**drawDrawableInDynamicBitmap**(**Drawable** & drawable, **BitmapId** bitmapId, const **Rect** & rect)
	Render a **Drawable** and its widgets into a dynamic bitmap.
void	**enableDMAAcceleration**(const bool enable)
	Sets a flag to allow use of DMA operations to speed up drawing operations.
void	**enableMCULoadCalculation**(bool enabled)
	This method sets a flag that determines if generic **HAL** should calculate MCU load based on concrete MCU instrumentation.
virtual void	**flushDMA**()
	This function blocks until the DMA queue (containing BlitOps) is empty.

void	**frontPorchEntered**()
	Has to be called from within the LCD IRQ routine when the Front Porch Entry is reached.
uint16_t *	**getAnimationStorage**() const
	Gets the optional framebuffer used for animation storage.
LCD *	**getAuxiliaryLCD**()
	Get the auxiliary LCD class attached to the **HAL** instance if any.
virtual **BlitOperations**	**getBlitCaps**()
	Function for obtaining the blit capabilities of the concrete **HAL** implementation.
ButtonController *	**getButtonController**() const
	Gets the associated **ButtonController**.
uint32_t	**getCPUCycles**()
	Gets the current cycle counter.
uint16_t	**getDisplayHeight**() const
	Gets display height.
DisplayOrientation	**getDisplayOrientation**() const
	Gets the current display orientation.
uint16_t	**getDisplayWidth**() const
	Gets display width.
virtual **DMAType**	**getDMAType**()
	Function for obtaining the DMA type of the concrete DMA implementation.
uint8_t	**getFingerSize**() const
	Gets the finger size in pixels.
virtual **FlashDataReader** *	**getFlashDataReader**() const
	Gets the flash data reader.
FrameBufferAllocator *	**getFrameBufferAllocator**()
	Gets the framebuffer allocator.
FrameRefreshStrategy	**getFrameRefreshStrategy**() const

	Used internally by TouchGFX core to manage the timing and process of drawing into the framebuffer.
Gestures *	**getGestures**()
	Get the Gesture class attached to the **HAL** instance.
uint32_t	**getLCDRefreshCount**()
	Returns the number of VSync interrupts between the current drawing operation and the last drawing operation, i.e.
uint8_t	**getMCULoadPct**() const
	Gets the current MCU load.
virtual uint16_t	**getTFTCurrentLine**()
	Get the current line (Y) of the TFT controller.
int8_t	**getTouchSampleRate**() const
	Gets the number of ticks between each touch screen sample.
	HAL(**DMA_Interface** & dmaInterface, **LCD** & display, **TouchController** & touchCtrl, uint16_t width, uint16_t height)
	Initializes a new instance of the **HAL** class.
void	**initialize**()
	This function is responsible for initializing the entire framework.
void	**lockDMAToFrontPorch**(bool enableLock)
	Function to set whether the DMA transfers are locked to the TFT update cycle.
virtual uint16_t *	**lockFrameBuffer**()
	Waits for the framebuffer to become available for use (i.e.
virtual void	**registerEventListener**(**UIEventListener** & listener)
	Registers an event handler implementation with the underlying event system.
void	**registerTaskDelayFunction**(void(*)(uint16_t) delayF)
	Registers a function capable of delaying GUI task execution.
void	**setAuxiliaryLCD**(**LCD** * auxLCD)
	Set an auxiliary LCD class to be used for offscreen rendering.
void	**setButtonController**(**ButtonController** * btnCtrl)

	Stores a pointer to an instance of a specific implementation of a **ButtonController**.
virtual void	**setDisplayOrientation**(**DisplayOrientation** orientation)
	Sets the desired display orientation (landscape or portrait).
void	**setDragThreshold**(uint8_t value)
	Configure the threshold for reporting drag events.
void	**setFingerSize**(uint8_t size)
	Sets the finger size in pixels.
void	**setFrameBufferAllocator**(**FrameBufferAllocator** * allocator)
	Sets a framebuffer allocator.
virtual void	**setFrameBufferStartAddresses**(void *frameBuffer, void* doubleBuffer, void * animationStorage)
	Sets framebuffer start addresses.
void	**setFrameRateCompensation**(bool enabled)
	Enables or disables compensation for lost frames.
bool	**setFrameRefreshStrategy**(**FrameRefreshStrategy** s)
	Set a specific strategy for handling timing and mechanism of framebuffer drawing.
void	**setMCUActive**(bool active)
	Register if MCU is active by measuring cpu cycles.
void	**setMCUInstrumentation**(**MCUInstrumentation** * mcuInstr)
	Stores a pointer to an instance of an MCU specific instrumentation class.
void	**setRenderingVariant**(**RenderingVariant** variant)
	Set current rendering variant for cache maintenance.
void	**setTouchSampleRate**(int8_t sampleRateInTicks)
	Sets the number of ticks between each touch screen sample.
void	**signalDMAInterrupt**()
	Notify the framework that a DMA interrupt has occurred.
void	**swapFrameBuffers**()
	Swaps the two framebuffers.

virtual void	**taskDelay**(uint16_t ms)
	Delay GUI task execution by number of milliseconds.
virtual void	**unlockFrameBuffer**()
	Unlocks the framebuffer (MUST be called exactly once for each call to **lockFrameBuffer()**).
void	**vSync**()
	Called by the VSync interrupt.
virtual	**~HAL**()
	Finalizes an instance of the **HAL** class.
HAL *	**getInstance**()
	Gets the **HAL** instance.
LCD &	**lcd**()
	Gets a reference to the LCD.

Protected Functions inherited from HAL

virtual bool	**beginFrame**()
	Called when beginning to rendering a frame.
virtual void	**endFrame**()
	Called when a rendering pass is completed.
virtual void	**FlushCache**()
	Flush D-Cache.
uint16_t *	**getClientFrameBuffer**()
	Gets client framebuffer.
virtual void	**InvalidateCache**()
	Invalidate D-Cache.
virtual void	**noTouch**()
	Called by the touch driver to indicate that no touch is currently detected.
virtual void	**tick**()
	This function is called at each timer tick, depending on platform implementation.

virtual void	**touch**(int32_t x, int32_t y)
	Called by the touch driver to indicate a touch.

Public Attributes inherited from HAL

uint16_t	**DISPLAY_HEIGHT**
	The height of the LCD display in pixels.
DisplayRotation	**DISPLAY_ROTATION**
	The rotation from display to framebuffer.
uint16_t	**DISPLAY_WIDTH**
	The width of the LCD display in pixels.
uint16_t	**FRAME_BUFFER_HEIGHT**
	The height of the framebuffer in pixels.
uint16_t	**FRAME_BUFFER_WIDTH**
	The width of the framebuffer in pixels.
bool	**USE_ANIMATION_STORAGE**
	Is animation storage enabled?
bool	**USE_DOUBLE_BUFFERING**
	Is double buffering enabled?

Protected Attributes inherited from HAL

LCD *	**auxiliaryLCD**
	Auxiliary LCD class used to render Drawables into dynamic bitmaps.
ButtonController *	**buttonController**
	A reference to an optional **ButtonController**.
DMA_Interface &	**dma**
	A reference to the DMA interface.
uint8_t	**fingerSize**
	The radius of the finger in pixels.
uint16_t *	**frameBuffer0**

Pointer to the first framebuffer.

uint16_t * **frameBuffer1**

Pointer to the second framebuffer.

uint16_t * **frameBuffer2**

Pointer to the optional third framebuffer used for animation storage.

FrameBufferAllocator * **frameBufferAllocator**

A reference to an optional **FrameBufferAllocator**.

bool **frameBufferUpdatedThisFrame**

True if something was drawn in the current frame.

Gestures **gestures**

Class for low-level interpretation of touch events.

LCD & **lcdRef**

A reference to the LCD.

bool **lockDMAToPorch**

Whether or not to lock DMA transfers with TFT porch signal.

MCUInstrumentation * **mcuInstrumentation**

A reference to an optional MCU instrumentation.

DisplayOrientation **nativeDisplayOrientation**

Contains the native display orientation. If desired orientation is different, apply rotation.

Rect **partialFrameBufferRect**

The region of the screen covered by the partial framebuffer.

FrameRefreshStrategy **refreshStrategy**

The selected display refresh strategy.

void(* **taskDelayFunc**

Pointer to a function that can delay GUI task for a number of milliseconds.

TouchController & **touchController**

A reference to the touch controller.

bool **isDrawing**

True if currently in the process of rendering a screen.

Public Functions Documentation

blockCopy

```
virtual bool blockCopy ( void *RESTRICT       dest ,
                         const void *RESTRICT src ,
                         uint32_t             numBytes
                         )
```

This function performs a platform-specific memcpy, if supported by the hardware.

Parameters:

dest	Pointer to destination memory.
src	Pointer to source memory.
numBytes	Number of bytes to copy.

Returns:

true if the copy succeeded, false if copy was not performed.

Reimplements: **touchgfx::HAL::blockCopy**

copyScreenshotToClipboard

```
virtual void copyScreenshotToClipboard (  )
```

Copies the screenshot to clipboard.

doSampleTouch

```
bool doSampleTouch ( int32_t & x ,       const
                     int32_t & y       const
                     )           const
```

Samples the position of the mouse cursor.

Parameters:

x	The x coordinate.
y	The y coordinate.

Returns:

True if touch detected, false otherwise.

flushFrameBuffer

virtual void flushFrameBuffer ( )

This function is called whenever the framework has performed a complete draw.

On some platforms, a local framebuffer needs to be pushed to the display through a SPI channel or similar. Implement that functionality here. This function is called whenever the framework has performed a complete draw.

Reimplements: **touchgfx::HAL::flushFrameBuffer**

flushFrameBuffer

virtual void flushFrameBuffer (const Rect & rect)

This function is called whenever the framework has performed a partial draw.

Parameters:

rect The area of the screen that has been drawn, expressed in absolute coordinates.

Reimplements: **touchgfx::HAL::flushFrameBuffer**

getConsoleVisible

bool getConsoleVisible ( ) const

Is console window visible?

Returns:

True if it is visible, false if it is hidden.

See also:

setConsoleVisible, **getWindowVisible**

getWindowVisible

bool getWindowVisible ( ) const

Is the window visible?

Returns:

True if it is visible, false if it is hidden.

See also:

setWindowVisible, **getConsoleVisible**

HALSDL2

```
HALSDL2 ( DMA_Interface &  dma ,
          LCD &            lcd ,
          TouchController & touchCtrl ,
          uint16_t         width ,
          uint16_t         height
          )
```

Initializes a new instance of the **HALSDL2** class.

Parameters:

dma	Reference to DMA interface.
lcd	Reference to the **LCD**.
touchCtrl	Reference to Touch Controller driver.
width	Width of the display.
height	Height of the display.

loadSkin

```
void loadSkin ( DisplayOrientation orientation ,
                int                x ,
                int                y
                )
```

Loads a skin for a given display orientation that will be rendered in the simulator window with the the TouchGFX framebuffer placed inside the bitmap at the given coordinates.

Different bitmaps can be loaded in landscape and portrait mode. If the provided bitmap cannot be loaded, the TouchGFX framebuffer will be displayed as normal. If the png files contain areas with alpha < 255, this will be used to create a shaped window.

Parameters:

orientation	The orientation.
x	The x coordinate.
y	The y coordinate.

NOTE

The skins must be named "portrait.png" and "landscape.png" and placed inside the "simulator/" folder. The build process of the simulator will automatically copy the skins to the folder where the executable simulator is generated. When as skin is set, the entire framebuffer is rendered through SDL whenever there is a change. Without a skin, only the areas with changes is rendered through SDL.

sampleKey

virtual bool sampleKey (uint8_t & key)

Sample key event from keyboard.

Parameters:

key	Output parameter that will be set to the key value if a key press was detected.

Returns:

True if a key press was detected and the "key" parameter is set to a value.

Reimplements: **touchgfx::HAL::sampleKey**

saveNextScreenshots

virtual void saveNextScreenshots (int n)

Copy the next N screenshots to disk.

On each screen update, the new screen is saved to disk.

Parameters:

n	Number of screenshots to save. These are added to any ongoing amount of screenshots in queue.

saveScreenshot

void saveScreenshot ( )

Saves a screenshot to the default folder and default filename.

saveScreenshot

```
virtual void saveScreenshot ( char * folder ,
                              char * filename
                              )
```

Saves a screenshot.

Parameters:

folder	Folder name to place the screenshot in.
filename	Filename to save the screenshot to.

sdl_init

```
virtual bool sdl_init ( int      argcount ,
                        char ** args
                        )
```

Initializes SDL.

Parameters:

argcount	Number of arguments.
args	Arguments.

Returns:

True if init went well, false otherwise.

setConsoleVisible

```
void setConsoleVisible ( bool visible ,
                         bool redrawWindow =true
                         )
```

Change visibility of console window (hidden vs.

shown).

Parameters:

visible	Should the window be visible?
redrawWindow	(Optional) Should the window be redrawn? Default is true.

See also:

setWindowVisible, **getConsoleVisible**

setVsyncInterval

void setVsyncInterval (float ms)

Sets vsync interval for simulating same tick speed as the real hardware.

Due to limitations in the granularity of SDL, the generated ticks in the simulator might not occur at the exact time, but accumulated over several ticks, the precision is very good.

Parameters:

ms The milliseconds between ticks.

NOTE

That you can also use HAL::setFrameRateCompensation() in the simulator. The effect of this can easily be demonstrated by dragging the console output window of the simulator (when running from Visual Studio) as this will pause the SDL and generate a lot of ticks when the console window is released. Beware that since the missed vsyncs are accumulated in an 8 bit counter, only up to 255 ticks may be missed, so at VsyncInterval = 16.6667, dragging the windows for more than 255 * 16.6667ms = 4250ms = 4.25s will not generate all the ticks that were actually missed. This situation is, however, not very realistic, as normally just a couple of vsyncs are skipped.

setWindowVisible

void setWindowVisible (bool visible ,
bool redrawWindow =true
)

Change visibility of window (hidden vs.

shown) as well as (due to backward compatibility) the visibility of the console window.

Parameters:

visible	Should the window be visible?
redrawWindow	(Optional) Should the window be redrawn? Default is true.

See also:

getWindowVisible, **setConsoleVisible**

taskEntry

virtual void taskEntry ( )

Main event loop.

Will wait for VSYNC signal, and then process next frame. Call this function from your GUI task.

NOTE

This function never returns!

Reimplements: **touchgfx::HAL::taskEntry**

doRotate

static uint8_t * doRotate (uint8_t * dst ,
uint8_t * src
)

Rotates a framebuffer if the display is rotated.

Parameters:

dst Destination for the rotated framebuffer. must be non-null if the screen is rotated.
src The framebuffer.

Returns:

Null if it fails, else a pointer to an uint8_t.

getArgv

static char ** getArgv (int * argc)

Gets the argc and argv for a Windows program.

Parameters:

argc Pointer to where to store number of arguments.

Returns:

The argv list of arguments.

getWindowTitle

static const char * getWindowTitle ( )

Gets window title.

Returns:

null "TouchGFX simulator" unless set to something else using **setWindowTitle()**.

See also:

setWindowTitle

isSingleStepping

static bool isSingleStepping ( )

Is single stepping.

Returns:

True if single stepping, false if not.

See also:

setSingleStepping

scaleTo24bpp

static uint8_t * scaleTo24bpp (uint8_t * dst ,
uint16_t * src ,
Bitmap::BitmapFormat format
)

Scale framebuffer to 24bpp.

The format of the framebuffer (src) is given in parameter format. The result is placed in the pre-allocated memory pointed to by parameter dst. If the frambebuffer is in format **Bitmap::RGB888**, parameter dst is not used and the parameter src is simply returned.

Parameters:

dst	Destination for the framebuffer. must be non-null unless format is **Bitmap::RGB888**.
src	The framebuffer.
format	Describes the format of the framebuffer (**lcd()**.framebufferFormat()).

Returns:

Null if it fails, else a pointer to an uint8_t.

setSingleStepping

static void setSingleStepping (bool singleStepping =true)

Single stepping enable/disable.

When single stepping is enabled, F10 will execute one tick and F9 will disable single stepping.

Parameters:

singleStepping (Optional) True to pause the simulation and start single stepping.

See also:

isSingleStepping

setWindowTitle

static void setWindowTitle (const char * title)

Sets window title.

Sets window title of the TouchGFX simulator.

Parameters:

title The title, if null the original "TouchGFX simulator" will be used.

See also:

getWindowTitle

singleStep

static void singleStep (uint16_t steps =1)

Single step a number of steps.

Only works if single stepping is already enabled.

Parameters:

steps (Optional) The steps Default is 1 step.

See also:

setSingleStepping, **isSingleStepping**

Protected Functions Documentation

configureInterrupts

virtual void configureInterrupts ( )

Configures the interrupts relevant for TouchGFX.

This primarily entails setting the interrupt priorities for the DMA and **LCD** interrupts.

Reimplements: **touchgfx::HAL::configureInterrupts**

configureLCDInterrupt

virtual void configureLCDInterrupt ( )

Configures LCD interrupt.

disableInterrupts

virtual void disableInterrupts ( )

Disables the DMA and LCD interrupts.

Reimplements: **touchgfx::HAL::disableInterrupts**

enableInterrupts

virtual void enableInterrupts ( )

Enables the DMA and LCD interrupts.

Reimplements: **touchgfx::HAL::enableInterrupts**

enableLCDControllerInterrupt

virtual void enableLCDControllerInterrupt ( )

Enables the LCD interrupt.

Reimplements: **touchgfx::HAL::enableLCDControllerInterrupt**

getTFTFrameBuffer

virtual uint16_t * getTFTFrameBuffer ( ) const

Gets TFT framebuffer.

Returns:

null if it fails, else the TFT framebuffer.

Reimplements: **touchgfx::HAL::getTFTFrameBuffer**

performDisplayOrientationChange

virtual void performDisplayOrientationChange ( )

Perform the actual display orientation change.

Reimplements: **touchgfx::HAL::performDisplayOrientationChange**

renderLCD_FrameBufferToMemory

virtual void renderLCD_FrameBufferToMemory (const Rect & _rectToUpdate ,
uint8_t * frameBuffer
)

Update framebuffer using an SDL Surface.

Parameters:

_rectToUpdate Area to update.
frameBuffer Target framebuffer.

setTFTFrameBuffer

virtual void setTFTFrameBuffer (uint16_t * addr)

Sets TFT framebuffer.

Parameters:

addr The address of the TFT framebuffer.

Reimplements: **touchgfx::HAL::setTFTFrameBuffer**

Version: 4.16

I2C

Platform independent interface for I2C drivers.

Public Functions

	I2C(uint8_t ch)
	Initializes a new instance of the **I2C** class.
virtual void	**init**() =0
	Initializes the **I2C** driver.
virtual bool	**readRegister**(uint8_t addr, uint8_t reg, uint8_t * data, uint32_t cnt) =0
	Reads the specified register on the device with the specified address.
virtual bool	**writeRegister**(uint8_t addr, uint8_t reg, uint8_t val) =0
	Writes the specified value in a register.
virtual	**~I2C**()
	Finalizes an instance of the **I2C** class.

Protected Attributes

uint8_t	**channel**
	I2c channel is stored in order to initialize and recover a specific **I2C** channel.

Public Functions Documentation

I2C

I2C (uint8_t ch)

Initializes a new instance of the **I2C** class.

Stores the channel of the **I2C** bus to be configured.

Parameters:

ch I2C channel.

init

```
virtual void init (  ) =0
```

Initializes the **I2C** driver.

readRegister

```
virtual bool readRegister ( uint8_t   addr ,  =0
                            uint8_t   reg ,   =0
                            uint8_t * data ,  =0
                            uint32_t  cnt     =0
                            )         =0
```

Reads the specified register on the device with the specified address.

Parameters:

addr The **I2C** device address.
reg The register.
data Pointer to buffer in which to place the result.
cnt Size of buffer in bytes.

Returns:

true on success, false otherwise.

writeRegister

```
virtual bool writeRegister ( uint8_t addr ,  =0
                             uint8_t reg ,   =0
                             uint8_t val     =0
                             )       =0
```

Writes the specified value in a register.

Parameters:

addr The **I2C** device address.
reg The register.
val The new value.

Returns:

true on success, false otherwise.

~I2C

virtual ~I2C ( )

Finalizes an instance of the **I2C** class.

Protected Attributes Documentation

channel

uint8_t channel

I2c channel is stored in order to initialize and recover a specific **I2C** channel.

Version: 4.16

I2CTouchController

Specific I2C-enabled type of Touch Controller.

See: TouchController

Inherits from: TouchController

Public Functions

	I2CTouchController(**I2C** & i2c)
	Constructor.
virtual void	**init**() =0
	Initializes touch controller.
virtual bool	**sampleTouch**(int32_t & x, int32_t & y) =0
	Checks whether the touch screen is being touched, and if so, what coordinates.
virtual	**~I2CTouchController**()

Protected Attributes

I2C &	**i2c**
	I2C driver.

Additional inherited members

Public Functions inherited from TouchController

virtual	**~TouchController**()
	Finalizes an instance of the **TouchController** class.

Public Functions Documentation

I2CTouchController

I2CTouchController (I2C & i2c)

Constructor.

Initializes **I2C** driver.

Parameters:

i2c **I2C** driver.

init

virtual void init ( ) =0

Initializes touch controller.

Reimplements: **touchgfx::TouchController::init**

sampleTouch

virtual bool sampleTouch (int32_t & x , =0
int32_t & y =0
) =0

Checks whether the touch screen is being touched, and if so, what coordinates.

Parameters:

x The x position of the touch.
y The y position of the touch.

Returns:

True if a touch has been detected, otherwise false.

Reimplements: **touchgfx::TouchController::sampleTouch**

~I2CTouchController

virtual ~I2CTouchController ( )

Protected Attributes Documentation

i2c

I2C & i2c
I2C driver.

Version: 4.16

IconButtonStyle

An icon button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show one of two icons depending on the state of the button (pressed or released).

To get a background behind the icon, use IconButtonStyle together with e.g. ImageButtonStyle: IconButtonStyle<ImageButtonStyle<ClickButtonTrigger> > myButton;

The IconButtonStyle will center the icon on the enclosing container (normally AbstractButtonContainer). Set the size of the button before setting the icons.

The position of the icon can be adjusted with setIconXY.

See: AbstractButtonContainer

Inherits from: T

Public Functions

Bitmap	**getCurrentlyDisplayedIcon**() const
	Gets currently displayed icon.
int16_t	**getIconX**() const
	Gets icon x coordinate.
int16_t	**getIconY**() const
	Gets icon y coordinate.
	IconButtonStyle()
virtual void	**setIconBitmaps**(const **Bitmap** & newIconReleased, const **Bitmap** & newIconPressed)
	Sets icon bitmaps.
void	**setIconX**(int16_t x)
	Sets icon x coordinate.
void	**setIconXY**(int16_t x, int16_t y)
	Sets the position of the icon.

void	**setIconY**(int16_t y)
	Sets icon y coordinate.

Protected Functions

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

Image	**iconImage**
	The icon image.
Bitmap	**iconPressed**
	Icon to display when button is pressed.
Bitmap	**iconReleased**
	Icon to display when button is not pressed.

Public Functions Documentation

getCurrentlyDisplayedIcon

Bitmap getCurrentlyDisplayedIcon ( ) const

Gets currently displayed icon.

Returns:

The currently displayed icon.

getIconX

int16_t getIconX ( ) const

Gets icon x coordinate.

Returns:

The icon x coordinate.

getIconY

int16_t getIconY ( ) const

Gets icon y coordinate.

Returns:

The icon y coordinate.

IconButtonStyle

IconButtonStyle ( )

setIconBitmaps

virtual void setIconBitmaps (const Bitmap & newIconReleased ,
const Bitmap & newIconPressed
)

Sets icon bitmaps.

Parameters:

newIconReleased The new icon released.
newIconPressed The new icon pressed.

setIconX

void setIconX (int16_t x)

Sets icon x coordinate.

Parameters:

x The x coordinate.

setIconXY

```
void setIconXY ( int16_t x ,
                 int16_t y
                 )
```

Sets the position of the icon.

Parameters:

x The x coordinate.
y The y coordinate.

setIconY

```
void setIconY ( int16_t y )
```

Sets icon y coordinate.

Parameters:

y The y coordinate.

Protected Functions Documentation

handleAlphaUpdated

```
virtual void handleAlphaUpdated (  )
```

Handles what should happen when the alpha is updated.

handlePressedUpdated

```
virtual void handlePressedUpdated (  )
```

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

iconImage

Image iconImage

The icon image.

iconPressed

Bitmap iconPressed

Icon to display when button is pressed.

iconReleased

Bitmap iconReleased

Icon to display when button is not pressed.

Version: 4.16

Image

Simple widget capable of showing a bitmap on the display. The bitmap can be alpha-blended with the background (or whichever other Drawable might be "underneath" the Image). The bitmap can and have areas of varying opacity.

The conversion from image.bmp or image.png to a bitmap that can be used in TouchGFX is handled by the Image Converter as part of compiling the project. Each image is assigned a unique BITMAP identifier which.

See: Bitmap

Inherits from: Widget, Drawable

Inherited by: AnimatedImage, ScalableImage, TextureMapper, TiledImage

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getBitmap**() const
	Gets the **Bitmap** currently assigned to the **Image** widget.
BitmapId	**getBitmapId**() const
	Gets the BitmapId currently assigned to the **Image** widget.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
	Image(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this **Image** and updates the width and height of this widget to match those of the **Bitmap**.

Protected Attributes

uint8_t	**alpha**
	The Alpha for this image.

Bitmap	**bitmap**
	The **Bitmap** to display.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.

virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.

int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()

	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.

bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

Reimplemented by: **touchgfx::ScalableImage::draw**, **touchgfx::TextureMapper::draw**, **touchgfx::TiledImage::draw**

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getBitmap

Bitmap getBitmap ( ) const

Gets the **Bitmap** currently assigned to the **Image** widget.

Returns:

The current **Bitmap** of the widget.

getBitmapId

BitmapId getBitmapId ( ) const

Gets the BitmapId currently assigned to the **Image** widget.

Returns:

The current BitmapId of the widget.

DEPRECATED

Use getBitmap() which is automatically converted to BitmapId on demand.

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

Reimplemented by: **touchgfx::ScalableImage::getSolidRect**, **touchgfx::TextureMapper::getSolidRect**, **touchgfx::TiledImage::getSolidRect**

Image

Image (const Bitmap & bitmap =Bitmap())

Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

If a **Bitmap** is passed to the constructor, the width and height of this widget is set to those of the bitmap.

Parameters:

bitmap (Optional) The bitmap to display.

See also:

setBitmap

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setBitmap

virtual void setBitmap (const Bitmap & bitmap)

Sets the bitmap for this **Image** and updates the width and height of this widget to match those of the **Bitmap**.

Parameters:

bitmap The bitmap instance.

NOTE

The user code must call **invalidate()** in order to update the image on the display.

Reimplemented by: touchgfx::AnimatedImage::setBitmap, touchgfx::TextureMapper::setBitmap, touchgfx::TiledImage::setBitmap

Protected Attributes Documentation

alpha

uint8_t alpha

The Alpha for this image.

bitmap

Bitmap bitmap

The **Bitmap** to display.

Version: 4.16

ImageButtonStyle

An image button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show one of two images depending on the state of the button (pressed or released).

The ImageButtonStyle will set the size of the enclosing container (normally AbstractButtonContainer) to the size of the pressed Bitmap. This can be overridden by calling setWidth/setHeight after setting the bitmaps.

The position of the bitmap can be adjusted with setBitmapXY (default is upper left corner).

Template Parameters:

- **T** Generic type parameter. Typically a AbstractButtonContainer subclass.

See: AbstractButtonContainer

Inherits from: T

Public Functions

Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
	ImageButtonStyle()
virtual void	**setBitmaps**(const **Bitmap** & bmpReleased, const **Bitmap** & bmpPressed)
	Sets the bitmaps.
void	**setBitmapXY**(uint16_t x, uint16_t y)
	Sets bitmap x and y.

Protected Functions

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.

virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

Image	**buttonImage**
	The button image.
Bitmap	**down**
	The image to display when button is pressed.
Bitmap	**up**
	The image to display when button is released.

Public Functions Documentation

getCurrentlyDisplayedBitmap

Bitmap getCurrentlyDisplayedBitmap ( ) const

Gets currently displayed bitmap.

Returns:

The currently displayed bitmap.

ImageButtonStyle

ImageButtonStyle ( )

setBitmaps

virtual void setBitmaps (const Bitmap & bmpReleased ,
const Bitmap & bmpPressed
)

Sets the bitmaps.

Parameters:

bmpReleased The bitmap released.
bmpPressed The bitmap pressed.

setBitmapXY

```
void setBitmapXY ( uint16_t x ,
                   uint16_t y
                   )
```

Sets bitmap x and y.

Parameters:

x An uint16_t to process.
y An uint16_t to process.

Protected Functions Documentation

handleAlphaUpdated

```
virtual void handleAlphaUpdated ( )
```

Handles what should happen when the alpha is updated.

handlePressedUpdated

```
virtual void handlePressedUpdated ( )
```

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

buttonImage

Image buttonImage

The button image.

down

Bitmap down

The image to display when button is pressed.

up

Bitmap up

The image to display when button is released.

Version: 4.16

ImageProgress

An image progress will show parts of an image as a progress indicator. The image can progress from the left, the right, the bottom or the top of the given area, and can visually be fixed with a larger and larger portion of the image showing, or it can be moved into view.

Inherits from: AbstractDirectionProgress, AbstractProgressIndicator, Container, Drawable

Public Functions

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual bool	**getAnchorAtZero**() const
	Gets the current anchor at zero setting.
virtual **BitmapId**	**getBitmap**() const
	Gets the bitmap id of the current image.
	ImageProgress()
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setAnchorAtZero**(bool anchorAtZero)
	Sets anchor at zero.
virtual void	**setBitmap**(**BitmapId** bitmapId)
	Sets the bitmap id to use for progress.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the actual progress indicator relative to the background image.
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.

Protected Attributes

Container	**container**
	The container for the image to allow for anchored images.

bool	**fixedPosition**
	true if the image should not move during progress

TiledImage	**image**
	The image.

Additional inherited members

Public Types inherited from AbstractDirectionProgress

enum	**DirectionType** { RIGHT, LEFT, DOWN, UP }
	Values that represent directions.

Public Functions inherited from AbstractDirectionProgress

	AbstractDirectionProgress()

virtual DirectionType	**getDirection**() const
	Gets the current direction for the progress indicator.

virtual void	**setDirection**(**DirectionType** direction)
	Sets a direction for the progress indicator.

Protected Attributes inherited from AbstractDirectionProgress

DirectionType	**progressDirection**
	The progress direction.

Public Functions inherited from AbstractProgressIndicator

	AbstractProgressIndicator()

	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.
virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.

void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.

int	**rangeMin**
	The range minimum.
uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.

virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**

	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getAnchorAtZero

virtual bool getAnchorAtZero ( ) const

Gets the current anchor at zero setting.

Returns:

true if the image is anchored at zero, false if it is anchored at current progress.

See also:

setAnchorAtZero

getBitmap

virtual BitmapId getBitmap ( ) const

Gets the bitmap id of the current image.

Returns:

The image.

See also:

setBitmap

ImageProgress

ImageProgress ( )

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setAnchorAtZero

virtual void setAnchorAtZero (bool anchorAtZero)

Sets anchor at zero.

Setting anchor at zero will force the image will be placed so that it is not moved during progress, only more and more of the image will become visible. If the image is not anchored at zero, it will be anchored at the current progress and will appear to slide into view.

Parameters:

anchorAtZero true to anchor at zero, false to anchor at current progress.

See also:

getAnchorAtZero

setBitmap

virtual void setBitmap (BitmapId bitmapId)

Sets the bitmap id to use for progress.

Please note that the bitmap is tiled which will allow smaller bitmaps to repeat on the display and save memory.

Parameters:

bitmapId The bitmap id.

See also:

getBitmap, **TiledImage**

setProgressIndicatorPosition

virtual void setProgressIndicatorPosition (int16_t x ,
int16_t y ,
int16_t width ,
int16_t height
)

Sets the position and dimensions of the actual progress indicator relative to the background image.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the box progress indicator.
height	The height of the box progress indicator.

See also:

getProgressIndicatorX, **getProgressIndicatorY**, **getProgressIndicatorWidth**, **getProgressIndicatorHeight**

Reimplements: **touchgfx::AbstractProgressIndicator::setProgressIndicatorPosition**

setValue

virtual void setValue (int value)

Sets the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. If a callback function has been set using setValueSetAction, that callback will be called (unless the new value is the same as the current value).

Parameters:

value The value.

NOTE

if value is equal to the current value, nothing happens, and the callback will not be called.

See also:

getValue, **updateValue**, **setValueSetAction**

Reimplements: **touchgfx::AbstractProgressIndicator::setValue**

Protected Attributes Documentation

container

Container container

The container for the image to allow for anchored images.

fixedPosition

bool fixedPosition

true if the image should not move during progress

image

TiledImage image

The image.

Version: 4.16

InternalFlashFont

An InternalFlashFont has both glyph table and glyph data placed in a flash which supports random access read (i.e. not a NAND flash)

See: Font, ConstFont

Inherits from: ConstFont, Font

Public Functions

virtual int8_t	**getKerning**(**Unicode::UnicodeChar** prevChar, const **GlyphNode** * glyph) const
	Gets the kerning distance between two characters.
virtual const uint8_t *	**getPixelData**(const **GlyphNode** * glyph) const
	Gets the pixel date associated with this glyph.
	InternalFlashFont(const **GlyphNode** *list, uint16_t size, uint16_t height, uint8_t pixBelowBase, uint8_t bitsPerPixel, uint8_t byteAlignRow, uint8_t maxLeft, uint8_t maxRight, const uint8_t* glyphDataInternalFlash, const **KerningNode** * kerningList, const **Unicode::UnicodeChar** fallbackChar, const **Unicode::UnicodeChar** ellipsisChar)
	Initializes a new instance of the **InternalFlashFont** class.

Additional inherited members

Public Functions inherited from ConstFont

	ConstFont(const **GlyphNode** * list, uint16_t size, uint16_t height, uint8_t pixBelowBase, uint8_t bitsPerPixel, uint8_t byteAlignRow, uint8_t maxLeft, uint8_t maxRight, const **Unicode::UnicodeChar** fallbackChar, const **Unicode::UnicodeChar** ellipsisChar)
	Initializes a new instance of the **ConstFont** class.
const **GlyphNode** *	**find**(**Unicode::UnicodeChar** unicode) const
	Finds the glyph data associated with the specified unicode.

const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode)
	Gets the glyph data associated with the specified Unicode.
virtual const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode, const uint8_t *& pixelData, uint8_t & bitsPerPixel) const
	Gets the glyph data associated with the specified Unicode.
const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode, const uint8_t *& pixelData, uint8_t & bitsPerPixel)
	Gets the glyph data associated with the specified Unicode.

Protected Attributes inherited from ConstFont

const **GlyphNode** *	**glyphList**
	The list of glyphs.
uint16_t	**listSize**
	The size of the list of glyphs.

Public Functions inherited from Font

virtual FORCE_INLINE_FUNCTION uint8_t	**getBitsPerPixel**() const
	Gets bits per pixel for this font.
virtual FORCE_INLINE_FUNCTION uint8_t	**getByteAlignRow**() const
	Are the glyphs saved with each glyph row byte aligned?
virtual uint16_t	**getCharWidth**(const **Unicode::UnicodeChar** c) const
	Gets the width in pixels of the specified character.
virtual **Unicode::UnicodeChar**	**getEllipsisChar**() const
	Gets ellipsis character for the given font.
virtual **Unicode::UnicodeChar**	**getFallbackChar**() const
	Gets fallback character for the given font.
virtual FORCE_INLINE_FUNCTION uint16_t	**getFontHeight**() const
	Returns the height in pixels of this font.
virtual const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode) const

	Gets the glyph data associated with the specified Unicode.
virtual const **GlyphNode** *	**getGlyph**(**Unicode::UnicodeChar** unicode, const uint8_t *& pixelData, uint8_t & bitsPerPixel) const =0
	Gets the glyph data associated with the specified Unicode.
virtual const uint16_t *	**getGSUBTable**() const
	Gets GSUB table.
FORCE_INLINE_FUNCTION uint8_t	**getMaxPixelsLeft**() const
	Gets maximum pixels left of any glyph in the font.
FORCE_INLINE_FUNCTION uint8_t	**getMaxPixelsRight**() const
	Gets maximum pixels right of any glyph in the font.
virtual uint16_t	**getMaxTextHeight**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the height of the highest character in a given string.
virtual FORCE_INLINE_FUNCTION uint16_t	**getMinimumTextHeight**() const
	Returns the minimum height needed for a text field that uses this font.
virtual uint16_t	**getNumberOfLines**(const **Unicode::UnicodeChar** * text, ...) const
	Count the number of lines in a given text.
virtual uint8_t	**getSpacingAbove**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the number of blank pixels at the top of the given text.
virtual uint16_t	**getStringWidth**(const **Unicode::UnicodeChar** * text, ...) const
	Gets the width in pixels of the specified string.
virtual uint16_t	**getStringWidth**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, ...) const
	Gets the width in pixels of the specified string.
virtual	**~Font**()
	Finalizes an instance of the **Font** class.

FORCE_INLINE_FUNCTION bool	**isInvisibleZeroWidth**(**Unicode::UnicodeChar** character)
	Query if 'character' is invisible, zero width.

Protected Functions inherited from Font

	Font(uint16_t height, uint8_t pixBelowBase, uint8_t bitsPerPixel, uint8_t byteAlignRow, uint8_t maxLeft, uint8_t maxRight, const **Unicode::UnicodeChar** fallbackChar, const **Unicode::UnicodeChar** ellipsisChar)
	Initializes a new instance of the **Font** class.
uint16_t	**getStringWidthLTR**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, va_list pArg) const
	Gets the width in pixels of the specified string.
uint16_t	**getStringWidthRTL**(**TextDirection** textDirection, const **Unicode::UnicodeChar** * text, va_list pArg) const
	Gets the width in pixels of the specified string.

Protected Attributes inherited from Font

uint8_t	**bAlignRow**
	The glyphs are saved with each row byte aligned.
uint8_t	**bPerPixel**
	The number of bits per pixel.
Unicode::UnicodeChar	**ellipsisCharacter**
	The ellipsis character used for truncating long texts.
Unicode::UnicodeChar	**fallbackCharacter**
	The fallback character to use when no glyph exists for the wanted character.
uint16_t	**fontHeight**
	The font height in pixels.
uint8_t	**maxPixelsLeft**
	The maximum number of pixels a glyph extends to the left.
uint8_t	**maxPixelsRight**
	The maximum number of pixels a glyph extends to the right.

uint8_t	**pixelsBelowBaseline**
	The number of pixels below the base line.

Public Functions Documentation

getKerning

virtual int8_t getKerning (Unicode::UnicodeChar prevChar , const
const GlyphNode * glyph const
) const

Gets the kerning distance between two characters.

Parameters:

prevChar The **Unicode** value of the previous character.
glyph the glyph object for the current character.

Returns:

The kerning distance between prevChar and glyph char.

Reimplements: **touchgfx::ConstFont::getKerning**

getPixelData

virtual const uint8_t * getPixelData (const GlyphNode * glyph)

Gets the pixel date associated with this glyph.

Parameters:

glyph The glyph to get the pixels data from.

Returns:

Pointer to the pixel data of this glyph.

Reimplements: **touchgfx::ConstFont::getPixelData**

InternalFlashFont

InternalFlashFont (const GlyphNode * list ,
uint16_t size ,

	uint16_t	height ,
	uint8_t	pixBelowBase ,
	uint8_t	bitsPerPixel ,
	uint8_t	byteAlignRow ,
	uint8_t	maxLeft ,
	uint8_t	maxRight ,
	const uint8_t *	glyphDataInternalFlash ,
	const KerningNode *	kerningList ,
	const Unicode::UnicodeChar	fallbackChar ,
	const Unicode::UnicodeChar	ellipsisChar
	)	

Initializes a new instance of the **InternalFlashFont** class.

Parameters:

list	The array of glyphs known to this font.
size	The number of glyphs in list.
height	The height in pixels of the highest character in this font.
pixBelowBase	The maximum number of pixels that can be drawn below the baseline in this font.
bitsPerPixel	The number of bits per pixel in this font.
byteAlignRow	The glyphs are saved with each row byte aligned.
maxLeft	The maximum a character extends to the left.
maxRight	The maximum a character extends to the right.
glyphDataInternalFlash	Pointer to the glyph data for the font, placed in internal flash.
kerningList	pointer to the kerning data for the font, placed in internal flash.
fallbackChar	The fallback character for the typography in case no glyph is available.
ellipsisChar	The ellipsis character used for truncating long texts.

Version: 4.16

KerningNode

Structure providing information about a kerning for a given pair of characters. Used by LCD when rendering, calculating text width etc.

Public Attributes

int8_t	**distance**
	The kerning distance.
Unicode::UnicodeChar	**unicodePrevChar**
	The Unicode for the first character in the kerning pair.

Public Attributes Documentation

distance

int8_t distance

The kerning distance.

unicodePrevChar

Unicode::UnicodeChar unicodePrevChar

The Unicode for the first character in the kerning pair.

Version: 4.16

Key

Mapping from rectangle to key id.

Public Attributes

BitmapId	highlightBitmapId
	A bitmap to show when the area is "pressed".
Rect	keyArea
	The area occupied by the key.
uint8_t	keyId
	The id of a key.

Public Attributes Documentation

highlightBitmapId

BitmapId highlightBitmapId

A bitmap to show when the area is "pressed".

keyArea

Rect keyArea

The area occupied by the key.

keyId

uint8_t keyId

The id of a key.

Version: 4.16

Keyboard

The keyboard provides text input for touch devices. It is configured using a Layout and a KeyMappingList, both of which can be changed at runtime. The class using the keyboard must provide a buffer where the entered text is placed. The Layout contains a bitmap id for the image to display and two mappings: rectangles to key ids and rectangles to callback methods.

The KeyMappingList maps key ids to Unicode characters. When the user presses a key, the keyboard looks in its layout for a rectangle containing the coordinates pressed. If it finds a mapping to a callback method, it will invoke that method. If it finds a mapping to a key it will look up the Unicode character for that key and place it in a text buffer. The sequence is: (x,y) -> KeyId -> UnicodeChar.

A keyboard with multiple key mappings e.g. lower case alpha, upper case alpha and numeric mappings can be created by implementing callback methods for shift and mode areas in the provided bitmap and then changing the KeyMappingList when those areas are pressed.

Inherits from: Container, Drawable

Public Classes

struct	**CallbackArea**
	Mapping from rectangle to a callback method to execute.
struct	**Key**
	Mapping from rectangle to key id.
struct	**KeyMapping**
	Mapping from key id to Unicode character.
struct	**KeyMappingList**
	List of KeyMappings to use.
struct	**Layout**
	Definition of the keyboard layout.

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Overrides the draw implementation on the **Container**.
Unicode::UnicodeChar *	**getBuffer**() const
	Gets the buffer.
uint16_t	**getBufferPosition**()
	Gets buffer position.
const **KeyMappingList** *	**getKeyMappingList**() const
	Gets key mapping list.
const **Layout** *	**getLayout**() const
	Gets the layout.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Overrides the handleClickEvent on the container.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Overrides the handleDragEvent on the container.
	Keyboard()
void	**setBuffer**(**Unicode::UnicodeChar** * newBuffer, uint16_t newBufferSize)
	Sets the buffer to be used by the keyboard.
void	**setBufferPosition**(uint16_t newPos)
	Change the buffer position i.e.
void	**setKeyListener**(**GenericCallback**< **Unicode::UnicodeChar** > & callback)
	Sets the callback for the keyboard.
void	**setKeymappingList**(const **KeyMappingList** * newKeyMappingList)
	Set/change the KeyMappingList to use.
void	**setLayout**(const **Layout** * newLayout)
	Set/change the Keyboard::Layout to use.The **Keyboard** will invalidate the space it occupies to request a redraw.
void	**setTextIndentation**()
	Sets text indentation by making the area for entered text slightly larger.

Protected Functions

CallbackArea	**getCallbackAreaForCoordinates**(int16_t x, int16_t y) const
	Gets the callback area defined by the layout for the specified coordinates.
Unicode::UnicodeChar	**getCharForKey**(uint8_t keyId) const
	Maps a keyId to the UnicodeChar being displayed by that key.
Key	**getKeyForCoordinates**(int16_t x, int16_t y) const
	Gets key for coordinates.

Protected Attributes

Unicode::UnicodeChar *	**buffer**
	Pointer to null-terminated buffer where the entered text is being displayed.
uint16_t	**bufferPosition**
	Current position in buffer.
uint16_t	**bufferSize**
	Size of the buffer.
bool	**cancelIsEmitted**
	Tells if a cancel is emitted to check when a key is released.
TextAreaWithOneWildcard	**enteredText**
	Widget capable of displaying the entered text buffer.
Image	**highlightImage**
	Image to display when a key is highlighted.
Image	**image**
	Layout bitmap.
GenericCallback< Unicode::UnicodeChar > *	**keyListener**
	Pointer to callback being executed when a key is pressed.
const KeyMappingList *	**keyMappingList**

	Pointer to key mapping.
const **Layout** *	**layout**
	Pointer to layout.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()

	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.

bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Overrides the draw implementation on the **Container**.

First invokes the container draw implementation to draw the keyboard bitmap and text area holding the entered text. If additional drawables have been added to the keyboard, they will also be draw. After invoking the container draw, the glyphs mapped to keys are drawn and if a key has been pressed, it will be highlighted.

Parameters:

invalidatedArea The area to draw.

Reimplements: **touchgfx::Container::draw**

getBuffer

Unicode::UnicodeChar * getBuffer ( ) const

Gets the buffer.

Returns:

The buffer containing entered text currently being displayed.

See also:

setBuffer

getBufferPosition

uint16_t getBufferPosition ( )

Gets buffer position.

Returns:

the buffer position, i.e. the current index where new characters will be placed.

See also:

setBufferPosition

getKeyMappingList

const KeyMappingList * getKeyMappingList ( ) const

Gets key mapping list.

Returns:

The **KeyMappingList** used by the **Keyboard**.

getLayout

const Layout * getLayout ( ) const

Gets the layout.

Returns:

The layout used by the **Keyboard**.

See also:

setLayout

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Overrides the handleClickEvent on the container.

The keyboard handles all click events internally and click events are *not* propagated to drawables added to the keyboard.

Parameters:

evt The **ClickEvent**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Overrides the handleDragEvent on the container.

The keyboard handles drag events to enable the container to, emit a CANCEL, if the user drags outside the currently pressed key.

Parameters:

evt The **DragEvent**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

Keyboard

```
Keyboard ( )
```

setBuffer

```
void setBuffer ( Unicode::UnicodeChar * newBuffer ,
                 uint16_t               newBufferSize
                 )
```

Sets the buffer to be used by the keyboard.

Keys entered are added to the buffer.

Parameters:

newBuffer	Pointer to a buffer holding the text edited by the keyboard. If the buffer is not empty, the edit position for the keyboard will be set to the end of the provided text.
newBufferSize	Length of the buffer, i.e. number of UnicodeChar's.

See also:

getBuffer

setBufferPosition

```
void setBufferPosition ( uint16_t newPos )
```

Change the buffer position i.e.

the next index to place a character when a key is pressed. This can be used to implement backspace functionality if the class using the **Keyboard** implements a callback and maps it to a backspace implementation. Setting the position will cause the **TextArea** displaying the text to be invalidated to request a redraw.

Parameters:

newPos The buffer position.

See also:

setTextIndentation

setKeyListener

void setKeyListener (GenericCallback< Unicode::UnicodeChar > & callback)

Sets the callback for the keyboard.

The callback will be executed every time a key is clicked. The callback argument contains the key that was just pressed.

Parameters:

callback The **Callback** to invoke.

NOTE

Backspace, shift and mode keys report a 0 as value.

setKeymappingList

void setKeymappingList (const KeyMappingList * newKeyMappingList)

Set/change the KeyMappingList to use.

The **Keyboard** will invalidate the space it occupies to request a redraw.

Parameters:

newKeyMappingList The new **KeyMappingList**.

setLayout

void setLayout (const Layout * newLayout)

Set/change the Keyboard::Layout to use.The **Keyboard** will invalidate the space it occupies to request a redraw.

Parameters:

newLayout The new layout.

See also:

getLayout

setTextIndentation

void setTextIndentation ( )

Sets text indentation by making the area for entered text slightly larger.

The result is that some characters (often 'j' and '_') will not be cut off. Indentation is added to both sides of the text area in case the text is right-to-left. Indentation is automatically set so all characters will display properly.

See also:

TextArea::setIndentation

Protected Functions Documentation

getCallbackAreaForCoordinates

CallbackArea getCallbackAreaForCoordinates (int16_t x , const
int16_t y const
) const

Gets the callback area defined by the layout for the specified coordinates.

Parameters:

x The x coordinate to perform key look up with.
y The y coordinate to perform key look up with.

Returns:

The **CallbackArea**, which is empty if not found.

getCharForKey

Unicode::UnicodeChar getCharForKey (uint8_t keyId)

Maps a keyId to the UnicodeChar being displayed by that key.

Parameters:

keyId The id of the key to perform lookup with.

Returns:

the UnicodeChar used for the specified key.

getKeyForCoordinates

```
Key getKeyForCoordinates ( int16_t x , const
                           int16_t y const
                           ) const
```

Gets key for coordinates.

Parameters:

x The x coordinate to perform key look up with.
y The y coordinate to perform key look up with.

Returns:

The key for the given coordinates.

Protected Attributes Documentation

buffer

Unicode::UnicodeChar * buffer

Pointer to null-terminated buffer where the entered text is being displayed.

bufferPosition

uint16_t bufferPosition

Current position in buffer.

bufferSize

uint16_t bufferSize

Size of the buffer.

cancelIsEmitted

bool cancelIsEmitted

Tells if a cancel is emitted to check when a key is released.

enteredText

TextAreaWithOneWildcard enteredText

Widget capable of displaying the entered text buffer.

highlightImage

Image highlightImage

Image to display when a key is highlighted.

image

Image image

Layout bitmap.

keyListener

GenericCallback< Unicode::UnicodeChar > * keyListener

Pointer to callback being executed when a key is pressed.

keyMappingList

const KeyMappingList * keyMappingList

Pointer to key mapping.

layout

const Layout * layout

Pointer to layout.

Version: 4.16

KeyMapping

Mapping from key id to Unicode character.

Public Attributes

uint8_t	**keyId**
	Id of a key.
Unicode::UnicodeChar	**keyValue**
	Unicode equivalent of the key id.

Public Attributes Documentation

keyId

uint8_t keyId

Id of a key.

keyValue

Unicode::UnicodeChar keyValue

Unicode equivalent of the key id.

Version: 4.16

KeyMappingList

List of KeyMappings to use.

Public Attributes

const KeyMapping *	keyMappingArray
	The array of key mappings used by the keyboard.
uint8_t	numberOfKeys
	The number of keys in the list.

Public Attributes Documentation

keyMappingArray

const KeyMapping * keyMappingArray

The array of key mappings used by the keyboard.

numberOfKeys

uint8_t numberOfKeys

The number of keys in the list.

Version: 4.16

Layout

Definition of the keyboard layout. The keyboard can handle changing layouts, so different keyboard modes can be implemented by changing layouts and key mappings.

Public Attributes

BitmapId	**bitmap**
	The bitmap used for the keyboard layout.
CallbackArea *	**callbackAreaArray**
	The array of areas and corresponding callbacks.
const Key *	**keyArray**
	The keys on the keyboard layout.
FontId	**keyFont**
	The font used for the keys.
colortype	**keyFontColor**
	The color used for the keys.
uint8_t	**numberOfCallbackAreas**
	The number of areas and corresponding callbacks.
uint8_t	**numberOfKeys**
	The number of keys on this keyboard layout.
TypedText	**textAreaFont**
	The font used for typing text.
colortype	**textAreaFontColor**
	The color used for the typing text.
Rect	**textAreaPosition**
	The area where text is written.

Public Attributes Documentation

bitmap

BitmapId bitmap

The bitmap used for the keyboard layout.

callbackAreaArray

CallbackArea * callbackAreaArray

The array of areas and corresponding callbacks.

keyArray

const Key * keyArray

The keys on the keyboard layout.

keyFont

FontId keyFont

The font used for the keys.

keyFontColor

colortype keyFontColor

The color used for the keys.

numberOfCallbackAreas

uint8_t numberOfCallbackAreas

The number of areas and corresponding callbacks.

numberOfKeys

uint8_t numberOfKeys

The number of keys on this keyboard layout.

textAreaFont

TypedText textAreaFont

The font used for typing text.

textAreaFontColor

colortype textAreaFontColor

The color used for the typing text.

textAreaPosition

Rect textAreaPosition

The area where text is written.

Version: 4.16

LCD

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles/boxes. Normally, these draw operations are called from widgets, which also keep track of logical states such as visibility etc.

The LCD class cannot be instantiated, instead use one of the subclasses which implements the LCD drawing operations for a specific display configuration.

Note: All coordinates sent to functions in the LCD class are expected to be in absolute coordinates, i.e. (0, 0) is upper left corner of the display.

Inherited by: LCD16bpp, LCD16bppSerialFlash, LCD1bpp, LCD24bpp, LCD2bpp, LCD32bpp, LCD4bpp, LCD8bpp_ABGR2222, LCD8bpp_ARGB2222, LCD8bpp_BGRA2222, LCD8bpp_RGBA2222

Public Classes

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions

virtual uint8_t	**bitDepth**() const =0
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels) =0
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels) =0
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & region, const **BitmapId** bitmapId =**BITMAP_ANIMATION_STORAGE**)
	Copies part of the framebuffer to the data section of a bitmap.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId) =0
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true) =0
	Draws all (or a part) of a *bitmap*.
void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.
virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255) =0
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const =0
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const =0
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const =0
	Gets the blue color part of a color.

virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const =0
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
virtual uint8_t	**getGreenColor**(**colortype** color) const =0
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const =0
	Gets the red color part of a color.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation) =0
	Private version of draw-glyph with explicit destination buffer pointer argument.

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
virtual **DrawTextureMapScanLineBase** *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.

uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const =0

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplemented by: **touchgfx::LCD16bpp::bitDepth**, **touchgfx::LCD16bppSerialFlash::bitDepth**, **touchgfx::LCD1bpp::bitDepth**, **touchgfx::LCD24bpp::bitDepth**, **touchgfx::LCD2bpp::bitDepth**, **touchgfx::LCD32bpp::bitDepth**, **touchgfx::LCD4bpp::bitDepth**, **touchgfx::LCD8bpp_ABGR2222::bitDepth**, **touchgfx::LCD8bpp_ARGB2222::bitDepth**, **touchgfx::LCD8bpp_BGRA2222::bitDepth**, **touchgfx::LCD8bpp_RGBA2222::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,          =0
                        const Rect &     source ,               =0
                        const Rect &     blitRect ,             =0
                        uint8_t          alpha ,                =0
                        bool             hasTransparentPixels   =0
                        )                =0
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplemented by: **touchgfx::LCD16bpp::blitCopy**, **touchgfx::LCD16bppSerialFlash::blitCopy**, **touchgfx::LCD1bpp::blitCopy**, **touchgfx::LCD24bpp::blitCopy**, **touchgfx::LCD2bpp::blitCopy**, **touchgfx::LCD32bpp::blitCopy**, **touchgfx::LCD4bpp::blitCopy**, **touchgfx::LCD8bpp_ABGR2222::blitCopy**, **touchgfx::LCD8bpp_ARGB2222::blitCopy**, **touchgfx::LCD8bpp_BGRA2222::blitCopy**, **touchgfx::LCD8bpp_RGBA2222::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,          =0
                        Bitmap::BitmapFormat     sourceFormat ,        =0
                        const Rect &             source ,              =0
                        const Rect &             blitRect ,            =0
                        uint8_t                  alpha ,               =0
                        bool                     hasTransparentPixels  =0
                        )                        =0
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplemented by: **touchgfx::LCD16bpp::blitCopy**, **touchgfx::LCD16bppSerialFlash::blitCopy**, **touchgfx::LCD1bpp::blitCopy**, **touchgfx::LCD24bpp::blitCopy**, **touchgfx::LCD2bpp::blitCopy**, **touchgfx::LCD32bpp::blitCopy**, **touchgfx::LCD4bpp::blitCopy**, **touchgfx::LCD8bpp_ABGR2222::blitCopy**, **touchgfx::LCD8bpp_ARGB2222::blitCopy**, **touchgfx::LCD8bpp_BGRA2222::blitCopy**, **touchgfx::LCD8bpp_RGBA2222::blitCopy**

copyFrameBufferRegionToMemory

```
uint16_t
* copyFrameBufferRegionToMemory ( const Rect &  region ,
                                  const BitmapId  bitmapId =BITMAP_ANIMATION_STORAGE
                                  )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP*ANIMATION_STORAGE). Only the part specified with by parameter _region* is copied.

If *region* has negative x/y coordinates of if width/height exceeds those of the given bitmap, only the visible and legal part of the framebuffer is copied. The rest of the bitmap image is left untouched.

Parameters:

region	The part of the framebuffer to copy.
bitmapId	(Optional) The bitmap to store the data in. Default is to use Animation Storage.

Returns:

A pointer to the copy.

NOTE

There is only one instance of animation storage. The content of the bitmap data (or animation storage) outside the given region is left untouched.

See also:

blitCopy

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,  =0
                                                   const Rect &    absRegion ,  =0
                                                   const BitmapId bitmapId     =0
                                                   )                =0
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplemented by: **touchgfx::LCD16bpp::copyFrameBufferRegionToMemory**, **touchgfx::LCD16bppSerialFlash::copyFrameBufferRegionToMemory**, **touchgfx::LCD1bpp::copyFrameBufferRegionToMemory**, **touchgfx::LCD24bpp::copyFrameBufferRegionToMemory**, **touchgfx::LCD2bpp::copyFrameBufferRegionToMemory**,

touchgfx::LCD32bpp::copyFrameBufferRegionToMemory,
touchgfx::LCD4bpp::copyFrameBufferRegionToMemory,
touchgfx::LCD8bpp_ABGR2222::copyFrameBufferRegionToMemory,
touchgfx::LCD8bpp_ARGB2222::copyFrameBufferRegionToMemory,
touchgfx::LCD8bpp_BGRA2222::copyFrameBufferRegionToMemory,
touchgfx::LCD8bpp_RGBA2222::copyFrameBufferRegionToMemory

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,          =0
                                 int16_t        x ,               =0
                                 int16_t        y ,               =0
                                 const Rect &   rect ,            =0
                                 uint8_t        alpha =255,       =0
                                 bool           useOptimized =true =0
                                 )              =0
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplemented by: **touchgfx::LCD16bpp::drawPartialBitmap**,
touchgfx::LCD16bppSerialFlash::drawPartialBitmap, **touchgfx::LCD1bpp::drawPartialBitmap**,
touchgfx::LCD24bpp::drawPartialBitmap, **touchgfx::LCD2bpp::drawPartialBitmap**,
touchgfx::LCD32bpp::drawPartialBitmap, **touchgfx::LCD4bpp::drawPartialBitmap**,
touchgfx::LCD8bpp_ABGR2222::drawPartialBitmap,
touchgfx::LCD8bpp_ARGB2222::drawPartialBitmap,
touchgfx::LCD8bpp_BGRA2222::drawPartialBitmap,
touchgfx::LCD8bpp_RGBA2222::drawPartialBitmap

drawString

```
void drawString ( Rect                          widgetArea ,
                  const Rect &                  invalidatedArea ,
                  const StringVisuals &         stringVisuals ,
                  const Unicode::UnicodeChar *  format ,
                  ...
                  )
```

Draws the specified Unicode string.

Breaks line on newline.

Parameters:

widgetArea	The area covered by the drawing widget in absolute coordinates.
invalidatedArea	The (sub)region of the widget area to draw, expressed relative to the widget area. If the widgetArea is x=10, y=10, width=20, height=20 and invalidatedArea is x=5, y=5, width=6, height=6 the widgetArea drawn on the **LCD** is x=15, y=15, width=6, height=6.
stringVisuals	The string visuals (font, alignment, line space, color) with which to draw this string.
format	A pointer to a null-terminated text string with optional additional wildcard arguments.
...	Variable arguments providing additional information.

drawTextureMapTriangle

```
virtual void drawTextureMapTriangle ( const DrawingSurface &  dest ,
                                      const Point3D *         vertices ,
                                      const TextureSurface &  texture ,
                                      const Rect &            absoluteRect ,
                                      const Rect &            dirtyAreaAbsolute ,
                                      RenderingVariant        renderVariant ,
                                      uint8_t                 alpha =255,
                                      uint16_t                subDivisionSize =12
                                      )
```

Texture map triangle.

Draw a perspective correct texture mapped triangle. The vertices describes the surface, the x,y,z coordinates and the u,v coordinates of the texture. The texture contains the image data to be drawn The triangle line will be placed and clipped using the absolute and dirty rectangles The alpha will determine how the triangle should be alpha blended. The subDivisionSize will determine the size of the piecewise affine texture mapped portions of the triangle.

Parameters:

dest	The description of where the texture is drawn - can be used to issue a draw off screen.
vertices	The vertices of the triangle.
texture	The texture.
absoluteRect	The containing rectangle in absolute coordinates.
dirtyAreaAbsolute	The dirty area in absolute coordinates.
renderVariant	The render variant - includes the algorithm and the pixel format.
alpha	(Optional) the alpha. Default is 255 (solid).
subDivisionSize	(Optional) the size of the subdivisions of the scan line. Default is 12.

fillRect

```
virtual void fillRect ( const Rect & rect ,          =0
                        colortype    color ,         =0
                        uint8_t      alpha =255      =0
                        )                            =0
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplemented by: **touchgfx::LCD16bpp::fillRect**, **touchgfx::LCD16bppSerialFlash::fillRect**, **touchgfx::LCD1bpp::fillRect**, **touchgfx::LCD24bpp::fillRect**, **touchgfx::LCD2bpp::fillRect**, **touchgfx::LCD32bpp::fillRect**, **touchgfx::LCD4bpp::fillRect**, **touchgfx::LCD8bpp_ABGR2222::fillRect**, **touchgfx::LCD8bpp_ARGB2222::fillRect**, **touchgfx::LCD8bpp_BGRA2222::fillRect**, **touchgfx::LCD8bpp_RGBA2222::fillRect**

framebufferFormat

```
virtual Bitmap::BitmapFormat framebufferFormat (  ) const =0
```

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplemented by: touchgfx::LCD16bpp::framebufferFormat, touchgfx::LCD16bppSerialFlash::framebufferFormat, touchgfx::LCD1bpp::framebufferFormat, touchgfx::LCD24bpp::framebufferFormat, touchgfx::LCD2bpp::framebufferFormat, touchgfx::LCD32bpp::framebufferFormat, touchgfx::LCD4bpp::framebufferFormat, touchgfx::LCD8bpp_ABGR2222::framebufferFormat, touchgfx::LCD8bpp_ARGB2222::framebufferFormat, touchgfx::LCD8bpp_BGRA2222::framebufferFormat, touchgfx::LCD8bpp_RGBA2222::framebufferFormat

framebufferStride

virtual uint16_t framebufferStride ( ) const =0

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplemented by: touchgfx::LCD16bpp::framebufferStride, touchgfx::LCD16bppSerialFlash::framebufferStride, touchgfx::LCD1bpp::framebufferStride, touchgfx::LCD24bpp::framebufferStride, touchgfx::LCD2bpp::framebufferStride, touchgfx::LCD32bpp::framebufferStride, touchgfx::LCD4bpp::framebufferStride, touchgfx::LCD8bpp_ABGR2222::framebufferStride, touchgfx::LCD8bpp_ARGB2222::framebufferStride, touchgfx::LCD8bpp_BGRA2222::framebufferStride, touchgfx::LCD8bpp_RGBA2222::framebufferStride

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplemented by: touchgfx::LCD16bpp::getBlueColor, touchgfx::LCD16bppSerialFlash::getBlueColor, touchgfx::LCD1bpp::getBlueColor, touchgfx::LCD24bpp::getBlueColor, touchgfx::LCD2bpp::getBlueColor, touchgfx::LCD32bpp::getBlueColor, touchgfx::LCD4bpp::getBlueColor, touchgfx::LCD8bpp_ABGR2222::getBlueColor, touchgfx::LCD8bpp_ARGB2222::getBlueColor, touchgfx::LCD8bpp_BGRA2222::getBlueColor, touchgfx::LCD8bpp_RGBA2222::getBlueColor

getColorFrom24BitRGB

```
virtual colortype getColorFrom24BitRGB ( uint8_t red ,      const =0
                                         uint8_t green ,    const =0
                                         uint8_t blue       const =0
                                         )          const =0
```

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplemented by: touchgfx::LCD16bpp::getColorFrom24BitRGB, touchgfx::LCD16bppSerialFlash::getColorFrom24BitRGB, touchgfx::LCD1bpp::getColorFrom24BitRGB, touchgfx::LCD24bpp::getColorFrom24BitRGB, touchgfx::LCD2bpp::getColorFrom24BitRGB, touchgfx::LCD32bpp::getColorFrom24BitRGB, touchgfx::LCD4bpp::getColorFrom24BitRGB, touchgfx::LCD8bpp_ABGR2222::getColorFrom24BitRGB, touchgfx::LCD8bpp_ARGB2222::getColorFrom24BitRGB, touchgfx::LCD8bpp_BGRA2222::getColorFrom24BitRGB, touchgfx::LCD8bpp_RGBA2222::getColorFrom24BitRGB

getDefaultColor

colortype getDefaultColor ( ) const

Gets default color previously set using setDefaultColor.

Returns:

The default color.

See also:

setDefaultColor

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The green part of the color.

Reimplemented by: **touchgfx::LCD16bpp::getGreenColor**, **touchgfx::LCD16bppSerialFlash::getGreenColor**, **touchgfx::LCD1bpp::getGreenColor**, **touchgfx::LCD24bpp::getGreenColor**, **touchgfx::LCD2bpp::getGreenColor**, **touchgfx::LCD32bpp::getGreenColor**, **touchgfx::LCD4bpp::getGreenColor**, **touchgfx::LCD8bpp_ABGR2222::getGreenColor**, **touchgfx::LCD8bpp_ARGB2222::getGreenColor**, **touchgfx::LCD8bpp_BGRA2222::getGreenColor**, **touchgfx::LCD8bpp_RGBA2222::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplemented by: **touchgfx::LCD16bpp::getRedColor**, **touchgfx::LCD16bppSerialFlash::getRedColor**, **touchgfx::LCD1bpp::getRedColor**, **touchgfx::LCD24bpp::getRedColor**, **touchgfx::LCD2bpp::getRedColor**, **touchgfx::LCD32bpp::getRedColor**, **touchgfx::LCD4bpp::getRedColor**, **touchgfx::LCD8bpp_ABGR2222::getRedColor**, **touchgfx::LCD8bpp_ARGB2222::getRedColor**, **touchgfx::LCD8bpp_BGRA2222::getRedColor**, **touchgfx::LCD8bpp_RGBA2222::getRedColor**

setDefaultColor

void setDefaultColor (colortype color)

Sets default color as used by alpha level only bitmap formats, e.g.

A4. The default color, if no color is set, is black.

Parameters:

color The color.

See also:

getDefaultColor

~LCD

virtual ~LCD ( )

Finalizes an instance of the **LCD** class.

div255

static FORCE_INLINE_FUNCTION uint8_t div255 (uint16_t num)

Approximates an integer division of a 16bit value by 255.

Divides numerator num (e.g. the sum resulting from an alpha-blending operation) by 255.

Parameters:

num The numerator to divide by 255.

Returns:

The result of a division by 255.

div255g

static FORCE_INLINE_FUNCTION uint32_t div255g (uint32_t pixelxAlpha)

Divides the green component of pixelxAlpha by 255.

Parameters:

pixelxAlpha The green component of a 32bit ARGB pixel multiplied by an alpha factor.

Returns:

pixelxAlpha with its green component divided by 255.

div255rb

static FORCE_INLINE_FUNCTION uint32_t div255rb (uint32_t pixelxAlpha)

Divides the red and blue components of pixelxAlpha by 255.

Parameters:

pixelxAlpha The red and blue components of a 32bit ARGB pixel multiplied by an alpha factor.

Returns:

pixelxAlpha with its red and blue components divided by 255.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *   wbuf16 ,      =0
                         Rect         widgetArea ,  =0
                         int16_t      x ,           =0
                         int16_t      y ,           =0
                         uint16_t     offsetX ,     =0
                         uint16_t     offsetY ,     =0
```

```
                                const Rect &          invalidatedArea ,  =0
                                const GlyphNode * glyph ,  =0
                                const uint8_t *       glyphData ,  =0
                                uint8_t               byteAlignRow ,  =0
                                colortype             color ,  =0
                                uint8_t               bitsPerPixel ,  =0
                                uint8_t               alpha ,  =0
                                TextRotation          rotation  =0
                                )                     =0
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplemented by: **touchgfx::LCD16bpp::drawGlyph**, **touchgfx::LCD16bppSerialFlash::drawGlyph**, **touchgfx::LCD1bpp::drawGlyph**, **touchgfx::LCD24bpp::drawGlyph**, **touchgfx::LCD2bpp::drawGlyph**, **touchgfx::LCD32bpp::drawGlyph**, **touchgfx::LCD4bpp::drawGlyph**, **touchgfx::LCD8bpp_ABGR2222::drawGlyph**, **touchgfx::LCD8bpp_ARGB2222::drawGlyph**, **touchgfx::LCD8bpp_BGRA2222::drawGlyph**, **touchgfx::LCD8bpp_RGBA2222::drawGlyph**

drawStringLTR

```
void drawStringLTR ( const Rect &                  widgetArea ,
                     const Rect &                  invalidatedArea ,
                     const StringVisuals &         visuals ,
                     const Unicode::UnicodeChar * format ,
                     va_list                       pArg
```

```
)
```

Draws the specified Unicode string.

Breaks line on newline. The string is assumed to contain only Latin characters written left-to-right.

Parameters:

widgetArea	The area covered by the drawing widget in absolute coordinates.
invalidatedArea	The (sub)region of the widget area to draw, expressed relative to the widget area. If the widgetArea is x=10, y=10, width=20, height=20 and invalidatedArea is x=5, y=5, width=6, height=6 the widgetArea drawn on the **LCD** is x=15, y=15, width=6, height=6.
visuals	The string visuals (font, alignment, line space, color) with which to draw this string.
format	A pointer to a null-terminated text string with optional additional wildcard arguments.
pArg	Variable arguments providing additional information.

See also:

drawString

drawStringRTL

```
void drawStringRTL ( const Rect &                widgetArea ,
                     const Rect &                invalidatedArea ,
                     const StringVisuals &       visuals ,
                     const Unicode::UnicodeChar * format ,
                     va_list                     pArg
                     )
```

Draws the specified Unicode string.

Breaks line on newline. The string can be either right-to-left or left-to-right and may contain sequences of Arabic / Hebrew and Latin characters.

Parameters:

widgetArea	The area covered by the drawing widget in absolute coordinates.
invalidatedArea	The (sub)region of the widget area to draw, expressed relative to the widget area. If the widgetArea is x=10, y=10, width=20, height=20 and invalidatedArea is x=5, y=5, width=6, height=6 the widgetArea drawn on the **LCD** is x=15, y=15, width=6, height=6.
visuals	The string visuals (font, alignment, line space, color) with which to draw this string.
format	A pointer to a null-terminated text string with optional additional wildcard arguments.
pArg	Variable arguments providing additional information.

See also:

drawString

drawTextureMapScanLine

```
virtual void drawTextureMapScanLine ( const DrawingSurface &     dest ,
                                      const Gradients &          gradients ,
                                      const Edge *               leftEdge ,
                                      const Edge *               rightEdge ,
                                      const TextureSurface &     texture ,
                                      const Rect &               absoluteRect ,
                                      const Rect &               dirtyAreaAbsolute ,
                                      RenderingVariant           renderVariant ,
                                      uint8_t                    alpha ,
                                      uint16_t                   subDivisionSize
                                      )
```

Draw scan line.

Draw one horizontal line of the texture map on screen. The scan line will be drawn using perspective correct texture mapping. The appearance of the line is determined by the left and right edge and the gradients structure. The edges contain the information about the x,y,z coordinates of the left and right side respectively and also information about the u,v coordinates of the texture map used. The gradients structure contains information about how to interpolate all the values across the scan line. The data drawn should be present in the texture argument.

The scan line will be drawn using the additional arguments. The scan line will be placed and clipped using the absolute and dirty rectangles The alpha will determine how the scan line should be alpha blended. The subDivisionSize will determine the size of the piecewise affine texture mapped lines.

Parameters:

dest	The description of where the texture is drawn - can be used to issue a draw off screen.
gradients	The gradients using in interpolation across the scan line.
leftEdge	The left edge of the scan line.
rightEdge	The right edge of the scan line.
texture	The texture.
absoluteRect	The containing rectangle in absolute coordinates.
dirtyAreaAbsolute	The dirty area in absolute coordinates.
renderVariant	The render variant - includes the algorithm and the pixel format.
alpha	The alpha.

subDivisionSize	The size of the subdivisions of the scan line. A value of 1 will give a completely perspective correct texture mapped scan line. A large value will give an affine texture mapped scan line.

Reimplemented by: **touchgfx::LCD1bpp::drawTextureMapScanLine**

getTextureMapperDrawScanLine

virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine (const TextureSurface & texture , RenderingVariant renderVariant , uint8_t alpha)

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplemented by: **touchgfx::LCD16bpp::getTextureMapperDrawScanLine**, **touchgfx::LCD16bppSerialFlash::getTextureMapperDrawScanLine**, **touchgfx::LCD24bpp::getTextureMapperDrawScanLine**, **touchgfx::LCD2bpp::getTextureMapperDrawScanLine**, **touchgfx::LCD32bpp::getTextureMapperDrawScanLine**, **touchgfx::LCD4bpp::getTextureMapperDrawScanLine**, **touchgfx::LCD8bpp_ABGR2222::getTextureMapperDrawScanLine**, **touchgfx::LCD8bpp_ARGB2222::getTextureMapperDrawScanLine**, **touchgfx::LCD8bpp_BGRA2222::getTextureMapperDrawScanLine**, **touchgfx::LCD8bpp_RGBA2222::getTextureMapperDrawScanLine**

getAlphaFromA4

static FORCE_INLINE_FUNCTION uint8_t getAlphaFromA4 (const uint16_t * data , uint32_t offset)

Gets alpha from A4 image at given offset.

The value is scaled up from range 0-15 to 0- 255.

Parameters:

data	A pointer to the start of the A4 data.
offset	The offset into the A4 image.

Returns:

The alpha from A4 (0-255).

getNumLines

```
static uint16_t getNumLines ( TextProvider &  textProvider ,
                              WideTextAction  wideTextAction ,
                              TextDirection   textDirection ,
                              const Font *    font ,
                              int16_t         width
                              )
```

Gets number of lines for a given text taking word wrap into consideration.

The font and width are required to find the number of lines in case word wrap is true.

Parameters:

textProvider	The text provider.
wideTextAction	The wide text action in case lines are longer than the width of the text area.
textDirection	The text direction (LTR or RTL).
font	The font.
width	The width.

Returns:

The number lines.

realX

```
static int realX ( const Rect &  widgetArea ,
                   int16_t       x ,
                   int16_t       y ,
                   TextRotation  rotation
                   )
```

Find the real, absolute x coordinate of a point inside a widget with regards to rotation.

Parameters:

widgetArea	The widget containing the point.
x	The x coordinate.
y	The y coordinate.
rotation	Rotation to perform.

Returns:

The absolute x coordinate after applying appropriate rotation.

realY

```
static int realY ( const Rect & widgetArea ,
                   int16_t      x ,
                   int16_t      y ,
                   TextRotation rotation
                   )
```

Find the real, absolute y coordinate of a point inside a widget with regards to rotation.

Parameters:

widgetArea	The widget containing the point.
x	The x coordinate.
y	The y coordinate.
rotation	Rotation to perform.

Returns:

The absolute y coordinate after applying appropriate rotation.

rotateRect

```
static void rotateRect ( Rect &       rect ,
                         const Rect & canvas ,
                         const TextRotation rotation
                         )
```

Rotate a rectangle inside another rectangle.

Parameters:

rect	The rectangle to rotate.
canvas	The rectangle containing the rect to rotate.
rotation	Rotation to perform on rect.

stringWidth

```
static uint16_t stringWidth ( TextProvider &  textProvider ,
                              const Font &    font ,
                              const int       numChars ,
                              TextDirection   textDirection
                              )
```

Find string width of the given number of ligatures read from the given TextProvider.

After the introduction of Arabic, Thai, Hindi and other languages, ligatures are counted instead of characters. For Latin languages, number of characters equal number of ligatures.

Parameters:

textProvider	The text provider.
font	The font.
numChars	Number of characters (ligatures).
textDirection	The text direction.

Returns:

An int16_t.

Protected Attributes Documentation

defaultColor

colortype defaultColor

Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.

newLine

const uint16_t newLine = 10

NewLine value.

Version: 4.16

LCD16bpp

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.

void **enableTextureMapperAll**()

Enables the texture mappers for all image formats.

void **enableTextureMapperARGB8888**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_BilinearInterpolation**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_NearestNeighbor**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_BilinearInterpolation**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_NearestNeighbor**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_RGB565**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB565_BilinearInterpolation**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB565_NearestNeighbor**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB888**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_BilinearInterpolation**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_NearestNeighbor**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperRGB565**()

Enables the texture mappers for RGB565 image format.

void **enableTextureMapperRGB565_NonOpaque_BilinearInterpolation**()

Enables the texture mappers for NonOpaque RGB565 image format.

void **enableTextureMapperRGB565_NonOpaque_NearestNeighbor**()

	Enables the texture mappers for NonOpaque RGB565 image format.
void	**enableTextureMapperRGB565_Opaque_BilinearInterpolation**()
	Enables the texture mappers for Opaque RGB565 image format.
void	**enableTextureMapperRGB565_Opaque_NearestNeighbor**()
	Enables the texture mappers for Opaque RGB565 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD16bpp()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.

FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea, int16_t x, int16_t y, uint16_t offsetX, uint16_t offsetY, const* ***Rect*** *& invalidatedArea, const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual **DrawTextureMapScanLineBase** *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyAlphaPerPixel**(const uint16_t *sourceData, const uint8_t* alphaData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.
void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.
void	**blitCopyL8**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.
void	**blitCopyL8_ARGB8888**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.
void	**blitCopyL8_RGB565**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)

	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB565 is not supported by the DMA a software blend is performed.
void	**blitCopyL8_RGB888**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB888 is not supported by the DMA a software blend is performed.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.
virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.

colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.

int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

virtual void blitCopy (const uint16_t * sourceData ,
const Rect & source ,

```
                    const Rect &    blitRect ,
                    uint8_t         alpha ,
                    bool            hasTransparentPixels
                    )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *         sourceData ,
                        Bitmap::BitmapFormat    sourceFormat ,
                        const Rect &            source ,
                        const Rect &            blitRect ,
                        uint8_t                 alpha ,
                        bool                    hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.

alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
```

```
                                uint8_t         alpha =255,
                                bool            useOptimized =true
                                )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation,
enableTextureMapperARGB8888_NearestNeighbor

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

enableTextureMapperL8_ARGB8888

void enableTextureMapperL8_ARGB8888 ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_ARGB8888_BilinearInterpolation,
enableTextureMapperL8_ARGB8888_NearestNeighbor

enableTextureMapperL8_ARGB8888_BilinearInterpolation

void enableTextureMapperL8_ARGB8888_BilinearInterpolation ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_NearestNeighbor**

enableTextureMapperL8_ARGB8888_NearestNeighbor

void enableTextureMapperL8_ARGB8888_NearestNeighbor ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_BilinearInterpolation**

enableTextureMapperL8_RGB565

void enableTextureMapperL8_RGB565 ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_RGB565_BilinearInterpolation,
enableTextureMapperL8_RGB565_NearestNeighbor

enableTextureMapperL8_RGB565_BilinearInterpolation

void enableTextureMapperL8_RGB565_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB565, **enableTextureMapperL8_RGB565_NearestNeighbor**

enableTextureMapperL8_RGB565_NearestNeighbor

void enableTextureMapperL8_RGB565_NearestNeighbor ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB565, **enableTextureMapperL8_RGB565_BilinearInterpolation**

enableTextureMapperL8_RGB888

void enableTextureMapperL8_RGB888 ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_RGB888_BilinearInterpolation,
enableTextureMapperL8_RGB888_NearestNeighbor

enableTextureMapperL8_RGB888_BilinearInterpolation

void enableTextureMapperL8_RGB888_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_NearestNeighbor**

enableTextureMapperL8_RGB888_NearestNeighbor

void enableTextureMapperL8_RGB888_NearestNeighbor ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_BilinearInterpolation**

enableTextureMapperRGB565

void enableTextureMapperRGB565 ( )

Enables the texture mappers for RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperRGB565_Opaque_BilinearInterpolation,
enableTextureMapperRGB565_Opaque_NearestNeighbor,
enableTextureMapperRGB565_NonOpaque_BilinearInterpolation,
enableTextureMapperRGB565_NonOpaque_NearestNeighbor

enableTextureMapperRGB565_NonOpaque_BilinearInterpolation

void enableTextureMapperRGB565_NonOpaque_BilinearInterpolation ( )

Enables the texture mappers for NonOpaque RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB565

enableTextureMapperRGB565_NonOpaque_NearestNeighbor

void enableTextureMapperRGB565_NonOpaque_NearestNeighbor ( )

Enables the texture mappers for NonOpaque RGB565 image format.

This allows drawing RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB565

enableTextureMapperRGB565_Opaque_BilinearInterpolation

void enableTextureMapperRGB565_Opaque_BilinearInterpolation ( )

Enables the texture mappers for Opaque RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB565

enableTextureMapperRGB565_Opaque_NearestNeighbor

void enableTextureMapperRGB565_Opaque_NearestNeighbor ( )

Enables the texture mappers for Opaque RGB565 image format.

This allows drawing RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB565

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

```
virtual colortype getColorFrom24BitRGB ( uint8_t red ,   const
                                         uint8_t green , const
                                         uint8_t blue  const
                                         )       const
```

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

```
virtual uint8_t getGreenColor ( colortype color )
```

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

```
virtual uint8_t getRedColor ( colortype color )
```

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD16bpp

LCD16bpp ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
```

	const uint8_t *	glyphData ,
	uint8_t	byteAlignRow ,
	colortype	color ,
	uint8_t	bitsPerPixel ,
	uint8_t	alpha ,
	TextRotation	rotation
	)	

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine	(	const TextureSurface &	texture ,
		RenderingVariant	renderVariant ,
		uint8_t	alpha
		)	

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: touchgfx::LCD::getTextureMapperDrawScanLine

blitCopyAlphaPerPixel

```
static void blitCopyAlphaPerPixel ( const uint16_t * sourceData ,
                                    const uint8_t *  alphaData ,
                                    const Rect &     source ,
                                    const Rect &     blitRect ,
                                    uint8_t          alpha
                                    )
```

Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.

Always performs a software blend.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 16- bits RGB565 values.
alphaData	The alpha channel array pointer (points to the beginning of the data)
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyARGB8888

```
static void blitCopyARGB8888 ( const uint32_t * sourceData ,
                               const Rect &     source ,
                               const Rect &     blitRect ,
                               uint8_t          alpha
                               )
```

Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.

If ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8

```
static void blitCopyL8 ( const uint8_t * sourceData ,
                         const uint8_t * clutData ,
                         const Rect &    source ,
                         const Rect &    blitRect ,
                         uint8_t         alpha
                         )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_ARGB8888

```
static void blitCopyL8_ARGB8888 ( const uint8_t * sourceData ,
                                  const uint8_t * clutData ,
                                  const Rect &    source ,
                                  const Rect &    blitRect ,
                                  uint8_t         alpha
                                  )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (ARGB8888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_RGB565

```
static void blitCopyL8_RGB565 ( const uint8_t * sourceData ,
                                const uint8_t * clutData ,
                                const Rect &    source ,
                                const Rect &    blitRect ,
```

```
                                uint8_t          alpha
                                )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB565 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer points to the beginning of the CLUT color format and size data followed by colors entries stored as 16- bits (RGB565) format. If the source have per pixel alpha channel, then alpha channel data will be following the clut entries data.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_RGB888

```
static void blitCopyL8_RGB888 ( const uint8_t * sourceData ,
                                const uint8_t * clutData ,
                                const Rect &    source ,
                                const Rect &    blitRect ,
                                uint8_t         alpha
                                )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (ARGB8888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool         rotatedDisplay ,
                      TextRotation textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?

textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation  textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD16bppSerialFlash

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.

void **enableTextureMapperAll**()

Enables the texture mappers for all image formats.

void **enableTextureMapperARGB8888**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_BilinearInterpolation**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_NearestNeighbor**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_BilinearInterpolation**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_NearestNeighbor**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_RGB565**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB565_BilinearInterpolation**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB565_NearestNeighbor**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB888**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_BilinearInterpolation**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_NearestNeighbor**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperRGB565**()

Enables the texture mappers for RGB565 image format.

void **enableTextureMapperRGB565_NonOpaque_BilinearInterpolation**()

Enables the texture mappers for NonOpaque RGB565 image format.

void **enableTextureMapperRGB565_NonOpaque_NearestNeighbor**()

	Enables the texture mappers for NonOpaque RGB565 image format.
void	**enableTextureMapperRGB565_Opaque_BilinearInterpolation**()
	Enables the texture mappers for Opaque RGB565 image format.
void	**enableTextureMapperRGB565_Opaque_NearestNeighbor**()
	Enables the texture mappers for Opaque RGB565 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD16bppSerialFlash(**FlashDataReader** & flashReader)
	Creates a **LCD16bppSerialFlash** object.
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.

FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.
void	**blitCopyL8**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.
void	**blitCopyL8_ARGB8888**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.
void	**blitCopyL8_RGB565**(const uint8_t *sourceData, const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB565 is not supported by the DMA a software blend is performed.
virtual void	**drawGlyph**(uint16_t *wbuf16, **Rect** widgetArea, int16_t x, int16_t y, uint16_t offsetX, uint16_t offsetY, const **Rect** & invalidatedArea, const ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)

	Find out how much to advance in the display buffer to get to the next pixel.

Protected Attributes

FlashDataReader &	**flashReader**
	Flash reader. Used by routines to read pixel data from the flash.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.
virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()

	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)

	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
                        bool                     hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of

alpha. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation,
enableTextureMapperARGB8888_NearestNeighbor

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

enableTextureMapperL8_ARGB8888

void enableTextureMapperL8_ARGB8888 ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_ARGB8888_BilinearInterpolation,
enableTextureMapperL8_ARGB8888_NearestNeighbor

enableTextureMapperL8_ARGB8888_BilinearInterpolation

void enableTextureMapperL8_ARGB8888_BilinearInterpolation ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_NearestNeighbor**

enableTextureMapperL8_ARGB8888_NearestNeighbor

void enableTextureMapperL8_ARGB8888_NearestNeighbor ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_BilinearInterpolation**

enableTextureMapperL8_RGB565

void enableTextureMapperL8_RGB565 ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_RGB565_BilinearInterpolation,
enableTextureMapperL8_RGB565_NearestNeighbor

enableTextureMapperL8_RGB565_BilinearInterpolation

void enableTextureMapperL8_RGB565_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB565, **enableTextureMapperL8_RGB565_NearestNeighbor**

enableTextureMapperL8_RGB565_NearestNeighbor

void enableTextureMapperL8_RGB565_NearestNeighbor ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB565, **enableTextureMapperL8_RGB565_BilinearInterpolation**

enableTextureMapperL8_RGB888

void enableTextureMapperL8_RGB888 ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation and NearestNeighbor algorithms.

See also:

enableTextureMapperL8_RGB888_BilinearInterpolation,
enableTextureMapperL8_RGB888_NearestNeighbor

enableTextureMapperL8_RGB888_BilinearInterpolation

void enableTextureMapperL8_RGB888_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_NearestNeighbor**

enableTextureMapperL8_RGB888_NearestNeighbor

void enableTextureMapperL8_RGB888_NearestNeighbor ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_BilinearInterpolation**

enableTextureMapperRGB565

void enableTextureMapperRGB565 ( )

Enables the texture mappers for RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperRGB565_Opaque_BilinearInterpolation,
enableTextureMapperRGB565_Opaque_NearestNeighbor,
enableTextureMapperRGB565_NonOpaque_BilinearInterpolation,
enableTextureMapperRGB565_NonOpaque_NearestNeighbor

enableTextureMapperRGB565_NonOpaque_BilinearInterpolation

void enableTextureMapperRGB565_NonOpaque_BilinearInterpolation ( )

Enables the texture mappers for NonOpaque RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB565

enableTextureMapperRGB565_NonOpaque_NearestNeighbor

void enableTextureMapperRGB565_NonOpaque_NearestNeighbor ( )

Enables the texture mappers for NonOpaque RGB565 image format.

This allows drawing RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB565

enableTextureMapperRGB565_Opaque_BilinearInterpolation

void enableTextureMapperRGB565_Opaque_BilinearInterpolation ( )

Enables the texture mappers for Opaque RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB565

enableTextureMapperRGB565_Opaque_NearestNeighbor

void enableTextureMapperRGB565_Opaque_NearestNeighbor ( )

Enables the texture mappers for Opaque RGB565 image format.

This allows drawing RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB565

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

virtual colortype getColorFrom24BitRGB (uint8_t red , const
uint8_t green , const
uint8_t blue const
) const

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16

or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD16bppSerialFlash

LCD16bppSerialFlash (FlashDataReader & flashReader)

Creates a **LCD16bppSerialFlash** object.

The **FlashDataReader** object is used to fetch data from the external flash.

Parameters:

flashReader Reference to a **FlashDataReader** object.

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

blitCopyARGB8888

```
void blitCopyARGB8888 ( const uint32_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha
                        )
```

Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.

If ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8

```
void blitCopyL8 ( const uint8_t * sourceData ,
                  const uint8_t * clutData ,
                  const Rect &    source ,
                  const Rect &    blitRect ,
                  uint8_t         alpha
                  )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_ARGB8888

```
void blitCopyL8_ARGB8888 ( const uint8_t * sourceData ,
                           const uint8_t * clutData ,
                           const Rect &    source ,
                           const Rect &    blitRect ,
                           uint8_t         alpha
                           )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (ARGB8888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_RGB565

```
void blitCopyL8_RGB565 ( const uint8_t * sourceData ,
                         const uint8_t * clutData ,
                         const Rect &    source ,
                         const Rect &    blitRect ,
                         uint8_t         alpha
                         )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB565 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer points to the beginning of the CLUT color format and size data followed by colors entries stored as 16- bits (RGB565) format. If the source have per pixel alpha channel, then alpha channel data will be following the clut entries data.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine (const TextureSurface & texture , RenderingVariant renderVariant , uint8_t alpha)

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

nextLine

static int nextLine (bool rotatedDisplay , TextRotation textRotation)

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation  textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Protected Attributes Documentation

flashReader

FlashDataReader & flashReader

Flash reader. Used by routines to read pixel data from the flash.

Version: 4.16

LCD16DebugPrinter

The class LCD16DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 16bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD1bpp

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 1 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.

virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**blitCopyRLE**(const uint16_t * _sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a run-length encoded2D source-array to the framebuffer if alpha > zero.
void	**copyRect**(const uint8_t *srcAddress, uint16_t srcStride, uint8_t srcPixelOffset, uint8_t* RESTRICT dstAddress, uint16_t dstStride, uint8_t dstPixelOffset, uint16_t width, uint16_t height) const
	Copies a rectangular area from the framebuffer til a givene memory address, which is typically in the animation storage or a dynamic bitmap.
virtual void	**drawGlyph**(uint16_t *wbuf16, **Rect** widgetArea, int16_t x, int16_t y, uint16_t offsetX, uint16_t offsetY, const **Rect** & invalidatedArea, const **GlyphNode** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const **Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
void	**fillMemory**(void *RESTRICT dst, **colortype** color, uint16_t bytesToFill)
	Fill memory efficiently.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**

Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.
virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.

void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual **DrawTextureMapScanLineBase** *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.

const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: touchgfx::LCD::blitCopy

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
                        bool                     hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: touchgfx::LCD::blitCopy

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId  bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.

x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

Currently texture mapping is not supported on 1bpp displays, so this function does not do anything. It is merely included to allow function **enableTextureMapperAll()** to be called on any subclass of **LCD**.

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

virtual colortype getColorFrom24BitRGB (uint8_t red , const
uint8_t green , const
uint8_t blue const

) const

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

blitCopyRLE

```
virtual void blitCopyRLE ( const uint16_t * _sourceData ,
                           const Rect &      source ,
                           const Rect &      blitRect ,
                           uint8_t           alpha
```

```
                                )
```

Blits a run-length encoded2D source-array to the framebuffer if alpha > zero.

Parameters:

_sourceData	The source-array pointer (points to the beginning of the data). Data stored in RLE format, where each byte indicates number of pixels with certain color, alternating between black and white. First byte represents black.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending (0 = invisible, otherwise solid).

copyRect

```
void copyRect ( const uint8_t *     srcAddress ,     const
                uint16_t            srcStride ,      const
                uint8_t             srcPixelOffset , const
                uint8_t *RESTRICT   dstAddress ,     const
                uint16_t            dstStride ,      const
                uint8_t             dstPixelOffset , const
                uint16_t            width ,          const
                uint16_t            height           const
                )                   const
```

Copies a rectangular area from the framebuffer til a givene memory address, which is typically in the animation storage or a dynamic bitmap.

Parameters:

srcAddress	Source address (byte address).
srcStride	Source stride (number of bytes to advance to next line).
srcPixelOffset	Source pixel offset (first pixel in first source byte).
dstAddress	If destination address (byte address).
dstStride	Destination stride (number of bytes to advance to next line).
dstPixelOffset	Destination pixel offset (first pixel in destination byte).
width	The width of area (in pixels).
height	The height of area (in pixels).

drawGlyph

```
virtual void drawGlyph ( uint16_t *     wbuf16 ,
                         Rect           widgetArea ,
                         int16_t        x ,
                         int16_t        y ,
```

```
                                uint16_t            offsetX ,
                                uint16_t            offsetY ,
                                const Rect &        invalidatedArea ,
                                const GlyphNode *   glyph ,
                                const uint8_t *     glyphData ,
                                uint8_t             byteAlignRow ,
                                colortype           color ,
                                uint8_t             bitsPerPixel ,
                                uint8_t             alpha ,
                                TextRotation        rotation
                                )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

drawTextureMapScanLine

```
virtual void drawTextureMapScanLine ( const DrawingSurface &  dest ,
                                      const Gradients &       gradients ,
                                      const Edge *            leftEdge ,
                                      const Edge *            rightEdge ,
                                      const TextureSurface &  texture ,
                                      const Rect &            absoluteRect ,
                                      const Rect &            dirtyAreaAbsolute ,
                                      RenderingVariant        renderVariant ,
```

```
                                uint8_t         alpha ,
                                uint16_t        subDivisionSize
                                )
```

Draw scan line.

Draw one horizontal line of the texture map on screen. The scan line will be drawn using perspective correct texture mapping. The appearance of the line is determined by the left and right edge and the gradients structure. The edges contain the information about the x,y,z coordinates of the left and right side respectively and also information about the u,v coordinates of the texture map used. The gradients structure contains information about how to interpolate all the values across the scan line. The data drawn should be present in the texture argument.

The scan line will be drawn using the additional arguments. The scan line will be placed and clipped using the absolute and dirty rectangles The alpha will determine how the scan line should be alpha blended. The subDivisionSize will determine the size of the piecewise affine texture mapped lines.

Parameters:

dest	The description of where the texture is drawn - can be used to issue a draw off screen.
gradients	The gradients using in interpolation across the scan line.
leftEdge	The left edge of the scan line.
rightEdge	The right edge of the scan line.
texture	The texture.
absoluteRect	The containing rectangle in absolute coordinates.
dirtyAreaAbsolute	The dirty area in absolute coordinates.
renderVariant	The render variant - includes the algorithm and the pixel format.
alpha	The alpha.
subDivisionSize	The size of the subdivisions of the scan line. A value of 1 will give a completely perspective correct texture mapped scan line. A large value will give an affine texture mapped scan line.

Reimplements: **touchgfx::LCD::drawTextureMapScanLine**

fillMemory

```
static void fillMemory ( void *RESTRICT dst ,
                         colortype      color ,
                         uint16_t       bytesToFill
                         )
```

Fill memory efficiently.

Try to get 32bit aligned or 16bit aligned and then copy as quickly as possible.

Parameters:

dst	Pointer to memory to fill.
color	**Color** to write to memory, either 0 => 0x00000000 or 1 => 0xFFFFFFFF.
bytesToFill	Number of bytes to fill.

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation  textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation  textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD1DebugPrinter

The class LCD1DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 24bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD24bpp

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.

void **enableTextureMapperA4_BilinearInterpolation**()

Enables the texture mappers for A4 image format.

void **enableTextureMapperA4_NearestNeighbor**()

Enables the texture mappers for A4 image format.

void **enableTextureMapperAll**()

Enables the texture mappers for all image formats.

void **enableTextureMapperARGB8888**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_BilinearInterpolation**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_NearestNeighbor**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_BilinearInterpolation**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_NearestNeighbor**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_RGB888**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_BilinearInterpolation**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_NearestNeighbor**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperRGB888**()

Enables the texture mappers for RGB888 image format.

void **enableTextureMapperRGB888_BilinearInterpolation**()

Enables the texture mappers for RGB888 image format.

void **enableTextureMapperRGB888_NearestNeighbor**()

	Enables the texture mappers for RGB888 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD24bpp()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Gets color from RGB.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** & *invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.
void	**blitCopyL8**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.
void	**blitCopyL8_ARGB8888**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.
void	**blitCopyL8_RGB888**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB888 is not supported by the DMA a software blend is performed.
void	**blitCopyRGB565**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)

	Blits a 2D source-array to the framebuffer.

int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.

int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.

virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.

colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.

void	**setDefaultColor**(**colortype** color)

	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.

int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
                        bool                     hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: touchgfx::LCD::copyFrameBufferRegionToMemory

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: touchgfx::LCD::drawPartialBitmap

enableTextureMapperA4

```
void enableTextureMapperA4 (  )
```

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation,
enableTextureMapperARGB8888_NearestNeighbor

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

enableTextureMapperL8_ARGB8888

void enableTextureMapperL8_ARGB8888 ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_ARGB8888_BilinearInterpolation,
enableTextureMapperL8_ARGB8888_NearestNeighbor

enableTextureMapperL8_ARGB8888_BilinearInterpolation

void enableTextureMapperL8_ARGB8888_BilinearInterpolation ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_NearestNeighbor**

enableTextureMapperL8_ARGB8888_NearestNeighbor

void enableTextureMapperL8_ARGB8888_NearestNeighbor ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_ARGB8888,
enableTextureMapperL8_ARGB8888_BilinearInterpolation

enableTextureMapperL8_RGB888

void enableTextureMapperL8_RGB888 ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_RGB888_BilinearInterpolation,
enableTextureMapperL8_RGB888_NearestNeighbor

enableTextureMapperL8_RGB888_BilinearInterpolation

void enableTextureMapperL8_RGB888_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_NearestNeighbor**

enableTextureMapperL8_RGB888_NearestNeighbor

void enableTextureMapperL8_RGB888_NearestNeighbor ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_BilinearInterpolation**

enableTextureMapperRGB888

void enableTextureMapperRGB888 ( )

Enables the texture mappers for RGB888 image format.

This allows drawing RGB888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperRGB888_BilinearInterpolation,
enableTextureMapperRGB888_NearestNeighbor

enableTextureMapperRGB888_BilinearInterpolation

void enableTextureMapperRGB888_BilinearInterpolation ( )

Enables the texture mappers for RGB888 image format.

This allows drawing RGB888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB888, **enableTextureMapperRGB888_NearestNeighbor**

enableTextureMapperRGB888_NearestNeighbor

void enableTextureMapperRGB888_NearestNeighbor ( )

Enables the texture mappers for RGB888 image format.

This allows drawing RGB888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB888, **enableTextureMapperRGB888_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

```
virtual Bitmap::BitmapFormat framebufferFormat (  ) const
```

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

```
virtual uint16_t framebufferStride (  ) const
```

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

virtual colortype getColorFrom24BitRGB (uint8_t red , const
uint8_t green , const
uint8_t blue const
) const

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD24bpp

LCD24bpp ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Gets color from RGB.

Parameters:

red The red.
green The green.
blue The blue.

Returns:

The color from RGB.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine	(const TextureSurface &	texture ,
	RenderingVariant	renderVariant ,
	uint8_t	alpha
	)	

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyARGB8888

```
static void blitCopyARGB8888 ( const uint32_t * sourceData ,
                               const Rect &     source ,
                               const Rect &     blitRect ,
                               uint8_t          alpha
                               )
```

Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8

```
static void blitCopyL8 ( const uint8_t * sourceData ,
                         const uint8_t * clutData ,
                         const Rect &    source ,
                         const Rect &    blitRect ,
                         uint8_t         alpha
                         )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_ARGB8888

```
static void blitCopyL8_ARGB8888 ( const uint8_t * sourceData ,
                                  const uint8_t * clutData ,
                                  const Rect &    source ,
                                  const Rect &    blitRect ,
                                  uint8_t         alpha
                                  )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (ARGB8888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_RGB888

```
static void blitCopyL8_RGB888 ( const uint8_t * sourceData ,
                                const uint8_t * clutData ,
                                const Rect &    source ,
                                const Rect &    blitRect ,
                                uint8_t         alpha
                                )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (RGB888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyRGB565

```
static void blitCopyRGB565 ( const uint16_t * sourceData16 ,
                             const Rect &      source ,
                             const Rect &      blitRect ,
                             uint8_t           alpha
                             )
```

Blits a 2D source-array to the framebuffer.

Per pixel alpha is not supported, only global alpha.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 16- bits RGB565 values.
source	The location and dimension of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation  textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation  textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD24DebugPrinter

The class LCD24DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 24bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD2bpp

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 2 bits per pixel grayscale displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.

void	**enableTextureMapperGRAY2**()
	Enables the texture mappers for GRAY2 image format.
void	**enableTextureMapperGRAY2_BilinearInterpolation**()
	Enables the texture mappers for GRAY2 image format.
void	**enableTextureMapperGRAY2_NearestNeighbor**()
	Enables the texture mappers for GRAY2 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD2bpp()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()

	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getPixel**(const uint16_t * addr, int offset)
	Get pixel from buffer/image.
FORCE_INLINE_FUNCTION uint8_t	**getPixel**(const uint8_t * addr, int offset)
	Get pixel from buffer/image.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.
FORCE_INLINE_FUNCTION void	**setPixel**(uint16_t * addr, int offset, uint8_t value)
	Set pixel in buffer.
FORCE_INLINE_FUNCTION void	**setPixel**(uint8_t * addr, int offset, uint8_t value)
	Set pixel in buffer.
FORCE_INLINE_FUNCTION int	**shiftVal**(int offset)
	Shift value to get the right pixel in a byte.

Protected Functions

void	**copyRect**(const uint8_t *srcAddress, uint16_t srcStride, uint8_t srcPixelOffset, uint8_t* RESTRICT dstAddress, uint16_t dstStride, uint8_t dstPixelOffset, uint16_t width, uint16_t height) const
	Copies a rectangular area.
virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

void	**blitCopyAlphaPerPixel**(const uint16_t *sourceData16, const uint8_t* sourceAlphaData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.

int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Protected Attributes

const uint8_t	**alphaTable2bpp**
	The alpha lookup table to avoid arithmetics when alpha blending.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)

	Draws the specified Unicode string.
virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.

virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
                        bool                     hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId  bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperGRAY2

void enableTextureMapperGRAY2 ( )

Enables the texture mappers for GRAY2 image format.

This allows drawing GRAY2 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperGRAY2_BilinearInterpolation, **enableTextureMapperGRAY2_NearestNeighbor**

enableTextureMapperGRAY2_BilinearInterpolation

void enableTextureMapperGRAY2_BilinearInterpolation ( )

Enables the texture mappers for GRAY2 image format.

This allows drawing GRAY2 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperGRAY2, **enableTextureMapperGRAY2_NearestNeighbor**

enableTextureMapperGRAY2_NearestNeighbor

void enableTextureMapperGRAY2_NearestNeighbor ( )

Enables the texture mappers for GRAY2 image format.

This allows drawing GRAY2 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperGRAY2, **enableTextureMapperGRAY2_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

```
virtual Bitmap::BitmapFormat framebufferFormat (  ) const
```

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

```
virtual uint16_t framebufferStride (  ) const
```

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

virtual colortype getColorFrom24BitRGB (uint8_t red , const
uint8_t green , const
uint8_t blue const
) const

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD2bpp

LCD2bpp ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getPixel

static FORCE_INLINE_FUNCTION uint8_t getPixel (const uint16_t * addr , int offset)

Get pixel from buffer/image.

Parameters:

addr The address.
offset The offset.

Returns:

The pixel value.

getPixel

static FORCE_INLINE_FUNCTION uint8_t getPixel (const uint8_t * addr , int offset)

Get pixel from buffer/image.

Parameters:

addr The address.
offset The offset.

Returns:

The pixel value.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

setPixel

```
static FORCE_INLINE_FUNCTION void setPixel ( uint16_t * addr ,
                                             int offset ,
                                             uint8_t value
                                             )
```

Set pixel in buffer.

Parameters:

addr The address.
offset The offset.
value The value.

setPixel

```
static FORCE_INLINE_FUNCTION void setPixel ( uint8_t * addr ,
                                             int offset ,
                                             uint8_t value
                                             )
```

Set pixel in buffer.

Parameters:

addr The address.
offset The offset.
value The value.

shiftVal

```
static FORCE_INLINE_FUNCTION int shiftVal ( int offset )
```

Shift value to get the right pixel in a byte.

Parameters:

offset The offset.

Returns:

The shift value.

Protected Functions Documentation

copyRect

```
void copyRect ( const uint8_t *     srcAddress ,     const
                uint16_t            srcStride ,      const
                uint8_t             srcPixelOffset , const
                uint8_t *RESTRICT   dstAddress ,     const
                uint16_t            dstStride ,      const
                uint8_t             dstPixelOffset , const
                uint16_t            width ,          const
                uint16_t            height           const
                )                   const
```

Copies a rectangular area.

Parameters:

srcAddress	Source address (byte address).
srcStride	Source stride (number of bytes to advance to next line).
srcPixelOffset	Source pixel offset (first pixel in first source byte).
dstAddress	If destination address (byte address).
dstStride	Destination stride (number of bytes to advance to next line).
dstPixelOffset	Destination pixel offset (first pixel in destination byte).
width	The width of area (in pixels).
height	The height of area (in pixels).

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
```

```
                                const GlyphNode * glyph ,
                                const uint8_t *   glyphData ,
                                uint8_t           byteAlignRow ,
                                colortype         color ,
                                uint8_t           bitsPerPixel ,
                                uint8_t           alpha ,
                                TextRotation      rotation
                                )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

```
virtual DrawTextureMapScanLineBase
* getTextureMapperDrawScanLine   (  const TextureSurface &   texture ,
                                   RenderingVariant         renderVariant ,
                                   uint8_t                  alpha
                                   )
```

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.

renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyAlphaPerPixel

```
static void blitCopyAlphaPerPixel ( const uint16_t * sourceData16 ,
                                    const uint8_t *  sourceAlphaData ,
                                    const Rect &     source ,
                                    const Rect &     blitRect ,
                                    uint8_t          alpha
                                    )
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 2bpp GRAY2 values.
sourceAlphaData	The alpha channel array pointer (points to the beginning of the data)
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation  textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation  textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Protected Attributes Documentation

alphaTable2bpp

const uint8_t alphaTable2bpp

The alpha lookup table to avoid arithmetics when alpha blending.

Version: 4.16

LCD2DebugPrinter

The class LCD2DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 24bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD32bpp

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.

void **enableTextureMapperAll**()

Enables the texture mappers for all image formats.

void **enableTextureMapperARGB8888**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_BilinearInterpolation**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperARGB8888_NearestNeighbor**()

Enables the texture mappers for ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_BilinearInterpolation**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_ARGB8888_NearestNeighbor**()

Enables the texture mappers for L8_ARGB8888 image format.

void **enableTextureMapperL8_RGB565**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB565_BilinearInterpolation**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB565_NearestNeighbor**()

Enables the texture mappers for L8_RGB565 image format.

void **enableTextureMapperL8_RGB888**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_BilinearInterpolation**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperL8_RGB888_NearestNeighbor**()

Enables the texture mappers for L8_RGB888 image format.

void **enableTextureMapperRGB565**()

Enables the texture mappers for RGB565 image format.

void **enableTextureMapperRGB565_NonOpaque_BilinearInterpolation**()

Enables the texture mappers for NonOpaque RGB565 image format.

void **enableTextureMapperRGB565_NonOpaque_NearestNeighbor**()

	Enables the texture mappers for NonOpaque RGB565 image format.
void	**enableTextureMapperRGB565_Opaque_BilinearInterpolation**()
	Enables the texture mappers for Opaque RGB565 image format.
void	**enableTextureMapperRGB565_Opaque_NearestNeighbor**()
	Enables the texture mappers for Opaque RGB565 image format.
void	**enableTextureMapperRGB888**()
	Enables the texture mappers for RGB888 image format.
void	**enableTextureMapperRGB888_BilinearInterpolation**()
	Enables the texture mappers for RGB888 image format.
void	**enableTextureMapperRGB888_NearestNeighbor**()
	Enables the texture mappers for RGB888 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD32bpp()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)

	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyL8**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.
void	**blitCopyL8_ARGB8888**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.
void	**blitCopyL8_RGB565**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB565 is not supported by the DMA a software blend is performed.
void	**blitCopyL8_RGB888**(const uint8_t *sourceData*, *const uint8_t* clutData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)

	Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB888 is not supported by the DMA a software blend is performed.
void	**blitCopyRGB565**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.
void	**blitCopyRGB888**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.

virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.

uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
                        bool                     hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

```
void enableTextureMapperA4 ( )
```

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

```
void enableTextureMapperA4_BilinearInterpolation ( )
```

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

enableTextureMapperL8_ARGB8888

void enableTextureMapperL8_ARGB8888 ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_ARGB8888_BilinearInterpolation,
enableTextureMapperL8_ARGB8888_NearestNeighbor

enableTextureMapperL8_ARGB8888_BilinearInterpolation

void enableTextureMapperL8_ARGB8888_BilinearInterpolation ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_NearestNeighbor**

enableTextureMapperL8_ARGB8888_NearestNeighbor

void enableTextureMapperL8_ARGB8888_NearestNeighbor ( )

Enables the texture mappers for L8_ARGB8888 image format.

This allows drawing L8_ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_ARGB8888, **enableTextureMapperL8_ARGB8888_BilinearInterpolation**

enableTextureMapperL8_RGB565

void enableTextureMapperL8_RGB565 ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_RGB565_BilinearInterpolation,
enableTextureMapperL8_RGB565_NearestNeighbor

enableTextureMapperL8_RGB565_BilinearInterpolation

void enableTextureMapperL8_RGB565_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB565, **enableTextureMapperL8_RGB565_NearestNeighbor**

enableTextureMapperL8_RGB565_NearestNeighbor

void enableTextureMapperL8_RGB565_NearestNeighbor ( )

Enables the texture mappers for L8_RGB565 image format.

This allows drawing L8_RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB565, **enableTextureMapperL8_RGB565_BilinearInterpolation**

enableTextureMapperL8_RGB888

void enableTextureMapperL8_RGB888 ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperL8_RGB888_BilinearInterpolation,
enableTextureMapperL8_RGB888_NearestNeighbor

enableTextureMapperL8_RGB888_BilinearInterpolation

void enableTextureMapperL8_RGB888_BilinearInterpolation ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_NearestNeighbor**

enableTextureMapperL8_RGB888_NearestNeighbor

void enableTextureMapperL8_RGB888_NearestNeighbor ( )

Enables the texture mappers for L8_RGB888 image format.

This allows drawing L8_RGB888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperL8_RGB888, **enableTextureMapperL8_RGB888_BilinearInterpolation**

enableTextureMapperRGB565

void enableTextureMapperRGB565 ( )

Enables the texture mappers for RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation and Nearest Neighbor algorithms.

enableTextureMapperRGB565_NonOpaque_BilinearInterpolation

void enableTextureMapperRGB565_NonOpaque_BilinearInterpolation ( )

Enables the texture mappers for NonOpaque RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB565, **enableTextureMapperRGB565_Opaque_BilinearInterpolation**

enableTextureMapperRGB565_NonOpaque_NearestNeighbor

void enableTextureMapperRGB565_NonOpaque_NearestNeighbor ( )

Enables the texture mappers for NonOpaque RGB565 image format.

This allows drawing RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB565, **enableTextureMapperRGB565_Opaque_NearestNeighbor**

enableTextureMapperRGB565_Opaque_BilinearInterpolation

void enableTextureMapperRGB565_Opaque_BilinearInterpolation ( )

Enables the texture mappers for Opaque RGB565 image format.

This allows drawing RGB565 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB565, **enableTextureMapperRGB565_NonOpaque_BilinearInterpolation**

enableTextureMapperRGB565_Opaque_NearestNeighbor

void enableTextureMapperRGB565_Opaque_NearestNeighbor ( )

Enables the texture mappers for Opaque RGB565 image format.

This allows drawing RGB565 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB565, **enableTextureMapperRGB565_NonOpaque_NearestNeighbor**

enableTextureMapperRGB888

void enableTextureMapperRGB888 ( )

Enables the texture mappers for RGB888 image format.

This allows drawing RGB888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperRGB888_BilinearInterpolation,
enableTextureMapperRGB888_NearestNeighbor

enableTextureMapperRGB888_BilinearInterpolation

void enableTextureMapperRGB888_BilinearInterpolation ( )

Enables the texture mappers for RGB888 image format.

This allows drawing RGB888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGB888, **enableTextureMapperRGB888_NearestNeighbor**

enableTextureMapperRGB888_NearestNeighbor

void enableTextureMapperRGB888_NearestNeighbor ( )

Enables the texture mappers for RGB888 image format.

This allows drawing RGB888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGB888, **enableTextureMapperRGB888_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

virtual colortype getColorFrom24BitRGB (uint8_t red , const
uint8_t green , const
uint8_t blue const
) const

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD32bpp

LCD32bpp ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color from RGB.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.

offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

```
virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine ( const TextureSurface & texture ,
                                                                    RenderingVariant renderVariant ,
                                                                    uint8_t alpha
                                                                    )
```

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyL8

```
static void blitCopyL8 ( const uint8_t * sourceData ,
                         const uint8_t * clutData ,
                         const Rect & source ,
                         const Rect & blitRect ,
                         uint8_t alpha
                         )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if indexed format is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_ARGB8888

```
static void blitCopyL8_ARGB8888 ( const uint8_t * sourceData ,
                                  const uint8_t * clutData ,
                                  const Rect &    source ,
                                  const Rect &    blitRect ,
                                  uint8_t         alpha
                                  )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (ARGB8888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_RGB565

```
static void blitCopyL8_RGB565 ( const uint8_t * sourceData ,
                                const uint8_t * clutData ,
                                const Rect &    source ,
                                const Rect &    blitRect ,
                                uint8_t         alpha
                                )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB565 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.

clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 16- bits (RGB565) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyL8_RGB888

```
static void blitCopyL8_RGB888 ( const uint8_t * sourceData ,
                                const uint8_t * clutData ,
                                const Rect &    source ,
                                const Rect &    blitRect ,
                                uint8_t         alpha
                                )
```

Blits a 2D indexed 8-bit source to the framebuffer performing alpha-blending per pixel as specified if L8_RGB888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-indexes pointer (points to the beginning of the data). The sourceData must be stored as 8- bits indexes.
clutData	The source-clut pointer (points to the beginning of the CLUT color format and size data followed by colors entries stored as 32- bits (RGB888) format.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyRGB565

```
static void blitCopyRGB565 ( const uint16_t * sourceData16 ,
                             const Rect &     source ,
                             const Rect &     blitRect ,
                             uint8_t          alpha
                             )
```

Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.

If! RGB565 is not supported by the DMA a software blend is performed.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 16- bits RGB565 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyRGB888

```
static void blitCopyRGB888 ( const uint16_t * sourceData16 ,
                             const Rect &      source ,
                             const Rect &      blitRect ,
                             uint8_t           alpha
                             )
```

Blits a 2D source-array to the framebuffer performing alpha-blending per pixel as specified.

If RGB888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 24- bits RGB888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool         rotatedDisplay ,
                      TextRotation textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool         rotatedDisplay ,
                       TextRotation textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD32DebugPrinter

The class LCD32DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 32bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD4bpp

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 4 bits per pixel grayscale displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.

void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.
void	**enableTextureMapperGRAY4**()
	Enables the texture mappers for GRAY4 image format.
void	**enableTextureMapperGRAY4_BilinearInterpolation**()
	Enables the texture mappers for GRAY4 image format.
void	**enableTextureMapperGRAY4_NearestNeighbor**()
	Enables the texture mappers for GRAY4 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD4bpp()

FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getPixel**(const uint16_t * addr, int offset)
	Get pixel from buffer/image.
FORCE_INLINE_FUNCTION uint8_t	**getPixel**(const uint8_t * addr, int offset)
	Get pixel from buffer/image.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.
FORCE_INLINE_FUNCTION void	**setPixel**(uint16_t * addr, int offset, uint8_t value)
	Set pixel in buffer.
FORCE_INLINE_FUNCTION void	**setPixel**(uint8_t * addr, int offset, uint8_t value)
	Set pixel in buffer.

Protected Functions

void	**copyRect**(const uint8_t *srcAddress, uint16_t srcStride, uint8_t srcPixelOffset, uint8_t* RESTRICT dstAddress, uint16_t dstStride, uint8_t dstPixelOffset, uint16_t width, uint16_t height) const
	Copies a rectangular area.
virtual void	**drawGlyph**(uint16_t *wbuf16, **Rect** widgetArea, int16_t x, int16_t y, uint16_t offsetX, uint16_t offsetY, const **Rect** & invalidatedArea, const **GlyphNode** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)

	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyAlphaPerPixel**(const uint16_t *sourceData16, const uint8_t* sourceAlphaData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)

	Draws the specified Unicode string.
virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.

virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.
FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
                        bool                     hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId  bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: touchgfx::LCD::drawPartialBitmap

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, enableTextureMapperA4_NearestNeighbor

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, enableTextureMapperA4_NearestNeighbor

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, enableTextureMapperA4_BilinearInterpolation

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperGRAY4

void enableTextureMapperGRAY4 ( )

Enables the texture mappers for GRAY4 image format.

This allows drawing GRAY4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperGRAY4_BilinearInterpolation,
enableTextureMapperGRAY4_NearestNeighbor

enableTextureMapperGRAY4_BilinearInterpolation

void enableTextureMapperGRAY4_BilinearInterpolation ( )

Enables the texture mappers for GRAY4 image format.

This allows drawing GRAY4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperGRAY4, **enableTextureMapperGRAY4_NearestNeighbor**

enableTextureMapperGRAY4_NearestNeighbor

void enableTextureMapperGRAY4_NearestNeighbor ( )

Enables the texture mappers for GRAY4 image format.

This allows drawing GRAY4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperGRAY4, **enableTextureMapperGRAY4_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype   color ,
```

```
                uint8_t         alpha =255
                )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

```
virtual Bitmap::BitmapFormat framebufferFormat (  ) const
```

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

```
virtual uint16_t framebufferStride (  ) const
```

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

```
virtual uint8_t getBlueColor ( colortype color )
```

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

```
virtual colortype getColorFrom24BitRGB ( uint8_t red ,   const
                                         uint8_t green , const
                                         uint8_t blue  const
                                         )       const
```

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red Value of the red part (0-255).
green Value of the green part (0-255).
blue Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

```
virtual uint8_t getGreenColor ( colortype color )
```

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD4bpp

LCD4bpp ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red , uint8_t green , uint8_t blue)

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color	The color.

Returns:

The green from color.

getPixel

```
static FORCE_INLINE_FUNCTION uint8_t getPixel ( const uint16_t * addr ,
                                                int             offset
                                                )
```

Get pixel from buffer/image.

Parameters:

addr The address.
offset The offset.

Returns:

The pixel value.

getPixel

```
static FORCE_INLINE_FUNCTION uint8_t getPixel ( const uint8_t * addr ,
                                                int             offset
                                                )
```

Get pixel from buffer/image.

Parameters:

addr The address.
offset The offset.

Returns:

The pixel value.

getRedFromColor

```
static FORCE_INLINE_FUNCTION uint8_t getRedFromColor ( colortype color )
```

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

setPixel

```
static FORCE_INLINE_FUNCTION void setPixel ( uint16_t * addr ,
                                             int        offset ,
                                             uint8_t    value
                                             )
```

Set pixel in buffer.

Parameters:

addr	The address.
offset	The offset.
value	The value.

setPixel

```
static FORCE_INLINE_FUNCTION void setPixel ( uint8_t * addr ,
                                             int        offset ,
                                             uint8_t    value
                                             )
```

Set pixel in buffer.

Parameters:

addr	The address.
offset	The offset.
value	The value.

Protected Functions Documentation

copyRect

```
void copyRect ( const uint8_t *      srcAddress ,      const
                uint16_t             srcStride ,       const
                uint8_t              srcPixelOffset ,  const
                uint8_t *RESTRICT    dstAddress ,      const
                uint16_t             dstStride ,       const
                uint8_t              dstPixelOffset ,  const
                uint16_t             width ,           const
                uint16_t             height            const
                )                    const
```

Copies a rectangular area.

Parameters:

srcAddress	Source address (byte address).
srcStride	Source stride (number of bytes to advance to next line).
srcPixelOffset	Source pixel offset (first pixel in first source byte).
dstAddress	If destination address (byte address).
dstStride	Destination stride (number of bytes to advance to next line).
dstPixelOffset	Destination pixel offset (first pixel in destination byte).
width	The width of area (in pixels).
height	The height of area (in pixels).

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)

byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine (const TextureSurface & texture , RenderingVariant renderVariant , uint8_t alpha)

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyAlphaPerPixel

static void blitCopyAlphaPerPixel (const uint16_t * sourceData16 , const uint8_t * sourceAlphaData , const Rect & source , const Rect & blitRect , uint8_t alpha)

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 4bpp GRAY4 values.
sourceAlphaData	The alpha channel array pointer (points to the beginning of the data)
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation  textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation  textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD4DebugPrinter

The class LCD4DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 8bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD8ABGR2222DebugPrinter

The class LCD8ABGR2222DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 8bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD8ARGB2222DebugPrinter

The class LCD8ARGB2222DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 8bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD8BGRA2222DebugPrinter

The class LCD8BGRA2222DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 8bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

LCD8bpp_ABGR2222

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.

void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperABGR2222**()
	Enables the texture mappers for ABGR2222 image format.
void	**enableTextureMapperABGR2222_BilinearInterpolation**()
	Enables the texture mappers for ABGR2222 image format.
void	**enableTextureMapperABGR2222_NearestNeighbor**()
	Enables the texture mappers for ABGR2222 image format.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.
void	**enableTextureMapperARGB8888**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_BilinearInterpolation**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_NearestNeighbor**()
	Enables the texture mappers for ARGB8888 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const

	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD8bpp_ABGR2222()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Gets color from RGB.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyAlphaPerPixel**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)

	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.
void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.

virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.

FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
```

```
                    bool                hasTransparentPixels
                    )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperABGR2222

void enableTextureMapperABGR2222 ( )

Enables the texture mappers for ABGR2222 image format.

This allows drawing ABGR2222 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperABGR2222_BilinearInterpolation,
enableTextureMapperABGR2222_NearestNeighbor

enableTextureMapperABGR2222_BilinearInterpolation

void enableTextureMapperABGR2222_BilinearInterpolation ( )

Enables the texture mappers for ABGR2222 image format.

This allows drawing ABGR2222 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperABGR2222, **enableTextureMapperABGR2222_NearestNeighbor**

enableTextureMapperABGR2222_NearestNeighbor

void enableTextureMapperABGR2222_NearestNeighbor ( )

Enables the texture mappers for ABGR2222 image format.

This allows drawing ABGR2222 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperABGR2222, **enableTextureMapperABGR2222_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect The rectangle to draw in absolute display coordinates.
color The rectangle color.
alpha (Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

virtual colortype getColorFrom24BitRGB (uint8_t red , const
uint8_t green , const
uint8_t blue const
) const

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

virtual uint8_t getGreenColor (colortype color)

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

virtual uint8_t getRedColor (colortype color)

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD8bpp_ABGR2222

LCD8bpp_ABGR2222 ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Gets color from RGB.

Parameters:

red	The red.
green	The green.
blue	The blue.

Returns:

The color from RGB.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color	The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color	The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: touchgfx::LCD::drawGlyph

getTextureMapperDrawScanLine

virtual DrawTextureMapScanLineBase * getTextureMapperDrawScanLine	(	const TextureSurface &	texture ,
		RenderingVariant	renderVariant ,
		uint8_t	alpha
		)	

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: touchgfx::LCD::getTextureMapperDrawScanLine

blitCopyAlphaPerPixel

static void blitCopyAlphaPerPixel	(	const uint16_t *	sourceData16 ,
		const Rect &	source ,
		const Rect &	blitRect ,
		uint8_t	alpha
		)	

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 8-bits ABGR2222 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyARGB8888

```
static void blitCopyARGB8888 ( const uint32_t * sourceData ,
                               const Rect &     source ,
                               const Rect &     blitRect ,
                               uint8_t          alpha
                               )
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool         rotatedDisplay ,
                      TextRotation textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool         rotatedDisplay ,
                       TextRotation textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD8bpp_ARGB2222

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.

void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.
void	**enableTextureMapperARGB2222**()
	Enables the texture mappers for ARGB2222 image format.
void	**enableTextureMapperARGB2222_BilinearInterpolation**()
	Enables the texture mappers for ARGB2222 image format.
void	**enableTextureMapperARGB2222_NearestNeighbor**()
	Enables the texture mappers for ARGB2222 image format.
void	**enableTextureMapperARGB8888**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_BilinearInterpolation**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_NearestNeighbor**()
	Enables the texture mappers for ARGB8888 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const

	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD8bpp_ARGB2222()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Gets color from RGB.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual **DrawTextureMapScanLineBase** *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

void	**blitCopyAlphaPerPixel**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.
void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.

virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.

FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
```

```
                    bool                hasTransparentPixels
                    )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB2222

void enableTextureMapperARGB2222 ( )

Enables the texture mappers for ARGB2222 image format.

This allows drawing ARGB2222 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB2222_BilinearInterpolation, **enableTextureMapperARGB2222_NearestNeighbor**

enableTextureMapperARGB2222_BilinearInterpolation

void enableTextureMapperARGB2222_BilinearInterpolation ( )

Enables the texture mappers for ARGB2222 image format.

This allows drawing ARGB2222 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB2222, **enableTextureMapperARGB2222_NearestNeighbor**

enableTextureMapperARGB2222_NearestNeighbor

void enableTextureMapperARGB2222_NearestNeighbor ( )

Enables the texture mappers for ARGB2222 image format.

This allows drawing ARGB2222 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB2222, **enableTextureMapperARGB2222_BilinearInterpolation**

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation,
enableTextureMapperARGB8888_NearestNeighbor

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

```
virtual colortype getColorFrom24BitRGB ( uint8_t red ,  const
                                         uint8_t green ,  const
                                         uint8_t blue  const
                                         )  const
```

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

```
virtual uint8_t getGreenColor ( colortype color )
```

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

```
virtual uint8_t getRedColor ( colortype color )
```

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD8bpp_ARGB2222

LCD8bpp_ARGB2222 ( )

getBlueFromColor

static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor (colortype color)

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

static FORCE_INLINE_FUNCTION colortype getColorFromRGB (uint8_t red ,
uint8_t green ,
uint8_t blue
)

Gets color from RGB.

Parameters:

red The red.
green The green.
blue The blue.

Returns:

The color from RGB.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

```
virtual DrawTextureMapScanLineBase
* getTextureMapperDrawScanLine ( const
                                  TextureSurface &   texture ,
```

```
                                    RenderingVariant    renderVariant
                                                        ,
                                    uint8_t             alpha
                                    )
```

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyAlphaPerPixel

```
static void blitCopyAlphaPerPixel ( const uint16_t * sourceData16 ,
                                    const Rect &     source ,
                                    const Rect &     blitRect ,
                                    uint8_t          alpha
                                    )
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 8-bits ARGB2222 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyARGB8888

```
static void blitCopyARGB8888 ( const uint32_t * sourceData ,
                               const Rect &     source ,
                               const Rect &     blitRect ,
                               uint8_t          alpha
```

```
)
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD8bpp_BGRA2222

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.

void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.
void	**enableTextureMapperARGB8888**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_BilinearInterpolation**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_NearestNeighbor**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperBGRA2222**()
	Enables the texture mappers for BGRA2222 image format.
void	**enableTextureMapperBGRA2222_BilinearInterpolation**()
	Enables the texture mappers for BGRA2222 image format.
void	**enableTextureMapperBGRA2222_NearestNeighbor**()
	Enables the texture mappers for BGRA2222 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const

	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD8bpp_BGRA2222()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Gets color from RGB.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyAlphaPerPixel**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)

	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.
void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.

virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.

FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
```

```
                    bool                hasTransparentPixels
                    )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

enableTextureMapperBGRA2222

void enableTextureMapperBGRA2222 ( )

Enables the texture mappers for BGRA2222 image format.

This allows drawing BGRA2222 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperBGRA2222_BilinearInterpolation,
enableTextureMapperBGRA2222_NearestNeighbor

enableTextureMapperBGRA2222_BilinearInterpolation

void enableTextureMapperBGRA2222_BilinearInterpolation ( )

Enables the texture mappers for BGRA2222 image format.

This allows drawing BGRA2222 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperBGRA2222, **enableTextureMapperBGRA2222_NearestNeighbor**

enableTextureMapperBGRA2222_NearestNeighbor

void enableTextureMapperBGRA2222_NearestNeighbor ( )

Enables the texture mappers for BGRA2222 image format.

This allows drawing BGRA2222 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperBGRA2222, **enableTextureMapperBGRA2222_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

```
virtual colortype getColorFrom24BitRGB ( uint8_t red ,  const
                                         uint8_t green ,  const
                                         uint8_t blue  const
                                         )  const
```

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

```
virtual uint8_t getGreenColor ( colortype color )
```

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

```
virtual uint8_t getRedColor ( colortype color )
```

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD8bpp_BGRA2222

```
LCD8bpp_BGRA2222 (  )
```

getBlueFromColor

```
static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor ( colortype color )
```

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

```
static FORCE_INLINE_FUNCTION colortype getColorFromRGB ( uint8_t red ,
                                                         uint8_t green ,
                                                         uint8_t blue
                                                         )
```

Gets color from RGB.

Parameters:

red The red.
green The green.
blue The blue.

Returns:

The color from RGB.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

```
virtual DrawTextureMapScanLineBase
* getTextureMapperDrawScanLine ( const
                                  TextureSurface &   texture ,
```

```
                                    RenderingVariant renderVariant
                                    ,
                                    uint8_t          alpha
                                    )
```

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyAlphaPerPixel

```
static void blitCopyAlphaPerPixel ( const uint16_t * sourceData16 ,
                                    const Rect &     source ,
                                    const Rect &     blitRect ,
                                    uint8_t          alpha
                                    )
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 8-bits BGRA2222 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyARGB8888

```
static void blitCopyARGB8888 ( const uint32_t * sourceData ,
                               const Rect &     source ,
                               const Rect &     blitRect ,
                               uint8_t          alpha
```

```
)
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD8bpp_RGBA2222

This class contains the various low-level drawing routines for drawing bitmaps, texts and rectangles on 16 bits per pixel displays.

See: LCD

Note: All coordinates are expected to be in absolute coordinates!

Inherits from: LCD

Public Functions

virtual uint8_t	**bitDepth**() const
	Number of bits per pixel used by the display.
virtual void	**blitCopy**(const uint16_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual void	**blitCopy**(const uint8_t * sourceData, **Bitmap::BitmapFormat** sourceFormat, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha, bool hasTransparentPixels)
	Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.
virtual uint16_t *	**copyFrameBufferRegionToMemory**(const **Rect** & visRegion, const **Rect** & absRegion, const **BitmapId** bitmapId)
	Copies part of the framebuffer to the data section of a bitmap.
virtual void	**drawPartialBitmap**(const **Bitmap** & bitmap, int16_t x, int16_t y, const **Rect** & rect, uint8_t alpha =255, bool useOptimized =true)
	Draws all (or a part) of a *bitmap*.
void	**enableTextureMapperA4**()
	Enables the texture mappers for A4 image format.

void	**enableTextureMapperA4_BilinearInterpolation**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperA4_NearestNeighbor**()
	Enables the texture mappers for A4 image format.
void	**enableTextureMapperAll**()
	Enables the texture mappers for all image formats.
void	**enableTextureMapperARGB8888**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_BilinearInterpolation**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperARGB8888_NearestNeighbor**()
	Enables the texture mappers for ARGB8888 image format.
void	**enableTextureMapperRGBA2222**()
	Enables the texture mappers for RGBA2222 image format.
void	**enableTextureMapperRGBA2222_BilinearInterpolation**()
	Enables the texture mappers for RGBA2222 image format.
void	**enableTextureMapperRGBA2222_NearestNeighbor**()
	Enables the texture mappers for RGBA2222 image format.
virtual void	**fillRect**(const **Rect** & rect, **colortype** color, uint8_t alpha =255)
	Draws a filled rectangle in the framebuffer in the specified color and opacity.
virtual **Bitmap::BitmapFormat**	**framebufferFormat**() const
	Framebuffer format used by the display.
virtual uint16_t	**framebufferStride**() const
	Framebuffer stride in bytes.
virtual uint8_t	**getBlueColor**(**colortype** color) const
	Gets the blue color part of a color.
virtual **colortype**	**getColorFrom24BitRGB**(uint8_t red, uint8_t green, uint8_t blue) const

	Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.
virtual uint8_t	**getGreenColor**(**colortype** color) const
	Gets the green color part of a color.
virtual uint8_t	**getRedColor**(**colortype** color) const
	Gets the red color part of a color.
	LCD8bpp_RGBA2222()
FORCE_INLINE_FUNCTION uint8_t	**getBlueFromColor**(**colortype** color)
	Gets blue from color.
FORCE_INLINE_FUNCTION **colortype**	**getColorFromRGB**(uint8_t red, uint8_t green, uint8_t blue)
	Gets color from RGB.
FORCE_INLINE_FUNCTION uint16_t	**getFramebufferStride**()
	Framebuffer stride in bytes.
FORCE_INLINE_FUNCTION uint8_t	**getGreenFromColor**(**colortype** color)
	Gets green from color.
FORCE_INLINE_FUNCTION uint8_t	**getRedFromColor**(**colortype** color)
	Gets red from color.

Protected Functions

virtual void	**drawGlyph**(uint16_t *wbuf16*, ***Rect*** *widgetArea*, *int16_t x*, *int16_t y*, *uint16_t offsetX*, *uint16_t offsetY*, *const* ***Rect*** *& invalidatedArea*, *const* ***GlyphNode*** glyph, const uint8_t * glyphData, uint8_t byteAlignRow, **colortype** color, uint8_t bitsPerPixel, uint8_t alpha, **TextRotation** rotation)
	Private version of draw-glyph with explicit destination buffer pointer argument.
virtual DrawTextureMapScanLineBase *	**getTextureMapperDrawScanLine**(const **TextureSurface** & texture, **RenderingVariant** renderVariant, uint8_t alpha)
	Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.
void	**blitCopyAlphaPerPixel**(const uint16_t * sourceData16, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)

	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.
void	**blitCopyARGB8888**(const uint32_t * sourceData, const **Rect** & source, const **Rect** & blitRect, uint8_t alpha)
	Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.
int	**nextLine**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next line.
int	**nextPixel**(bool rotatedDisplay, **TextRotation** textRotation)
	Find out how much to advance in the display buffer to get to the next pixel.

Additional inherited members

Public Classes inherited from LCD

struct	**StringVisuals**
	The visual elements when writing a string.

Protected Classes inherited from LCD

class	**DrawTextureMapScanLineBase**
	Base class for drawing scanline by the texture mapper.

Public Functions inherited from LCD

void	**drawString**(**Rect** widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & stringVisuals, const **Unicode::UnicodeChar** * format, ...)
	Draws the specified Unicode string.

virtual void	**drawTextureMapTriangle**(const **DrawingSurface** & dest, const **Point3D** * vertices, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha =255, uint16_t subDivisionSize =12)
	Texture map triangle.
colortype	**getDefaultColor**() const
	Gets default color previously set using setDefaultColor.
void	**setDefaultColor**(**colortype** color)
	Sets default color as used by alpha level only bitmap formats, e.g.
virtual	**~LCD**()
	Finalizes an instance of the **LCD** class.
FORCE_INLINE_FUNCTION uint8_t	**div255**(uint16_t num)
	Approximates an integer division of a 16bit value by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255g**(uint32_t pixelxAlpha)
	Divides the green component of pixelxAlpha by 255.
FORCE_INLINE_FUNCTION uint32_t	**div255rb**(uint32_t pixelxAlpha)
	Divides the red and blue components of pixelxAlpha by 255.

Protected Functions inherited from LCD

void	**drawStringLTR**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
void	**drawStringRTL**(const **Rect** & widgetArea, const **Rect** & invalidatedArea, const **StringVisuals** & visuals, const **Unicode::UnicodeChar** * format, va_list pArg)
	Draws the specified Unicode string.
virtual void	**drawTextureMapScanLine**(const **DrawingSurface** & dest, const **Gradients** & gradients, const **Edge** *leftEdge, const* ***Edge*** rightEdge, const **TextureSurface** & texture, const **Rect** & absoluteRect, const **Rect** & dirtyAreaAbsolute, **RenderingVariant** renderVariant, uint8_t alpha, uint16_t subDivisionSize)
	Draw scan line.

FORCE_INLINE_FUNCTION uint8_t	**getAlphaFromA4**(const uint16_t * data, uint32_t offset)
	Gets alpha from A4 image at given offset.
uint16_t	**getNumLines**(**TextProvider** & textProvider, **WideTextAction** wideTextAction, **TextDirection** textDirection, const **Font** * font, int16_t width)
	Gets number of lines for a given text taking word wrap into consideration.
int	**realX**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute x coordinate of a point inside a widget with regards to rotation.
int	**realY**(const **Rect** & widgetArea, int16_t x, int16_t y, **TextRotation** rotation)
	Find the real, absolute y coordinate of a point inside a widget with regards to rotation.
void	**rotateRect**(**Rect** & rect, const **Rect** & canvas, const **TextRotation** rotation)
	Rotate a rectangle inside another rectangle.
uint16_t	**stringWidth**(**TextProvider** & textProvider, const **Font** & font, const int numChars, **TextDirection** textDirection)
	Find string width of the given number of ligatures read from the given TextProvider.

Protected Attributes inherited from LCD

colortype	**defaultColor**
	Default **Color** to use when displaying transparency-only elements, e.g. A4 bitmaps.
const uint16_t	**newLine**
	NewLine value.

Public Functions Documentation

bitDepth

virtual uint8_t bitDepth ( ) const

Number of bits per pixel used by the display.

Returns:

The number of bits per pixel.

Reimplements: **touchgfx::LCD::bitDepth**

blitCopy

```
virtual void blitCopy ( const uint16_t * sourceData ,
                        const Rect &     source ,
                        const Rect &     blitRect ,
                        uint8_t          alpha ,
                        bool             hasTransparentPixels
                        )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha != 255 (solid).

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

blitCopy

```
virtual void blitCopy ( const uint8_t *          sourceData ,
                        Bitmap::BitmapFormat     sourceFormat ,
                        const Rect &             source ,
                        const Rect &             blitRect ,
                        uint8_t                  alpha ,
```

```
                    bool                hasTransparentPixels
                    )
```

Blits (directly copies) a block of data to the framebuffer, performing alpha blending (and tranparency keying) as specified.

Performs a software blend if **HAL** does not support BLIT_COPY_WITH_ALPHA and alpha < 255 (solid).

If the display does not support the specified *sourceFormat*, an *assert* will be raised.

Parameters:

sourceData	The source array pointer (points to the beginning of the data). The sourceData must be stored in a format suitable for the selected display.
sourceFormat	The bitmap format used in the source data.
source	The position and dimensions of the source. The x and y of this rect should both be 0.
blitRect	A rectangle describing what region of the *sourceData* is to be copied to the framebuffer.
alpha	The alpha value to use for blending ranging from 0=invisible to 255=solid=no blending.
hasTransparentPixels	If true, this data copy contains transparent pixels and require hardware support for that to be enabled.

Reimplements: **touchgfx::LCD::blitCopy**

copyFrameBufferRegionToMemory

```
virtual uint16_t * copyFrameBufferRegionToMemory ( const Rect &    visRegion ,
                                                   const Rect &    absRegion ,
                                                   const BitmapId bitmapId
                                                   )
```

Copies part of the framebuffer to the data section of a bitmap.

The bitmap must be a dynamic bitmap or animation storage (BITMAP_ANIMATION_STORAGE). The two regions given are the visible region and the absolute region on screen. This is used to copy only a part of the framebuffer. This might be the case if a **SnapshotWidget** is placed inside a **Container** where parts of the SnapshowWidget is outside the area defined by the **Container**. The visible region must be completely inside the absolute region.

Parameters:

visRegion	The visible region.
absRegion	The absolute region.
bitmapId	Identifier for the bitmap.

Returns:

Null if it fails, else a pointer to the data in the given bitmap.

NOTE

There is only one instance of animation storage. The content of the bitmap data /animation storage outside the given region is left untouched.

See also:

blitCopy

Reimplements: **touchgfx::LCD::copyFrameBufferRegionToMemory**

drawPartialBitmap

```
virtual void drawPartialBitmap ( const Bitmap & bitmap ,
                                 int16_t        x ,
                                 int16_t        y ,
                                 const Rect &   rect ,
                                 uint8_t        alpha =255,
                                 bool           useOptimized =true
                                 )
```

Draws all (or a part) of a *bitmap*.

The coordinates of the corner of the bitmap is given in (*x*, *y*) and *rect* describes which part of the *bitmap* should be drawn. The bitmap can be drawn as it is or more or less transparent depending on the value of *alpha*. The value of *alpha* is independent of the transparency of the individual pixels of the given *bitmap*.

Parameters:

bitmap	The bitmap to draw.
x	The absolute x coordinate to place (0, 0) of the bitmap on the screen.
y	The absolute y coordinate to place (0, 0) of the bitmap on the screen.
rect	A rectangle describing what region of the bitmap is to be drawn.
alpha	(Optional) Optional alpha value ranging from 0=invisible to 255=solid. Default is 255 (solid).
useOptimized	(Optional) if false, do not attempt to substitute (parts of) this bitmap with faster fillrects.

Reimplements: **touchgfx::LCD::drawPartialBitmap**

enableTextureMapperA4

void enableTextureMapperA4 ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperA4_BilinearInterpolation, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_BilinearInterpolation

void enableTextureMapperA4_BilinearInterpolation ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_NearestNeighbor**

enableTextureMapperA4_NearestNeighbor

void enableTextureMapperA4_NearestNeighbor ( )

Enables the texture mappers for A4 image format.

This allows drawing A4 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperA4, **enableTextureMapperA4_BilinearInterpolation**

enableTextureMapperAll

void enableTextureMapperAll ( )

Enables the texture mappers for all image formats.

This allows drawing any image using Bilinear Interpolation and Nearest Neighbor algorithms, but might use a lot of memory for the drawing algorithms.

enableTextureMapperARGB8888

void enableTextureMapperARGB8888 ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperARGB8888_BilinearInterpolation, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_BilinearInterpolation

void enableTextureMapperARGB8888_BilinearInterpolation ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_NearestNeighbor**

enableTextureMapperARGB8888_NearestNeighbor

void enableTextureMapperARGB8888_NearestNeighbor ( )

Enables the texture mappers for ARGB8888 image format.

This allows drawing ARGB8888 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperARGB8888, **enableTextureMapperARGB8888_BilinearInterpolation**

enableTextureMapperRGBA2222

void enableTextureMapperRGBA2222 ( )

Enables the texture mappers for RGBA2222 image format.

This allows drawing RGBA2222 images using Bilinear Interpolation and Nearest Neighbor algorithms.

See also:

enableTextureMapperRGBA2222_BilinearInterpolation,
enableTextureMapperRGBA2222_NearestNeighbor

enableTextureMapperRGBA2222_BilinearInterpolation

void enableTextureMapperRGBA2222_BilinearInterpolation ( )

Enables the texture mappers for RGBA2222 image format.

This allows drawing RGBA2222 images using Bilinear Interpolation algorithm.

See also:

enableTextureMapperRGBA2222, **enableTextureMapperRGBA2222_NearestNeighbor**

enableTextureMapperRGBA2222_NearestNeighbor

void enableTextureMapperRGBA2222_NearestNeighbor ( )

Enables the texture mappers for RGBA2222 image format.

This allows drawing RGBA2222 images using Nearest Neighbor algorithm.

See also:

enableTextureMapperRGBA2222, **enableTextureMapperRGBA2222_BilinearInterpolation**

fillRect

```
virtual void fillRect ( const Rect & rect ,
                        colortype    color ,
                        uint8_t      alpha =255
                        )
```

Draws a filled rectangle in the framebuffer in the specified color and opacity.

By default the rectangle will be drawn as a solid box. The rectangle can be drawn with transparency by specifying alpha from 0=invisible to 255=solid.

Parameters:

rect	The rectangle to draw in absolute display coordinates.
color	The rectangle color.
alpha	(Optional) The rectangle opacity, from 0=invisible to 255=solid.

Reimplements: **touchgfx::LCD::fillRect**

framebufferFormat

virtual Bitmap::BitmapFormat framebufferFormat ( ) const

Framebuffer format used by the display.

Returns:

A **Bitmap::BitmapFormat**.

Reimplements: **touchgfx::LCD::framebufferFormat**

framebufferStride

virtual uint16_t framebufferStride ( ) const

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

Reimplements: **touchgfx::LCD::framebufferStride**

getBlueColor

virtual uint8_t getBlueColor (colortype color)

Gets the blue color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The 16 bit color value.

Returns:

The blue part of the color.

Reimplements: **touchgfx::LCD::getBlueColor**

getColorFrom24BitRGB

```
virtual colortype getColorFrom24BitRGB ( uint8_t red ,  const
                                         uint8_t green ,  const
                                         uint8_t blue  const
                                         )  const
```

Generates a color representation to be used on the **LCD**, based on 24 bit RGB values.

Depending on your chosen color bit depth, the color will be interpreted internally as either a 16 bit or 24 bit color value. This function can be safely used regardless of whether your application is configured for 16 or 24 bit colors.

Parameters:

red	Value of the red part (0-255).
green	Value of the green part (0-255).
blue	Value of the blue part (0-255).

Returns:

The color representation depending on **LCD** color format.

Reimplements: **touchgfx::LCD::getColorFrom24BitRGB**

getGreenColor

```
virtual uint8_t getGreenColor ( colortype color )
```

Gets the green color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color	The 16 bit color value.

Returns:

The green part of the color.

Reimplements: **touchgfx::LCD::getGreenColor**

getRedColor

```
virtual uint8_t getRedColor ( colortype color )
```

Gets the red color part of a color.

As this function must work for all color depths, it can be somewhat slow if used in speed critical sections. Consider finding the color in another way, if possible.

Parameters:

color The color value.

Returns:

The red part of the color.

Reimplements: **touchgfx::LCD::getRedColor**

LCD8bpp_RGBA2222

```
LCD8bpp_RGBA2222 (  )
```

getBlueFromColor

```
static FORCE_INLINE_FUNCTION uint8_t getBlueFromColor ( colortype color )
```

Gets blue from color.

Parameters:

color The color.

Returns:

The blue from color.

getColorFromRGB

```
static FORCE_INLINE_FUNCTION colortype getColorFromRGB ( uint8_t red ,
                                                         uint8_t green ,
                                                         uint8_t blue
                                                         )
```

Gets color from RGB.

Parameters:

red The red.
green The green.
blue The blue.

Returns:

The color from RGB.

getFramebufferStride

static FORCE_INLINE_FUNCTION uint16_t getFramebufferStride ( )

Framebuffer stride in bytes.

The distance (in bytes) from the start of one framebuffer row, to the next.

Returns:

The number of bytes in one framebuffer row.

getGreenFromColor

static FORCE_INLINE_FUNCTION uint8_t getGreenFromColor (colortype color)

Gets green from color.

Parameters:

color The color.

Returns:

The green from color.

getRedFromColor

static FORCE_INLINE_FUNCTION uint8_t getRedFromColor (colortype color)

Gets red from color.

Parameters:

color The color.

Returns:

The red from color.

Protected Functions Documentation

drawGlyph

```
virtual void drawGlyph ( uint16_t *          wbuf16 ,
                         Rect                widgetArea ,
                         int16_t             x ,
                         int16_t             y ,
                         uint16_t            offsetX ,
                         uint16_t            offsetY ,
                         const Rect &        invalidatedArea ,
                         const GlyphNode *   glyph ,
                         const uint8_t *     glyphData ,
                         uint8_t             byteAlignRow ,
                         colortype           color ,
                         uint8_t             bitsPerPixel ,
                         uint8_t             alpha ,
                         TextRotation        rotation
                         )
```

Private version of draw-glyph with explicit destination buffer pointer argument.

For all parameters (except the buffer pointer) see the public function **drawString()**.

Parameters:

wbuf16	The destination (frame) buffer to draw to.
widgetArea	The canvas to draw the glyph inside.
x	Horizontal offset to start drawing the glyph.
y	Vertical offset to start drawing the glyph.
offsetX	Horizontal offset in the glyph to start drawing from.
offsetY	Vertical offset in the glyph to start drawing from.
invalidatedArea	The area to draw inside.
glyph	Specifications of the glyph to draw.
glyphData	Data containing the actual glyph (dense format)
byteAlignRow	Each row of glyph data starts in a new byte.
color	The color of the glyph.
bitsPerPixel	Bit depth of the glyph.
alpha	The transparency of the glyph.
rotation	Rotation to do before drawing the glyph.

Reimplements: **touchgfx::LCD::drawGlyph**

getTextureMapperDrawScanLine

```
virtual DrawTextureMapScanLineBase
* getTextureMapperDrawScanLine ( const
                                  TextureSurface &   texture ,
```

```
                                    RenderingVariant    renderVariant
                                                        ,
                                    uint8_t             alpha
                                    )
```

Gets pointer to object that can draw a scan line which allows for highly specialized and optimized implementation.

Parameters:

texture	The texture Surface.
renderVariant	The render variant.
alpha	The global alpha.

Returns:

Null if it fails, else the pointer to the texture mapper draw scan line object.

Reimplements: **touchgfx::LCD::getTextureMapperDrawScanLine**

blitCopyAlphaPerPixel

```
static void blitCopyAlphaPerPixel ( const uint16_t * sourceData16 ,
                                    const Rect &     source ,
                                    const Rect &     blitRect ,
                                    uint8_t          alpha
                                    )
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified Performs always a software blend.

Parameters:

sourceData16	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 8-bit RGBA2222 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

blitCopyARGB8888

```
static void blitCopyARGB8888 ( const uint32_t * sourceData ,
                               const Rect &     source ,
                               const Rect &     blitRect ,
                               uint8_t          alpha
```

```
)
```

Blit a 2D source-array to the framebuffer performing alpha-blending per pixel as specified if ARGB8888 is not supported by the DMA a software blend is performed.

Parameters:

sourceData	The source-array pointer (points to the beginning of the data). The sourceData must be stored as 32- bits ARGB8888 values.
source	The location and dimensions of the source.
blitRect	A rectangle describing what region is to be drawn.
alpha	The alpha value to use for blending applied to the whole image (255 = solid, no blending)

nextLine

```
static int nextLine ( bool          rotatedDisplay ,
                      TextRotation textRotation
                      )
```

Find out how much to advance in the display buffer to get to the next line.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next line.

nextPixel

```
static int nextPixel ( bool          rotatedDisplay ,
                       TextRotation textRotation
                       )
```

Find out how much to advance in the display buffer to get to the next pixel.

Parameters:

rotatedDisplay	Is the display running in portrait mode?
textRotation	Rotation to perform.

Returns:

How much to advance to get to the next pixel.

Version: 4.16

LCD8RGBA2222DebugPrinter

The class LCD8RGBA2222DebugPrinter implements the DebugPrinter interface for printing debug messages on top of 8bit framebuffer.

See: DebugPrinter

Inherits from: DebugPrinter

Public Functions

virtual void	**draw**(const **Rect** & rect) const
	Draws the debug string on top of the framebuffer content.

Additional inherited members

Public Functions inherited from DebugPrinter

	DebugPrinter()
	Initializes a new instance of the **DebugPrinter** class.
const **Rect** &	**getRegion**() const
	Returns the region where the debug string is displayed.
void	**setColor**(**colortype** fg)
	Sets the foreground color of the debug string.
void	**setPosition**(uint16_t x, uint16_t y, uint16_t w, uint16_t h)
	Sets the position onscreen where the debug string will be displayed.
void	**setScale**(uint8_t scale)
	Sets the font scale of the debug string.
void	**setString**(const char * string)
	Sets the debug string to be displayed on top of the framebuffer.
virtual	**~DebugPrinter**()

	Finalizes an instance of the **DebugPrinter** class.

Protected Functions inherited from DebugPrinter

uint16_t	**getGlyph**(uint8_t c) const
	Gets a glyph (15 bits) arranged with 3 bits wide, 5 bits high in a single uint16_t value.

Protected Attributes inherited from DebugPrinter

colortype	**debugForegroundColor**
	Font color to use when displaying the debug string.
Rect	**debugRegion**
	Region onscreen where the debug message is displayed.
uint8_t	**debugScale**
	Font scaling factor to use when displaying the debug string.
const char *	**debugString**
	Debug string to be displayed onscreen.

Public Functions Documentation

draw

virtual void draw (const Rect & rect)

Draws the debug string on top of the framebuffer content.

Parameters:

rect The rect to draw inside.

Reimplements: **touchgfx::DebugPrinter::draw**

Version: 4.16

Line

Simple CanvasWidget capable of drawing a line from one point to another point. The end points can be moved to new locations and the line width can be set and changed. A 10 pixel long line along the top of the screen with a width on 1 pixel has endpoints in (0, 0.5) and (10, 0.5) and line width 1. The Line class calculates the corners of the shape, which in this case would be (0, 0), (10, 0), (10, 1) and (0, 1) and tells CanvasWidgetRenderer to moveTo() the first coordinate and then lineTo() the next coordinates in order. Finally it tells CWR to render the inside of the shape using the set Painter object.

The Line class caches the four corners of the shape to speed up redrawing. In general, drawing lines involve some extra mathematics for calculating the normal vector of the line and this computation would slow down re-draws if not cached.

Note: All coordinates are internally handled as CWRUtil::Q5 which means that floating point values are rounded down to a fixed number of binary digits, for example:

```
Line line;
line.setStart(1.1f, 1.1f); // Will use (35/32, 35/32) = (1.09375f, 1.09375f)
int x, y;
line.getStart(&amp;x, &amp;y); // Will return (1, 1)
```

.

Inherits from: CanvasWidget, Widget, Drawable

Public Types

enum	**LINE_ENDING_STYLE** { BUTT_CAP_ENDING, ROUND_CAP_ENDING, SQUARE_CAP_ENDING }
	Values that represent line ending styles.

Public Functions

virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
template \<typename T \> void	**getEnd**(T & x, T & y) const

	Gets the endpoint coordinates for the line.
LINE_ENDING_STYLE	**getLineEndingStyle**() const
	Gets line ending style.
template \<typename T \> T	**getLineWidth**() const
	Gets line width.
template \<typename T \> void	**getLineWidth**(T & width) const
	Gets line width.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
template \<typename T \> void	**getStart**(T & x, T & y) const
	Gets the starting point of the line as either integers or floats.
	Line()
void	**setCapPrecision**(int precision)
	Sets a precision of the arc at the ends of the **Line**.
void	**setEnd**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5)
	Sets the endpoint coordinates of the line.
template \<typename T \> void	**setEnd**(T x, T y)
	Sets the endpoint coordinates of the line.
template \<typename T \> void	**setLine**(T startX, T startY, T endX, T endY)
	Sets the starting point and ending point of the line.
void	**setLineEndingStyle**(**LINE_ENDING_STYLE** lineEnding)
	Sets line ending style.
void	**setLineWidth**(**CWRUtil::Q5** widthQ5)
	Sets the width for this **Line**.
template \<typename T \> void	**setLineWidth**(T width)

	Sets the width for this **Line**.
void	**setStart**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5)
	Sets the starting point of the line.
template \<typename T \> void	**setStart**(T x, T y)
	Sets the starting point of the line.
void	**updateEnd**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5)
	Update the endpoint for this **Line**.
template \<typename T \> void	**updateEnd**(T x, T y)
	Update the endpoint for this **Line**.
void	**updateLengthAndAngle**(**CWRUtil::Q5** length, **CWRUtil::Q5** angle)
	Update the end point for this **Line** given the new length and angle.
void	**updateLineWidth**(**CWRUtil::Q5** widthQ5)
	Update the width for this **Line**.
template \<typename T \> void	**updateLineWidth**(T width)
	Update the width for this **Line** and invalidates the minimal rectangle surrounding the line on screen.
void	**updateStart**(**CWRUtil::Q5** xQ5, **CWRUtil::Q5** yQ5)
	Update the start point for this **Line**.
template \<typename T \> void	**updateStart**(T x, T y)
	Update the start point for this **Line**.

Additional inherited members

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const

	Draws the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const
	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const

	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)

	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

LINE_ENDING_STYLE

enum LINE_ENDING_STYLE

Values that represent line ending styles.

BUTT_CAP_ENDING	The line ending is cut 90 degrees at the end of the line.
ROUND_CAP_ENDING	The line ending is rounded as a circle with center at the end of the line.
SQUARE_CAP_ENDING	The line ending is cut 90 degrees, but extends half the width of the line.

Public Functions Documentation

drawCanvasWidget

virtual bool drawCanvasWidget (const Rect & invalidatedArea)

Draw canvas widget for the given invalidated area.

Similar to **draw()**, but might be invoked several times with increasingly smaller areas to due to memory constraints from the underlying **CanvasWidgetRenderer**.

Parameters:

invalidatedArea The invalidated area.

Returns:

true if the widget was drawn properly, false if not.

See also:

draw

Reimplements: **touchgfx::CanvasWidget::drawCanvasWidget**

getEnd

```
void getEnd ( T & x ,       const
              T & y         const
              )     const
```

Gets the endpoint coordinates for the line.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate rounded down to the precision of T.
y The y coordinate rounded down to the precision of T.

See also:

setEnd, **updateEnd**

getLineEndingStyle

```
LINE_ENDING_STYLE getLineEndingStyle (  ) const
```

Gets line ending style.

Returns:

The line ending style.

See also:

LINE_ENDING_STYLE, **setLineEndingStyle**

getLineWidth

T getLineWidth ( ) const

Gets line width.

Template Parameters:

T Generic type parameter, either int or float.

Returns:

The line width rounded down to the precision of T.

See also:

setLineWidth

getLineWidth

void getLineWidth (T & width)

Gets line width.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

width The line width rounded down to the precision of T.

See also:

setLineWidth

getMinimalRect

virtual Rect getMinimalRect ( ) const

Gets minimal rectangle containing the shape drawn by this widget.

Default implementation returns the size of the entire widget, but this function should be overwritten in subclasses and return the minimal rectangle containing the shape. See classes such as **Circle** for example implementations.

Returns:

The minimal rectangle containing the shape drawn.

Reimplements: **touchgfx::CanvasWidget::getMinimalRect**

getStart

```
void getStart ( T & x ,      const
                T & y        const
                )    const
```

Gets the starting point of the line as either integers or floats.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate rounded down to the precision of T.
y The y coordinate rounded down to the precision of T.

See also:

setStart, **setLine**

Line

```
Line (  )
```

setCapPrecision

```
void setCapPrecision ( int precision )
```

Sets a precision of the arc at the ends of the **Line**.

This only works for ROUND_CAP_ENDING. The precision is given in degrees where 18 is the default which results in a nice half circle with 10 line segments. 90 will draw "an arrow head", 180 will look exactly like a BUTT_CAP_ENDING.

Parameters:

precision The new ROUND_CAP_ENDING precision.

NOTE

The line is not invalidated. This is only used if line ending is set to ROUND_CAP_ENDING.

setEnd

```
void setEnd ( CWRUtil::Q5 xQ5 ,
              CWRUtil::Q5 yQ5
              )
```

Sets the endpoint coordinates of the line.

Parameters:

xQ5 The x coordinate of the end point in Q5 format.
yQ5 The y coordinate of the end point in Q5 format.

NOTE

The area containing the **Line** is not invalidated.

See also:

updateEnd, **getEnd**

setEnd

```
void setEnd ( T x ,
              T y
              )
```

Sets the endpoint coordinates of the line.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of the end point.
y The y coordinate of the end point.

NOTE

The area containing the **Line** is not invalidated.

See also:

updateEnd, **getEnd**

setLine

```
void setLine ( T startX ,
               T startY ,
               T endX ,
               T endY
               )
```

Sets the starting point and ending point of the line.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

startX The x coordinate of the start point.
startY The y coordinate of the start point.
endX The x coordinate of the end point.
endY The y coordinate of the end point.

NOTE

The area containing the Line is not invalidated.

See also:

setStart, **setEnd**

setLineEndingStyle

```
void setLineEndingStyle ( LINE_ENDING_STYLE lineEnding )
```

Sets line ending style.

The same style is applied to both ends of the line.

Parameters:

lineEnding The line ending style.

NOTE

The area containing the Line is not invalidated.

See also:

LINE_ENDING_STYLE, **getLineEndingStyle**

setLineWidth

void setLineWidth (CWRUtil::Q5 widthQ5)

Sets the width for this **Line**.

Parameters:

widthQ5 The width of the line measured in pixels in Q5 format.

NOTE

The area containing the Line is not invalidated.

See also:

updateLineWidth

setLineWidth

void setLineWidth (T width)

Sets the width for this **Line**.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

width The width of the line measured in pixels.

NOTE

The area containing the Line is not invalidated.

See also:

updateLineWidth

setStart

void setStart (CWRUtil::Q5 xQ5 ,
CWRUtil::Q5 yQ5
)

Sets the starting point of the line.

Parameters:

xQ5 The x coordinate of the start point in Q5 format.
yQ5 The y coordinate of the start point in Q5 format.

NOTE

The area containing the Line is not invalidated.

See also:

updateStart, **getStart**, **setLine**, **setEnd**

setStart

```
void setStart ( T x ,
                T y
                )
```

Sets the starting point of the line.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of the start point.
y The y coordinate of the start point.

NOTE

The area containing the Line is not invalidated.

See also:

updateStart, **getStart**, **setLine**, **setEnd**

updateEnd

```
void updateEnd ( CWRUtil::Q5 xQ5 ,
                 CWRUtil::Q5 yQ5
                 )
```

Update the endpoint for this **Line**.

The rectangle that surrounds the line before and after will be invalidated.

Parameters:

xQ5 The x coordinate of the end point in Q5 format.
yQ5 The y coordinate of the end point in Q5 format.

NOTE

The area containing the Line is invalidated before and after the change.

See also:

setEnd, **updateStart**

updateEnd

```
void updateEnd ( T x ,
                 T y
                 )
```

Update the endpoint for this **Line**.

The rectangle that surrounds the line before and after will be invalidated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of the end point.
y The y coordinate of the end point.

NOTE

The area containing the Line is invalidated before and after the change.

See also:

setEnd, **updateStart**

updateLengthAndAngle

```
void updateLengthAndAngle ( CWRUtil::Q5 length ,
                            CWRUtil::Q5 angle
                            )
```

Update the end point for this **Line** given the new length and angle.

The rectangle that surrounds the line before and after will be invalidated. The starting coordinates will be fixed but the ending point will be updated. This is simply a different way to update the ending point.

Parameters:

length The new length of the line in Q5 format.
angle The new angle of the line in Q5 format.

NOTE

The area containing the Line is invalidated before and after the change.

See also:

updateEnd

updateLineWidth

void updateLineWidth (CWRUtil::Q5 widthQ5)

Update the width for this **Line**.

Update the width for this **Line** and invalidates the minimal rectangle surrounding the line on screen.

Parameters:

widthQ5 The width of the line measured in pixels in Q5 format.

NOTE

The area containing the Line is invalidated before and after the change.

See also:

setLineWidth

updateLineWidth

void updateLineWidth (T width)

Update the width for this **Line** and invalidates the minimal rectangle surrounding the line on screen.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

width The width of the line measured in pixels.

NOTE

The area containing the Line is invalidated before and after the change.

See also:

setLineWidth

updateStart

```
void updateStart ( CWRUtil::Q5 xQ5 ,
                   CWRUtil::Q5 yQ5
                   )
```

Update the start point for this **Line**.

The rectangle that surrounds the line before and after will be invalidated.

Parameters:

xQ5 The x coordinate of the start point in **CWRUtil::Q5** format.
yQ5 The y coordinate of the start point in **CWRUtil::Q5** format.

NOTE

The area containing the Line is invalidated before and after the change.

See also:

setStart, **updateEnd**

updateStart

```
void updateStart ( T x ,
                   T y
                   )
```

Update the start point for this **Line**.

The rectangle that surrounds the line before and after will be invalidated.

Template Parameters:

T Generic type parameter, either int or float.

Parameters:

x The x coordinate of the start point.
y The y coordinate of the start point.

NOTE

The area containing the **Line** is invalidated before and after the change.

See also:

setStart, **updateEnd**

Version: 4.16

LineProgress

Using Line from CanvasWidgetRenderer, progress will be rendered as a line. This means that the user must create a painter for painting the circle. The line does not need to horizontal or vertical, but can start at any coordinate and finish at any coordinate.

Note: As LineProgress uses CanvasWidgetRenderer, it is important that a buffer is set up by calling CanvasWidgetRendere::setBuffer().

Inherits from: AbstractProgressIndicator, Container, Drawable

Public Functions

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**getEnd**(int & x, int & y) const
	Gets the coordinates of the end point of the line.
virtual **Line::LINE_ENDING_STYLE**	**getLineEndingStyle**() const
	Gets line ending style.
virtual int	**getLineWidth**() const
	Gets the line width.
virtual void	**getStart**(int & x, int & y) const
	Gets the coordinates of the starting point of the line.
	LineProgress()
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setEnd**(int x, int y)
	Sets the end point for the line.
virtual void	**setLineEndingStyle**(**Line::LINE_ENDING_STYLE** lineEndingStyle)
	Sets line ending style.
virtual void	**setLineWidth**(int width)

	Sets the line width.
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter to be used for drawing the line.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the actual progress indicator relative to the background image.
virtual void	**setStart**(int x, int y)
	Sets a starting point for the line.
virtual void	**setValue**(int value)
	Sets the current value in the range (min..max) set by **setRange()**.

Protected Attributes

CWRUtil::Q5	**endX**
	The end x coordinate.
CWRUtil::Q5	**endY**
	The end y coordinate.
Line	**line**
	The line.
CWRUtil::Q5	**startX**
	The start x coordinate.
CWRUtil::Q5	**startY**
	The start y coordinate.

Additional inherited members

Public Functions inherited from AbstractProgressIndicator

AbstractProgressIndicator()

	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.
virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.

void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.
int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.

int	**rangeMin**
	The range minimum.
uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.

virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)

	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**

	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getEnd

virtual void getEnd (int & x , const
int & y const
) const

Gets the coordinates of the end point of the line.

Beware that this is not the coordinates of the current progress of the line, but the coordinates when the line is at 100%.

Parameters:

x The x coordinate.
y The y coordinate.

getLineEndingStyle

virtual Line::LINE_ENDING_STYLE getLineEndingStyle ( ) const

Gets line ending style.

Returns:

The line ending style.

getLineWidth

virtual int getLineWidth ( ) const

Gets the line width.

Returns:

The line width.

getStart

virtual void getStart (int & x , const
int & y const
) const

Gets the coordinates of the starting point of the line.

Parameters:

x The x coordinate.
y The y coordinate.

LineProgress

LineProgress ( )

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setEnd

virtual void setEnd (int x ,
int y
)

Sets the end point for the line.

When progress is at 100%, the line will go from the coordinates set by **setStart()** to the coordinates set by **setEnd()**

Parameters:

x The x coordinate of the end point.
y The y coordinate of the end point.

See also:

setStart

setLineEndingStyle

virtual void setLineEndingStyle (Line::LINE_ENDING_STYLE lineEndingStyle)

Sets line ending style.

Parameters:

lineEndingStyle The line ending style.

See also:

Line::setLineEndingStyle

setLineWidth

virtual void setLineWidth (int width)

Sets the line width.

Parameters:

width The width.

See also:

Line::setLineWidth

setPainter

virtual void setPainter (AbstractPainter & painter)

Sets a painter to be used for drawing the line.

This can be any Painter, a simple single color painter, a bitmap painter or a custom painter.

Parameters:

painter The painter.

setProgressIndicatorPosition

virtual void setProgressIndicatorPosition (int16_t x ,
int16_t y ,
int16_t width ,
int16_t height
)

Sets the position and dimensions of the actual progress indicator relative to the background image.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the box progress indicator.
height	The height of the box progress indicator.

See also:

getProgressIndicatorX, **getProgressIndicatorY**, **getProgressIndicatorWidth**, **getProgressIndicatorHeight**

Reimplements: **touchgfx::AbstractProgressIndicator::setProgressIndicatorPosition**

setStart

```
virtual void setStart ( int x ,
                        int y
                        )
```

Sets a starting point for the line.

Parameters:

x The x coordinate of the start point.
y The y coordinate of the start point.

See also:

setEnd

setValue

```
virtual void setValue ( int value )
```

Sets the current value in the range (min..max) set by **setRange()**.

Values lower than min are mapped to min, values higher than max are mapped to max. If a callback function has been set using setValueSetAction, that callback will be called (unless the new value is the same as the current value).

Parameters:

value The value.

NOTE

if value is equal to the current value, nothing happens, and the callback will not be called.

See also:

getValue, **updateValue**, **setValueSetAction**

Reimplements: **touchgfx::AbstractProgressIndicator::setValue**

Protected Attributes Documentation

endX

CWRUtil::Q5 endX

The end x coordinate.

endY

CWRUtil::Q5 endY

The end y coordinate.

line

Line line

The line.

startX

CWRUtil::Q5 startX

The start x coordinate.

startY

CWRUtil::Q5 startY

The start y coordinate.

Version: 4.16

ListLayout

This class provides a layout mechanism for arranging Drawable instances adjacent in the specified Direction. The first element in the ListLayout is positioned in the ListLayout origin (0,0). The dimensions of this class is automatically expanded to cover the area of the added Drawable instances, which may grow larger than the dimensions of the physical screen. Place the ListLayout inside e.g. a ScrollableContainer to allow all the children to be viewed.

See: ScrollableContainer

Inherits from: Container, Drawable

Public Functions

virtual void	**add**(**Drawable** & d)
	Adds a **Drawable** instance to the end of the list.
virtual **Direction**	**getDirection**() const
	Gets the direction of the **ListLayout**.
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a **Drawable** after a specific child node.
	ListLayout(const **Direction** d =**SOUTH**)
	Initializes a new instance of the **ListLayout** class.
virtual void	**remove**(**Drawable** & d)
	Removes a **Drawable**.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**setDirection**(const **Direction** d)
	Sets the direction of the **ListLayout**.

Additional inherited members

Public Functions inherited from Container

	Container()

virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.

virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.

virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.

virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.

virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const

	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

add

virtual void add (Drawable & d)

Adds a **Drawable** instance to the end of the list.

The **Drawable** dimensions shall be set prior to addition. The coordinates of the **Drawable** will be updated to reflect the position in the **ListLayout**.

Parameters:

d The **Drawable** to add.

Reimplements: **touchgfx::Container::add**

getDirection

virtual Direction getDirection ( ) const

Gets the direction of the **ListLayout**.

Returns:

The current direction to grow in when added children (either **SOUTH** or **EAST**).

See also:

setDirection

insert

virtual void insert (Drawable * previous ,
Drawable & d
)

Inserts a **Drawable** after a specific child node.

If previous child node is 0, the drawable will be inserted as the first element in the list. The first element in the list of children is the element drawn first, so this makes it possible to insert a **Drawable***behind* all previously added children.

Parameters:

previous	The **Drawable** to insert after. If null, insert as header.
d	The **Drawable** to insert.

NOTE

As with add, do not add the same drawable twice.

Reimplements: **touchgfx::Container::insert**

ListLayout

ListLayout (const Direction d =SOUTH)

Initializes a new instance of the **ListLayout** class.

Parameters:

d (Optional) The direction to place the elements. **SOUTH** (Default) places the elements vertically, **EAST** places the elements horizontally.

See also:

setDirection

remove

virtual void remove (Drawable & d)

Removes a **Drawable**.

Safe to call even if drawable has not been added. Other **Drawable** elements in the **ListLayout** are repositioned and the size of the **ListLayout** is adjusted.

Parameters:

d The drawable to remove.

Reimplements: **touchgfx::Container::remove**

removeAll

virtual void removeAll ( )

Removes all children in the **Container** by resetting their parent and sibling pointers.

Please note that this is not done recursively, so any child which is itself a **Container** is not emptied.

Reimplements: **touchgfx::Container::removeAll**

setDirection

virtual void setDirection (const Direction d)

Sets the direction of the **ListLayout**.

If elements have already been added to the **ListLayout**, these elements will be repositioned to adhere to the new direction.

Parameters:

d The new Direction to grow in when added children (either **SOUTH** or **EAST**).

See also:

getDirection

Version: 4.16

LockFreeDMA_Queue

This implements a simple lock-free FIFO queue (single producer, single consumer)

See: DMA_Queue

Inherits from: DMA_Queue

Public Functions

virtual bool	**isEmpty**()
	Query if this object is empty.
virtual bool	**isFull**()
	Query if this object is full.
	LockFreeDMA_Queue(**BlitOp** * mem, atomic_t n)
	Constructs a lock-free queue.
virtual void	**pushCopyOf**(const **BlitOp** & op)
	Adds the specified blitop to the queue.

Protected Functions

virtual const **BlitOp** *	**first**()
	Gets the first element in the queue.
virtual void	**pop**()
	Pops an element from the queue.

Protected Attributes

atomic_t	**capacity**
	The number of elements the queue can contain.
atomic_t	**head**

	Index to the head element.
BlitOp *	**q**
	Pointer to the queue memory.
atomic_t	**tail**
	Index to the tail element.

Additional inherited members

Public Functions inherited from DMA_Queue

virtual	**~DMA_Queue**()
	Finalizes an instance of the **DMA_Queue** class.

Protected Functions inherited from DMA_Queue

DMA_Queue()
Initializes a new instance of the **DMA_Queue** class.

Public Functions Documentation

isEmpty

virtual bool isEmpty ( )

Query if this object is empty.

Returns:

true if the queue is empty.

Reimplements: **touchgfx::DMA_Queue::isEmpty**

isFull

virtual bool isFull ( )

Query if this object is full.

Returns:

true if the queue is full.

Reimplements: **touchgfx::DMA_Queue::isFull**

LockFreeDMA_Queue

```
LockFreeDMA_Queue ( BlitOp *  mem ,
                    atomic_t n
                    )
```

Constructs a lock-free queue.

Parameters:

mem Pointer to the memory used by the queue to store elements.
n Number of elements the memory provided can contain.

pushCopyOf

```
virtual void pushCopyOf ( const BlitOp & op )
```

Adds the specified blitop to the queue.

Parameters:

op The blitop to add.

Reimplements: **touchgfx::DMA_Queue::pushCopyOf**

Protected Functions Documentation

first

```
virtual const BlitOp * first (  )
```

Gets the first element in the queue.

Returns:

The first element in the queue.

Reimplements: touchgfx::DMA_Queue::first

pop

virtual void pop ( )

Pops an element from the queue.

Reimplements: touchgfx::DMA_Queue::pop

Protected Attributes Documentation

capacity

atomic_t capacity

The number of elements the queue can contain.

head

atomic_t head

Index to the head element.

q

BlitOp * q

Pointer to the queue memory.

tail

atomic_t tail

Index to the tail element.

Version: 4.16

ManyBlockAllocator

This class is partial framebuffer allocator using multiple blocks. New buffers can be allocated until no free blocks are available. After transfer to LCD, a block is queued for allocation again.

See: FrameBufferAllocator

Inherits from: FrameBufferAllocator

Public Functions

virtual uint16_t	**allocateBlock**(const uint16_t x, const uint16_t y, const uint16_t width, const uint16_t height, uint8_t ** block)
	Allocates a framebuffer block.
virtual void	**freeBlockAfterTransfer**()
	Free a block after transfer to the **LCD**.
virtual const uint8_t *	**getBlockForTransfer**(**Rect** & rect)
	Get the block ready for transfer.
virtual bool	**hasBlockReadyForTransfer**()
	Check if a block is ready for transfer to the **LCD**.
virtual bool	**hasEmptyBlock**()
	Check if a block is ready for drawing (the block is empty).
	ManyBlockAllocator()
virtual void	**markBlockReadyForTransfer**()
	Marks a previously allocated block as ready to be transferred to the **LCD**.
virtual const **Rect** &	**peekBlockForTransfer**()
	Get the Rect of the next block to transfer.

Additional inherited members

Protected Types inherited from FrameBufferAllocator

enum	**BlockState** { EMPTY, ALLOCATED, DRAWN, SENDING }
	BlockState is used for internal state of each block.

Public Functions inherited from FrameBufferAllocator

virtual	**~FrameBufferAllocator**()
	Finalizes an instance of the **FrameBufferAllocator** class.

Public Functions Documentation

allocateBlock

```
virtual uint16_t allocateBlock ( const uint16_t x ,
                                 const uint16_t y ,
                                 const uint16_t width ,
                                 const uint16_t height ,
                                 uint8_t **       block
                                 )
```

Allocates a framebuffer block.

The block will have at least the width requested. The height of the allocated block can be lower than requested if not enough memory is available.

Parameters:

x	The absolute x coordinate of the block on the screen.
y	The absolute y coordinate of the block on the screen.
width	The width of the block.
height	The height of the block.
block	Pointer to pointer to return the block address in.

Returns:

The height of the allocated block.

Reimplements: **touchgfx::FrameBufferAllocator::allocateBlock**

freeBlockAfterTransfer

virtual void freeBlockAfterTransfer ( )

Free a block after transfer to the **LCD**.

Marks a previously allocated block as transferred and ready to reuse.

Reimplements: **touchgfx::FrameBufferAllocator::freeBlockAfterTransfer**

getBlockForTransfer

virtual const uint8_t * getBlockForTransfer (Rect & rect)

Get the block ready for transfer.

Parameters:

rect Reference to rect to write block x, y, width, and height.

Returns:

Returns the address of the block ready for transfer.

Reimplements: **touchgfx::FrameBufferAllocator::getBlockForTransfer**

hasBlockReadyForTransfer

virtual bool hasBlockReadyForTransfer ( )

Check if a block is ready for transfer to the **LCD**.

Returns:

True if a block is ready for transfer.

Reimplements: **touchgfx::FrameBufferAllocator::hasBlockReadyForTransfer**

hasEmptyBlock

virtual bool hasEmptyBlock ( )

Check if a block is ready for drawing (the block is empty).

Returns:

True if a block is empty.

Reimplements: **touchgfx::FrameBufferAllocator::hasEmptyBlock**

ManyBlockAllocator

ManyBlockAllocator ( )

markBlockReadyForTransfer

virtual void markBlockReadyForTransfer ( )

Marks a previously allocated block as ready to be transferred to the **LCD**.

Reimplements: **touchgfx::FrameBufferAllocator::markBlockReadyForTransfer**

peekBlockForTransfer

virtual const Rect & peekBlockForTransfer ( )

Get the Rect of the next block to transfer.

Returns:

Rect ready for transfer.

NOTE

This function should only be called when the allocator has a block ready for transfer.

See also:

hasBlockReadyForTransfer

Reimplements: **touchgfx::FrameBufferAllocator::peekBlockForTransfer**

Version: 4.16

Matrix4x4

This class represents row major 4x4 homogeneous matrices.

Public Functions

Matrix4x4 &	**concatenateXRotation**(float radians)
	Concatenate x coordinate rotation.
Matrix4x4 &	**concatenateXScale**(float distance)
	Concatenate x coordinate scale.
Matrix4x4 &	**concatenateXTranslation**(float distance)
	Concatenate x coordinate translation.
Matrix4x4 &	**concatenateYRotation**(float radians)
	Concatenate y coordinate rotation.
Matrix4x4 &	**concatenateYScale**(float distance)
	Concatenate y coordinate scale.
Matrix4x4 &	**concatenateYTranslation**(float distance)
	Concatenate y coordinate translation.
Matrix4x4 &	**concatenateZRotation**(float radians)
	Concatenate z coordinate rotation.
Matrix4x4 &	**concatenateZScale**(float distance)
	Concatenate z coordinate scale.
Matrix4x4 &	**concatenateZTranslation**(float distance)
	Concatenate z coordinate translation.
FORCE_INLINE_FUNCTION float	**getElement**(int row, int column) const
	Gets an element.
	Matrix4x4()
	Initializes a new instance of the Point4 class.
FORCE_INLINE_FUNCTION **Matrix4x4**	**setElement**(int row, int column, float value)

	Sets an element.
void	**setViewDistance**(float distance)
	Sets view distance.

Protected Attributes

float	**elements**
	The elements[4][4].

Public Functions Documentation

concatenateXRotation

Matrix4x4 & concatenateXRotation (float radians)

Concatenate x coordinate rotation.

Parameters:

radians The radians.

Returns:

A matrix_4x4&

concatenateXScale

Matrix4x4 & concatenateXScale (float distance)

Concatenate x coordinate scale.

Parameters:

distance The distance.

Returns:

A matrix_4x4&

concatenateXTranslation

Matrix4x4 & concatenateXTranslation (float distance)

Concatenate x coordinate translation.

Parameters:

distance The distance.

Returns:

A matrix_4x4&

concatenateYRotation

Matrix4x4 & concatenateYRotation (float radians)

Concatenate y coordinate rotation.

Parameters:

radians The radians.

Returns:

A matrix_4x4&

concatenateYScale

Matrix4x4 & concatenateYScale (float distance)

Concatenate y coordinate scale.

Parameters:

distance The distance.

Returns:

A matrix_4x4&

concatenateYTranslation

Matrix4x4 & concatenateYTranslation (float distance)

Concatenate y coordinate translation.

Parameters:

distance The distance.

Returns:

A matrix_4x4&

concatenateZRotation

Matrix4x4 & concatenateZRotation (float radians)

Concatenate z coordinate rotation.

Parameters:

radians The radians.

Returns:

A matrix_4x4&

concatenateZScale

Matrix4x4 & concatenateZScale (float distance)

Concatenate z coordinate scale.

Parameters:

distance The distance.

Returns:

A matrix_4x4&

concatenateZTranslation

Matrix4x4 & concatenateZTranslation (float distance)

Concatenate z coordinate translation.

Parameters:

distance The distance.

Returns:

A matrix_4x4&

getElement

```
FORCE_INLINE_FUNCTION float getElement ( int row ,     const
                                         int column    const
                                         ) const
```

Gets an element.

Parameters:

row	The row (0-3).
column	The column (0-3).

Returns:

The element.

Matrix4x4

```
Matrix4x4 ( )
```

Initializes a new instance of the Point4 class.

setElement

```
FORCE_INLINE_FUNCTION Matrix4x4 setElement ( int   row ,
                                             int   column ,
                                             float value
                                             )
```

Sets an element.

Parameters:

row	The row.
column	The column.
value	The value.

Returns:

A matrix_4x4&

setViewDistance

```
void setViewDistance ( float distance )
```

Sets view distance.

Parameters:

distance The distance.

Protected Attributes Documentation

elements

float elements

The elements[4][4].

Version: 4.16

MCUInstrumentation

Interface for instrumenting processors to measure MCU load via measured CPU cycles.

Public Functions

virtual uint32_t	**getCCConsumed**()
	Gets number of consumed clock cycles.
virtual unsigned int	**getCPUCycles**(void) =0
	Gets CPU cycles from register.
virtual unsigned int	**getElapsedUS**(unsigned int start, unsigned int now, unsigned int clockfrequency) =0
	Gets elapsed microseconds based on clock frequency.
virtual void	**init**() =0
	Initialize.
	MCUInstrumentation()
	Initializes a new instance of the **MCUInstrumentation** class.
virtual void	**setCCConsumed**(uint32_t val)
	Sets number of consumed clock cycles.
virtual void	**setMCUActive**(bool active)
	Sets MCU activity high.
virtual	**~MCUInstrumentation**()
	Finalizes an instance of the **MCUInstrumentation** class.

Protected Attributes

uint32_t	**cc_consumed**
	Amount of consumed CPU cycles.
uint32_t	**cc_in**
	Current CPU cycles.

Public Functions Documentation

getCCConsumed

virtual uint32_t getCCConsumed ( )

Gets number of consumed clock cycles.

Returns:

clock cycles.

getCPUCycles

virtual unsigned int getCPUCycles (void)

Gets CPU cycles from register.

Returns:

CPU cycles.

getElapsedUS

virtual unsigned int getElapsedUS (unsigned int start , =0
unsigned int now , =0
unsigned int clockfrequency =0
) =0

Gets elapsed microseconds based on clock frequency.

Parameters:

start	Start time.
now	Current time.
clockfrequency	Clock frequency of the system expressed in MHz.

Returns:

Elapsed microseconds start and now.

init

virtual void init ( ) =0

Initialize.

MCUInstrumentation

MCUInstrumentation ( )

Initializes a new instance of the **MCUInstrumentation** class.

setCCConsumed

virtual void setCCConsumed (uint32_t val)

Sets number of consumed clock cycles.

Parameters:

val number of clock cycles.

setMCUActive

virtual void setMCUActive (bool active)

Sets MCU activity high.

Parameters:

active if True, inactive otherwise.

~MCUInstrumentation

virtual ~MCUInstrumentation ( )

Finalizes an instance of the **MCUInstrumentation** class.

Protected Attributes Documentation

cc_consumed

uint32_t cc_consumed

Amount of consumed CPU cycles.

cc_in

uint32_t cc_in

Current CPU cycles.

Version: 4.16

ModalWindow

Container for displaying a modal window and hijacking touch event and prevent them from reaching the underlying view and widgets. The container has a background image and a surrounding box that acts as a shade on top of the rest of the screen. The background image must be set (using setBackground()) and the shade can be adjusted (using setShadeAlpha() and setShadeColor()).

The ModalWindow can either be used directly by adding widgets/containers to the ModalWindow from your view or by sub-classing it if you need a specific ModalWindow with predefined behavior across your application.

The ModalWindow should be instantiated in the view class and added as the last element (to always be on top, i.e. be modal). The ModalWindow will fill up the entire screen so it should always be placed at x=0, y=0 on the display.

To control the visibility of the ModalWindow use the show and hide methods.

Inherits from: Container, Drawable

Public Functions

virtual void	**add**(**Drawable** & d)
	Adds a **Drawable** instance as child to this **Container**.
virtual uint16_t	**getBackgroundHeight**() const
	Gets the height of the actual window (the background images).
virtual uint16_t	**getBackgroundWidth**() const
	Gets the width of the actual window (the background images).
virtual uint8_t	**getShadeAlpha**() const
	Gets the alpha value of the background shade.
virtual **colortype**	**getShadeColor**() const
	Gets the color of the background shade.
virtual void	**hide**()
	Make the **ModalWindow** invisible.
	ModalWindow()

virtual void	**remove**(**Drawable** & d)
	Removes a **Drawable** from the container by removing it from the linked list of children.
virtual void	**setBackground**(const **BitmapId** & bmpId)
	Sets the background of the actual window.
virtual void	**setBackground**(const **BitmapId** & bmpId, int16_t backgroundX, int16_t backgroundY)
	Sets the background of the actual window.
virtual void	**setShadeAlpha**(uint8_t alpha)
	Sets the alpha value of the background shade.
virtual void	**setShadeColor**(**colortype** color)
	Sets the color of the background shade.
virtual void	**show**()
	Make the **ModalWindow** visible.

Protected Attributes

Box	**backgroundShade**
	The background shade.
Image	**windowBackground**
	The window background.
Container	**windowContainer**
	The window container that defines the active container area where both the windowBackground and added drawables are placed.

Additional inherited members

Public Functions inherited from Container

Container()

virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()

	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)

	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

add

virtual void add (Drawable & d)

Adds a **Drawable** instance as child to this **Container**.

The **Drawable** added will be placed as the element to be drawn last, and thus appear on top of all previously added drawables in the **Container**.

Parameters:

d The **Drawable** to add.

NOTE

Never add a drawable more than once!

Reimplements: **touchgfx::Container::add**

getBackgroundHeight

virtual uint16_t getBackgroundHeight ( ) const

Gets the height of the actual window (the background images).

Whereas the **getHeight()** method will return the height including the shade.

Returns:

The height of the actual window.

getBackgroundWidth

virtual uint16_t getBackgroundWidth ( ) const

Gets the width of the actual window (the background images).

Whereas the **getWidth()** method will return the width including the shade.

Returns:

The width of the actual window.

getShadeAlpha

virtual uint8_t getShadeAlpha ( ) const

Gets the alpha value of the background shade.

Returns:

The background shades alpha.

getShadeColor

virtual colortype getShadeColor ( ) const

Gets the color of the background shade.

Returns:

The color of the background shade.

hide

virtual void hide ( )

Make the **ModalWindow** invisible.

ModalWindow

ModalWindow ( )

remove

virtual void remove (Drawable & d)

Removes a **Drawable** from the container by removing it from the linked list of children.

If the **Drawable** is not in the list of children, nothing happens. It is possible to remove an element from whichever **Container** it is a member of using:

```
if (d.getParent()) d.getParent()->remove(d);
```

The **Drawable** will have the parent and next sibling cleared, but is otherwise left unaltered.

Parameters:

d The **Drawable** to remove.

NOTE

This is safe to call even if d is not a child of this Container (in which case nothing happens).

Reimplements: **touchgfx::Container::remove**

setBackground

virtual void setBackground (const BitmapId & bmpId)

Sets the background of the actual window.

The remaining area of the screen will be covered by the shade. The background image is centered on the screen.

Parameters:

bmpId Identifier for the background bitmap.

setBackground

```
virtual void setBackground ( const BitmapId & bmpId ,
                             int16_t          backgroundX ,
                             int16_t          backgroundY
                             )
```

Sets the background of the actual window.

The remaining area of the screen will be covered by the shade. The background image will be placed at the backgroundX and backgroundY coordinate.

Parameters:

bmpId Identifier for the bitmap.
backgroundX The background x coordinate.
backgroundY The background y coordinate.

setShadeAlpha

virtual void setShadeAlpha (uint8_t alpha)

Sets the alpha value of the background shade.

Default, if not set, is 96.

Parameters:

alpha The new alpha.

setShadeColor

virtual void setShadeColor (colortype color)

Sets the color of the background shade.

Default is black.

Parameters:

color The new color.

show

virtual void show ( )

Make the **ModalWindow** visible.

Protected Attributes Documentation

backgroundShade

Box backgroundShade

The background shade.

windowBackground

Image windowBackground

The window background.

windowContainer

Container windowContainer

The window container that defines the active container area where both the windowBackground and added drawables are placed.

Version: 4.16

MoveAnimator

A MoveAnimator makes the template class T able to animate a movement from its current position to a specified end position. The speed of the movement in both the X and Y direction can be controlled by supplying EasingEquations. The MoveAnimator performs a callback when the animation has finished.

This mixin can be used on any Drawable.

Inherits from: T

Public Functions

void	**cancelMoveAnimation**()
	Cancel move animation and leave the **Drawable** in its current position.
void	**clearMoveAnimationEndedAction**()
	Clears the move animation ended action previously set by setMoveAnimationEndedAction.
virtual uint16_t	**getMoveAnimationDelay**() const
	Gets the current animation delay.
virtual void	**handleTickEvent**()
	The tick handler that handles the actual animation steps.
bool	**isMoveAnimationRunning**() const
	Gets whether or not the move animation is running.
	MoveAnimator()
virtual void	**setMoveAnimationDelay**(uint16_t delay)
	Sets a delay on animations done by the **MoveAnimator**.
void	**setMoveAnimationEndedAction**(**GenericCallback**< const **MoveAnimator**< T > & > & callback)
	Associates an action to be performed when the animation ends.

void	**startMoveAnimation**(int16_t endX, int16_t endY, uint16_t duration, **EasingEquation** xProgressionEquation =&**EasingEquations::linearEaseNone**, **EasingEquation** yProgressionEquation =&**EasingEquations::linearEaseNone**)
	Starts the move animation from the current position to the specified end position.

Protected Functions

void	**nextMoveAnimationStep**()
	Execute next step in move animation and stop the timer if the animation has finished.

Protected Attributes

uint16_t	**moveAnimationCounter**
	Counter that is equal to the current step in the animation.
uint16_t	**moveAnimationDelay**
	The delay applied before animation start. Expressed in ticks.
uint16_t	**moveAnimationDuration**
	The complete duration of the actual animation. Expressed in ticks.
GenericCallback< const **MoveAnimator**< T > & > *	**moveAnimationEndedCallback**
	Animation ended **Callback**.
int16_t	**moveAnimationEndX**
	The X value at the end of the animation.
int16_t	**moveAnimationEndY**
	The Y value at the end of the animation.
bool	**moveAnimationRunning**
	True if the animation is running.
int16_t	**moveAnimationStartX**
	The X value at the beginning of the animation.
int16_t	**moveAnimationStartY**

	The Y value at the beginning of the animation.
EasingEquation	**moveAnimationXEquation**
	EasingEquation expressing the development of the X value during the animation.
EasingEquation	**moveAnimationYEquation**
	EasingEquation expressing the development of the Y value during the animation.

Public Functions Documentation

cancelMoveAnimation

void cancelMoveAnimation ( )

Cancel move animation and leave the **Drawable** in its current position.

If the animation is not running, nothing is done.

clearMoveAnimationEndedAction

void clearMoveAnimationEndedAction ( )

Clears the move animation ended action previously set by setMoveAnimationEndedAction.

The effect is that any action set using **setMoveAnimationEndedAction()** will not be executed.

See also:

setMoveAnimationEndedAction

getMoveAnimationDelay

virtual uint16_t getMoveAnimationDelay ( ) const

Gets the current animation delay.

Returns:

The current animation delay.

See also:

setMoveAnimationDelay

handleTickEvent

virtual void handleTickEvent ( )

The tick handler that handles the actual animation steps.

isMoveAnimationRunning

bool isMoveAnimationRunning ( ) const

Gets whether or not the move animation is running.

Returns:

true if the move animation is running.

MoveAnimator

MoveAnimator ( )

setMoveAnimationDelay

virtual void setMoveAnimationDelay (uint16_t delay)

Sets a delay on animations done by the **MoveAnimator**.

Parameters:

delay The delay in ticks.

See also:

getMoveAnimationDelay

setMoveAnimationEndedAction

void setMoveAnimationEndedAction (GenericCallback< const MoveAnimator< T > & > & callback)

Associates an action to be performed when the animation ends.

Parameters:

callback	The callback to be executed. The callback will be given a reference to the **MoveAnimator**.

See also:

clearMoveAnimationEndedAction

startMoveAnimation

```
void startMoveAnimation ( int16_t endX ,
                          int16_t endY ,
                          uint16_t duration ,
                          EasingEquation xProgressionEquation =&EasingEquations::linearEaseNone,
                          EasingEquation yProgressionEquation =&EasingEquations::linearEaseNone
                          )
```

Starts the move animation from the current position to the specified end position.

The development of the position (X, Y) during the animation is described by the supplied **EasingEquations**. If no easing equation is given, the movement is performed linear.

Parameters:

endX	The X position at animation end.
endY	The Y position at animation end.
duration	The duration of the animation measured in ticks.
xProgressionEquation	(Optional) The equation that describes the development of the X position during the animation. Default is **EasingEquations::linearEaseNone**.
yProgressionEquation	(Optional) The equation that describes the development of the Y position during the animation. Default is **EasingEquations::linearEaseNone**.

Protected Functions Documentation

nextMoveAnimationStep

```
void nextMoveAnimationStep (  )
```

Execute next step in move animation and stop the timer if the animation has finished.

Protected Attributes Documentation

moveAnimationCounter

uint16_t moveAnimationCounter

Counter that is equal to the current step in the animation.

moveAnimationDelay

uint16_t moveAnimationDelay

The delay applied before animation start. Expressed in ticks.

moveAnimationDuration

uint16_t moveAnimationDuration

The complete duration of the actual animation. Expressed in ticks.

moveAnimationEndedCallback

GenericCallback< const MoveAnimator< T > & > * moveAnimationEndedCallback

Animation ended **Callback**.

moveAnimationEndX

int16_t moveAnimationEndX

The X value at the end of the animation.

moveAnimationEndY

int16_t moveAnimationEndY

The Y value at the end of the animation.

moveAnimationRunning

bool moveAnimationRunning

True if the animation is running.

moveAnimationStartX

int16_t moveAnimationStartX

The X value at the beginning of the animation.

moveAnimationStartY

int16_t moveAnimationStartY

The Y value at the beginning of the animation.

moveAnimationXEquation

EasingEquation moveAnimationXEquation

EasingEquation expressing the development of the X value during the animation.

moveAnimationYEquation

EasingEquation moveAnimationYEquation

EasingEquation expressing the development of the Y value during the animation.

Version: 4.16

MVPApplication

A specialization of the TouchGFX Application class that provides the necessary glue for transitioning between presenter/view pairs. It maintains a callback for transitioning and evaluates this at each tick.

See: Application

Inherits from: Application, UIEventListener

Public Functions

virtual void	**handlePendingScreenTransition**()
	Handles the pending screen transition.
	MVPApplication()
	Initializes a new instance of the **MVPApplication** class.

Protected Functions

void	**evaluatePendingScreenTransition**()
	Evaluates the pending **Callback** instances.

Protected Attributes

Presenter *	**currentPresenter**
	Pointer to the currently active presenter.
GenericCallback *	**pendingScreenTransitionCallback**
	Callback for screen transitions. Will be set to something valid when a transition request is made.

Additional inherited members

Protected Types inherited from Application

typedef Vector< Rect, 8 >	RectVector_t
	Type to ensure the same number of rects are in the Vector.

Public Functions inherited from Application

DebugPrinter *	getDebugPrinter()
	Returns the DebugPrinter object associated with the application.
Application *	getInstance()
	Gets the single instance application.
void	invalidateDebugRegion()
	Sets the debug string to be displayed onscreen on top of the framebuffer.
void	setDebugPrinter(DebugPrinter * printer)
	Sets the DebugPrinter object to be used by the application to print debug messages.
void	setDebugString(const char * string)
	Sets the debug string to be displayed onscreen on top of the framebuffer.
virtual void	appSwitchScreen(uint8_t screenId)
	An application specific function for switching screen.
virtual void	cacheDrawOperations(bool enableCache)
	This function allows for deferring draw operations to a later time.
void	clearAllTimerWidgets()
	Clears all currently registered timer widgets.
void	copyInvalidatedAreasFromTFTToClientBuffer()
	This function copies the parts that were updated in the previous frame (in the tft buffer) to the active framebuffer (client buffer).
virtual void	draw()
	Initiate a draw operation of the entire screen.
virtual void	draw(Rect & rect)
	Initiate a draw operation of the specified region of the screen.

Screen *	**getCurrentScreen**()
	Gets the current screen.
uint16_t	**getNumberOfRegisteredTimerWidgets**() const
	gets the number of timer widgets that has been registered.
uint16_t	**getTimerWidgetCountForDrawable**(const **Drawable** * w) const
	Gets the number of timer events registered to a widget, i.e.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Handle a click event.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Handle drag events.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Handle gestures.
virtual void	**handleKeyEvent**(uint8_t c)
	Handle an incoming character received by the HAL layer.
virtual void	**handleTickEvent**()
	Handle tick.
void	**registerTimerWidget**(**Drawable** * w)
	Adds a widget to the list of widgets receiving ticks every frame (typically 16.67ms)
virtual void	**requestRedraw**()
	An application specific function for requesting redraw of entire screen.
virtual void	**requestRedraw**(**Rect** & rect)
	An application specific function for requesting redraw of given Rect.
virtual void	**switchScreen**(**Screen** * newScreen)
	Switch to another Screen.
void	**unregisterTimerWidget**(const **Drawable** * w)
	Removes a widget from the list of widgets receiving ticks every frame (typically 16.67ms) milliseconds.

Protected Functions inherited from Application

	Application()
	Protected constructor.

void	**invalidateArea**(**Rect** area)
	Invalidates this area.

Public Attributes inherited from Application

const uint8_t	**MAX_TIMER_WIDGETS**
	Maximum number of widgets receiving ticks.

const uint16_t	**TICK_INTERVAL_MS**
	Deprecated, do not use this constant. Tick interval depends on VSYNC of your target platform.

Protected Attributes inherited from Application

RectVector_t	**cachedDirtyAreas**
	When draw caching is enabled, these rects keeps track of the dirty screen area.

bool	**drawCacheEnabled**
	True when draw caching is active.

RectVector_t	**lastRects**
	The dirty areas from last frame that needs to be redrawn because we have swapped frame buffers.

Rect	**redraw**
	Rect describing application requested invalidate area.

uint8_t	**timerWidgetCounter**
	A counter for each potentially registered timer widget. Increase when registering for timer events, decrease when unregistering.

Vector< **Drawable** *, **MAX_TIMER_WIDGETS** >	**timerWidgets**
	List of widgets that receive timer ticks.

bool	**transitionHandled**

	True if the transition is done and Screen::afterTransition has been called.
Screen *	**currentScreen**
	Pointer to currently displayed Screen.
Transition *	**currentTransition**
	Pointer to current transition.
DebugPrinter *	**debugPrinter**
	Pointer to the DebugPrinter instance.
Rect	**debugRegionInvalidRect**
	Invalidated Debug Region.
Application *	**instance**
	Pointer to the instance of the Application-derived subclass.

Public Functions inherited from UIEventListener

virtual void	**handleClickEvent**(const **ClickEvent** & event)
	This handler is invoked when a mouse click or display touch event has been detected by the system.
virtual void	**handleDragEvent**(const **DragEvent** & event)
	This handler is invoked when a drag event has been detected by the system.
virtual void	**handleGestureEvent**(const **GestureEvent** & event)
	This handler is invoked when a gesture event has been detected by the system.
virtual void	**handleKeyEvent**(uint8_t c)
	This handler is invoked when a key (or button) event has been detected by the system.
virtual void	**handleTickEvent**()
	This handler is invoked when a system tick event has been generated.
virtual	**~UIEventListener**()
	Finalizes an instance of the **UIEventListener** class.

Public Functions Documentation

handlePendingScreenTransition

virtual void handlePendingScreenTransition ( )

Handles the pending screen transition.

Delegates the work to **evaluatePendingScreenTransition()**

Reimplements: **touchgfx::Application::handlePendingScreenTransition**

MVPApplication

MVPApplication ( )

Initializes a new instance of the **MVPApplication** class.

Protected Functions Documentation

evaluatePendingScreenTransition

void evaluatePendingScreenTransition ( )

Evaluates the pending **Callback** instances.

If a callback is valid, it is executed and a **Screen** transition is executed.

Protected Attributes Documentation

currentPresenter

Presenter * currentPresenter

Pointer to the currently active presenter.

pendingScreenTransitionCallback

GenericCallback * pendingScreenTransitionCallback

Callback for screen transitions. Will be set to something valid when a transition request is made.

Version: 4.16

MVPHeap

Generic heap class for MVP applications. Serves as a way of obtaining the memory storage areas for presenters, screens, transitions and the concrete application.

Subclassed by an application-specific heap which provides the actual storage areas. This generic interface is used only in makeTransition.

Public Functions

	MVPHeap(**AbstractPartition** & pres, **AbstractPartition** & scr, **AbstractPartition** & tra, **MVPApplication** & app)
	Initializes a new instance of the **MVPHeap** class.
virtual	**~MVPHeap**()
	Finalizes an instance of the **MVPHeap** class.

Public Attributes

MVPApplication &	**frontendApplication**
	A reference to the **MVPApplication** instance.
AbstractPartition &	**presenterStorage**
	A memory partition containing enough memory to hold the largest presenter.
AbstractPartition &	**screenStorage**
	A memory partition containing enough memory to hold the largest view.
AbstractPartition &	**transitionStorage**
	A memory partition containing enough memory to hold the largest transition.

Public Functions Documentation

MVPHeap

```
MVPHeap ( AbstractPartition & pres ,
          AbstractPartition & scr ,
          AbstractPartition & tra ,
          MVPApplication &  app
          )
```

Initializes a new instance of the **MVPHeap** class.

Parameters:

pres	A memory partition containing enough memory to hold the largest presenter.
scr	A memory partition containing enough memory to hold the largest view.
tra	A memory partition containing enough memory to hold the largest transition.
app	A reference to the **MVPApplication** instance.

~MVPHeap

```
virtual ~MVPHeap (  )
```

Finalizes an instance of the **MVPHeap** class.

Public Attributes Documentation

frontendApplication

MVPApplication & frontendApplication

A reference to the **MVPApplication** instance.

presenterStorage

AbstractPartition & presenterStorage

A memory partition containing enough memory to hold the largest presenter.

screenStorage

AbstractPartition & screenStorage

A memory partition containing enough memory to hold the largest view.

transitionStorage

AbstractPartition & transitionStorage

A memory partition containing enough memory to hold the largest transition.

Version: 4.16

NoDMA

This is an "empty" DMA subclass that does nothing except assert if accidentally used. An instance of this object can be used if DMA support is not desired.

See: DMA_Interface

Inherits from: DMA_Interface

Public Functions

virtual void	**flush**()
	Block until all DMA transfers are complete.
virtual **BlitOperations**	**getBlitCaps**()
	No blit operations supported by this DMA implementation.
	NoDMA()
virtual void	**setupDataCopy**(const **BlitOp** & blitOp)
	Asserts if used.
virtual void	**setupDataFill**(const **BlitOp** & blitOp)
	Asserts if used.
virtual void	**signalDMAInterrupt**()
	Does nothing.

Additional inherited members

Public Functions inherited from DMA_Interface

virtual void	**addToQueue**(const **BlitOp** & op)
	Inserts a **BlitOp** for processing.
bool	**getAllowed**() const
	Gets whether a DMA operation is allowed to begin.

virtual **DMAType**	**getDMAType**(void)
	Function for obtaining the DMA type of the concrete **DMA_Interface** implementation.
virtual void	**initialize**()
	Perform initialization.
uint8_t	**isDmaQueueEmpty**()
	Query if the DMA queue is empty.
uint8_t	**isDmaQueueFull**()
	Query if the DMA queue is full.
bool	**isDMARunning**()
	Query if the DMA is running.
void	**setAllowed**(bool allowed)
	Sets whether or not a DMA operation is allowed to begin.
virtual void	**start**()
	Signals that DMA transfers can start.
virtual	**~DMA_Interface**()
	Finalizes an instance of the **DMA_Interface** class.

Protected Functions inherited from DMA_Interface

virtual void	**disableAlpha**()
	Configures blit-op hardware for solid operation (no alpha-blending)
	DMA_Interface(**DMA_Queue** & dmaQueue)
	Constructs a DMA Interface object.
virtual void	**enableAlpha**(uint8_t alpha)
	Configures blit-op hardware for alpha-blending.
virtual void	**enableCopyWithTransparentPixels**(uint8_t alpha)
	Configures blit-op hardware for alpha-blending while simultaneously skipping transparent pixels.
virtual void	**execute**()
	Performs a queued blit-op.

virtual void	**executeCompleted**()
	To be called when blit-op has been performed.
virtual void	**seedExecution**()
	Called when elements are added to the DMA-queue.
virtual void	**waitForFrameBufferSemaphore**()
	Waits until framebuffer semaphore is available (i.e.

Protected Attributes inherited from DMA_Interface

bool	**isAllowed**
	true if DMA transfers are currently allowed.
bool	**isRunning**
	true if a DMA transfer is currently ongoing.
DMA_Queue &	**queue**
	Reference to the DMA queue.

Public Functions Documentation

flush

virtual void flush ( )

Block until all DMA transfers are complete.

Since this particular DMA does not do anything, return immediately.

Reimplements: **touchgfx::DMA_Interface::flush**

getBlitCaps

virtual BlitOperations getBlitCaps ( )

No blit operations supported by this DMA implementation.

Returns:

Zero (no blit ops supported).

Reimplements: **touchgfx::DMA_Interface::getBlitCaps**

NoDMA

NoDMA ( )

setupDataCopy

virtual void setupDataCopy (const BlitOp & blitOp)

Asserts if used.

Parameters:

blitOp The blit operation to be performed by this DMA instance.

Reimplements: **touchgfx::DMA_Interface::setupDataCopy**

setupDataFill

virtual void setupDataFill (const BlitOp & blitOp)

Asserts if used.

Parameters:

blitOp The blit operation to be performed by this DMA instance.

Reimplements: **touchgfx::DMA_Interface::setupDataFill**

signalDMAInterrupt

virtual void signalDMAInterrupt ( )

Does nothing.

Reimplements: **touchgfx::DMA_Interface::signalDMAInterrupt**

Version: 4.16

NoTouchController

Empty TouchController implementation which does nothing. Use this if your display does not have touch input capabilities.

Inherits from: TouchController

Public Functions

virtual void	**init**()
	Initializes touch controller.
virtual bool	**sampleTouch**(int32_t & x, int32_t & y)
	Checks whether the touch screen is being touched, and if so, what coordinates.

Additional inherited members

Public Functions inherited from TouchController

virtual	**~TouchController**()
	Finalizes an instance of the **TouchController** class.

Public Functions Documentation

init

virtual void init ( )

Initializes touch controller.

Reimplements: **touchgfx::TouchController::init**

sampleTouch

```
virtual bool sampleTouch ( int32_t & x ,
                           int32_t & y
                           )
```

Checks whether the touch screen is being touched, and if so, what coordinates.

Parameters:

x The x position of the touch.
y The y position of the touch.

Returns:

True if a touch has been detected, otherwise false.

Reimplements: **touchgfx::TouchController::sampleTouch**

Version: 4.16

NoTransition

The most simple Transition without any visual effects. THe screen transition is done by immediately replace the current Screen with a new Screen.

See: Transition

Inherits from: Transition

Public Functions

virtual void	**handleTickEvent**()
	Indicates that the transition is done after the first tick.

Additional inherited members

Public Functions inherited from Transition

virtual void	**init**()
	Initializes the transition.
virtual void	**invalidate**()
	Invalidates the screen when starting the **Transition**.
bool	**isDone**() const
	Query if the transition is done transitioning.
virtual void	**setScreenContainer**(**Container** & cont)
	Sets the **ScreenContainer**.
virtual void	**tearDown**()
	Tears down the Animation.
	Transition()
	Initializes a new instance of the **Transition** class.
virtual	**~Transition**()

	Finalizes an instance of the **Transition** class.

Protected Attributes inherited from Transition

bool	**done**
	Flag that indicates when the transition is done. This should be set by implementing classes.
Container *	**screenContainer**
	The screen **Container** of the **Screen** transitioning to.

Public Functions Documentation

handleTickEvent

virtual void handleTickEvent ( )

Indicates that the transition is done after the first tick.

Reimplements: **touchgfx::Transition::handleTickEvent**

Version: 4.16

OSWrappers

This class specifies OS wrappers for dealing with the framebuffer semaphore and the VSYNC signal.

Public Functions

void	**giveFrameBufferSemaphore**()
	Release the framebuffer semaphore.
void	**giveFrameBufferSemaphoreFromISR**()
	Release the framebuffer semaphore in a way that is safe in interrupt context.
void	**initialize**()
	Initialize framebuffer semaphore and queue/mutex for VSYNC signal.
bool	**isVSyncAvailable**()
	This function checks if a VSync occurred after last rendering.
void	**signalRenderingDone**()
	Signal that the rendering of the frame has completed.
void	**signalVSync**()
	Signal that a VSYNC has occurred.
void	**takeFrameBufferSemaphore**()
	Take the framebuffer semaphore.
void	**taskDelay**(uint16_t ms)
	A function that causes executing task to sleep for a number of milliseconds.
void	**tryTakeFrameBufferSemaphore**()
	Attempt to obtain the framebuffer semaphore.
void	**waitForVSync**()
	This function blocks until a VSYNC occurs.

Public Functions Documentation

giveFrameBufferSemaphore

static void giveFrameBufferSemaphore ( )

Release the framebuffer semaphore.

giveFrameBufferSemaphoreFromISR

static void giveFrameBufferSemaphoreFromISR ( )

Release the framebuffer semaphore in a way that is safe in interrupt context.

Called from ISR.

initialize

static void initialize ( )

Initialize framebuffer semaphore and queue/mutex for VSYNC signal.

isVSyncAvailable

static bool isVSyncAvailable ( )

This function checks if a VSync occurred after last rendering.

The function is used in systems that cannot wait in waitForVSync (because they are also checking other event sources.

Returns:

True if VSync occurred.

NOTE

signalRenderingDone is typically used together with this function.

signalRenderingDone

static void signalRenderingDone ( )

Signal that the rendering of the frame has completed.

Used by some systems to avoid using any previous vsync.

signalVSync

static void signalVSync ( )

Signal that a VSYNC has occurred.

Should make the vsync queue/mutex available.

NOTE

This function is called from an ISR, and should (depending on OS) trigger a scheduling.

takeFrameBufferSemaphore

static void takeFrameBufferSemaphore ( )

Take the framebuffer semaphore.

Blocks until semaphore is available.

taskDelay

static void taskDelay (uint16_t ms)

A function that causes executing task to sleep for a number of milliseconds.

This function is OPTIONAL. It is only used by the TouchGFX in the case of a specific frame refresh strategy (REFRESH_STRATEGY_OPTIM_SINGLE_BUFFER_TFT_CTRL). Due to backwards compatibility, in order for this function to be usable by the **HAL** the function must be explicitly registered: hal.registerTaskDelayFunction(&OSWrappers::taskDelay)

Parameters:

ms The number of milliseconds to sleep.

See also:

HAL::setFrameRefreshStrategy, **HAL::registerTaskDelayFunction**

tryTakeFrameBufferSemaphore

static void tryTakeFrameBufferSemaphore ( )

Attempt to obtain the framebuffer semaphore.

If semaphore is not available, do nothing.

NOTE

must return immediately! This function does not care who has the taken the semaphore, it only serves to make sure that the semaphore is taken by someone.

waitForVSync

static void waitForVSync ( )

This function blocks until a VSYNC occurs.

NOTE

This function must first clear the mutex/queue and then wait for the next one to occur.

Version: 4.16

PainterABGR2222

The PainterABGR2222 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterABGR2222, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterABGR2222(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterABGR2222** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterBlue**
	The blue part of the color, scaled up to [0..255].
uint8_t	**painterColor**
	The color.

uint8_t	**painterGreen**
	The green part of the color, scaled up to [0..255].
uint8_t	**painterRed**
	The red part of the color, scaled up to [0..255].

Additional inherited members

Public Functions inherited from AbstractPainterABGR2222

	AbstractPainterABGR2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterABGR2222

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterABGR2222

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**

	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**

The alpha of the widget using the painter.

Public Functions Documentation

getColor

```
colortype getColor ( ) const
```

Gets the current color.

Returns:

The color.

PainterABGR2222

```
PainterABGR2222 ( colortype color =0,
                  uint8_t   alpha =255
                  )
```

Initializes a new instance of the **PainterABGR2222** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterABGR2222::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color	The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.

green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterABGR2222::renderNext**

Protected Attributes Documentation

painterBlue

uint8_t painterBlue

The blue part of the color, scaled up to [0..255].

painterColor

uint8_t painterColor

The color.

painterGreen

uint8_t painterGreen

The green part of the color, scaled up to [0..255].

painterRed

uint8_t painterRed

The red part of the color, scaled up to [0..255].

Version: 4.16

PainterABGR2222Bitmap

PainterABGR2222Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterABGR2222, AbstractPainter

Public Functions

	PainterABGR2222Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterABGR2222Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapABGR2222Pointer**
	Pointer to the bitmap (ABGR2222)

Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterABGR2222

	AbstractPainterABGR2222()

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterABGR2222

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterABGR2222

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterABGR2222Bitmap

```
PainterABGR2222Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
                        )
```

Initializes a new instance of the **PainterABGR2222Bitmap** class.

Parameters:

bmp (Optional) the bitmap, default is **BITMAP_INVALID**.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainterABGR2222::render

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: touchgfx::AbstractPainterABGR2222::renderInit

renderNext

virtual bool renderNext (uint8_t & red ,
uint8_t & green ,
uint8_t & blue ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterABGR2222::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapABGR2222Pointer

const uint8_t * bitmapABGR2222Pointer

Pointer to the bitmap (ABGR2222)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterARGB2222

The PainterARGB2222 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterARGB2222, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterARGB2222(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterARGB2222** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterBlue**
	The blue part of the color, scaled up to [0..255].
uint8_t	**painterColor**
	The color.

uint8_t	**painterGreen**
	The green part of the color, scaled up to [0..255].
uint8_t	**painterRed**
	The red part of the color, scaled up to [0..255].

Additional inherited members

Public Functions inherited from AbstractPainterARGB2222

	AbstractPainterARGB2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterARGB2222

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterARGB2222

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**

	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

getColor

```
colortype getColor ( ) const
```

Gets the current color.

Returns:

The color.

PainterARGB2222

```
PainterARGB2222 ( colortype color =0,
                  uint8_t   alpha =255
                  )
```

Initializes a new instance of the **PainterARGB2222** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterARGB2222::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color	The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.

green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterARGB2222::renderNext**

Protected Attributes Documentation

painterBlue

uint8_t painterBlue

The blue part of the color, scaled up to [0..255].

painterColor

uint8_t painterColor

The color.

painterGreen

uint8_t painterGreen

The green part of the color, scaled up to [0..255].

painterRed

uint8_t painterRed

The red part of the color, scaled up to [0..255].

Version: 4.16

PainterARGB2222Bitmap

PainterARGB2222Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterARGB2222, AbstractPainter

Public Functions

	PainterARGB2222Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Constructor.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapARGB2222Pointer**
	Pointer to the bitmap (ARGB2222)

Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterARGB2222

	AbstractPainterARGB2222()

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterARGB2222

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterARGB2222

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterARGB2222Bitmap

```
PainterARGB2222Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
                        )
```

Constructor.

Parameters:

bmp (Optional) The bitmap, default is **BITMAP_INVALID**.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainterARGB2222::render

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: touchgfx::AbstractPainterARGB2222::renderInit

renderNext

virtual bool renderNext (uint8_t & red ,
uint8_t & green ,
uint8_t & blue ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterARGB2222::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapARGB2222Pointer

const uint8_t * bitmapARGB2222Pointer

Pointer to the bitmap (ARGB2222)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterARGB8888

The PainterARGB8888 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterARGB8888, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterARGB8888(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterARGB8888** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterBlue**
	The blue part of the color.
uint8_t	**painterGreen**
	The green part of the color.

uint8_t	**painterRed**
	The red part of the color.

Additional inherited members

Public Functions inherited from AbstractPainterARGB8888

	AbstractPainterARGB8888()

Protected Functions inherited from AbstractPainterARGB8888

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.
virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue, uint8_t alpha)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterARGB8888

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.

virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

getColor

colortype getColor ( ) const

Gets the current color.

Returns:

The color.

PainterARGB8888

```
PainterARGB8888 ( colortype color =0,
                  uint8_t   alpha =255
                  )
```

Initializes a new instance of the **PainterARGB8888** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: **touchgfx::AbstractPainterARGB8888::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterARGB8888::renderNext**

Protected Attributes Documentation

painterBlue

uint8_t painterBlue

The blue part of the color.

painterGreen

uint8_t painterGreen

The green part of the color.

painterRed

uint8_t painterRed

The red part of the color.

Version: 4.16

PainterARGB8888Bitmap

PainterARGB8888Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterARGB8888, AbstractPainter

Public Functions

	PainterARGB8888Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterARGB8888Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint32_t *	**bitmapARGB8888Pointer**
	Pointer to the bitmap (ARGB8888)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.
const uint16_t *	**bitmapRGB565Pointer**
	Pointer to the bitmap (RGB565)
const uint8_t *	**bitmapRGB888Pointer**
	Pointer to the bitmap (RGB888)

Additional inherited members

Public Functions inherited from AbstractPainterARGB8888

AbstractPainterARGB8888()

Protected Functions inherited from AbstractPainterARGB8888

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.
virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue, uint8_t alpha)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterARGB8888

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterARGB8888Bitmap

```
PainterARGB8888Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
                        )
```

Initializes a new instance of the **PainterARGB8888Bitmap** class.

Parameters:

bmp (Optional) The bitmap, default is **BITMAP_INVALID**.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainterARGB8888::render

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: touchgfx::AbstractPainterARGB8888::renderInit

renderNext

virtual bool renderNext (uint8_t & red ,
uint8_t & green ,
uint8_t & blue ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterARGB8888::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapARGB8888Pointer

const uint32_t * bitmapARGB8888Pointer

Pointer to the bitmap (ARGB8888)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

bitmapRGB565Pointer

const uint16_t * bitmapRGB565Pointer

Pointer to the bitmap (RGB565)

bitmapRGB888Pointer

const uint8_t * bitmapRGB888Pointer

Pointer to the bitmap (RGB888)

Version: 4.16

PainterARGB8888L8Bitmap

PainterARGB8888L8Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterARGB8888, AbstractPainter

Public Functions

	PainterARGB8888L8Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterARGB8888L8Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapExtraPointer**
	Pointer to the CLUT (L8)
const uint8_t *	**bitmapPointer**
	Pointer to the bitmap (L8)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterARGB8888

AbstractPainterARGB8888()

Protected Functions inherited from AbstractPainterARGB8888

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.
virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue, uint8_t alpha)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterARGB8888

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

AbstractPainter()
Initializes a new instance of the **AbstractPainter** class.

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterARGB8888L8Bitmap

```
PainterARGB8888L8Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                          uint8_t        alpha =255
                          )
```

Initializes a new instance of the **PainterARGB8888L8Bitmap** class.

Parameters:

bmp	(Optional) The bitmap, default is **BITMAP_INVALID**.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterARGB8888::render**

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterARGB8888::renderInit**

renderNext

virtual bool renderNext (uint8_t & red ,
uint8_t & green ,
uint8_t & blue ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterARGB8888::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapExtraPointer

const uint8_t * bitmapExtraPointer

Pointer to the CLUT (L8)

bitmapPointer

const uint8_t * bitmapPointer

Pointer to the bitmap (L8)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterBGRA2222

The PainterBGRA2222 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterBGRA2222, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterBGRA2222(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterBGRA2222** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterBlue**
	The blue part of the color, scaled up to [0..255].
uint8_t	**painterColor**
	The color.

uint8_t	**painterGreen**
	The green part of the color, scaled up to [0..255].
uint8_t	**painterRed**
	The red part of the color, scaled up to [0..255].

Additional inherited members

Public Functions inherited from AbstractPainterBGRA2222

	AbstractPainterBGRA2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterBGRA2222

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterBGRA2222

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**

	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**

The alpha of the widget using the painter.

Public Functions Documentation

getColor

```
colortype getColor ( ) const
```

Gets the current color.

Returns:

The color.

PainterBGRA2222

```
PainterBGRA2222 ( colortype color =0,
                  uint8_t   alpha =255
                  )
```

Initializes a new instance of the **PainterBGRA2222** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterBGRA2222::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color	The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.

green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterBGRA2222::renderNext**

Protected Attributes Documentation

painterBlue

uint8_t painterBlue

The blue part of the color, scaled up to [0..255].

painterColor

uint8_t painterColor

The color.

painterGreen

uint8_t painterGreen

The green part of the color, scaled up to [0..255].

painterRed

uint8_t painterRed

The red part of the color, scaled up to [0..255].

Version: 4.16

PainterBGRA2222Bitmap

PainterBGRA2222Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterBGRA2222, AbstractPainter

Public Functions

	PainterBGRA2222Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterBGRA2222Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapBGRA2222Pointer**
	Pointer to the bitmap (BGRA2222)

Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterBGRA2222

	AbstractPainterBGRA2222()

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterBGRA2222

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterBGRA2222

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterBGRA2222Bitmap

```
PainterBGRA2222Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
                        )
```

Initializes a new instance of the **PainterBGRA2222Bitmap** class.

Parameters:

bmp (Optional) The bitmap, default is **BITMAP_INVALID**.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainterBGRA2222::render

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: touchgfx::AbstractPainterBGRA2222::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterBGRA2222::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapBGRA2222Pointer

const uint8_t * bitmapBGRA2222Pointer

Pointer to the bitmap (BGRA2222)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterBW

PainterBW is used for drawing one 1bpp displays. The color is either on or off. No transparency is supported.

See: AbstractPainter

Inherits from: AbstractPainterBW, AbstractPainter

Public Functions

unsigned	**bw**(unsigned red, unsigned green, unsigned blue)
	Converts the selected color to either white (1) or black (0) depending on the gray representation of the RGB color.
colortype	**getColor**() const
	Gets the current color.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & color)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterColor**
	The color to use when painting.

Additional inherited members

Public Functions inherited from AbstractPainterBW

	AbstractPainterBW()

Protected Functions inherited from AbstractPainterBW

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.

Protected Attributes inherited from AbstractPainterBW

uint16_t	**currentX**
	Current x coordinate relative to the widget.
uint16_t	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

bw

```
static unsigned bw ( unsigned red ,
                     unsigned green ,
                     unsigned blue
                     )
```

Converts the selected color to either white (1) or black (0) depending on the gray representation of the RGB color.

Parameters:

red	The red color.
green	The green color.
blue	The blue color.

Returns:

1 (white) if the brightness of the RGB color is above 50% and 0 (black) otherwise.

getColor

colortype getColor ( ) const

Gets the current color.

Returns:

The color.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: **touchgfx::AbstractPainterBW::render**

setColor

void setColor (colortype color)

Sets color to use when drawing the **CanvasWidget**.

Parameters:

color The color, 0=black, otherwise white.

Protected Functions Documentation

renderNext

virtual bool renderNext (uint8_t & color)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

color The color (0 or 1).

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterBW::renderNext**

Protected Attributes Documentation

painterColor

uint8_t painterColor

The color to use when painting.

Version: 4.16

PainterBWBitmap

PainterBWBitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterBW, AbstractPainter

Public Functions

	PainterBWBitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**))
	Initializes a new instance of the **PainterBWBitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & color)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.
const uint8_t *	**bitmapBWPointer**

	Pointer to the bitmap (BW)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.
LCD1bpp::bwRLEdata	**bw_rle**
	Pointer to class for walking through bw_rle image.

Additional inherited members

Public Functions inherited from AbstractPainterBW

	AbstractPainterBW()

Protected Attributes inherited from AbstractPainterBW

uint16_t	**currentX**
	Current x coordinate relative to the widget.
uint16_t	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterBWBitmap

PainterBWBitmap (const Bitmap & bmp =Bitmap(BITMAP_INVALID))

Initializes a new instance of the **PainterBWBitmap** class.

Parameters:

bmp (Optional) The bitmap, default is **BITMAP_INVALID**.

render

virtual void render (uint8_t * ptr ,
int x ,

```
                int             xAdjust ,
                int             y ,
                unsigned        count ,
                const uint8_t * covers
                )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterBW::render**

setBitmap

```
void setBitmap ( const Bitmap & bmp )
```

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp	The bitmap.

Protected Functions Documentation

renderInit

```
virtual bool renderInit (  )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterBW::renderInit**

renderNext

virtual bool renderNext (uint8_t & color)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

color The color (0 or 1).

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterBW::renderNext**

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapBWPointer

const uint8_t * bitmapBWPointer

Pointer to the bitmap (BW)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

bw_rle

LCD1bpp::bwRLEdata bw_rle

Pointer to class for walking through bw_rle image.

Version: 4.16

PainterGRAY2

The PainterGRAY2 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterGRAY2, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterGRAY2(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterGRAY2** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & gray, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterGray**
	The gray color.

Additional inherited members

Public Functions inherited from AbstractPainterGRAY2

	AbstractPainterGRAY2()

Protected Functions inherited from AbstractPainterGRAY2

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint8_t * p, uint16_t offset, uint8_t gray)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterGRAY2

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

getColor

colortype getColor ( ) const

Gets the current color.

Returns:

The color.

PainterGRAY2

```
PainterGRAY2 ( colortype color =0,
               uint8_t   alpha =255
```

```
)
```

Initializes a new instance of the **PainterGRAY2** class.

Parameters:

color	(Optional) the color, default is black.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterGRAY2::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & gray ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

gray The gray color (0-3).
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterGRAY2::renderNext**

Protected Attributes Documentation

painterGray

uint8_t painterGray

The gray color.

Version: 4.16

PainterGRAY2Bitmap

PainterGRAY2Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterGRAY2, AbstractPainter

Public Functions

	PainterGRAY2Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterGRAY2Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & gray, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapAlphaPointer**
	Pointer to the bitmap alpha data for GRAY2.
const uint8_t *	**bitmapGRAY2Pointer**
	Pointer to the bitmap (GRAY2)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterGRAY2

	AbstractPainterGRAY2()

Protected Functions inherited from AbstractPainterGRAY2

virtual void	**renderPixel**(uint8_t * p, uint16_t offset, uint8_t gray)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterGRAY2

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)

	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterGRAY2Bitmap

```
PainterGRAY2Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                     uint8_t        alpha =255
                     )
```

Initializes a new instance of the **PainterGRAY2Bitmap** class.

Parameters:

bmp	(Optional) The bitmap, default is **BITMAP_INVALID**.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterGRAY2::render**

setBitmap

```
void setBitmap ( const Bitmap & bmp )
```

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterGRAY2::renderInit**

renderNext

virtual bool renderNext (uint8_t & gray ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

gray The gray color (0-3).
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterGRAY2::renderNext**

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapAlphaPointer

const uint8_t * bitmapAlphaPointer

Pointer to the bitmap alpha data for GRAY2.

bitmapGRAY2Pointer

const uint8_t * bitmapGRAY2Pointer

Pointer to the bitmap (GRAY2)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterGRAY4

The PainterGRAY4 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterGRAY4, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterGRAY4(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterGRAY4** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & gray, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterGray**
	The gray color.

Additional inherited members

Public Functions inherited from AbstractPainterGRAY4

	AbstractPainterGRAY4()

Protected Functions inherited from AbstractPainterGRAY4

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint8_t * p, uint16_t offset, uint8_t gray)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterGRAY4

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

getColor

colortype getColor ( ) const

Gets the current color.

Returns:

The color.

PainterGRAY4

```
PainterGRAY4 ( colortype color =0,
               uint8_t   alpha =255
```

```
)
```

Initializes a new instance of the **PainterGRAY4** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterGRAY4::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & gray ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

gray The gray color (0-15).
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterGRAY4::renderNext**

Protected Attributes Documentation

painterGray

uint8_t painterGray

The gray color.

Version: 4.16

PainterGRAY4Bitmap

PainterGRAY4Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRendere (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterGRAY4, AbstractPainter

Public Functions

	PainterGRAY4Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterGRAY4Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & gray, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapAlphaPointer**
	Pointer to the bitmap alpha data for GRAY4.
const uint8_t *	**bitmapGRAY4Pointer**
	Pointer to the bitmap (GRAY4)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterGRAY4

	AbstractPainterGRAY4()

Protected Functions inherited from AbstractPainterGRAY4

virtual void	**renderPixel**(uint8_t * p, uint16_t offset, uint8_t gray)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterGRAY4

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)

	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterGRAY4Bitmap

```
PainterGRAY4Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                     uint8_t        alpha =255
                     )
```

Initializes a new instance of the **PainterGRAY4Bitmap** class.

Parameters:

bmp	(Optional) The bitmap, default is **BITMAP_INVALID**.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterGRAY4::render**

setBitmap

```
void setBitmap ( const Bitmap & bmp )
```

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterGRAY4::renderInit**

renderNext

virtual bool renderNext (uint8_t & gray ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

gray The gray color (0-15).
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterGRAY4::renderNext**

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapAlphaPointer

const uint8_t * bitmapAlphaPointer

Pointer to the bitmap alpha data for GRAY4.

bitmapGRAY4Pointer

const uint8_t * bitmapGRAY4Pointer

Pointer to the bitmap (GRAY4)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterRGB565

The PainterRGB565 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterRGB565, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterRGB565(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterRGB565** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint16_t	**painterBlue**
	The blue part of the color.
uint16_t	**painterColor**
	The color.

uint16_t	**painterGreen**
	The green part of the color.

uint16_t	**painterRed**
	The red part of the color.

Additional inherited members

Public Functions inherited from AbstractPainterRGB565

	AbstractPainterRGB565()

FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t newpix, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t R, uint16_t G, uint16_t B, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterRGB565

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Public Attributes inherited from AbstractPainterRGB565

const uint16_t	**BMASK**
	Mask for blue (0000000000011111)

const uint16_t	**GMASK**
	Mask for green (0000011111100000)

const uint16_t	**RMASK**
	Mask for red (1111100000000000)

Protected Attributes inherited from AbstractPainterRGB565

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

getColor

```
colortype getColor ( ) const
```

Gets the current color.

Returns:

The color.

PainterRGB565

```
PainterRGB565 ( colortype color =0,
                uint8_t   alpha =255
                )
```

Initializes a new instance of the **PainterRGB565** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
```

```
                int             y ,
                unsigned        count ,
                const uint8_t * covers
                )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterRGB565::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color	The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
```

```
                                uint8_t & blue ,
                                uint8_t & alpha
                                )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterRGB565::renderNext**

Protected Attributes Documentation

painterBlue

uint16_t painterBlue

The blue part of the color.

painterColor

uint16_t painterColor

The color.

painterGreen

uint16_t painterGreen

The green part of the color.

painterRed

uint16_t painterRed

The red part of the color.

Version: 4.16

PainterRGB565Bitmap

PainterRGB565Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterRGB565, AbstractPainter

Public Functions

	PainterRGB565Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterRGB565Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapAlphaPointer**
	Pointer to the bitmap alpha data for RGB565.
const uint32_t *	**bitmapARGB8888Pointer**
	Pointer to the bitmap (ARGB8888)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.
const uint16_t *	**bitmapRGB565Pointer**
	Pointer to the bitmap (RGB565)

Additional inherited members

Public Functions inherited from AbstractPainterRGB565

	AbstractPainterRGB565()
FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t newpix, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t R, uint16_t G, uint16_t B, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterRGB565

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Public Attributes inherited from AbstractPainterRGB565

const uint16_t	**BMASK**
	Mask for blue (0000000000011111)

const uint16_t	**GMASK**
	Mask for green (0000011111100000)
const uint16_t	**RMASK**
	Mask for red (1111100000000000)

Protected Attributes inherited from AbstractPainterRGB565

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)

	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterRGB565Bitmap

```
PainterRGB565Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                      uint8_t        alpha =255
                      )
```

Initializes a new instance of the **PainterRGB565Bitmap** class.

Parameters:

bmp (Optional) The bitmap, default is **BITMAP_INVALID**.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
```

)

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterRGB565::render**

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp	The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: touchgfx::AbstractPainterRGB565::renderInit

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterRGB565::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapAlphaPointer

const uint8_t * bitmapAlphaPointer

Pointer to the bitmap alpha data for RGB565.

bitmapARGB8888Pointer

const uint32_t * bitmapARGB8888Pointer

Pointer to the bitmap (ARGB8888)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

bitmapRGB565Pointer

const uint16_t * bitmapRGB565Pointer

Pointer to the bitmap (RGB565)

Version: 4.16

PainterRGB565L8Bitmap

PainterRGB565L8Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterRGB565, AbstractPainter

Public Functions

	PainterRGB565L8Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterRGB565L8Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapExtraPointer**
	Pointer to the bitmap alpha data for RGB565 / CLUT for L8.
const uint8_t *	**bitmapPointer**
	Pointer to the bitmap (L8)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterRGB565

	AbstractPainterRGB565()
FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t newpix, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint16_t	**mixColors**(uint16_t R, uint16_t G, uint16_t B, uint16_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterRGB565

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Public Attributes inherited from AbstractPainterRGB565

const uint16_t	**BMASK**
	Mask for blue (0000000000011111)
const uint16_t	**GMASK**
	Mask for green (0000011111100000)

const uint16_t	**RMASK**
	Mask for red (1111100000000000)

Protected Attributes inherited from AbstractPainterRGB565

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t **areaOffsetX**

The offset x coordinate of the area being drawn.

int16_t **areaOffsetY**

The offset y coordinate of the area being drawn.

uint8_t **painterAlpha**

The alpha value for the painter.

uint8_t **widgetAlpha**

The alpha of the widget using the painter.

Public Functions Documentation

PainterRGB565L8Bitmap

```
PainterRGB565L8Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
                        )
```

Initializes a new instance of the **PainterRGB565L8Bitmap** class.

Parameters:

bmp (Optional) The bitmap, default is **BITMAP_INVALID**.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of **render()** in the **AbstractPainter** classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of **AbstractPainter** for better performance.

Reimplements: **touchgfx::AbstractPainterRGB565::render**

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp	The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterRGB565::renderInit**

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.
green	The green.
blue	The blue.
alpha	The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterRGB565::renderNext**

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapExtraPointer

const uint8_t * bitmapExtraPointer

Pointer to the bitmap alpha data for RGB565 / CLUT for L8.

bitmapPointer

const uint8_t * bitmapPointer

Pointer to the bitmap (L8)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterRGB888

The PainterRGB888 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterRGB888, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterRGB888(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterRGB888** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterBlue**
	The blue part of the color.
uint8_t	**painterGreen**
	The green part of the color.

uint8_t	**painterRed**
	The red part of the color.

Additional inherited members

Public Functions inherited from AbstractPainterRGB888

	AbstractPainterRGB888()

Protected Functions inherited from AbstractPainterRGB888

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterRGB888

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.

virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.

FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.

int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.

uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

getColor

colortype getColor ( ) const

Gets the current color.

Returns:

The color.

PainterRGB888

```
PainterRGB888 ( colortype color =0,
                uint8_t   alpha =255
                )
```

Initializes a new instance of the **PainterRGB888** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainterRGB888::render

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterRGB888::renderNext

Protected Attributes Documentation

painterBlue

uint8_t painterBlue

The blue part of the color.

painterGreen

uint8_t painterGreen

The green part of the color.

painterRed

uint8_t painterRed

The red part of the color.

Version: 4.16

PainterRGB888Bitmap

PainterRGB888Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterRGB888, AbstractPainter

Public Functions

	PainterRGB888Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterRGB888Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint32_t *	**bitmapARGB8888Pointer**
	Pointer to the bitmap (ARGB8888)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.
const uint8_t *	**bitmapRGB888Pointer**
	Pointer to the bitmap (RGB888)

Additional inherited members

Public Functions inherited from AbstractPainterRGB888

AbstractPainterRGB888()

Protected Functions inherited from AbstractPainterRGB888

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterRGB888

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)

	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterRGB888Bitmap

```
PainterRGB888Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                      uint8_t        alpha =255
                      )
```

Initializes a new instance of the **PainterRGB888Bitmap** class.

Parameters:

bmp	(Optional) The bitmap, default is **BITMAP_INVALID**.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterRGB888::render**

setBitmap

```
void setBitmap ( const Bitmap & bmp )
```

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

```
virtual bool renderInit ( )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterRGB888::renderInit**

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterRGB888::renderNext**

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapARGB8888Pointer

const uint32_t * bitmapARGB8888Pointer

Pointer to the bitmap (ARGB8888)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

bitmapRGB888Pointer

const uint8_t * bitmapRGB888Pointer

Pointer to the bitmap (RGB888)

Version: 4.16

PainterRGB888L8Bitmap

PainterRGB888L8Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterRGB888, AbstractPainter

Public Functions

	PainterRGB888L8Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterRGB888L8Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

const uint8_t *	**bitmapExtraPointer**
	Pointer to the CLUT (L8)
const uint8_t *	**bitmapPointer**
	Pointer to the bitmap (L8)
Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

Additional inherited members

Public Functions inherited from AbstractPainterRGB888

	AbstractPainterRGB888()

Protected Functions inherited from AbstractPainterRGB888

virtual void	**renderPixel**(uint16_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterRGB888

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.

virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterRGB888L8Bitmap

```
PainterRGB888L8Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
```

```
                                )
```

Initializes a new instance of the **PainterRGB888L8Bitmap** class.

Parameters:

bmp	(Optional) The bitmap, default is **BITMAP_INVALID**.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterRGB888::render**

setBitmap

```
void setBitmap ( const Bitmap & bmp )
```

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

```
virtual bool renderInit ( )
```

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: **touchgfx::AbstractPainterRGB888::renderInit**

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterRGB888::renderNext**

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapExtraPointer

const uint8_t * bitmapExtraPointer

Pointer to the CLUT (L8)

bitmapPointer

const uint8_t * bitmapPointer

Pointer to the bitmap (L8)

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

Version: 4.16

PainterRGBA2222

The PainterRGBA2222 class allows a shape to be filled with a given color and alpha value. This allows transparent, anti-aliased elements to be drawn.

See: AbstractPainter

Inherits from: AbstractPainterRGBA2222, AbstractPainter

Public Functions

colortype	**getColor**() const
	Gets the current color.
	PainterRGBA2222(**colortype** color =0, uint8_t alpha =255)
	Initializes a new instance of the **PainterRGBA2222** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setColor**(**colortype** color)
	Sets color and alpha to use when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

uint8_t	**painterBlue**
	The blue part of the color, scaled up to [0..255].
uint8_t	**painterColor**
	The color.

uint8_t	**painterGreen**
	The green part of the color, scaled up to [0..255].
uint8_t	**painterRed**
	The red part of the color, scaled up to [0..255].

Additional inherited members

Public Functions inherited from AbstractPainterRGBA2222

	AbstractPainterRGBA2222()
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.
FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterRGBA2222

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterRGBA2222

int	**currentX**
	Current x coordinate relative to the widget.
int	**currentY**

	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.

uint8_t	**widgetAlpha**

The alpha of the widget using the painter.

Public Functions Documentation

getColor

```
colortype getColor ( ) const
```

Gets the current color.

Returns:

The color.

PainterRGBA2222

```
PainterRGBA2222 ( colortype color =0,
                  uint8_t   alpha =255
                  )
```

Initializes a new instance of the **PainterRGBA2222** class.

Parameters:

color (Optional) the color, default is black.
alpha (Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: **touchgfx::AbstractPainterRGBA2222::render**

setColor

```
void setColor ( colortype color )
```

Sets color and alpha to use when drawing the **CanvasWidget**.

Parameters:

color	The color.

Protected Functions Documentation

renderNext

```
virtual bool renderNext ( uint8_t & red ,
                          uint8_t & green ,
                          uint8_t & blue ,
                          uint8_t & alpha
                          )
```

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red	The red.

green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: **touchgfx::AbstractPainterRGBA2222::renderNext**

Protected Attributes Documentation

painterBlue

uint8_t painterBlue

The blue part of the color, scaled up to [0..255].

painterColor

uint8_t painterColor

The color.

painterGreen

uint8_t painterGreen

The green part of the color, scaled up to [0..255].

painterRed

uint8_t painterRed

The red part of the color, scaled up to [0..255].

Version: 4.16

PainterRGBA2222Bitmap

PainterRGBA2222Bitmap will take the color for a given point in the shape from a bitmap. Please be aware, the the bitmap is used by the CanvasWidgetRenderer (not Shape), so any rotation you might specify for a CanvasWidget (e.g. Shape) is not applied to the bitmap as CWR is not aware of this rotation.

See: AbstractPainter

Inherits from: AbstractPainterRGBA2222, AbstractPainter

Public Functions

	PainterRGBA2222Bitmap(const **Bitmap** & bmp =**Bitmap**(**BITMAP_INVALID**), uint8_t alpha =255)
	Initializes a new instance of the **PainterRGBA2222Bitmap** class.
virtual void	**render**(uint8_t *ptr, int x, int xAdjust, int y, unsigned count, const uint8_t* covers)
	Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.
void	**setBitmap**(const **Bitmap** & bmp)
	Sets a bitmap to be used when drawing the **CanvasWidget**.

Protected Functions

virtual bool	**renderInit**()
	Initialize rendering of a single scan line of pixels for the render.
virtual bool	**renderNext**(uint8_t & red, uint8_t & green, uint8_t & blue, uint8_t & alpha)
	Get the color of the next pixel in the scan line to blend into the framebuffer.

Protected Attributes

Bitmap	**bitmap**
	The bitmap to be used when painting.

Rect	**bitmapRectToFrameBuffer**
	Bitmap rectangle translated to framebuffer coordinates.

const uint8_t *	**bitmapRGBA2222Pointer**
	Pointer to the bitmap (RGBA2222)

Additional inherited members

Public Functions inherited from AbstractPainterRGBA2222

	AbstractPainterRGBA2222()

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t newpix, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

FORCE_INLINE_FUNCTION uint8_t	**mixColors**(uint8_t R, uint8_t G, uint8_t B, uint8_t bufpix, uint8_t alpha)
	Mix colors from a new pixel and a buffer pixel with the given alpha applied to the new pixel, and the inverse alpha applied to the buffer pixel.

Protected Functions inherited from AbstractPainterRGBA2222

virtual void	**renderPixel**(uint8_t * p, uint8_t red, uint8_t green, uint8_t blue)
	Renders (writes) the specified color into the framebuffer.

Protected Attributes inherited from AbstractPainterRGBA2222

int	**currentX**
	Current x coordinate relative to the widget.

int	**currentY**
	Current y coordinate relative to the widget.

Public Functions inherited from AbstractPainter

	AbstractPainter()
	Initializes a new instance of the **AbstractPainter** class.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setOffset**(uint16_t offsetX, uint16_t offsetY)
	Sets the offset of the area being drawn.
virtual	**~AbstractPainter**()
	Finalizes an instance of the **AbstractPainter** class.

Protected Functions inherited from AbstractPainter

void	**setWidgetAlpha**(const uint8_t alpha)
	Sets the widget alpha to allow an entire canvas widget to easily be faded without changing the painter of the widget.
FORCE_INLINE_FUNCTION bool	**compatibleFramebuffer**(**Bitmap::BitmapFormat** format)
	Helper function to check if the provided bitmap format matches the current framebuffer format.

Protected Attributes inherited from AbstractPainter

int16_t	**areaOffsetX**
	The offset x coordinate of the area being drawn.
int16_t	**areaOffsetY**
	The offset y coordinate of the area being drawn.
uint8_t	**painterAlpha**
	The alpha value for the painter.
uint8_t	**widgetAlpha**
	The alpha of the widget using the painter.

Public Functions Documentation

PainterRGBA2222Bitmap

```
PainterRGBA2222Bitmap ( const Bitmap & bmp =Bitmap(BITMAP_INVALID),
                        uint8_t        alpha =255
                        )
```

Initializes a new instance of the **PainterRGBA2222Bitmap** class.

Parameters:

bmp	(Optional) The bitmap, default is **BITMAP_INVALID**.
alpha	(Optional) the alpha, default is 255 i.e. solid.

render

```
virtual void render ( uint8_t *       ptr ,
                      int             x ,
                      int             xAdjust ,
                      int             y ,
                      unsigned        count ,
                      const uint8_t * covers
                      )
```

Paint a designated part of the RenderingBuffer with respect to the amount of coverage of each pixel given by the parameter covers.

The cover is the alpha for each pixel, which is what makes it possible to have smooth anti-aliased edges on the shapes drawn with **CanvasWidgetRenderer**.

Parameters:

ptr	Pointer to the row in the RenderingBuffer.
x	The x coordinate.
xAdjust	The minor adjustment of x (used when a pixel is smaller than a byte to specify that the *ptr* should have been advanced "xAdjust" pixels futher into the byte).
y	The y coordinate.
count	Number of pixels to fill.
covers	The coverage in of each pixel.

NOTE

The implementation of render() in the AbstractPainter classes is a generic (i.e. slow) implementation that should be completely implemented in subclasses of AbstractPainter for better performance.

Reimplements: touchgfx::AbstractPainterRGBA2222::render

setBitmap

void setBitmap (const Bitmap & bmp)

Sets a bitmap to be used when drawing the **CanvasWidget**.

Parameters:

bmp The bitmap.

Protected Functions Documentation

renderInit

virtual bool renderInit ( )

Initialize rendering of a single scan line of pixels for the render.

If renderInit returns false, the scanline will not be rendered.

Returns:

true if it succeeds, false if it fails.

Reimplements: touchgfx::AbstractPainterRGBA2222::renderInit

renderNext

virtual bool renderNext (uint8_t & red ,
uint8_t & green ,
uint8_t & blue ,
uint8_t & alpha
)

Get the color of the next pixel in the scan line to blend into the framebuffer.

Parameters:

red The red.
green The green.
blue The blue.
alpha The alpha.

Returns:

true if the pixel should be painted, false otherwise.

Reimplements: touchgfx::AbstractPainterRGBA2222::renderNext

Protected Attributes Documentation

bitmap

Bitmap bitmap

The bitmap to be used when painting.

bitmapRectToFrameBuffer

Rect bitmapRectToFrameBuffer

Bitmap rectangle translated to framebuffer coordinates.

bitmapRGBA2222Pointer

const uint8_t * bitmapRGBA2222Pointer

Pointer to the bitmap (RGBA2222)

Version: 4.16

Pair

A simple struct for holding pairs of data.

Template Parameters:

- **T1** The type of the first element.
- **T2** The type of the second element.

Public Functions

	Pair()
	Constructor initializing the elements it holds, using their default constructors.
template \<class U ,class V \>	**Pair**(const **Pair**< U, V > & p)
	Copy constructor.
	Pair(const T1 & x, const T2 & y)
	Constructor initializing the elements it holds, using their copy constructor.

Public Attributes

T1	**first**
	The first element.
T2	**second**
	The second element.

Public Functions Documentation

Pair

```
Pair ( )
```

Constructor initializing the elements it holds, using their default constructors.

Pair

```
Pair ( const Pair< U, V > & p )
```

Copy constructor.

Template Parameters:

U Generic type parameter.
V Generic type parameter.

Parameters:

p The pair to copy from.

Pair

```
Pair ( const T1 & x ,
       const T2 & y
       )
```

Constructor initializing the elements it holds, using their copy constructor.

Parameters:

x Reference to the first element.
y Reference to the second element.

Public Attributes Documentation

first

T1 first

The first element.

second

T2 second

The second element.

Version: 4.16

PartialFrameBufferManager

This class specifies strategies for transmitting block to the display using Partial Frame Buffer.

Public Functions

void	**transmitRemainingBlocks**()
	Transmit all remaining drawn Framebuffer blocks.
void	**tryTransmitBlock**()
	Tries to transmit a drawn block.
void	**tryTransmitBlockFromIRQ**()
	Tries to transmit a drawn block in interrupt context.

Public Functions Documentation

transmitRemainingBlocks

static void transmitRemainingBlocks ( )

Transmit all remaining drawn Framebuffer blocks.

NOTE

This function does not return before all blocks have been transmitted.

tryTransmitBlock

static void tryTransmitBlock ( )

Tries to transmit a drawn block.

NOTE

Will return immediately if already transmitting.

tryTransmitBlockFromIRQ

static void tryTransmitBlockFromIRQ ( )

Tries to transmit a drawn block in interrupt context.

NOTE

Will transmit next block immediately if drawn.

Version: 4.16

Partition

This type provides a concrete Partition of memory-slots capable of holding any of the specified list of types. The Partition is not aware of the types stored in the Partition memory, hence it provides no mechanism for deleting C++ objects when the Partition is clear()'ed.

This class implements AbstractPartition.

Template Parameters:

- **ListOfTypes** Type of the list of types.
- **NUMBER_OF_ELEMENTS** Type of the number of elements.

See: AbstractPartition

Inherits from: AbstractPartition

Public Types

enum	**@0** { INTS_PR_ELEMENT = (sizeof(typename meta::select_type_maxsize<SupportedTypesList>::type) + sizeof(int) - 1) / sizeof(int), SIZE_OF_ELEMENT = INTS_PR_ELEMENT * sizeof(int) }
	Compile-time generated constants specifying the "element" or "slot" size used by this partition.
typedef ListOfTypes	**SupportedTypesList**
	Provides a generic public type containing the list of supported types.

Public Functions

virtual uint16_t	**capacity**() const
	Gets the capacity, i.e.
virtual uint32_t	**element_size**()
	Access to concrete element-size.

Protected Functions

virtual void *	**element**(uint16_t index)
	Access to stored element.
virtual const void *	**element**(uint16_t index) const
	Access to stored element, const version.

Additional inherited members

Public Functions inherited from AbstractPartition

void *	**allocate**()
	Gets the address of the next available storage slot.
virtual void *	**allocate**(uint16_t size)
	Gets the address of the next available storage slot.
void *	**allocateAt**(uint16_t index)
	Gets the address of the specified storage slot.
virtual void *	**allocateAt**(uint16_t index, uint16_t size)
	Gets the address of the specified index.
T &	**at**(const uint16_t index)
	Gets the object at the specified index.
const T &	**at**(const uint16_t index) const
	const version of **at()**.
virtual void	**clear**()
	Prepares the Partition for new allocations.
void	**dec**()
	Decreases number of allocations.
Pair< T *, uint16_t >	**find**(const void * pT)
	Determines if the specified object could have been previously allocated in the partition.

virtual uint16_t	**getAllocationCount**() const
	Gets allocation count.
virtual uint16_t	**indexOf**(const void * address)
	Determines index of previously allocated location.
virtual	**~AbstractPartition**()
	Finalizes an instance of the **AbstractPartition** class.

Protected Functions inherited from AbstractPartition

AbstractPartition()
Initializes a new instance of the **AbstractPartition** class.

Public Types Documentation

@0

enum @0

Compile-time generated constants specifying the "element" or "slot" size used by this partition.

INTS_PR_ELEMENT
SIZE_OF_ELEMENT

SupportedTypesList

typedef ListOfTypes SupportedTypesList

Provides a generic public type containing the list of supported types.

Public Functions Documentation

capacity

virtual uint16_t capacity ( ) const

Gets the capacity, i.e.

the maximum allocation count.

Returns:

The maximum allocation count.

Reimplements: **touchgfx::AbstractPartition::capacity**

element_size

virtual uint32_t element_size ( )

Access to concrete element-size.

Used internally.

Returns:

An uint32_t.

Reimplements: **touchgfx::AbstractPartition::element_size**

Protected Functions Documentation

element

virtual void * element (uint16_t index)

Access to stored element.

Used internally.

Parameters:

index Zero-based index of the.

Returns:

null if it fails, else a void*.

Reimplements: **touchgfx::AbstractPartition::element**

element

virtual const void * element (uint16_t index)

Access to stored element, const version.

Parameters:

index Zero-based index of the.

Returns:

null if it fails, else a void*.

Reimplements: **touchgfx::AbstractPartition::element**

Version: 4.16

PixelDataWidget

A widget for displaying a buffer of pixel data. This can also be though of as a dynamic bitmap where the dimensions of the bitmap is the same as the dimensions of the widget and the actual bitmap data can be set and updated dynamically. The size of the buffer must match the number of bytes required for the widget calculated as WIDTH x HEIGHT x BYTES_PER_PIXEL. If the LCD is 16 bit per pixel the buffer must hold 2 bytes for each pixel. If the LCD is 24 bit the buffer must hold 3 bytes for each pixel.

Inherits from: Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
	PixelDataWidget()
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setBitmapFormat**(**Bitmap::BitmapFormat** format)
	Set the format of the pixel data.
void	**setPixelData**(uint8_t *const data)
	Set the pixel data to display.

Protected Attributes

uint8_t	**alpha**
	The Alpha for this widget.

uint8_t *	**buffer**
	The buffer where the pixels are copied from.

Bitmap::BitmapFormat	**format**
	The pixel format for the data.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.

Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.

virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.

int16_t	**getHeight**() const
	Gets the height of this **Drawable**.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.

Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.

bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the **Drawable**, meaning that a completely solid widget should return the full size **Rect**(0, 0, **getWidth()**, **getHeight()**). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

PixelDataWidget

PixelDataWidget ( )

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call **invalidate()** in order to update the display.

See also:

getAlpha

setBitmapFormat

void setBitmapFormat (Bitmap::BitmapFormat format)

Set the format of the pixel data.

The supported formats depend on the display type. For example grayscale displays do not support color images.

Parameters:

format Describes the format to use when reading the pixel data.

setPixelData

void setPixelData (uint8_t *const data)

Set the pixel data to display.

The given pointer must contain WIDTH x HEIGHT x BYTES_PER_PIXEL bytes of addressable image data.

Parameters:

data **Image** data.

See also:

setBitmapFormat

Protected Attributes Documentation

alpha

uint8_t alpha

The Alpha for this widget.

buffer

uint8_t * buffer

The buffer where the pixels are copied from.

format

Bitmap::BitmapFormat format

The pixel format for the data.

Version: 4.16

Point

A simple struct containing coordinates.

Public Functions

unsigned	**dist_sqr**(struct **Point** & o)
	The squared distance from this **Point** to another **Point**.

Public Attributes

int32_t	**x**
	The x coordinate.
int32_t	**y**
	The y coordinate.

Public Functions Documentation

dist_sqr

unsigned dist_sqr (struct Point & o)

The squared distance from this **Point** to another **Point**.

Parameters:

- **o** The point to get the squared distance to.

Returns:

The squared distance.

Public Attributes Documentation

x

int32_t x

The x coordinate.

y

int32_t y

The y coordinate.

Version: 4.16

Point3D

A 3D point.

Public Attributes

float	**U**
	The U coordinate.
float	**V**
	The V coordinate.
fixed28_4	**X**
	The X coordinate.
fixed28_4	**Y**
	The Y coordinate.
float	**Z**
	The Z coordinate.

Public Attributes Documentation

U

float U

The U coordinate.

V

float V

The V coordinate.

X

fixed28_4 X

The X coordinate.

Y

fixed28_4 Y

The Y coordinate.

Z

float Z

The Z coordinate.

Version: 4.16

Point4

This class represents a homogeneous 3D point.

See: Quadruple

Inherits from: Quadruple

Public Functions

FORCE_INLINE_FUNCTION	**Point4**()
	Initializes a new instance of the **Point4** class.
FORCE_INLINE_FUNCTION	**Point4**(float x, float y, float z)
	Initializes a new instance of the **Point4** class.

Additional inherited members

Public Functions inherited from Quadruple

FORCE_INLINE_FUNCTION float	**getElement**(int row) const
	Gets an element.
FORCE_INLINE_FUNCTION float	**getW**() const
	Get w coordinate.
FORCE_INLINE_FUNCTION float	**getX**() const
	Get x coordinate.
FORCE_INLINE_FUNCTION float	**getY**() const
	Get y coordinate.
FORCE_INLINE_FUNCTION float	**getZ**() const
	Get z coordinate.
FORCE_INLINE_FUNCTION void	**setElement**(int row, float value)
	Sets an element.

FORCE_INLINE_FUNCTION void	**setW**(float value)
	Sets a w coordinate.
FORCE_INLINE_FUNCTION void	**setX**(float value)
	Sets an x coordinate.
FORCE_INLINE_FUNCTION void	**setY**(float value)
	Sets a y coordinate.
FORCE_INLINE_FUNCTION void	**setZ**(float value)
	Sets a z coordinate.

Protected Functions inherited from Quadruple

FORCE_INLINE_FUNCTION	**Quadruple**()
	Initializes a new instance of the **Quadruple** class.
FORCE_INLINE_FUNCTION	**Quadruple**(float x, float y, float z, float w)
	Initializes a new instance of the **Quadruple** class.

Protected Attributes inherited from Quadruple

float	**elements**
	The elements[4].

Public Functions Documentation

Point4

FORCE_INLINE_FUNCTION Point4 ( )

Initializes a new instance of the **Point4** class.

Point4

FORCE_INLINE_FUNCTION Point4 (float x ,
float y ,
float z

)

Initializes a new instance of the **Point4** class.

Parameters:

- **x** The x coordinate.
- **y** The y coordinate.
- **z** The z coordinate.

Version: 4.16

Presenter

The Presenter base class that all application-specific presenters should derive from. Only contains activate and deactivate virtual functions which are called automatically during screen transition.

Public Functions

virtual void	**activate**()
	Place initialization code for the **Presenter** here.
virtual void	**deactivate**()
	Place cleanup code for the **Presenter** here.
virtual	**~Presenter**()
	Finalizes an instance of the **Presenter** class.

Protected Functions

Presenter()
Initializes a new instance of the **Presenter** class.

Public Functions Documentation

activate

virtual void activate ( )

Place initialization code for the **Presenter** here.

The activate function is called automatically when a screen transition causes this **Presenter** to become active. Place initialization code for the **Presenter** here.

deactivate

virtual void deactivate ( )

Place cleanup code for the **Presenter** here.

The deactivate function is called automatically when a screen transition causes this **Presenter** to become inactive. Place cleanup code for the **Presenter** here.

~Presenter

virtual ~Presenter ( )

Finalizes an instance of the **Presenter** class.

Protected Functions Documentation

Presenter

Presenter ( )

Initializes a new instance of the **Presenter** class.

Version: 4.16

Q10

Defines a "floating point number" with 10 bits reserved for the fractional part of the decimal number. Q10 implements some simple arithmetic operations, most yielding a Q10 number and some yielding a Q5 number or a Q15 number as a result.

Q5*Q5=Q10, Q10/Q5=Q5, ...

See: Q5, Q15

Public Functions

	operator int() const
	Gets the **Q10** as an integer without conversion.
Q15	**operator***(const **Q5** & q5) const
	Multiplication operator.
Q10	**operator+**(const **Q10** & q10) const
	Addition operator.
Q10	**operator-**() const
	Negation operator.
Q5	**operator/**(const **Q5** & q5) const
	Division operator.
	Q10()
	Initializes a new instance of the **Q10** class.
	Q10(int i)
	Constructor from integer.
template \<typename T \> T	**to**() const
	Converts the **Q10** value to an int or a float.

Public Functions Documentation

operator int

operator int ( ) const

Gets the **Q10** as an integer without conversion.

Returns:

The unconverted **Q10** value.

operator*

Q15 operator* (const Q5 & q5)

Multiplication operator.

The result is a **Q15**, not a **Q10**, for increased precision.

Parameters:

q5 The **Q5** to multiply this with.

Returns:

The result of the operation.

operator+

Q10 operator+ (const Q10 & q10)

Addition operator.

Parameters:

q10 The **Q10** to add to this.

Returns:

The result of the operation.

operator-

Q10 operator- ( ) const

Negation operator.

Returns:

The negative value of this.

operator/

Q5 operator/ (const Q5 & q5)

Division operator.

Parameters:

q5 The **Q5** to divide this by.

Returns:

The result of the operation.

Q10

Q10 ( )

Initializes a new instance of the **Q10** class.

Q10

explicit Q10 (int i)

Constructor from integer.

No conversion is done - the integer is assumed to already be in **Q10** format.

Parameters:

i int pre-formattet in **Q10** format.

to

T to ( ) const

Converts the **Q10** value to an int or a float.

Convert the **Q10** value to an integer by removing the 10 bits used for the fraction, or to a floating point value by dividing by 32 * 32, depending on the type specified as T.

Template Parameters:

T Either int or float.

Returns:

Q10 value as a type T.

Version: 4.16

Q15

Defines a "floating point number" with 15 bits reserved for the fractional part of the decimal number. Q15 is only used for sine/cosine and for intermediate calculations when multiplying.

Q5*Q5=Q10, Q10/Q5=Q5, ...

See: Q5, Q10

Public Functions

	operator int() const
	Gets the **Q15** as an integer without conversion.
Q15	**operator+**(const **Q15** & q15) const
	Addition operator.
Q15	**operator-**() const
	Negation operator.
Q10	**operator/**(const **Q5** & q5) const
	Calculate **Q15** / Q5 as a **Q10**.
	Q15(int i)
	Constructor from integer.

Public Functions Documentation

operator int

operator int ( ) const

Gets the **Q15** as an integer without conversion.

Returns:

The unconverted **Q15** value.

operator+

Q15 operator+ (const Q15 & q15)

Addition operator.

Parameters:

q15 The **Q15** to add to this.

Returns:

The result of the operation.

operator-

Q15 operator- ( ) const

Negation operator.

Returns:

The negative value of this.

operator/

Q10 operator/ (const Q5 & q5)

Calculate **Q15** / Q5 as a **Q10**.

Parameters:

q5 The **Q5** to divide this by.

Returns:

The result of the operation.

Q15

explicit Q15 (int i)

Constructor from integer.

No conversion is done - the integer is assumed to already be in **Q15** format.

Parameters:

i int pre-formattet in **Q15** format.

Version: 4.16

Q5

Defines a "floating point number" with 5 bits reserved for the fractional part of the decimal number. Q5 implements some simple arithmetic operations, most yielding a Q5 number and some yielding a Q10 number as a result. Other operations also work with Q15 numbers.

See: Q10, Q15

Public Functions

int	**ceil**() const
	Convert the **Q5** value to an integer by removing the 5 bits used for the fraction.
	operator int() const
	Gets the **Q5** as an integer without conversion.
Q5	**operator***(const int i) const
	Multiplication operator.
Q5	**operator***(const **Q15** & q15) const
	Multiplication operator.
Q10	**operator***(const **Q5** & q5) const
	Multiplication operator.
Q5	**operator+**(const **Q5** & q5) const
	Addition operator.
Q5	**operator-**() const
	Negation operator.
Q5	**operator-**(const **Q5** & q5) const
	Subtraction operator.
Q5	**operator/**(const int i) const
	Division operator.
Q5	**operator/**(const **Q5** q5) const
	Division operator.

	Q5()
	Initializes a new instance of the **Q5** class.
	Q5(const **Q10** q10)
	Constructor from **Q10**.
	Q5(int i)
	Constructor from integer.
int	**round**() const
	Round the **Q5** value to the nearest integer value.
template \<typename T \> T	**to**() const
	Convert the **Q5** value to an integer by removing the 5 bits used for the fraction, or to a floating point value by dividing by 32, depending on the type specified as T.

Public Functions Documentation

ceil

int ceil ( ) const

Convert the **Q5** value to an integer by removing the 5 bits used for the fraction.

The number is rounded up to the nearest integer.

Returns:

The first integer value higher than (or equal to) the **Q5** value.

operator int

operator int ( ) const

Gets the **Q5** as an integer without conversion.

Returns:

The unconverted **Q5** value.

operator*

Q5 operator* (const int i)

Multiplication operator.

Parameters:

i The integer to multiply this with.

Returns:

The result of the operation.

operator*

Q5 operator* (const Q15 & q15)

Multiplication operator.

Often used in relation with sine and cosine calculation which are pre-calculated as **Q15**. As the result is needed as a **Q5**, this operator multiplies with the given **Q15** and converts the result to a **Q5**.

Parameters:

q15 The **Q15** to multiply this with.

Returns:

The result of the operation.

See also:

Q15

operator*

Q10 operator* (const Q5 & q5)

Multiplication operator.

The result is a **Q10**, not a **Q5**, for increased precision.

Parameters:

q5 The **Q5** to multiply this with.

Returns:

The result of the operation.

See also:

Q10

operator+

Q5 operator+ (const Q5 & q5)

Addition operator.

Parameters:

q5 The **Q5** to add to this.

Returns:

The result of the operation.

operator-

Q5 operator- ( ) const

Negation operator.

Returns:

The negative value of this.

operator-

Q5 operator- (const Q5 & q5)

Subtraction operator.

Parameters:

q5 The **Q5** to subtract from this.

Returns:

The result of the operation.

operator/

Q5 operator/ (const int i)

Division operator.

Parameters:

i The integer to divide this by.

Returns:

The result of the operation.

operator/

Q5 operator/ (const Q5 q5)

Division operator.

Internally this **Q5** is converted to **Q10** before the division to increased precision.

Parameters:

q5 The **Q5** to divide this by.

Returns:

The result of the operation.

See also:

Q10

Q5

Q5 ( )

Initializes a new instance of the **Q5** class.

Q5

Q5 (const Q10 q10)

Constructor from **Q10**.

The **Q10** is shifted down to convert it to **Q5**, thus the value is rounded down in the conversion.

Parameters:

q10 The **Q10** value to convert to a **Q5** value.

See also:

Q10

Q5

explicit Q5 (int i)

Constructor from integer.

No conversion is done - the integer is assumed to already be in **Q5** format.

Parameters:

i Integer pre-formattet in **Q5** format.

round

int round ( ) const

Round the **Q5** value to the nearest integer value.

Returns:

The integer closest to the **Q5** value.

to

T to ( ) const

Convert the **Q5** value to an integer by removing the 5 bits used for the fraction, or to a floating point value by dividing by 32, depending on the type specified as T.

Template Parameters:

T Either int or float.

Returns:

Q5 value as a type T.

NOTE

Using "to<int16_t>()" result in loss of precision. Use "(int16_t)to<int>()" instead.

Version: 4.16

Quadruple

Base class for homogeneous vectors and points.

Inherited by: Point4, Vector4

Public Functions

FORCE_INLINE_FUNCTION float	**getElement**(int row) const
	Gets an element.
FORCE_INLINE_FUNCTION float	**getW**() const
	Get w coordinate.
FORCE_INLINE_FUNCTION float	**getX**() const
	Get x coordinate.
FORCE_INLINE_FUNCTION float	**getY**() const
	Get y coordinate.
FORCE_INLINE_FUNCTION float	**getZ**() const
	Get z coordinate.
FORCE_INLINE_FUNCTION void	**setElement**(int row, float value)
	Sets an element.
FORCE_INLINE_FUNCTION void	**setW**(float value)
	Sets a w coordinate.
FORCE_INLINE_FUNCTION void	**setX**(float value)
	Sets an x coordinate.
FORCE_INLINE_FUNCTION void	**setY**(float value)
	Sets a y coordinate.
FORCE_INLINE_FUNCTION void	**setZ**(float value)
	Sets a z coordinate.

Protected Functions

FORCE_INLINE_FUNCTION	**Quadruple**()
	Initializes a new instance of the **Quadruple** class.
FORCE_INLINE_FUNCTION	**Quadruple**(float x, float y, float z, float w)
	Initializes a new instance of the **Quadruple** class.

Protected Attributes

float	**elements**
	The elements[4].

Public Functions Documentation

getElement

FORCE_INLINE_FUNCTION float getElement (int row)

Gets an element.

Parameters:

row The row (0-3).

Returns:

The element.

getW

FORCE_INLINE_FUNCTION float getW ( ) const

Get w coordinate.

Returns:

The w coordinate.

getX

FORCE_INLINE_FUNCTION float getX ( ) const

Get x coordinate.

Returns:

The x coordinate.

getY

FORCE_INLINE_FUNCTION float getY ( ) const

Get y coordinate.

Returns:

The y coordinate.

getZ

FORCE_INLINE_FUNCTION float getZ ( ) const

Get z coordinate.

Returns:

The z coordinate.

setElement

FORCE_INLINE_FUNCTION void setElement (int row ,
float value
)

Sets an element.

Parameters:

row The row (0-3).
value The new value.

setW

FORCE_INLINE_FUNCTION void setW (float value)

Sets a w coordinate.

Parameters:

value The new value.

setX

FORCE_INLINE_FUNCTION void setX (float value)

Sets an x coordinate.

Parameters:

value The new value.

setY

FORCE_INLINE_FUNCTION void setY (float value)

Sets a y coordinate.

Parameters:

value The new value.

setZ

FORCE_INLINE_FUNCTION void setZ (float value)

Sets a z coordinate.

Parameters:

value The new value.

Protected Functions Documentation

Quadruple

FORCE_INLINE_FUNCTION Quadruple ( )

Initializes a new instance of the **Quadruple** class.

Quadruple

```
FORCE_INLINE_FUNCTION Quadruple ( float x ,
                                  float y ,
                                  float z ,
                                  float w
                                  )
```

Initializes a new instance of the **Quadruple** class.

Parameters:

- **x** The x coordinate.
- **y** The y coordinate.
- **z** The z coordinate.
- **w** The w coordinate.

Protected Attributes Documentation

elements

float elements

The elements[4].

Version: 4.16

RadioButton

Radio button with two states. A RadioButton is a button that changes appearance (state) when it has been pushed. Pushing the RadioButton again will return the to original state.

To make managing radio buttons much easier, they can be added to a RadioButtonGroup which then automates deselecting radio buttons when a new radio button is pressed.

See: RadioButtonGroup

Inherits from: AbstractButton, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
void	**executeDeselectedAction**()
	Executes the previously set action.
uint8_t	**getAlpha**() const
	Gets the current alpha value, as previously set using setAlpha.
Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
bool	**getDeselectionEnabled**() const
	Gets the current deselectionEnabled state.
bool	**getSelected**() const
	Gets the current selected state.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
	RadioButton()

void	**setAlpha**(uint8_t alpha)
	Sets the alpha channel for the **RadioButton**, i.e.
virtual void	**setBitmaps**(const **Bitmap** & bmpUnselected, const **Bitmap** & bmpUnselectedPressed, const **Bitmap** & bmpSelected, const **Bitmap** & bmpSelectedPressed)
	Sets the four bitmaps used by this button.
void	**setDeselectedAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action to be performed when the **RadioButton** is deselected.
void	**setDeselectionEnabled**(bool state)
	Sets whether or not it is possible to deselect the **RadioButton** by clicking it.
void	**setSelected**(bool newSelected)
	Sets the radio buttons selected state.

Protected Attributes

uint8_t	**alpha**
	The current alpha value. 255=solid, 0=invisible.
Bitmap	**bitmapSelected**
	The image to display when radio button selected and released.
Bitmap	**bitmapSelectedPressed**
	The image to display when radio button selected and pressed.
Bitmap	**bitmapUnselected**
	The image to display when radio button unselected and released.
Bitmap	**bitmapUnselectedPressed**
	The image to display when radio button unselected and pressed.
GenericCallback< const **AbstractButton** & > *	**deselectedAction**
	The callback to be executed when this **AbstractButton** is deselected.
bool	**deselectionEnabled**

	Is it possible to deselect by pressing a selected **RadioButton**.
bool	**selected**
	The current selected state.

Additional inherited members

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const

	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)

	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

executeDeselectedAction

void executeDeselectedAction ( )

Executes the previously set action.

See also:

setDeselectedAction

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value, as previously set using setAlpha.

The default alpha value (if the alpha value has not been changed using setAlpha) is 255=solid.

Returns:

The current alpha value ranging from 255=solid to 0=invisible.

See also:

setAlpha

getCurrentlyDisplayedBitmap

Bitmap getCurrentlyDisplayedBitmap ( ) const

Gets currently displayed bitmap.

This depends on whether the **RadioButton** is currently selected or not and whether it is being pressed or not, i.e. it depends on the radio button's pressed and selected state.

Returns:

The bitmap currently displayed.

getDeselectionEnabled

bool getDeselectionEnabled ( ) const

Gets the current deselectionEnabled state.

Returns:

The current deselectionEnabled state.

See also:

setDeselectionEnabled

getSelected

bool getSelected ( ) const

Gets the current selected state.

Returns:

The current selected state.

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Updates the current state of the button.

The state can be either pressed or released, and if the new state is different from the current state, the button is also invalidated to force a redraw.

If the button state is changed from **ClickEvent::PRESSED** to **ClickEvent::RELEASED**, the associated action (if any) is also executed.

Parameters:

event Information about the click.

Reimplements: touchgfx::AbstractButton::handleClickEvent

RadioButton

RadioButton ( )

setAlpha

void setAlpha (uint8_t alpha)

Sets the alpha channel for the **RadioButton**, i.e.

all the images used. The default alpha value on a **RadioButton** is 255.

Parameters:

alpha The alpha value ranging from 255=solid to 0=invisible.

See also:

getAlpha

setBitmaps

virtual void setBitmaps (const Bitmap & bmpUnselected ,
const Bitmap & bmpUnselectedPressed ,
const Bitmap & bmpSelected ,
const Bitmap & bmpSelectedPressed
)

Sets the four bitmaps used by this button.

The first two bitmaps must show the unselected **Button** when it is released and pressed. The last two bitmaps must show the selected **Button** when it is released and pressed.

Parameters:

bmpUnselected	**Bitmap** to use when button is unselected and released.
bmpUnselectedPressed	**Bitmap** to use when button is unselected and pressed.
bmpSelected	**Bitmap** to use when button is selected and released.
bmpSelectedPressed	**Bitmap** to use when button is selected and pressed.

NOTE

It is not uncommon to have the same bitmap for released (normal) and pressed state.

setDeselectedAction

void setDeselectedAction (GenericCallback< const AbstractButton & > & callback)

Associates an action to be performed when the **RadioButton** is deselected.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **AbstractButton**.

NOTE

The action performed when the RadioButton is selected, is set using setAction().

setDeselectionEnabled

void setDeselectionEnabled (bool state)

Sets whether or not it is possible to deselect the **RadioButton** by clicking it.

By default it is not possible to deselect a **RadioButton**. The meaning of this is most clear when the **RadioButton** is used in a **RadioButtonGroup** where exactly one **RadioButton** should always be selected. Pressing the currently selected **RadioButton** should not deselect it, but rather select it again. This makes the button "sticky", i.e. a button can only be deselected by selecting another **RadioButton** in the same **RadioButtonGroup**.

Parameters:

state true if it should be possible to deselect by click. Default is false.

See also:

getDeselectionEnabled

setSelected

void setSelected (bool newSelected)

Sets the radio buttons selected state.

Note that the associated action is also performed.

Parameters:

newSelected The new selected state.

NOTE

If the RadioButton is part of a RadioButtonGroup, setting the selected state of individual RadioButtons is not recommended.

See also:

setAction, setDeselectedAction, RadioButtonGroup

Protected Attributes Documentation

alpha

uint8_t alpha

The current alpha value. 255=solid, 0=invisible.

bitmapSelected

Bitmap bitmapSelected

The image to display when radio button selected and released.

bitmapSelectedPressed

Bitmap bitmapSelectedPressed

The image to display when radio button selected and pressed.

bitmapUnselected

Bitmap bitmapUnselected

The image to display when radio button unselected and released.

bitmapUnselectedPressed

Bitmap bitmapUnselectedPressed

The image to display when radio button unselected and pressed.

deselectedAction

GenericCallback< const AbstractButton & > * deselectedAction

The callback to be executed when this **AbstractButton** is deselected.

deselectionEnabled

bool deselectionEnabled

Is it possible to deselect by pressing a selected **RadioButton**.

selected

bool selected

The current selected state.

Version: 4.16

RadioButtonGroup

Class for handling a collection of RadioButton objects. The RadioButtonGroup handles the automatic deselection of other radio buttons when a new RadioButton is selected. A callback is executed when a new selection occurs reporting the newly selected RadioButton.

Template Parameters:

- **CAPACITY** The number of RadioButtons to store in the RadioButtonGroup.

See: RadioButton

Public Functions

virtual void	**add**(**RadioButton** & radioButton)
	Add the **RadioButton** to the **RadioButtonGroup**.
virtual bool	**getDeselectionEnabled**() const
	Gets the current deselectionEnabled state.
virtual **RadioButton** *	**getRadioButton**(uint16_t index) const
	Gets the **RadioButton** at the specified index.
virtual **RadioButton** *	**getSelectedRadioButton**() const
	Gets the currently selected **RadioButton**.
virtual int32_t	**getSelectedRadioButtonIndex**() const
	Gets the index of the currently selected **RadioButton**.
	RadioButtonGroup()
	Initializes a new instance of the **RadioButtonGroup** class.
virtual void	**setDeselectionEnabled**(bool deselectionEnabled)
	Sets whether or not it is possible to deselect RadioButtons by clicking them when they are selected.
void	**setRadioButtonDeselectedHandler**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action to be performed when a radio button belonging to this group transition from selected to unselected.

void	**setRadioButtonSelectedHandler**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action to be performed when a radio button belonging to this group is selected.
virtual void	**setSelected**(**RadioButton** & radioButton)
	Sets the specified **RadioButton** to be selected.
virtual	**~RadioButtonGroup**()
	Finalizes an instance of the **RadioButtonGroup** class.

Protected Functions

virtual void	**radioButtonClickedHandler**(const **AbstractButton** & radioButton)
	Handles the event that a **RadioButton** has been selected.
virtual void	**radioButtonDeselectedHandler**(const **AbstractButton** & radioButton)
	Handles the event that a **RadioButton** has been deselected.

Protected Attributes

Callback< **RadioButtonGroup**, const **AbstractButton** & >	**radioButtonClicked**
	Callback that is attached to the RadioButtons.
GenericCallback< const **AbstractButton** & > *	**radioButtonDeselectedCallback**
	The callback to be executed when a radio button belonging to this group is deselected.
RadioButton *	**radioButtons**
	The list of added RadioButtons.
GenericCallback< const **AbstractButton** & > *	**radioButtonSelectedCallback**
	The callback to be executed when a radio button belonging to this group is selected.
Callback< **RadioButtonGroup**, const **AbstractButton** & >	**radioButtonUnselected**

	Callback that is attached to the RadioButtons.
uint16_t	**size**
	The current number of added RadioButtons.

Public Functions Documentation

add

virtual void add (RadioButton & radioButton)

Add the **RadioButton** to the **RadioButtonGroup**.

Adding more radio buttons than the *CAPACITY* of the **RadioButtonGroup** raises an assert.

Parameters:

radioButton The **RadioButton** to add.

getDeselectionEnabled

virtual bool getDeselectionEnabled ( ) const

Gets the current deselectionEnabled state.

Returns:

The current deselectionEnabled state.

See also:

setDeselectionEnabled

getRadioButton

virtual RadioButton * getRadioButton (uint16_t index)

Gets the **RadioButton** at the specified index.

Parameters:

index the index of the **RadioButton** to return.

Returns:

the **RadioButton** at the specified index. Returns 0 if the index is illegal.

getSelectedRadioButton

virtual RadioButton * getSelectedRadioButton ( ) const

Gets the currently selected **RadioButton**.

Returns:

a pointer to the selected **RadioButton**. Returns 0 if no **RadioButton** is selected.

getSelectedRadioButtonIndex

virtual int32_t getSelectedRadioButtonIndex ( ) const

Gets the index of the currently selected **RadioButton**.

Returns:

the index of the selected **RadioButton**. Returns -1 if no **RadioButton** is selected.

RadioButtonGroup

RadioButtonGroup ( )

Initializes a new instance of the **RadioButtonGroup** class.

setDeselectionEnabled

virtual void setDeselectionEnabled (bool deselectionEnabled)

Sets whether or not it is possible to deselect RadioButtons by clicking them when they are selected.

If deselection is enabled, it will be possible to select a **RadioButton** (and as a result deselect all other radio buttons) and the push the same **RadioButton** again to deselect it. The result is that no **RadioButton** is selected.

Parameters:

deselectionEnabled true if it should be possible to deselect by click.

See also:

getDeselectionEnabled

setRadioButtonDeselectedHandler

void setRadioButtonDeselectedHandler (GenericCallback< const AbstractButton & > & callback)

Associates an action to be performed when a radio button belonging to this group transition from selected to unselected.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **RadioButton** that was selected.

See also:

GenericCallback

setRadioButtonSelectedHandler

void setRadioButtonSelectedHandler (GenericCallback< const AbstractButton & > & callback)

Associates an action to be performed when a radio button belonging to this group is selected.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **RadioButton** that was selected.

See also:

GenericCallback

setSelected

virtual void setSelected (RadioButton & radioButton)

Sets the specified **RadioButton** to be selected.

Sets the specified **RadioButton** to be selected and all other radio buttons to be deselected. Do not call this function before all **RadioButton** objects have been added to the **RadioButtonGroup**. Will call the radioButtonSelected callback.

Parameters:

radioButton the **RadioButton** to be selected.

~RadioButtonGroup

virtual ~RadioButtonGroup ( )

Finalizes an instance of the **RadioButtonGroup** class.

Protected Functions Documentation

radioButtonClickedHandler

virtual void radioButtonClickedHandler (const AbstractButton & radioButton)

Handles the event that a **RadioButton** has been selected.

deselects all other RadioButtons.

Parameters:

radioButton the **RadioButton** that has been selected.

radioButtonDeselectedHandler

virtual void radioButtonDeselectedHandler (const AbstractButton & radioButton)

Handles the event that a **RadioButton** has been deselected.

Parameters:

radioButton the **RadioButton** that has been deselected.

Protected Attributes Documentation

radioButtonClicked

Callback< RadioButtonGroup, const AbstractButton & > radioButtonClicked

Callback that is attached to the RadioButtons.

radioButtonDeselectedCallback

GenericCallback< const AbstractButton & > * radioButtonDeselectedCallback

The callback to be executed when a radio button belonging to this group is deselected.

radioButtons

RadioButton * radioButtons

The list of added RadioButtons.

radioButtonSelectedCallback

GenericCallback< const AbstractButton & > * radioButtonSelectedCallback

The callback to be executed when a radio button belonging to this group is selected.

radioButtonUnselected

Callback< RadioButtonGroup, const AbstractButton & > radioButtonUnselected

Callback that is attached to the RadioButtons.

size

uint16_t size

The current number of added RadioButtons.

Version: 4.16

Rect

Class representing a Rectangle with a few convenient methods.

Public Functions

uint32_t	**area**() const
	Calculate the area of the rectangle.
FORCE_INLINE_FUNCTION int16_t	**bottom**() const
	Gets the y coordinate of the bottom edge of the **Rect**.
void	**expandToFit**(const **Rect** & other)
	Increases the area covered by this rectangle to encompass the area covered by supplied rectangle.
bool	**includes**(const **Rect** & other) const
	Determines whether the specified rectangle is completely included in this rectangle.
bool	**intersect**(const **Rect** & other) const
	Determines whether specified rectangle intersects with this rectangle.
bool	**intersect**(int16_t otherX, int16_t otherY) const
	Determines whether specified point lies inside this rectangle.
bool	**isEmpty**() const
	Query if this object is empty.
bool	**operator!=**(const **Rect** & other) const
	Opposite of the == operator.
Rect	**operator&**(const **Rect** & other) const
	Gets a rectangle describing the intersecting area between this rectangle and the supplied rectangle.
void	**operator&=**(const **Rect** & other)
	Assigns this **Rect** to the intersection of the current **Rect** and the assigned **Rect**.

bool	**operator==**(const **Rect** & other) const
	Compares equality of two **Rect** by the dimensions and position of these.
	Rect()
	Default constructor.
	Rect(int16_t x, int16_t y, int16_t width, int16_t height)
	Initializes a new instance of the **Rect** class.
FORCE_INLINE_FUNCTION int16_t	**right**() const
	Gets the x coordinate of the right edge of the **Rect**.

Public Attributes

int16_t	**height**
	The height.
int16_t	**width**
	The width.
int16_t	**x**
	The x coordinate.
int16_t	**y**
	The y coordinate.

Public Functions Documentation

area

uint32_t area ( ) const

Calculate the area of the rectangle.

Returns:

area of the rectangle.

bottom

FORCE_INLINE_FUNCTION int16_t bottom ( ) const

Gets the y coordinate of the bottom edge of the **Rect**.

Returns:

y coordinate of the buttom edge.

expandToFit

void expandToFit (const Rect & other)

Increases the area covered by this rectangle to encompass the area covered by supplied rectangle.

Parameters:

other The other rectangle.

includes

bool includes (const Rect & other)

Determines whether the specified rectangle is completely included in this rectangle.

Parameters:

other The other rectangle.

Returns:

true if the specified rectangle is completely included.

intersect

bool intersect (const Rect & other)

Determines whether specified rectangle intersects with this rectangle.

Parameters:

other The other rectangle.

Returns:

true if the two rectangles intersect.

intersect

```
bool intersect ( int16_t otherX ,  const
                 int16_t otherY    const
                 )         const
```

Determines whether specified point lies inside this rectangle.

Parameters:

otherX The x coordinate of the point.
otherY The y coordinate of the point.

Returns:

true if point lies inside rectangle.

isEmpty

```
bool isEmpty (   ) const
```

Query if this object is empty.

Returns:

true if any of the dimensions are 0.

operator!=

```
bool operator!= ( const Rect & other )
```

Opposite of the == operator.

Parameters:

other The **Rect** to compare with.

Returns:

true if the compared **Rect** differ in dimensions or coordinates.

operator&

```
Rect operator& ( const Rect & other )
```

Gets a rectangle describing the intersecting area between this rectangle and the supplied rectangle.

Parameters:

other The other rectangle.

Returns:

Intersecting rectangle or Rect(0, 0, 0, 0) in case of no intersection.

operator&=

void operator&= (const Rect & other)

Assigns this **Rect** to the intersection of the current **Rect** and the assigned **Rect**.

The assignment will result in a Rect(0, 0, 0, 0) if they do not intersect.

Parameters:

other The rect to intersect with.

operator==

bool operator== (const Rect & other)

Compares equality of two **Rect** by the dimensions and position of these.

Parameters:

other The **Rect** to compare with.

Returns:

true if the compared **Rect** have the same dimensions and coordinates.

Rect

Rect ( )

Default constructor.

Resulting in an empty **Rect** with coordinates 0,0.

Rect

Rect (int16_t x ,

```
    int16_t y ,
    int16_t width ,
    int16_t height
    )
```

Initializes a new instance of the **Rect** class.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width.
height	The height.

right

```
FORCE_INLINE_FUNCTION int16_t right (  ) const
```

Gets the x coordinate of the right edge of the **Rect**.

Returns:

x coordinate of the right edge.

Public Attributes Documentation

height

int16_t height

The height.

width

int16_t width

The width.

x

int16_t x

The x coordinate.

y

int16_t y

The y coordinate.

Version: 4.16

RepeatButton

A RepeatButton is similar to a regular Button, but it will 'repeat' if pressed for a long period of time. The RepeatButton differs from a regular Button with regards to activation. A Button is activated when the button is released, whereas a RepeatButton is activated immediately when pressed and then at regular intervals. A RepeatButton does not activate when released.

As for other well-known repeat buttons, the interval from the first activation until the second activation as well as the subsequent interval between activations can be set for the RepeatButton.

The default values for initial delay is 10 ticks, and the default value for the following delays between button activations is 5 ticks.

Inherits from: Button, AbstractButton, Widget, Drawable

Public Functions

virtual int	**getDelay**()
	Gets the delay in ticks from first button activation until next activation.
virtual int	**getInterval**()
	The interval between repeated activations, measured in ticks.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
	RepeatButton()
virtual void	**setDelay**(int delay)
	Sets the delay (in number of ticks) from the first button activation until the next time it will be automatically activated.
virtual void	**setInterval**(int interval)
	Sets the interval in number of ticks between each each activation of the pressed button after the second activation.

Additional inherited members

Public Functions inherited from Button

	Button()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBitmaps**(const **Bitmap** & bitmapReleased, const **Bitmap** & bitmapPressed)
	Sets the two bitmaps used by this button.

Protected Attributes inherited from Button

uint8_t	**alpha**
	The current alpha value. 255=solid, 0=invisible.
Bitmap	**down**
	The image to display when button is pressed.
Bitmap	**up**
	The image to display when button is released (normal state).

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.

virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.

virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.

bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getDelay

virtual int getDelay ( )

Gets the delay in ticks from first button activation until next activation.

Returns:

The delay, measured in ticks, between first activation and second activation.

See also:

setDelay

getInterval

virtual int getInterval ( )

The interval between repeated activations, measured in ticks.

This is the number of ticks between the an activation beyond the first and the following activation.

Returns:

The interval between repeated activations, measured in ticks.

See also:

setInterval

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Updates the current state of the button.

The state can be either pressed or released, and if the new state is different from the current state, the button is also invalidated to force a redraw.

If the button state is changed from **ClickEvent::PRESSED** to **ClickEvent::RELEASED**, the associated action (if any) is also executed.

Parameters:

event Information about the click.

Reimplements: **touchgfx::AbstractButton::handleClickEvent**

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

RepeatButton

RepeatButton ( )

setDelay

virtual void setDelay (int delay)

Sets the delay (in number of ticks) from the first button activation until the next time it will be automatically activated.

Parameters:

delay The delay, measured in ticks, between first activation and second activation.

See also:

setInterval, **getDelay**

setInterval

virtual void setInterval (int interval)

Sets the interval in number of ticks between each each activation of the pressed button after the second activation.

Parameters:

`interval` The interval between repeated activations, measured in ticks.

See also:

setDelay, **getInterval**

Version: 4.16

RepeatButtonTrigger

A repeat button trigger. This trigger will create a button that reacts to a consistent touch. This means it will call the set action repeatedly as long as it is touched. The RepeatButtonTrigger can be combined with one or more of the ButtonStyle classes to create a fully functional button.

Inherits from: AbstractButtonContainer, Container, Drawable

Public Functions

int	**getDelay**()
	Gets the delay in ticks from first button activation until next activation.
int	**getInterval**()
	The interval between repeated activations, measured in ticks.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
	RepeatButtonTrigger()
void	**setDelay**(int delay)
	Sets the delay (in number of ticks) from the first button activation until the next time it will be automatically activated.
void	**setInterval**(int interval)
	Sets the interval in number of ticks between each each activation of the pressed button after the second activation.

Additional inherited members

Public Functions inherited from AbstractButtonContainer

	AbstractButtonContainer()
virtual void	**executeAction**() Executes the previously set action.
uint8_t	**getAlpha**() const Gets the current alpha value of the widget.
bool	**getPressed**() Gets the pressed state.
void	**setAction**(**GenericCallback**< const **AbstractButtonContainer** & > & callback) Sets an action callback to be executed by the subclass of AbstractContainerButton.
void	**setAlpha**(uint8_t newAlpha) Sets the opacity (alpha value).
void	**setPressed**(bool isPressed) Sets the pressed state to the given state.

Protected Functions inherited from AbstractButtonContainer

virtual void	**handleAlphaUpdated**() Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**() Handles what should happen when the pressed state is updated.

Protected Attributes inherited from AbstractButtonContainer

GenericCallback< const **AbstractButtonContainer** & > *	**action** The action to be executed.
uint8_t	**alpha** The current alpha value. 255 denotes solid, 0 denotes completely invisible.
bool	**pressed** True if pressed.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const

	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()

	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const

	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getDelay

int getDelay ( )

Gets the delay in ticks from first button activation until next activation.

Returns:

The delay, measured in ticks, between first activation and second activation.

See also:

setDelay

getInterval

int getInterval ( )

The interval between repeated activations, measured in ticks.

This is the number of ticks between the an activation beyond the first and the following activation.

Returns:

The interval between repeated activations, measured in ticks.

See also:

setInterval

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: touchgfx::Drawable::handleTickEvent

RepeatButtonTrigger

RepeatButtonTrigger ( )

setDelay

void setDelay (int delay)

Sets the delay (in number of ticks) from the first button activation until the next time it will be automatically activated.

Parameters:

delay The delay, measured in ticks, between first activation and second activation.

See also:

setInterval, **getDelay**

setInterval

void setInterval (int interval)

Sets the interval in number of ticks between each each activation of the pressed button after the second activation.

Parameters:

interval The interval between repeated activations, measured in ticks.

See also:

setDelay, **getInterval**

Version: 4.16

ScalableImage

Widget for representing a scaled version of a bitmap. Simply change the width/height of the widget to resize the image. The quality of the scaled image depends of the rendering algorithm used. The rendering algorithm can be changed dynamically. Please note that scaling images is done at runtime and may require a lot of calculations.

Note: Note that this widget does not support 1 bit per pixel color depth.

Inherits from: Image, Widget, Drawable

Public Types

enum	**ScalingAlgorithm** { NEAREST_NEIGHBOR, BILINEAR_INTERPOLATION }
	Rendering algorithm to use when scaling the bitmap.

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual **ScalingAlgorithm**	**getScalingAlgorithm**()
	Gets the algorithm used when rendering.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
	ScalableImage(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **ScalableImage** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
virtual void	**setScalingAlgorithm**(**ScalingAlgorithm** algorithm)
	Sets the algorithm to be used.

Protected Attributes

ScalingAlgorithm	**currentScalingAlgorithm**
	The current scaling algorithm.

Additional inherited members

Public Functions inherited from Image

uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getBitmap**() const
	Gets the **Bitmap** currently assigned to the **Image** widget.
BitmapId	**getBitmapId**() const
	Gets the BitmapId currently assigned to the **Image** widget.
	Image(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this **Image** and updates the width and height of this widget to match those of the **Bitmap**.

Protected Attributes inherited from Image

uint8_t	**alpha**
	The Alpha for this image.
Bitmap	**bitmap**
	The **Bitmap** to display.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const

	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

ScalingAlgorithm

enum ScalingAlgorithm

Rendering algorithm to use when scaling the bitmap.

NEAREST_NEIGHBOR	Fast but not a very good image quality. Good for fast animations.
BILINEAR_INTERPOLATION	Slower but better image quality. Good for static representation of a scaled image.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Image::draw**

getScalingAlgorithm

virtual ScalingAlgorithm getScalingAlgorithm ( )

Gets the algorithm used when rendering.

Returns:

The algorithm used when rendering.

See also:

ScalingAlgorithm

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to

be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Image::getSolidRect**

ScalableImage

ScalableImage (const Bitmap & bitmap =Bitmap())

Constructs a new **ScalableImage** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

If a **Bitmap** is passed to the constructor, the width and height of this widget is set to those of the bitmap.

Parameters:

bitmap (Optional) The bitmap to display.

See also:

setBitmap

setScalingAlgorithm

virtual void setScalingAlgorithm (ScalingAlgorithm algorithm)

Sets the algorithm to be used.

In short, there is currently a value for fast (nearest neighbor) and a value for slow (bi-linear interpolation).

Parameters:

algorithm The algorithm to use when rendering.

See also:

ScalingAlgorithm

Protected Attributes Documentation

currentScalingAlgorithm

ScalingAlgorithm currentScalingAlgorithm

The current scaling algorithm.

Version: 4.16

Screen

A Screen represents a full-screen drawable area. Applications create specific screens by subclassing this class. Each Screen has a root container to which drawables are added. The Screen makes sure to delegate draw requests and various events to the appropriate drawables in correct order.

Inherited by: View< T >

Public Functions

virtual void	**afterTransition**()
	Called by **Application::handleTickEvent()** when the transition to the screen is done.
void	**bindTransition**(**Transition** & trans)
	Enables the transition to access the containers.
void	**draw**()
	Tells the screen to draw its entire area.
virtual void	**draw**(**Rect** & rect)
	Tell the screen to draw the specified area.
Container &	**getRootContainer**()
	Obtain a reference to the root container of this screen.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Traverse the drawables in reverse z-order and notify them of a click event.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Traverse the drawables in reverse z-order and notify them of a drag event.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Handle gestures.
virtual void	**handleKeyEvent**(uint8_t key)
	Called by the **Application** on the reception of a "key", the meaning of which is platform/application specific.
virtual void	**handleTickEvent**()

	Called by the **Application** on the current screen with a frequency of **Application::TICK_INTERVAL_MS**.
void	**JSMOC**(const **Rect** & invalidatedArea, **Drawable** * widgetToDraw)
	Recursive JSMOC function.
	Screen()
	Initializes a new instance of the **Screen** class.
virtual void	**setupScreen**()
	Called by **Application::switchScreen()** when this screen is going to be displayed.
void	**startSMOC**(const **Rect** & invalidatedArea)
	Starts a JSMOC run, analyzing what parts of what widgets should be redrawn.
virtual void	**tearDownScreen**()
	Called by **Application::switchScreen()** when this screen will no longer be displayed.
bool	**usingSMOC**() const
	Determines if using JSMOC.
virtual	**~Screen**()
	Finalizes an instance of the **Screen** class.

Protected Functions

void	**add**(**Drawable** & d)
	Add a drawable to the content container.
void	**remove**(**Drawable** & d)
	Removes a drawable from the content container.
void	**useSMOCDrawing**(bool enabled)
	Determines whether to use JSMOC or painter's algorithm for drawing.

Protected Attributes

Container	**container**
	The container contains the contents of the screen.

Drawable * focus
The drawable currently in focus (set when DOWN_PRESSED is received).

Public Functions Documentation

afterTransition

virtual void afterTransition ( )

Called by **Application::handleTickEvent()** when the transition to the screen is done.

Base version does nothing, but override to do screen specific initialization code that has to be done after the transition to the screen.

See also:

Application::handleTickEvent

bindTransition

void bindTransition (Transition & trans)

Enables the transition to access the containers.

Parameters:

trans The transition to bind.

draw

void draw ( )

Tells the screen to draw its entire area.

NOTE

The more specific draw(Rect&) version is preferred when possible.

draw

virtual void draw (Rect & rect)

Tell the screen to draw the specified area.

Will traverse the drawables tree from in z-order and delegate draw to them.

Parameters:

rect The area in absolute coordinates.

NOTE

The given rect must be in absolute coordinates.

getRootContainer

Container & getRootContainer ( )

Obtain a reference to the root container of this screen.

Returns:

The root container.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Traverse the drawables in reverse z-order and notify them of a click event.

Parameters:

evt The event to handle.

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Traverse the drawables in reverse z-order and notify them of a drag event.

Parameters:

evt The event to handle.

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Handle gestures.

Traverses drawables in reverse-z and notifies them of the gesture.

Parameters:

evt The event to handle.

handleKeyEvent

virtual void handleKeyEvent (uint8_t key)

Called by the **Application** on the reception of a "key", the meaning of which is platform/application specific.

Default implementation does nothing.

Parameters:

key The key to handle.

handleTickEvent

virtual void handleTickEvent ( )

Called by the **Application** on the current screen with a frequency of **Application::TICK_INTERVAL_MS**.

JSMOC

void JSMOC (const Rect & invalidatedArea ,
Drawable * widgetToDraw
)

Recursive JSMOC function.

This is the actual occlusion culling implementation.

Parameters:

invalidatedArea The area to redraw, expressed in absolute coordinates.
widgetToDraw **Widget** currently being drawn.

NOTE

JSMOC is an abbreviation of *Jesper, Sren & Martin's Occlusion Culling*.

Screen

Screen ( )

Initializes a new instance of the **Screen** class.

setupScreen

virtual void setupScreen ( )

Called by **Application::switchScreen()** when this screen is going to be displayed.

Base version does nothing, but place any screen specific initialization code in an overridden version.

See also:

Application::switchScreen

startSMOC

void startSMOC (const Rect & invalidatedArea)

Starts a JSMOC run, analyzing what parts of what widgets should be redrawn.

Parameters:

invalidatedArea The area to redraw, expressed in absolute coordinates.

NOTE

SMOC is an abbreviation of *Sren & Martin's Occlusion Culling*.

tearDownScreen

virtual void tearDownScreen ( )

Called by **Application::switchScreen()** when this screen will no longer be displayed.

Base version does nothing, but place any screen specific cleanup code in an overridden version.

See also:

Application::switchScreen

usingSMOC

bool usingSMOC ( ) const

Determines if using JSMOC.

Returns:

true if this screen uses the JSMOC drawing algorithm.

~Screen

virtual ~Screen ( )

Finalizes an instance of the **Screen** class.

Protected Functions Documentation

add

void add (Drawable & d)

Add a drawable to the content container.

Parameters:

d The **Drawable** to add.

NOTE

Must not be called with a Drawable that was already added to the screen. If in doubt, call remove() first.

remove

void remove (Drawable & d)

Removes a drawable from the content container.

Safe to call even if the drawable was never added (in which case nothing happens).

Parameters:

d The **Drawable** to remove.

useSMOCDrawing

void useSMOCDrawing (bool enabled)

Determines whether to use JSMOC or painter's algorithm for drawing.

Parameters:

enabled true if JSMOC should be enabled, false if disabled (meaning painter's algorithm is employed instead).

Protected Attributes Documentation

container

Container container

The container contains the contents of the screen.

focus

Drawable * focus

The drawable currently in focus (set when DOWN_PRESSED is received).

Version: 4.16

ScrollableContainer

A ScrollableContainer is a container that allows its contents to be scrolled. It will intercept drag operations and move child nodes accordingly.

A standard Container will simply clip children that are either larger than the container itself, or children that extend beyond the borders of the container or children that are placed outside the borders of the container. A ScrollableContainer behaves much like a Container, except it enables the user to scroll the children and thereby act like a viewport. When the contents of the ScrollableContainer is scrollable, scrollbars can be seen near the edge of the ScrollableContainer.

See: Container

Note: The ScrollableContainer will consume all DragEvents in the area covered by the container.

Inherits from: Container, Drawable

Public Functions

virtual void	**add**(**Drawable** & d)
	Adds a **Drawable** instance as child to this **Container**.
virtual void	**childGeometryChanged**()
	Used to signal that the size or position of one or more children have changed.
void	**enableHorizontalScroll**(bool enable)
	Enables horizontal scrolling.
void	**enableVerticalScroll**(bool enable)
	Enables vertical scrolling.
virtual **Rect**	**getContainedArea**() const
	Gets the area that contains all children added to the **ScrollableContainer**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
uint16_t	**getScrollDurationSlowdown**() const
	Gets scroll duration speedup divisor.

uint16_t	**getScrollDurationSpeedup**() const
	Gets scroll duration speedup multiplier.
int16_t	**getScrolledX**() const
	Gets the distance scrolled for the x-axis.
int16_t	**getScrolledY**() const
	Gets the distance scrolled for the y-axis.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**isScrollableXY**(bool & scrollX, bool & scrollY)
	Is the ClickableContainer scrollable in either direction? Takes the width of the contained elements into account and also checks to see if horizontal or vertical scrolling is allowed.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.
void	**reset**()
	Resets the **ScrollableContainer** to its original state, before the user started dragging the contents.
	ScrollableContainer()
void	**setMaxVelocity**(uint16_t max)
	Sets the maximum velocity of a scroll due to a swipe.
void	**setScrollbarPadding**(uint8_t padding)
	Sets the amount of space between the scrollbar and the edge of the **ScrollableContainer**.
void	**setScrollbarsAlpha**(uint8_t alpha)

Sets the alpha value (transparency) of the scrollbars.

void **setScrollbarsColor**(**colortype** color)

Sets the color of the scrollbars.

void **setScrollbarsPermanentlyVisible**(bool permanentlyVisible =true)

Make scrollbars permanently visible regardless of the size and position of the children of the **ScrollableContainer**.

void **setScrollbarsVisible**(bool newVisible)

Sets the visibility of the scrollbars, when the scrollable area is pressed.

void **setScrollbarWidth**(uint8_t width)

Sets the width of the scrollbar measured in pixels.

void **setScrollDurationSlowdown**(uint16_t slowdown)

Sets scroll duration speedup divisor.

void **setScrollDurationSpeedup**(uint16_t speedup)

Sets scroll duration speedup multiplier.

void **setScrollThreshold**(int16_t t)

Change the threshold which the first drag event received must exceed before initiating a scroll.

Protected Functions

virtual bool **doScroll**(int16_t deltaX, int16_t deltaY)

Method to actually scroll the container.

Rect **getXBorder**(const **Rect** & xBar, const **Rect** & yBar) const

Gets the area where the horizontal scrollbar can move.

Rect **getXScrollbar**() const

Gets x coordinate of the scrollbar.

Rect **getYBorder**(const **Rect** & xBar, const **Rect** & yBar) const

Gets the area where the vertical scrollbar can move.

Rect **getYScrollbar**() const

Gets y coordinate of the scrollbar.

void	**invalidateScrollbars**()
	Invalidate the scrollbars.

Protected Attributes

GestureEvent::GestureEventType	**accelDirection**
	The current direction (horizontal or vertical) of scroll.
bool	**animate**
	Is scroll animation currently active.
uint16_t	**animationCounter**
	Current step/tick in scroll animation.
int16_t	**beginningValue**
	Initial X or Y for calculated values in scroll animation.
int16_t	**fingerAdjustmentX**
	How much should the finger be adjusted horizontally.
int16_t	**fingerAdjustmentY**
	and how much vertically
bool	**hasIssuedCancelEvent**
	true if the pressed drawable has received cancel event
bool	**isPressed**
	Is the container currently pressed (maybe show scrollbars)
bool	**isScrolling**
	Is the container scrolling (i.e. has overcome the initial larger drag that is required to initiate a scroll).
Drawable *	**lastDraggableChild**
	The drawable child of this container which should receive drag events. Note that only drag events in directions which cannot be scrolled by this **ScrollableContainer** will be forwarded to children.
uint16_t	**maxVelocity**
	The maximum velocity of a scroll (due to a swipe)

Drawable *	**pressedDrawable**
	The drawable child of this container which received the last **ClickEvent::PRESSED** notification. When scrolling, send this drawable a CANCEL event if the new x/y coords no longer matches this drawable.
int16_t	**pressedX**
	The x coordinate where the last **ClickEvent::PRESSED** was received.
int16_t	**pressedY**
	The y coordinate where the last **ClickEvent::PRESSED** was received.
bool	**scrollableX**
	Is the container scrollable in the horizontal direction.
bool	**scrollableY**
	Is the container scrollable in the vertical direction.
uint8_t	**scrollbarAlpha**
	The scrollbar is semitransparent.
colortype	**scrollbarColor**
	The color of the scrollbar.
uint8_t	**scrollbarPadding**
	The amount of padding. The scrollbar will have a bit of space to the borders of the container.
bool	**scrollbarsPermanentlyVisible**
	Are scrollbars always visible.
bool	**scrollbarsVisible**
	Are scrollbars visible.
uint8_t	**scrollbarWidth**
	The width of the scrollbar.
uint16_t	**scrollDuration**
	Number of ticks the scroll animation should use.
uint16_t	**scrollDurationSlowdown**
	The scroll durations is divided by this number.

uint16_t	**scrollDurationSpeedup**
	The scroll durations is multipled by this number.
int16_t	**scrolledXDistance**
	The scrolled horizontal distance.
int16_t	**scrolledYDistance**
	The scrolled vertical distance.
int16_t	**scrollThreshold**
	The threshold which the first drag event received must exceed before scrolling. Default is 5.
int16_t	**targetValue**
	Target X or Y value for scroll animation.
Box	**xSlider**
	The horizontal scrollbar drawable.
Box	**ySlider**
	The vertical scrollbar drawable.
const uint8_t	**SCROLLBAR_LINE**
	The scrollbar line.
const uint16_t	**SCROLLBAR_MAX_VELOCITY**
	The (default) maximum velocity of a scroll due to a swipe.
const uint16_t	**SCROLLBAR_MIN_VELOCITY**
	The minimum velocity of a scroll due to a swipe.

Additional inherited members

Public Functions inherited from Container

	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)

	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const

	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

add

virtual void add (Drawable & d)

Adds a **Drawable** instance as child to this **Container**.

The **Drawable** added will be placed as the element to be drawn last, and thus appear on top of all previously added drawables in the **Container**.

Parameters:

d The **Drawable** to add.

NOTE

Never add a drawable more than once!

Reimplements: **touchgfx::Container::add**

childGeometryChanged

virtual void childGeometryChanged ( )

Used to signal that the size or position of one or more children have changed.

This function can be called on parent nodes to signal that the size of one or more of its children have changed.

Reimplements: **touchgfx::Drawable::childGeometryChanged**

enableHorizontalScroll

void enableHorizontalScroll (bool enable)

Enables horizontal scrolling.

By default, scrolling in either direction is enabled, provided that the content is larger than the size of the **ScrollableContainer**. This function can be used to explicitly (dis)allow horizontal scrolling, even if the content is larger than the container.

Parameters:

enable If true (default), horizontal scrolling is enabled. If false, horizontal scrolling is disabled.

See also:

enableVerticalScroll

enableVerticalScroll

void enableVerticalScroll (bool enable)

Enables vertical scrolling.

By default, scrolling in either direction is enabled, provided that the content is larger than the size of the **ScrollableContainer**. This function can be used to explicitly (dis)allow vertical scrolling, even if the content is larger than the container.

Parameters:

enable If true (default), vertical scrolling is enabled. If false, vertical scrolling is disabled.

See also:

enableHorizontalScroll

getContainedArea

virtual Rect getContainedArea ( ) const

Gets the area that contains all children added to the **ScrollableContainer**.

The scrollbars are not considered in this operation.

Returns:

The contained area.

Reimplements: **touchgfx::Container::getContainedArea**

getLastChild

```
virtual void getLastChild ( int16_t      x ,
                            int16_t      y ,
                            Drawable ** last
                            )
```

Gets the last child in the list of children in this **Container**.

If this **Container** is touchable (**isTouchable()**), it will be passed back as the result. Otherwise all *visible* children are traversed recursively to find the **Drawable** that intersects with the given coordinate.

Parameters:

x	The x coordinate of the intersection.
y	The y coordinate of the intersection.
last	out parameter in which the result is placed.

See also:

isVisible, **isTouchable**

Reimplements: **touchgfx::Container::getLastChild**

getScrollDurationSlowdown

uint16_t getScrollDurationSlowdown ( ) const

Gets scroll duration speedup divisor.

Returns:

The scroll duration speedup divisor.

See also:

setScrollDurationSlowdown

getScrollDurationSpeedup

uint16_t getScrollDurationSpeedup ( ) const

Gets scroll duration speedup multiplier.

Returns:

The swipe acceleration.

See also:

setScrollDurationSpeedup, **getScrollDurationSlowdown**

getScrolledX

int16_t getScrolledX ( ) const

Gets the distance scrolled for the x-axis.

Returns:

the distance scrolled for the x-axis.

getScrolledY

int16_t getScrolledY ( ) const

Gets the distance scrolled for the y-axis.

Returns:

the distance scrolled for the y-axis.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Defines the event handler interface for GestureEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **GestureEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleGestureEvent**

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

isScrollableXY

virtual void isScrollableXY (bool & scrollX ,
bool & scrollY
)

Is the ClickableContainer scrollable in either direction? Takes the width of the contained elements into account and also checks to see if horizontal or vertical scrolling is allowed.

Parameters:

scrollX True if the container is able to scroll horizontally.
scrollY True if the container is able to scroll vertically.

See also:

enableHorizontalScroll, **enableVerticalScroll**

moveChildrenRelative

virtual void moveChildrenRelative (int16_t deltaX ,
int16_t deltaY
)

Calls moveRelative on all children.

Parameters:

deltaX Horizontal displacement.
deltaY Vertical displacement.

NOTE

Takes care not to move the scrollbars, which are also children.

Reimplements: **touchgfx::Container::moveChildrenRelative**

reset

void reset ( )

Resets the **ScrollableContainer** to its original state, before the user started dragging the contents.

This reset the x/y coordinates of children to the position they were in before the first drag event was received.

ScrollableContainer

ScrollableContainer ( )

setMaxVelocity

void setMaxVelocity (uint16_t max)

Sets the maximum velocity of a scroll due to a swipe.

This can be used to force smooth scrolling by limiting the speed of any swipe gesture.

Parameters:

max The maximum velocity of the scroll.

See also:

GestureEvent::getVelocity

setScrollbarPadding

void setScrollbarPadding (uint8_t padding)

Sets the amount of space between the scrollbar and the edge of the **ScrollableContainer**.

Parameters:

padding The padding.

setScrollbarsAlpha

void setScrollbarsAlpha (uint8_t alpha)

Sets the alpha value (transparency) of the scrollbars.

Parameters:

alpha The alpha value. 255 being completely solid, 0 being completely invisible.

setScrollbarsColor

void setScrollbarsColor (colortype color)

Sets the color of the scrollbars.

Parameters:

color The color of the box.

setScrollbarsPermanentlyVisible

void setScrollbarsPermanentlyVisible (bool permanentlyVisible =true)

Make scrollbars permanently visible regardless of the size and position of the children of the **ScrollableContainer**.

Normally the scrollbars are hidden and only shown when dragging the contents of the **ScrollableContainer** (unless prohibited using **setScrollbarsVisible()**).

Parameters:

permanentlyVisible (Optional) True to show the scrollbars permanently, false for default behavior.

See also:

setScrollbarsVisible

setScrollbarsVisible

void setScrollbarsVisible (bool newVisible)

Sets the visibility of the scrollbars, when the scrollable area is pressed.

By default the scrollbars are hidden, but shown when the contents of the **ScrollableContainer** is being dragged around. Using setScrollbarsVisible, it is possble to hide the scrollbars when dragging the contents.

Parameters:

newVisible If true (default), the scrollbars are visible when scrollable area is pressed. If false, scrollbars are always hidden.

See also:

setScrollbarsPermanentlyVisible

setScrollbarWidth

void setScrollbarWidth (uint8_t width)

Sets the width of the scrollbar measured in pixels.

Parameters:

width The width of the scrollbar.

setScrollDurationSlowdown

void setScrollDurationSlowdown (uint16_t slowdown)

Sets scroll duration speedup divisor.

Default value is 1.

Parameters:

slowdown The scroll duration speedup divisor.

See also:

setScrollDurationSpeedup, **getScrollDurationSlowdown**

setScrollDurationSpeedup

void setScrollDurationSpeedup (uint16_t speedup)

Sets scroll duration speedup multiplier.

Default value is 7 which gives a nice speedup on gestures.

Parameters:

speedup The scroll duration speedup multiplier.

See also:

getScrollDurationSpeedup, **setScrollDurationSlowdown**

setScrollThreshold

void setScrollThreshold (int16_t t)

Change the threshold which the first drag event received must exceed before initiating a scroll.

This can be used to avoid touching the screen and moving the finger only a few pixels resulting in the contents being scrolled.

Parameters:

t The new threshold value.

NOTE

All subsequent scrolls will be processed regardless of threshold value until a ClickEvent::RELEASED is received.

Protected Functions Documentation

doScroll

virtual bool doScroll (int16_t deltaX ,
int16_t deltaY
)

Method to actually scroll the container.

Passing negative values will scroll the items in the **ScrollableContainer** up / left, whereas positive values will scroll items down / right.

If the distance is larger than allowed, the deltas are adjusted down to make sure the contained items stay inside view.

Parameters:

deltaX	The horizontal amount to scroll.
deltaY	The vertical amount to scroll.

Returns:

did the container actually scroll. The call doScroll(0,0) will always return false.

getXBorder

```
Rect getXBorder ( const Rect & xBar , const
                  const Rect & yBar const
                  )              const
```

Gets the area where the horizontal scrollbar can move.

Parameters:

xBar	The current horizontal scrollbar, supplied for caching reasons.
yBar	The current vertical scrollbar, supplied for caching reasons.

Returns:

The area.

getXScrollbar

```
Rect getXScrollbar (  ) const
```

Gets x coordinate of the scrollbar.

Returns:

The horizontal scrollbar area.

getYBorder

```
Rect getYBorder ( const Rect & xBar , const
                  const Rect & yBar const
```

) const

Gets the area where the vertical scrollbar can move.

Parameters:

xBar The current horizontal scrollbar, supplied for caching reasons.
yBar The current vertical scrollbar, supplied for caching reasons.

Returns:

The area.

getYScrollbar

Rect getYScrollbar ( ) const

Gets y coordinate of the scrollbar.

Returns:

The vertical scrollbar area.

invalidateScrollbars

void invalidateScrollbars ( )

Invalidate the scrollbars.

Protected Attributes Documentation

accelDirection

GestureEvent::GestureEventType accelDirection

The current direction (horizontal or vertical) of scroll.

animate

bool animate

Is scroll animation currently active.

animationCounter

uint16_t animationCounter

Current step/tick in scroll animation.

beginningValue

int16_t beginningValue

Initial X or Y for calculated values in scroll animation.

fingerAdjustmentX

int16_t fingerAdjustmentX

How much should the finger be adjusted horizontally.

fingerAdjustmentY

int16_t fingerAdjustmentY

and how much vertically

hasIssuedCancelEvent

bool hasIssuedCancelEvent

true if the pressed drawable has received cancel event

isPressed

bool isPressed

Is the container currently pressed (maybe show scrollbars)

isScrolling

bool isScrolling

Is the container scrolling (i.e. has overcome the initial larger drag that is required to initiate a scroll).

lastDraggableChild

Drawable * lastDraggableChild

The drawable child of this container which should receive drag events. Note that only drag events in directions which cannot be scrolled by this **ScrollableContainer** will be forwarded to children.

maxVelocity

uint16_t maxVelocity

The maximum velocity of a scroll (due to a swipe)

pressedDrawable

Drawable * pressedDrawable

The drawable child of this container which received the last **ClickEvent::PRESSED** notification. When scrolling, send this drawable a CANCEL event if the new x/y coords no longer matches this drawable.

pressedX

int16_t pressedX

The x coordinate where the last **ClickEvent::PRESSED** was received.

pressedY

int16_t pressedY

The y coordinate where the last **ClickEvent::PRESSED** was received.

scrollableX

bool scrollableX

Is the container scrollable in the horizontal direction.

scrollableY

bool scrollableY

Is the container scrollable in the vertical direction.

scrollbarAlpha

uint8_t scrollbarAlpha

The scrollbar is semitransparent.

scrollbarColor

colortype scrollbarColor

The color of the scrollbar.

scrollbarPadding

uint8_t scrollbarPadding

The amount of padding. The scrollbar will have a bit of space to the borders of the container.

scrollbarsPermanentlyVisible

bool scrollbarsPermanentlyVisible

Are scrollbars always visible.

scrollbarsVisible

bool scrollbarsVisible

Are scrollbars visible.

scrollbarWidth

uint8_t scrollbarWidth

The width of the scrollbar.

scrollDuration

uint16_t scrollDuration

Number of ticks the scroll animation should use.

scrollDurationSlowdown

uint16_t scrollDurationSlowdown

The scroll durations is divided by this number.

scrollDurationSpeedup

uint16_t scrollDurationSpeedup

The scroll durations is multipled by this number.

scrolledXDistance

int16_t scrolledXDistance

The scrolled horizontal distance.

scrolledYDistance

int16_t scrolledYDistance

The scrolled vertical distance.

scrollThreshold

int16_t scrollThreshold

The threshold which the first drag event received must exceed before scrolling. Default is 5.

targetValue

int16_t targetValue

Target X or Y value for scroll animation.

xSlider

Box xSlider

The horizontal scrollbar drawable.

ySlider

Box ySlider

The vertical scrollbar drawable.

SCROLLBAR_LINE

const uint8_t SCROLLBAR_LINE = 0

The scrollbar line.

SCROLLBAR_MAX_VELOCITY

const uint16_t SCROLLBAR_MAX_VELOCITY = 17

The (default) maximum velocity of a scroll due to a swipe.

SCROLLBAR_MIN_VELOCITY

const uint16_t SCROLLBAR_MIN_VELOCITY = 5

The minimum velocity of a scroll due to a swipe.

Version: 4.16

ScrollBase

The ScrollBase class is an abstract class used for Widgets that needs to show (a lot of) elements in a DrawableList that can be scrolled. Due to memory limitations, this is implemented by re-using the Drawables in the DrawableList - once an element is moved off screen, it is filled with new content and moved to the other end and the of the scrolling list.

Lists can be horizontal or vertical and the can be circular (infinite scrolling).

See:

- ScrollList, ScrollWheel, ScrollWheelWithSelectionStyle
- ScrollWheelBase, DrawableList

Inherits from: Container, Drawable

Inherited by: ScrollList, ScrollWheelBase

Protected Types

enum	**AnimationState** { NO_ANIMATION, ANIMATING_GESTURE, ANIMATING_DRAG }
	Values that represent animation states.

Public Functions

void	**allowHorizontalDrag**(bool enable)
	Enables horizontal scrolling to be passed to the children in the list (in case a child widget is able to handle drag events).
void	**allowVerticalDrag**(bool enable)
	Enables the vertical scroll.
virtual void	**animateToItem**(int16_t itemIndex, int16_t animationSteps =-1)
	Go to a specific item, possibly with animation.
uint16_t	**getAnimationSteps**() const
	Gets animation steps as set in setAnimationSteps.

virtual bool	**getCircular**() const
	Gets the circular setting, previously set using setCircular().
uint16_t	**getDragAcceleration**() const
	Gets drag acceleration (times 10).
virtual int16_t	**getDrawableMargin**() const
	Gets drawable margin as set through the second parameter in most recent call to setDrawableSize().
virtual int16_t	**getDrawableSize**() const
	Gets drawable size as set through the first parameter in most recent call to setDrawableSize().
virtual bool	**getHorizontal**() const
	Gets the orientation of the drawables, previously set using setHorizontal().
uint16_t	**getMaxSwipeItems**() const
	Gets maximum swipe items as set by setMaxSwipeItems.
virtual int16_t	**getNumberOfItems**() const
	Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().
uint16_t	**getSwipeAcceleration**() const
	Gets swipe acceleration (times 10).
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**initialize**()
	Removed all drawables and initializes the content of these items.
bool	**isAnimating**() const
	Query if an animation is ongoing.
virtual void	**itemChanged**(int itemIndex)

	Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.
	ScrollBase()
void	**setAnimationEndedCallback**(**GenericCallback**<> & callback)
	Callback, called when the set animation ended.
void	**setAnimationSteps**(int16_t steps)
	Sets animation steps (in ticks) when moving to a new selected item.
virtual void	**setCircular**(bool circular)
	Sets whether the list is circular (infinite) or not.
void	**setDragAcceleration**(uint16_t acceleration)
	Sets drag acceleration times 10, so "10" means "1", "15" means "1.5".
void	**setDrawableSize**(int16_t drawableSize, int16_t drawableMargin)
	Sets drawables size.
void	**setEasingEquation**(**EasingEquation** equation)
	Sets easing equation when changing the selected item, for example via swipe or AnimateTo.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setHorizontal**(bool horizontal)
	Sets a horizontal or vertical layout.
void	**setItemPressedCallback**(**GenericCallback**< int16_t > & callback)
	Set **Callback** which will be called when a item is pressed.
void	**setItemSelectedCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the selected item is clicked.
void	**setMaxSwipeItems**(uint16_t maxItems)
	Sets maximum swipe items.
virtual void	**setNumberOfItems**(int16_t numberOfItems)
	Sets number of items in the **DrawableList**.
void	**setSwipeAcceleration**(uint16_t acceleration)

	Sets swipe acceleration (times 10).
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**stopAnimation**()
	Stops an animation if one is ongoing.

Protected Functions

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.
virtual int32_t	**getNearestAlignedOffset**(int32_t offset) const
	Gets nearest offset aligned to a multiple of itemSize.
int	**getNormalizedOffset**(int offset) const
	Gets normalized offset from a given offset from 0 down to -numItems*itemSize.
virtual int32_t	**getOffset**() const
	Gets display offset of first item.
virtual int32_t	**getPositionForItem**(int16_t itemIndex) =0
	Get the position for an item.
virtual int32_t	**keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const =0
	Keep offset inside limits.
virtual void	**setOffset**(int32_t offset)
	Sets display offset of first item.

Protected Attributes

GenericCallback *	**animationEndedCallback**
	The animation ended callback.
AnimationState	**currentAnimationState**
	The current animation state.

uint16_t **defaultAnimationSteps**

The animation steps.

int16_t **distanceBeforeAlignedItem**

The distance before aligned item.

uint16_t **dragAcceleration**

The drag acceleration x10.

bool **draggableX**

Is the container draggable in the horizontal direction.

bool **draggableY**

Is the container draggable in the vertical direction.

EasingEquation **easingEquation**

The easing equation used for animation.

int **gestureEnd**

The gesture end.

int **gestureStart**

The gesture start.

int **gestureStep**

The current gesture step.

int **gestureStepsTotal**

The total gesture steps.

int32_t **initialSwipeOffset**

The initial swipe offset.

GenericCallback * **itemLockedInCallback**

The item locked in callback.

GenericCallback< int16_t > * **itemPressedCallback**

The item pressed callback.

GenericCallback< int16_t > * **itemSelectedCallback**

The item selected callback.

int16_t **itemSize**

	Size of the item (including margin)
DrawableList	**list**
	The list.
uint16_t	**maxSwipeItems**
	The maximum swipe items.
int16_t	**numberOfDrawables**
	Number of drawables.
uint16_t	**swipeAcceleration**
	The swipe acceleration x10.
int16_t	**xClick**
	The x coordinate of a click.
int16_t	**yClick**
	The y coordinate of a click.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const

	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
void	**setWidthHeight**(const **Bitmap** & bitmap)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**

	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Protected Types Documentation

AnimationState

enum AnimationState

Values that represent animation states.

NO_ANIMATION	No animation.
ANIMATING_GESTURE	Animating a gesture.
ANIMATING_DRAG	Animating a drag.

Public Functions Documentation

allowHorizontalDrag

void allowHorizontalDrag (bool enable)

Enables horizontal scrolling to be passed to the children in the list (in case a child widget is able to handle drag events).

By default, scrolling in either direction is disabled. This function can be used to explicitly (dis)allow scrolling in the horizontal direction.

Parameters:

enable If true, horizontal scrolling is enabled. If false (default), scrolling is disabled.

allowVerticalDrag

void allowVerticalDrag (bool enable)

Enables the vertical scroll.

Enables the vertical scroll to be passed to the children in the list (in case a child widget is able to handle drag events). By default, scrolling in either direction is disabled. This function can be used to explicitly (dis)allow scrolling in the vertical direction.

Parameters:

enable If true, vertical scrolling is enabled. If false (default), scrolling is disabled.

animateToItem

```
virtual void animateToItem ( int16_t itemIndex ,
                             int16_t animationSteps =-1
                             )
```

Go to a specific item, possibly with animation.

The given item index is scrolled into view. If animationSteps is omitted, the default number of animation steps is used. If animationSteps is 0 no animation will be used, otherwise the number of animation steps specified is used.

Parameters:

itemIndex	Zero-based index of the item.
animationSteps	(Optional) The steps to use for the animation. 0 means no animation. If omitted, default animation steps are used.

See also:

setAnimationSteps

getAnimationSteps

```
uint16_t getAnimationSteps (  ) const
```

Gets animation steps as set in setAnimationSteps.

Returns:

The animation steps.

See also:

setAnimationSteps, **setEasingEquation**

getCircular

virtual bool getCircular ( ) const

Gets the circular setting, previously set using setCircular().

Returns:

True if the list is circular (infinite), false if the list is not circular (finite).

See also:

DrawableList::getCircular, **setCircular**

getDragAcceleration

uint16_t getDragAcceleration ( ) const

Gets drag acceleration (times 10).

Returns:

The drag acceleration.

NOTE

The reason for multiplying the acceleration by 10 is to avoid introducing floating point arithmetic.

See also:

setDragAcceleration

getDrawableMargin

virtual int16_t getDrawableMargin ( ) const

Gets drawable margin as set through the second parameter in most recent call to setDrawableSize().

Returns:

The drawable margin.

See also:

setDrawableSize

getDrawableSize

virtual int16_t getDrawableSize ( ) const

Gets drawable size as set through the first parameter in most recent call to setDrawableSize().

Returns:

The drawable size.

See also:

setDrawableSize

getHorizontal

virtual bool getHorizontal ( ) const

Gets the orientation of the drawables, previously set using setHorizontal().

Returns:

True if it horizontal, false if it is vertical.

See also:

DrawableList::getHorizontal, **setHorizontal**

getMaxSwipeItems

uint16_t getMaxSwipeItems ( ) const

Gets maximum swipe items as set by setMaxSwipeItems.

Returns:

The maximum swipe items, 0 means "no limit".

See also:

setMaxSwipeItems

getNumberOfItems

virtual int16_t getNumberOfItems ( ) const

Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().

Returns:

The number of items.

See also:

setNumberOfItems, **DrawableList::getNumberOfItems**

getSwipeAcceleration

uint16_t getSwipeAcceleration ( ) const

Gets swipe acceleration (times 10).

Returns:

The swipe acceleration.

NOTE

The reason for multiplying the acceleration by 10 is to avoid introducing floating point arithmetic.

See also:

setSwipeAcceleration

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

Reimplemented by: **touchgfx::ScrollWheelBase::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Defines the event handler interface for GestureEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **GestureEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleGestureEvent**

Reimplemented by: **touchgfx::ScrollWheelBase::handleGestureEvent**

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

initialize

virtual void initialize ( )

Removed all drawables and initializes the content of these items.

Reimplemented by: **touchgfx::ScrollWheelWithSelectionStyle::initialize**

isAnimating

bool isAnimating ( ) const

Query if an animation is ongoing.

This can be good to know if getSelectedItem() is called, as the result might not be as expected if **isAnimating()** returns true, since the display is not showing the selected item in the right place yet.

Returns:

true if animating, false if not.

itemChanged

virtual void itemChanged (int itemIndex)

Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.

This is important as a circular list with very few items might display the same item more than once and all these items should be updated.

Parameters:

itemIndex Zero-based index of the changed item.

Reimplemented by: **touchgfx::ScrollWheelWithSelectionStyle::itemChanged**

ScrollBase

ScrollBase ( )

setAnimationEndedCallback

void setAnimationEndedCallback (GenericCallback<> & callback)

Callback, called when the set animation ended.

Parameters:

callback The ended callback.

setAnimationSteps

void setAnimationSteps (int16_t steps)

Sets animation steps (in ticks) when moving to a new selected item.

The default value is 30.

Parameters:

steps The animation steps.

See also:

setEasingEquation, **getAnimationSteps**

setCircular

virtual void setCircular (bool circular)

Sets whether the list is circular (infinite) or not.

A circular list is a list where the first drawable re-appears after the last item in the list - and the last item in the list appears before the first item in the list.

Parameters:

circular True if the list should be circular, false if the list should not be circular.

See also:

DrawableList::setCircular, getCircular

Reimplemented by: touchgfx::ScrollWheelWithSelectionStyle::setCircular

setDragAcceleration

void setDragAcceleration (uint16_t acceleration)

Sets drag acceleration times 10, so "10" means "1", "15" means "1.5".

10 makes the containers follow the finger, higher values makes the containers move faster. This can often be useful if the list is very long.

Parameters:

acceleration The drag acceleration.

NOTE

The reason for multiplying the acceleration by 10 is to avoid introducing floating point arithmetic.

See also:

getDragAcceleration

setDrawableSize

void setDrawableSize (int16_t drawableSize ,
int16_t drawableMargin
)

Sets drawables size.

The drawable is is the size of each drawable in the list in the set direction of the list (this is enforced by the **DrawableList** class). The specified margin is added above and below each item for spacing. The entire size of an item is thus size + 2 * spacing.

For a horizontal list each element will be *drawableSize* high and have the same width as set using **setWidth()**. For a vertical list each element will be *drawableSize* wide and have the same height as set using **setHeight()**.

Parameters:

drawableSize	The size of the drawable.
drawableMargin	The margin around drawables (margin before and margin after).

See also:

setWidth, **setHeight**, **setHorizontal**

setEasingEquation

void setEasingEquation (EasingEquation equation)

Sets easing equation when changing the selected item, for example via swipe or AnimateTo.

Parameters:

equation	The equation.

See also:

setAnimationSteps, **getAnimationSteps**

setHeight

virtual void setHeight (int16_t height)

Sets the height of this drawable.

Parameters:

height	The new height.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.Also sets the height of the children.

Reimplements: **touchgfx::Drawable::setHeight**

Reimplemented by: touchgfx::ScrollWheelWithSelectionStyle::setHeight

setHorizontal

virtual void setHorizontal (bool horizontal)

Sets a horizontal or vertical layout.

If parameter horizontal is set true, all drawables are arranged side by side. If horizontal is set false, the drawables are arranged above and below each other (vertically).

Parameters:

horizontal True to align drawables horizontal, false to align drawables vertically.

NOTE

Default value is false, i.e. vertical layout.

See also:

DrawableList::setHorizontal, getHorizontal

Reimplemented by: touchgfx::ScrollWheelWithSelectionStyle::setHorizontal

setItemPressedCallback

void setItemPressedCallback (GenericCallback< int16_t > & callback)

Set **Callback** which will be called when a item is pressed.

Parameters:

callback The callback.

setItemSelectedCallback

void setItemSelectedCallback (GenericCallback< int16_t > & callback)

Sets **Callback** which will be called when the selected item is clicked.

Parameters:

callback The callback.

setMaxSwipeItems

void setMaxSwipeItems (uint16_t maxItems)

Sets maximum swipe items.

Often useful when there are e.g. five visible items on the screen and a swipe action should at most swipe the next/previous five items into view to achieve sort of a paging effect.

Parameters:

maxItems The maximum items, 0 means "no limit" (which is also the default).

See also:

getMaxSwipeItems

setNumberOfItems

virtual void setNumberOfItems (int16_t numberOfItems)

Sets number of items in the **DrawableList**.

This forces all drawables to be updated to ensure that the content is correct. For example a date selector might switch number of days between 28, 29, 30, and 31 depending on the month. A circular list might show 27-28-29-30-31 and might need to update this to show 27-28-1-2-3.

Parameters:

numberOfItems Number of items.

NOTE

The DrawableList is refreshed to reflect the change.

Reimplemented by: **touchgfx::ScrollWheelWithSelectionStyle::setNumberOfItems**

setSwipeAcceleration

void setSwipeAcceleration (uint16_t acceleration)

Sets swipe acceleration (times 10).

Default value, if not set, is 10, i.e. 1.0.

Parameters:

acceleration The acceleration times 10, so "9" means "0.9" and "75" means "7.5".

NOTE

The reason for multiplying the acceleration by 10 is to avoid introducing floating point arithmetic.

See also:

getSwipeAcceleration

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most **Drawable** widgets, changing this does normally not automatically yield a redraw.Also sets the width of the children.

Reimplements: touchgfx::Drawable::setWidth

Reimplemented by: touchgfx::ScrollWheelWithSelectionStyle::setWidth

stopAnimation

void stopAnimation ( )

Stops an animation if one is ongoing.

Immediately moves to the item which is being animated to.

Protected Functions Documentation

animateToPosition

```
virtual void animateToPosition ( int32_t position ,
                                 int16_t steps =-1
                                 )
```

Animate to a new position/offset using the given number of steps.

Parameters:

position The new position.
steps (Optional) The steps.

Reimplemented by: **touchgfx::ScrollWheelBase::animateToPosition**

getNearestAlignedOffset

virtual int32_t getNearestAlignedOffset (int32_t offset)

Gets nearest offset aligned to a multiple of itemSize.

Parameters:

offset The offset.

Returns:

The nearest aligned offset.

Reimplemented by: **touchgfx::ScrollList::getNearestAlignedOffset**

getNormalizedOffset

int getNormalizedOffset (int offset)

Gets normalized offset from a given offset from 0 down to -numItems*itemSize.

Parameters:

offset The offset.

Returns:

The normalized offset.

getOffset

virtual int32_t getOffset ( ) const

Gets display offset of first item.

Returns:

The offset.

getPositionForItem

virtual int32_t getPositionForItem (int16_t itemIndex)

Get the position for an item.

The position should ensure that the item is in view as defined by the semantics of the actual scroll class. If the item is already in view, the current offset is returned and not the offset of the given item.

Parameters:

itemIndex Zero-based index of the item.

Returns:

The position for item.

Reimplemented by: **touchgfx::ScrollList::getPositionForItem**, **touchgfx::ScrollWheelBase::getPositionForItem**

keepOffsetInsideLimits

virtual int32_t keepOffsetInsideLimits (int32_t newOffset , const =0
int16_t overShoot const =0
) const =0

Keep offset inside limits.

Return the new offset that is inside the limits of the scroll list, with the overShoot value added at both ends of the list.

Parameters:

newOffset The new offset.
overShoot The over shoot.

Returns:

The new offset inside the limits.

Reimplemented by: **touchgfx::ScrollList::keepOffsetInsideLimits**, **touchgfx::ScrollWheelBase::keepOffsetInsideLimits**

setOffset

virtual void setOffset (int32_t offset)

Sets display offset of first item.

Parameters:

offset The offset.

Reimplemented by: touchgfx::ScrollWheelWithSelectionStyle::setOffset

Protected Attributes Documentation

animationEndedCallback

GenericCallback * animationEndedCallback

The animation ended callback.

currentAnimationState

AnimationState currentAnimationState

The current animation state.

defaultAnimationSteps

uint16_t defaultAnimationSteps

The animation steps.

distanceBeforeAlignedItem

int16_t distanceBeforeAlignedItem

The distance before aligned item.

dragAcceleration

uint16_t dragAcceleration

The drag acceleration x10.

draggableX

bool draggableX

Is the container draggable in the horizontal direction.

draggableY

bool draggableY

Is the container draggable in the vertical direction.

easingEquation

EasingEquation easingEquation

The easing equation used for animation.

gestureEnd

int gestureEnd

The gesture end.

gestureStart

int gestureStart

The gesture start.

gestureStep

int gestureStep

The current gesture step.

gestureStepsTotal

int gestureStepsTotal

The total gesture steps.

initialSwipeOffset

int32_t initialSwipeOffset

The initial swipe offset.

itemLockedInCallback

GenericCallback * itemLockedInCallback

The item locked in callback.

itemPressedCallback

GenericCallback< int16_t > * itemPressedCallback

The item pressed callback.

itemSelectedCallback

GenericCallback< int16_t > * itemSelectedCallback

The item selected callback.

itemSize

int16_t itemSize

Size of the item (including margin)

list

DrawableList list

The list.

maxSwipeItems

uint16_t maxSwipeItems

The maximum swipe items.

numberOfDrawables

int16_t numberOfDrawables

Number of drawables.

swipeAcceleration

uint16_t swipeAcceleration

The swipe acceleration x10.

xClick

int16_t xClick

The x coordinate of a click.

yClick

int16_t yClick

The y coordinate of a click.

Version: 4.16

ScrollList

A simple list of scrolling drawables. Since a long list of drawables only display a few of items at any one time, the drawables are re-used to preserve resources.

See: DrawableList

Inherits from: ScrollBase, Container, Drawable

Public Functions

int16_t	**getItem**(int16_t drawableIndex)
	Gets an item.
int16_t	**getPaddingAfter**() const
	Gets distance after last drawable in **ScrollList**.
int16_t	**getPaddingBefore**() const
	Gets distance before first drawable in **ScrollList**.
bool	**getSnapping**() const
	Gets the current snap stetting.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
	ScrollList()
virtual void	**setDrawables**(**DrawableListItemsInterface** & drawableListItems, **GenericCallback**< **DrawableListItemsInterface** *, int16_t, int16_t > & updateDrawableCallback)
	Setup a list of drawables and provide a function to call to update a given **Drawable** with new contents.
void	**setPadding**(int16_t paddingBefore, int16_t paddingAfter)
	Sets distance offset before and after the "visible" drawables in the **ScrollList**.
void	**setSnapping**(bool snap)
	Set snapping.
void	**setWindowSize**(int16_t items)

Sets window size, i.e.

Protected Functions

virtual int32_t **getNearestAlignedOffset**(int32_t offset) const

Gets nearest offset aligned to a multiple of itemSize.

virtual int32_t **getPositionForItem**(int16_t itemIndex)

Get the position for an item.

virtual int32_t **keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const

Keep offset inside limits.

Protected Attributes

int16_t **paddingAfterLastItem**

The distance after last item.

bool **snapping**

Is snapping enabled?

int **windowSize**

Number of items that should always be visible.

Additional inherited members

Protected Types inherited from ScrollBase

enum **AnimationState** { NO_ANIMATION, ANIMATING_GESTURE, ANIMATING_DRAG }

Values that represent animation states.

Public Functions inherited from ScrollBase

void **allowHorizontalDrag**(bool enable)

	Enables horizontal scrolling to be passed to the children in the list (in case a child widget is able to handle drag events).
void	**allowVerticalDrag**(bool enable)
	Enables the vertical scroll.
virtual void	**animateToItem**(int16_t itemIndex, int16_t animationSteps =-1)
	Go to a specific item, possibly with animation.
uint16_t	**getAnimationSteps**() const
	Gets animation steps as set in setAnimationSteps.
virtual bool	**getCircular**() const
	Gets the circular setting, previously set using setCircular().
uint16_t	**getDragAcceleration**() const
	Gets drag acceleration (times 10).
virtual int16_t	**getDrawableMargin**() const
	Gets drawable margin as set through the second parameter in most recent call to setDrawableSize().
virtual int16_t	**getDrawableSize**() const
	Gets drawable size as set through the first parameter in most recent call to setDrawableSize().
virtual bool	**getHorizontal**() const
	Gets the orientation of the drawables, previously set using setHorizontal().
uint16_t	**getMaxSwipeItems**() const
	Gets maximum swipe items as set by setMaxSwipeItems.
virtual int16_t	**getNumberOfItems**() const
	Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().
uint16_t	**getSwipeAcceleration**() const
	Gets swipe acceleration (times 10).
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.

virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**initialize**()
	Removed all drawables and initializes the content of these items.
bool	**isAnimating**() const
	Query if an animation is ongoing.
virtual void	**itemChanged**(int itemIndex)
	Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.
	ScrollBase()
void	**setAnimationEndedCallback**(**GenericCallback**<> & callback)
	Callback, called when the set animation ended.
void	**setAnimationSteps**(int16_t steps)
	Sets animation steps (in ticks) when moving to a new selected item.
virtual void	**setCircular**(bool circular)
	Sets whether the list is circular (infinite) or not.
void	**setDragAcceleration**(uint16_t acceleration)
	Sets drag acceleration times 10, so "10" means "1", "15" means "1.5".
void	**setDrawableSize**(int16_t drawableSize, int16_t drawableMargin)
	Sets drawables size.
void	**setEasingEquation**(**EasingEquation** equation)
	Sets easing equation when changing the selected item, for example via swipe or AnimateTo.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setHorizontal**(bool horizontal)
	Sets a horizontal or vertical layout.
void	**setItemPressedCallback**(**GenericCallback**< int16_t > & callback)
	Set **Callback** which will be called when a item is pressed.

void	**setItemSelectedCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the selected item is clicked.
void	**setMaxSwipeItems**(uint16_t maxItems)
	Sets maximum swipe items.
virtual void	**setNumberOfItems**(int16_t numberOfItems)
	Sets number of items in the **DrawableList**.
void	**setSwipeAcceleration**(uint16_t acceleration)
	Sets swipe acceleration (times 10).
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**stopAnimation**()
	Stops an animation if one is ongoing.

Protected Functions inherited from ScrollBase

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.
int	**getNormalizedOffset**(int offset) const
	Gets normalized offset from a given offset from 0 down to -numItems*itemSize.
virtual int32_t	**getOffset**() const
	Gets display offset of first item.
virtual void	**setOffset**(int32_t offset)
	Sets display offset of first item.

Protected Attributes inherited from ScrollBase

GenericCallback *	**animationEndedCallback**
	The animation ended callback.
AnimationState	**currentAnimationState**
	The current animation state.
uint16_t	**defaultAnimationSteps**

	The animation steps.
int16_t	**distanceBeforeAlignedItem**
	The distance before aligned item.
uint16_t	**dragAcceleration**
	The drag acceleration x10.
bool	**draggableX**
	Is the container draggable in the horizontal direction.
bool	**draggableY**
	Is the container draggable in the vertical direction.
EasingEquation	**easingEquation**
	The easing equation used for animation.
int	**gestureEnd**
	The gesture end.
int	**gestureStart**
	The gesture start.
int	**gestureStep**
	The current gesture step.
int	**gestureStepsTotal**
	The total gesture steps.
int32_t	**initialSwipeOffset**
	The initial swipe offset.
GenericCallback *	**itemLockedInCallback**
	The item locked in callback.
GenericCallback< int16_t > *	**itemPressedCallback**
	The item pressed callback.
GenericCallback< int16_t > *	**itemSelectedCallback**
	The item selected callback.
int16_t	**itemSize**
	Size of the item (including margin)

DrawableList	**list**
	The list.
uint16_t	**maxSwipeItems**
	The maximum swipe items.
int16_t	**numberOfDrawables**
	Number of drawables.
uint16_t	**swipeAcceleration**
	The swipe acceleration x10.
int16_t	**xClick**
	The x coordinate of a click.
int16_t	**yClick**
	The y coordinate of a click.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const

	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)

	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**

	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getItem

int16_t getItem (int16_t drawableIndex)

Gets an item.

Parameters:

drawableIndex Zero-based index of the drawable.

Returns:

The item.

getPaddingAfter

int16_t getPaddingAfter ( ) const

Gets distance after last drawable in **ScrollList**.

Returns:

The distance after the last drawable in the **ScrollList**.

See also:

setPadding, **getPaddingBefore**

getPaddingBefore

int16_t getPaddingBefore ( ) const

Gets distance before first drawable in **ScrollList**.

Returns:

The distance before.

See also:

setPadding, **getPaddingAfter**

getSnapping

bool getSnapping ( ) const

Gets the current snap stetting.

Returns:

true if snapping is set, false otherwise.

See also:

setSnapping

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

ScrollList

ScrollList ( )

setDrawables

```
virtual void setDrawables ( DrawableListItemsInterface &                    drawableListItems ,
                            GenericCallback< DrawableListItemsInterface
                            *, int16_t, int16_t > &                         updateDrawableCallback
                            )
```

Setup a list of drawables and provide a function to call to update a given **Drawable** with new contents.

Parameters:

drawableListItems	The drawables allocated.
updateDrawableCallback	A callback to update the contents of a specific drawable with a specific item.

See also:

DrawableList::setDrawables

setPadding

```
void setPadding ( int16_t paddingBefore ,
                  int16_t paddingAfter
                  )
```

Sets distance offset before and after the "visible" drawables in the **ScrollList**.

This allows the actual area where widgets are placed to have a little extra area where parts of drawables can be seen. For example if the **ScrollList** is 200, each drawable is 50 and distance before and distance after are 25, then there is room for three visible drawables inside the **ScrollList**. When scrolling, part of the scrolled out drawables can be seen before and after the three drawables. In this case 25/50 = 50% of a drawable can be seen before and after the three drawables in the **ScrollList**.

Parameters:

paddingBefore	The distance before the first drawable in the **ScrollList**.
paddingAfter	The distance after the last drawable in the **ScrollList**.

See also:

getPaddingBefore, **getPaddingAfter**

setSnapping

void setSnapping (bool snap)

Set snapping.

If snapping is false, the items can flow freely. If snapping is true, the items will snap into place so an item is always in the "selected" spot.

Parameters:

snap true to snap.

See also:

getSnapping

setWindowSize

void setWindowSize (int16_t items)

Sets window size, i.e.

the number of items that should always be visible. The default value is 1. If three items are visible on the display and window size is set to three, no part of the screen will be blank (unless the list contains less than three items and the list is not circular).

Parameters:

items The number of items that should always be visible.

NOTE

This only applies to non-circular lists.

Protected Functions Documentation

getNearestAlignedOffset

virtual int32_t getNearestAlignedOffset (int32_t offset)

Gets nearest offset aligned to a multiple of itemSize.

Parameters:

offset The offset.

Returns:

The nearest aligned offset.

Reimplements: touchgfx::ScrollBase::getNearestAlignedOffset

getPositionForItem

virtual int32_t getPositionForItem (int16_t itemIndex)

Get the position for an item.

The position should ensure that the item is in view as defined by the semantics of the actual scroll class. If the item is already in view, the current offset is returned and not the offset of the given item.

Parameters:

itemIndex Zero-based index of the item.

Returns:

The position for item.

Reimplements: touchgfx::ScrollBase::getPositionForItem

keepOffsetInsideLimits

virtual int32_t keepOffsetInsideLimits (int32_t newOffset , const int16_t overShoot const) const

Keep offset inside limits.

Return the new offset that is inside the limits of the scroll list, with the overShoot value added at both ends of the list.

Parameters:

newOffset The new offset.
overShoot The over shoot.

Returns:

The new offset inside the limits.

Reimplements: touchgfx::ScrollBase::keepOffsetInsideLimits

Protected Attributes Documentation

paddingAfterLastItem

int16_t paddingAfterLastItem

The distance after last item.

snapping

bool snapping

Is snapping enabled?

windowSize

int windowSize

Number of items that should always be visible.

Version: 4.16

ScrollWheel

A scroll wheel is very much like the digit selector on a padlock with numbers. The digits always snap into place and exactly one number is always the "selected" number. Thus, a scroll wheel is a list of identically styled drawables which can be scrolled through. One of the items in the list is the "selected" one, and scrolling through the list can be done in various ways. The ScrollWheel uses the DrawableList to make it possible to handle a huge number of items using only a limited number of drawables by reusing drawables that are no longer in view.

See: ScrollWheelBase, DrawableList, ScrollWheelWithSelectionStyle

Inherits from: ScrollWheelBase, ScrollBase, Container, Drawable

Public Functions

virtual void	**setDrawables**(**DrawableListItemsInterface** & drawableListItems, **GenericCallback**< **DrawableListItemsInterface** *, int16_t, int16_t > & updateDrawableCallback)
	Sets the drawables used by the scroll wheel.

Additional inherited members

Public Functions inherited from ScrollWheelBase

int	**getSelectedItem**() const
	Gets selected item.
virtual int16_t	**getSelectedItemOffset**() const
	Gets offset of selected item measured in pixels relative to the start of the widget.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.

virtual int32_t	**keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const
	Keep offset inside limits.
	ScrollWheelBase()
void	**setAnimateToCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the **ScrollWheel** animates to a new item.
virtual void	**setSelectedItemOffset**(int16_t offset)
	Sets selected item offset, measured in pixels, from the edge of the widget.

Protected Functions inherited from ScrollWheelBase

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.
virtual int32_t	**getPositionForItem**(int16_t itemIndex)
	Get the position for an item.

Protected Attributes inherited from ScrollWheelBase

GenericCallback< int16_t > *	**animateToCallback**
	The animate to callback.

Protected Types inherited from ScrollBase

enum	**AnimationState** { NO_ANIMATION, ANIMATING_GESTURE, ANIMATING_DRAG }
	Values that represent animation states.

Public Functions inherited from ScrollBase

void	**allowHorizontalDrag**(bool enable)
	Enables horizontal scrolling to be passed to the children in the list (in case a child widget is able to handle drag events).
void	**allowVerticalDrag**(bool enable)
	Enables the vertical scroll.

virtual void	**animateToItem**(int16_t itemIndex, int16_t animationSteps =-1)
	Go to a specific item, possibly with animation.
uint16_t	**getAnimationSteps**() const
	Gets animation steps as set in setAnimationSteps.
virtual bool	**getCircular**() const
	Gets the circular setting, previously set using setCircular().
uint16_t	**getDragAcceleration**() const
	Gets drag acceleration (times 10).
virtual int16_t	**getDrawableMargin**() const
	Gets drawable margin as set through the second parameter in most recent call to setDrawableSize().
virtual int16_t	**getDrawableSize**() const
	Gets drawable size as set through the first parameter in most recent call to setDrawableSize().
virtual bool	**getHorizontal**() const
	Gets the orientation of the drawables, previously set using setHorizontal().
uint16_t	**getMaxSwipeItems**() const
	Gets maximum swipe items as set by setMaxSwipeItems.
virtual int16_t	**getNumberOfItems**() const
	Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().
uint16_t	**getSwipeAcceleration**() const
	Gets swipe acceleration (times 10).
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**initialize**()

	Removed all drawables and initializes the content of these items.
bool	**isAnimating**() const
	Query if an animation is ongoing.
virtual void	**itemChanged**(int itemIndex)
	Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.
	ScrollBase()
void	**setAnimationEndedCallback**(**GenericCallback**<> & callback)
	Callback, called when the set animation ended.
void	**setAnimationSteps**(int16_t steps)
	Sets animation steps (in ticks) when moving to a new selected item.
virtual void	**setCircular**(bool circular)
	Sets whether the list is circular (infinite) or not.
void	**setDragAcceleration**(uint16_t acceleration)
	Sets drag acceleration times 10, so "10" means "1", "15" means "1.5".
void	**setDrawableSize**(int16_t drawableSize, int16_t drawableMargin)
	Sets drawables size.
void	**setEasingEquation**(**EasingEquation** equation)
	Sets easing equation when changing the selected item, for example via swipe or AnimateTo.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setHorizontal**(bool horizontal)
	Sets a horizontal or vertical layout.
void	**setItemPressedCallback**(**GenericCallback**< int16_t > & callback)
	Set **Callback** which will be called when a item is pressed.
void	**setItemSelectedCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the selected item is clicked.
void	**setMaxSwipeItems**(uint16_t maxItems)

	Sets maximum swipe items.
virtual void	**setNumberOfItems**(int16_t numberOfItems)
	Sets number of items in the **DrawableList**.
void	**setSwipeAcceleration**(uint16_t acceleration)
	Sets swipe acceleration (times 10).
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**stopAnimation**()
	Stops an animation if one is ongoing.

Protected Functions inherited from ScrollBase

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.
virtual int32_t	**getNearestAlignedOffset**(int32_t offset) const
	Gets nearest offset aligned to a multiple of itemSize.
int	**getNormalizedOffset**(int offset) const
	Gets normalized offset from a given offset from 0 down to -numItems*itemSize.
virtual int32_t	**getOffset**() const
	Gets display offset of first item.
virtual int32_t	**getPositionForItem**(int16_t itemIndex) =0
	Get the position for an item.
virtual int32_t	**keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const =0
	Keep offset inside limits.
virtual void	**setOffset**(int32_t offset)
	Sets display offset of first item.

Protected Attributes inherited from ScrollBase

GenericCallback *	**animationEndedCallback**
	The animation ended callback.

AnimationState	**currentAnimationState**
	The current animation state.
uint16_t	**defaultAnimationSteps**
	The animation steps.
int16_t	**distanceBeforeAlignedItem**
	The distance before aligned item.
uint16_t	**dragAcceleration**
	The drag acceleration x10.
bool	**draggableX**
	Is the container draggable in the horizontal direction.
bool	**draggableY**
	Is the container draggable in the vertical direction.
EasingEquation	**easingEquation**
	The easing equation used for animation.
int	**gestureEnd**
	The gesture end.
int	**gestureStart**
	The gesture start.
int	**gestureStep**
	The current gesture step.
int	**gestureStepsTotal**
	The total gesture steps.
int32_t	**initialSwipeOffset**
	The initial swipe offset.
GenericCallback *	**itemLockedInCallback**
	The item locked in callback.
GenericCallback< int16_t > *	**itemPressedCallback**
	The item pressed callback.
GenericCallback< int16_t > *	**itemSelectedCallback**

	The item selected callback.
int16_t	**itemSize**
	Size of the item (including margin)
DrawableList	**list**
	The list.
uint16_t	**maxSwipeItems**
	The maximum swipe items.
int16_t	**numberOfDrawables**
	Number of drawables.
uint16_t	**swipeAcceleration**
	The swipe acceleration x10.
int16_t	**xClick**
	The x coordinate of a click.
int16_t	**yClick**
	The y coordinate of a click.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.

virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const

	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void **setXY**(int16_t x, int16_t y)

Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void **setY**(int16_t y)

Sets the y coordinate of this **Drawable**, relative to its parent.

virtual void **translateRectToAbsolute**(**Rect** & r) const

	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

setDrawables

virtual void setDrawables (DrawableListItemsInterface & drawableListItems ,
GenericCallback< DrawableListItemsInterface *, int16_t, int16_t > & updateDrawableCallback
)

Sets the drawables used by the scroll wheel.

The drawables are updated through a callback will put the right data in the drawable.

Parameters:

drawableListItems	Number of drawables.
updateDrawableCallback	The update drawable callback.

Version: 4.16

ScrollWheelBase

A base class for a scroll wheel. A scroll wheel is very much like the digit selector on a padlock with numbers. The digits always snap into place and exactly one number is always the "selected" number. Using ScrollWheel, all elements look the same, but an underlying bitmap can help emphasize the "selected" element. The ScrollWheelWithSelectionStyle can have a completely different style on the "selected" item - the font can be larger or bold and images can change color - this can help to give a kind of 3D effect using very few resources.

See: ScrollWheel, ScrollWheelWithSelectionStyle

Inherits from: ScrollBase, Container, Drawable

Inherited by: ScrollWheel, ScrollWheelWithSelectionStyle

Public Functions

int	**getSelectedItem**() const
	Gets selected item.
virtual int16_t	**getSelectedItemOffset**() const
	Gets offset of selected item measured in pixels relative to the start of the widget.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual int32_t	**keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const
	Keep offset inside limits.
	ScrollWheelBase()
void	**setAnimateToCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the **ScrollWheel** animates to a new item.

virtual void	**setSelectedItemOffset**(int16_t offset)
	Sets selected item offset, measured in pixels, from the edge of the widget.

Protected Functions

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.

virtual int32_t	**getPositionForItem**(int16_t itemIndex)
	Get the position for an item.

Protected Attributes

GenericCallback< int16_t > *	**animateToCallback**
	The animate to callback.

Additional inherited members

Protected Types inherited from ScrollBase

enum	**AnimationState** { NO_ANIMATION, ANIMATING_GESTURE, ANIMATING_DRAG }
	Values that represent animation states.

Public Functions inherited from ScrollBase

void	**allowHorizontalDrag**(bool enable)
	Enables horizontal scrolling to be passed to the children in the list (in case a child widget is able to handle drag events).

void	**allowVerticalDrag**(bool enable)
	Enables the vertical scroll.

virtual void	**animateToItem**(int16_t itemIndex, int16_t animationSteps =-1)
	Go to a specific item, possibly with animation.

uint16_t	**getAnimationSteps**() const
	Gets animation steps as set in setAnimationSteps.
virtual bool	**getCircular**() const
	Gets the circular setting, previously set using setCircular().
uint16_t	**getDragAcceleration**() const
	Gets drag acceleration (times 10).
virtual int16_t	**getDrawableMargin**() const
	Gets drawable margin as set through the second parameter in most recent call to setDrawableSize().
virtual int16_t	**getDrawableSize**() const
	Gets drawable size as set through the first parameter in most recent call to setDrawableSize().
virtual bool	**getHorizontal**() const
	Gets the orientation of the drawables, previously set using setHorizontal().
uint16_t	**getMaxSwipeItems**() const
	Gets maximum swipe items as set by setMaxSwipeItems.
virtual int16_t	**getNumberOfItems**() const
	Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().
uint16_t	**getSwipeAcceleration**() const
	Gets swipe acceleration (times 10).
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**initialize**()
	Removed all drawables and initializes the content of these items.
bool	**isAnimating**() const
	Query if an animation is ongoing.
virtual void	**itemChanged**(int itemIndex)
	Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.

ScrollBase()

void **setAnimationEndedCallback**(**GenericCallback**<> & callback)

Callback, called when the set animation ended.

void **setAnimationSteps**(int16_t steps)

Sets animation steps (in ticks) when moving to a new selected item.

virtual void **setCircular**(bool circular)

Sets whether the list is circular (infinite) or not.

void **setDragAcceleration**(uint16_t acceleration)

Sets drag acceleration times 10, so "10" means "1", "15" means "1.5".

void **setDrawableSize**(int16_t drawableSize, int16_t drawableMargin)

Sets drawables size.

void **setEasingEquation**(**EasingEquation** equation)

Sets easing equation when changing the selected item, for example via swipe or AnimateTo.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

virtual void **setHorizontal**(bool horizontal)

Sets a horizontal or vertical layout.

void **setItemPressedCallback**(**GenericCallback**< int16_t > & callback)

Set **Callback** which will be called when a item is pressed.

void **setItemSelectedCallback**(**GenericCallback**< int16_t > & callback)

Sets **Callback** which will be called when the selected item is clicked.

void **setMaxSwipeItems**(uint16_t maxItems)

Sets maximum swipe items.

virtual void **setNumberOfItems**(int16_t numberOfItems)

Sets number of items in the **DrawableList**.

void **setSwipeAcceleration**(uint16_t acceleration)

Sets swipe acceleration (times 10).

virtual void **setWidth**(int16_t width)

	Sets the width of this drawable.
void	**stopAnimation**()
	Stops an animation if one is ongoing.

Protected Functions inherited from ScrollBase

virtual int32_t	**getNearestAlignedOffset**(int32_t offset) const
	Gets nearest offset aligned to a multiple of itemSize.
int	**getNormalizedOffset**(int offset) const
	Gets normalized offset from a given offset from 0 down to -numItems*itemSize.
virtual int32_t	**getOffset**() const
	Gets display offset of first item.
virtual void	**setOffset**(int32_t offset)
	Sets display offset of first item.

Protected Attributes inherited from ScrollBase

GenericCallback *	**animationEndedCallback**
	The animation ended callback.
AnimationState	**currentAnimationState**
	The current animation state.
uint16_t	**defaultAnimationSteps**
	The animation steps.
int16_t	**distanceBeforeAlignedItem**
	The distance before aligned item.
uint16_t	**dragAcceleration**
	The drag acceleration x10.
bool	**draggableX**
	Is the container draggable in the horizontal direction.
bool	**draggableY**
	Is the container draggable in the vertical direction.

EasingEquation	**easingEquation**
	The easing equation used for animation.
int	**gestureEnd**
	The gesture end.
int	**gestureStart**
	The gesture start.
int	**gestureStep**
	The current gesture step.
int	**gestureStepsTotal**
	The total gesture steps.
int32_t	**initialSwipeOffset**
	The initial swipe offset.
GenericCallback *	**itemLockedInCallback**
	The item locked in callback.
GenericCallback< int16_t > *	**itemPressedCallback**
	The item pressed callback.
GenericCallback< int16_t > *	**itemSelectedCallback**
	The item selected callback.
int16_t	**itemSize**
	Size of the item (including margin)
DrawableList	**list**
	The list.
uint16_t	**maxSwipeItems**
	The maximum swipe items.
int16_t	**numberOfDrawables**
	Number of drawables.
uint16_t	**swipeAcceleration**
	The swipe acceleration x10.
int16_t	**xClick**

	The x coordinate of a click.
int16_t	**yClick**
	The y coordinate of a click.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()

	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const

Gets whether this **Drawable** receives touch events or not.

bool **isVisible**() const

Gets whether this **Drawable** is visible.

virtual void **moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getSelectedItem

int getSelectedItem ( ) const

Gets selected item.

If an animation is in progress, the item that is being scrolled to is returned, not the item that happens to be flying by at the time.

Returns:

The selected item.

getSelectedItemOffset

virtual int16_t getSelectedItemOffset ( ) const

Gets offset of selected item measured in pixels relative to the start of the widget.

Returns:

The selected item offset.

See also:

setSelectedItemOffset

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::ScrollBase::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Defines the event handler interface for GestureEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **GestureEvent** received from the **HAL**.

Reimplements: **touchgfx::ScrollBase::handleGestureEvent**

keepOffsetInsideLimits

virtual int32_t keepOffsetInsideLimits (int32_t newOffset , const int16_t overShoot const) const

Keep offset inside limits.

Return the new offset that is inside the limits of the scroll list, with the overShoot value added at both ends of the list.

Parameters:

newOffset The new offset.
overShoot The over shoot.

Returns:

The new offset inside the limits.

Reimplements: **touchgfx::ScrollBase::keepOffsetInsideLimits**

ScrollWheelBase

ScrollWheelBase ( )

setAnimateToCallback

void setAnimateToCallback (GenericCallback< int16_t > & callback)

Sets **Callback** which will be called when the **ScrollWheel** animates to a new item.

Parameters:

callback The callback.

setSelectedItemOffset

virtual void setSelectedItemOffset (int16_t offset)

Sets selected item offset, measured in pixels, from the edge of the widget.

The offset is the relative x coordinate if the **ScrollWheel** is horizontal, otherwise it is the relative y coordinate. If this value is zero, the selected item is placed at the very start of the widget.

Parameters:

offset The offset.

See also:

getSelectedItemOffset

Reimplemented by: **touchgfx::ScrollWheelWithSelectionStyle::setSelectedItemOffset**

Protected Functions Documentation

animateToPosition

virtual void animateToPosition (int32_t position ,
int16_t steps =-1
)

Animate to a new position/offset using the given number of steps.

Parameters:

position The new position.
steps (Optional) The steps.

Reimplements: touchgfx::ScrollBase::animateToPosition

getPositionForItem

virtual int32_t getPositionForItem (int16_t itemIndex)

Get the position for an item.

The position should ensure that the item is in view as defined by the semantics of the actual scroll class. If the item is already in view, the current offset is returned and not the offset of the given item.

Parameters:

itemIndex Zero-based index of the item.

Returns:

The position for item.

Reimplements: touchgfx::ScrollBase::getPositionForItem

Protected Attributes Documentation

animateToCallback

GenericCallback< int16_t > * animateToCallback

The animate to callback.

Version: 4.16

ScrollWheelWithSelectionStyle

A scroll wheel is very much like the digit selector on a padlock with numbers. The digits always snap into place and exactly one number is always the "selected" number. Similar to an ordinary ScrollWheel, but with a different style for the selected item which can thus be bold, have a different color or similar effect to highlight it and help create a 3D effect using very few resources.

See: DrawableList, ScrollWheel

Inherits from: ScrollWheelBase, ScrollBase, Container, Drawable

Public Functions

virtual int16_t	**getSelectedItemExtraSizeAfter**() const
	Gets selected item extra size after.
virtual int16_t	**getSelectedItemExtraSizeBefore**() const
	Gets selected item extra size before.
virtual int16_t	**getSelectedItemMarginAfter**() const
	Gets selected item margin after.
virtual int16_t	**getSelectedItemMarginBefore**() const
	Gets selected item margin before.
virtual void	**initialize**()
	Removed all drawables and initializes the content of these items.
virtual void	**itemChanged**(int itemIndex)
	Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.
	ScrollWheelWithSelectionStyle()
virtual void	**setCircular**(bool circular)
	Sets whether the list is circular (infinite) or not.
virtual void	**setDrawables**(**DrawableListItemsInterface** & drawableListItems, **GenericCallback**< **DrawableListItemsInterface** , *int16_t, int16_t* > & *updateDrawableCallback*, ***DrawableListItemsInterface*** & *centerDrawableListItems*, ***GenericCallback***< ***DrawableListItemsInterface*** , int16_t, int16_t > & updateCenterDrawableCallback)
	Setups the widget.
virtual void	**setDrawableSize**(int16_t drawableSize, int16_t drawableMargin)

	Sets drawables size.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setHorizontal**(bool horizontal)
	Sets a horizontal or vertical layout.
virtual void	**setNumberOfItems**(int16_t numberOfItems)
	Sets number of items in the **DrawableList**.
virtual void	**setSelectedItemExtraSize**(int16_t extraSizeBefore, int16_t extraSizeAfter)
	Sets selected item extra size to make the size of the area for the center drawables larger.
virtual void	**setSelectedItemMargin**(int16_t marginBefore, int16_t marginAfter)
	Sets margin around selected item.
virtual void	**setSelectedItemOffset**(int16_t offset)
	Sets selected item offset, measured in pixels, from the edge of the widget.
virtual void	**setSelectedItemPosition**(int16_t offset, int16_t extraSizeBefore, int16_t extraSizeAfter, int16_t marginBefore, int16_t marginAfter)
	Sets the selected item offset.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.

Protected Functions

void	**refreshDrawableListsLayout**()
	Refresh drawable lists layout.
virtual void	**setOffset**(int32_t offset)
	Sets display offset of first item.

Protected Attributes

DrawableListItemsInterface *	**centerDrawables**
	The drawables at the center of the scroll wheel.
DrawableListItemsInterface *	**drawables**
	The drawables at the beginning and end of the scroll wheel.

int16_t	**drawablesInFirstList**
	List of drawables in firsts.
int16_t	**extraSizeAfterSelectedItem**
	The distance after selected item.
int16_t	**extraSizeBeforeSelectedItem**
	The distance before selected item.
DrawableList	**list1**
	The center list.
DrawableList	**list2**
	The list for items not in the center.
int16_t	**marginAfterSelectedItem**
	The distance after selected item.
int16_t	**marginBeforeSelectedItem**
	The distance before selected item.
GenericCallback< **DrawableListItemsInterface** , *int16_t, int16_t* >	**originalUpdateCenterDrawableCallback**
	The original update center drawable callback.
GenericCallback< **DrawableListItemsInterface** , *int16_t, int16_t* >	**originalUpdateDrawableCallback**
	The original update drawable callback.

Additional inherited members

Public Functions inherited from ScrollWheelBase

int	**getSelectedItem**() const
	Gets selected item.
virtual int16_t	**getSelectedItemOffset**() const
	Gets offset of selected item measured in pixels relative to the start of the widget.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)

	Defines the event handler interface for GestureEvents.
virtual int32_t	**keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const
	Keep offset inside limits.
	ScrollWheelBase()
void	**setAnimateToCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the **ScrollWheel** animates to a new item.

Protected Functions inherited from ScrollWheelBase

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.
virtual int32_t	**getPositionForItem**(int16_t itemIndex)
	Get the position for an item.

Protected Attributes inherited from ScrollWheelBase

GenericCallback< int16_t > *	**animateToCallback**
	The animate to callback.

Protected Types inherited from ScrollBase

enum	**AnimationState** { NO_ANIMATION, ANIMATING_GESTURE, ANIMATING_DRAG }
	Values that represent animation states.

Public Functions inherited from ScrollBase

void	**allowHorizontalDrag**(bool enable)
	Enables horizontal scrolling to be passed to the children in the list (in case a child widget is able to handle drag events).
void	**allowVerticalDrag**(bool enable)
	Enables the vertical scroll.
virtual void	**animateToItem**(int16_t itemIndex, int16_t animationSteps =-1)
	Go to a specific item, possibly with animation.
uint16_t	**getAnimationSteps**() const
	Gets animation steps as set in setAnimationSteps.

virtual bool	**getCircular**() const
	Gets the circular setting, previously set using setCircular().
uint16_t	**getDragAcceleration**() const
	Gets drag acceleration (times 10).
virtual int16_t	**getDrawableMargin**() const
	Gets drawable margin as set through the second parameter in most recent call to setDrawableSize().
virtual int16_t	**getDrawableSize**() const
	Gets drawable size as set through the first parameter in most recent call to setDrawableSize().
virtual bool	**getHorizontal**() const
	Gets the orientation of the drawables, previously set using setHorizontal().
uint16_t	**getMaxSwipeItems**() const
	Gets maximum swipe items as set by setMaxSwipeItems.
virtual int16_t	**getNumberOfItems**() const
	Gets number of items in the **DrawableList**, as previously set using setNumberOfItems().
uint16_t	**getSwipeAcceleration**() const
	Gets swipe acceleration (times 10).
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
bool	**isAnimating**() const
	Query if an animation is ongoing.
	ScrollBase()
void	**setAnimationEndedCallback**(**GenericCallback**<> & callback)
	Callback, called when the set animation ended.
void	**setAnimationSteps**(int16_t steps)
	Sets animation steps (in ticks) when moving to a new selected item.
void	**setDragAcceleration**(uint16_t acceleration)
	Sets drag acceleration times 10, so "10" means "1", "15" means "1.5".

void	**setEasingEquation**(**EasingEquation** equation)
	Sets easing equation when changing the selected item, for example via swipe or AnimateTo.
void	**setItemPressedCallback**(**GenericCallback**< int16_t > & callback)
	Set **Callback** which will be called when a item is pressed.
void	**setItemSelectedCallback**(**GenericCallback**< int16_t > & callback)
	Sets **Callback** which will be called when the selected item is clicked.
void	**setMaxSwipeItems**(uint16_t maxItems)
	Sets maximum swipe items.
void	**setSwipeAcceleration**(uint16_t acceleration)
	Sets swipe acceleration (times 10).
void	**stopAnimation**()
	Stops an animation if one is ongoing.

Protected Functions inherited from ScrollBase

virtual void	**animateToPosition**(int32_t position, int16_t steps =-1)
	Animate to a new position/offset using the given number of steps.
virtual int32_t	**getNearestAlignedOffset**(int32_t offset) const
	Gets nearest offset aligned to a multiple of itemSize.
int	**getNormalizedOffset**(int offset) const
	Gets normalized offset from a given offset from 0 down to -numItems*itemSize.
virtual int32_t	**getOffset**() const
	Gets display offset of first item.
virtual int32_t	**getPositionForItem**(int16_t itemIndex) =0
	Get the position for an item.
virtual int32_t	**keepOffsetInsideLimits**(int32_t newOffset, int16_t overShoot) const =0
	Keep offset inside limits.

Protected Attributes inherited from ScrollBase

GenericCallback *	**animationEndedCallback**
	The animation ended callback.
AnimationState	**currentAnimationState**

	The current animation state.
uint16_t	**defaultAnimationSteps**
	The animation steps.
int16_t	**distanceBeforeAlignedItem**
	The distance before aligned item.
uint16_t	**dragAcceleration**
	The drag acceleration x10.
bool	**draggableX**
	Is the container draggable in the horizontal direction.
bool	**draggableY**
	Is the container draggable in the vertical direction.
EasingEquation	**easingEquation**
	The easing equation used for animation.
int	**gestureEnd**
	The gesture end.
int	**gestureStart**
	The gesture start.
int	**gestureStep**
	The current gesture step.
int	**gestureStepsTotal**
	The total gesture steps.
int32_t	**initialSwipeOffset**
	The initial swipe offset.
GenericCallback *	**itemLockedInCallback**
	The item locked in callback.
GenericCallback< int16_t > *	**itemPressedCallback**
	The item pressed callback.
GenericCallback< int16_t > *	**itemSelectedCallback**
	The item selected callback.
int16_t	**itemSize**
	Size of the item (including margin)
DrawableList	**list**

	The list.
uint16_t	**maxSwipeItems**
	The maximum swipe items.
int16_t	**numberOfDrawables**
	Number of drawables.
uint16_t	**swipeAcceleration**
	The swipe acceleration x10.
int16_t	**xClick**
	The x coordinate of a click.
int16_t	**yClick**
	The y coordinate of a click.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)

	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.

int16_t **getHeight**() const

Gets the height of this **Drawable**.

virtual void **getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

Function for obtaining the the last child of this drawable that intersects with the specified point.

Drawable * **getNextSibling**()

Gets the next sibling node.

Drawable * **getParent**() const

Returns the parent node.

const **Rect** & **getRect**() const

Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.

virtual **Rect** **getSolidRect**() const =0

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

virtual **Rect** **getSolidRectAbsolute**()

Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.

virtual void **getVisibleRect**(**Rect** & rect) const

Function for finding the visible part of this drawable.

int16_t **getWidth**() const

Gets the width of this **Drawable**.

int16_t **getX**() const

Gets the x coordinate of this **Drawable**, relative to its parent.

int16_t **getY**() const

Gets the y coordinate of this **Drawable**, relative to its parent.

virtual void **handleClickEvent**(const **ClickEvent** & evt)

Defines the event handler interface for ClickEvents.

virtual void **handleDragEvent**(const **DragEvent** & evt)

Defines the event handler interface for DragEvents.

virtual void **handleGestureEvent**(const **GestureEvent** & evt)

Defines the event handler interface for GestureEvents.

virtual void **handleTickEvent**()

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getSelectedItemExtraSizeAfter

virtual int16_t getSelectedItemExtraSizeAfter ( ) const

Gets selected item extra size after.

Returns:

The selected item extra size after.

See also:

setSelectedItemExtraSize

getSelectedItemExtraSizeBefore

virtual int16_t getSelectedItemExtraSizeBefore ( ) const

Gets selected item extra size before.

Returns:

The selected item extra size before.

See also:

setSelectedItemExtraSize

getSelectedItemMarginAfter

virtual int16_t getSelectedItemMarginAfter ( ) const

Gets selected item margin after.

Returns:

The selected item margin after.

See also:

setSelectedItemMargin

getSelectedItemMarginBefore

virtual int16_t getSelectedItemMarginBefore ( ) const

Gets selected item margin before.

Returns:

The selected item margin before.

See also:

setSelectedItemMargin

initialize

virtual void initialize ( )

Removed all drawables and initializes the content of these items.

Reimplements: **touchgfx::ScrollBase::initialize**

itemChanged

virtual void itemChanged (int itemIndex)

Inform the scroll list that the contents of an item has changed and force all drawables with the given item index to be updated via the callback provided.

This is important as a circular list with very few items might display the same item more than once and all these items should be updated.

Parameters:

itemIndex Zero-based index of the changed item.

Reimplements: **touchgfx::ScrollBase::itemChanged**

ScrollWheelWithSelectionStyle

ScrollWheelWithSelectionStyle ( )

setCircular

virtual void setCircular (bool circular)

Sets whether the list is circular (infinite) or not.

A circular list is a list where the first drawable re-appears after the last item in the list - and the last item in the list appears before the first item in the list.

Parameters:

circular True if the list should be circular, false if the list should not be circular.

See also:

DrawableList::setCircular, **getCircular**

Reimplements: **touchgfx::ScrollBase::setCircular**

setDrawables

virtual void setDrawables (DrawableListItemsInterface & drawableListItems ,
GenericCallback< DrawableListItemsInterface *, int16_t, int16_t > & updateDrawableCallback ,
DrawableListItemsInterface & centerDrawableListItems ,
GenericCallback< DrawableListItemsInterface *, int16_t, int16_t > & updateCenterDrawableCallback

)

Setups the widget.

Numerous parameters control the position of the widget, the two scroll lists inside and the values in them.

Parameters:

drawableListItems	Number of drawables in outer array.
updateDrawableCallback	The callback to update a drawable.
centerDrawableListItems	Number of drawables in center array.
updateCenterDrawableCallback	The callback to update a center drawable.

setDrawableSize

```
virtual void setDrawableSize ( int16_t drawableSize ,
                               int16_t drawableMargin
                               )
```

Sets drawables size.

The drawable is is the size of each drawable in the list in the set direction of the list (this is enforced by the **DrawableList** class). The specified margin is added above and below each item for spacing. The entire size of an item is thus size + 2 * spacing.

For a horizontal list each element will be *drawableSize* high and have the same width as set using **setWidth()**. For a vertical list each element will be *drawableSize* wide and have the same height as set using **setHeight()**.

Parameters:

drawableSize	The size of the drawable.
drawableMargin	The margin around drawables (margin before and margin after).

See also:

setWidth, **setHeight**, **setHorizontal**

setHeight

```
virtual void setHeight ( int16_t height )
```

Sets the height of this drawable.

Parameters:

height	The new height.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.Also sets the height of the children.

Reimplements: touchgfx::ScrollBase::setHeight

setHorizontal

virtual void setHorizontal (bool horizontal)

Sets a horizontal or vertical layout.

If parameter horizontal is set true, all drawables are arranged side by side. If horizontal is set false, the drawables are arranged above and below each other (vertically).

Parameters:

horizontal True to align drawables horizontal, false to align drawables vertically.

NOTE

Default value is false, i.e. vertical layout.

See also:

DrawableList::setHorizontal, getHorizontal

Reimplements: touchgfx::ScrollBase::setHorizontal

setNumberOfItems

virtual void setNumberOfItems (int16_t numberOfItems)

Sets number of items in the **DrawableList**.

This forces all drawables to be updated to ensure that the content is correct. For example a date selector might switch number of days between 28, 29, 30, and 31 depending on the month. A circular list might show 27-28-29-30-31 and might need to update this to show 27-28-1-2-3.

Parameters:

numberOfItems Number of items.

NOTE

The **DrawableList** is refreshed to reflect the change.

Reimplements: touchgfx::ScrollBase::setNumberOfItems

setSelectedItemExtraSize

virtual void setSelectedItemExtraSize (int16_t extraSizeBefore ,
int16_t extraSizeAfter

)

Sets selected item extra size to make the size of the area for the center drawables larger.

Parameters:

extraSizeBefore The extra size before.
extraSizeAfter The extra size after.

See also:

setSelectedItemOffset

setSelectedItemMargin

virtual void setSelectedItemMargin (int16_t marginBefore ,
int16_t marginAfter
)

Sets margin around selected item.

This like an invisible area added before and after the selected item (including extra size).

Parameters:

marginBefore The margin before.
marginAfter The margin after.

See also:

setSelectedItemOffset, **setSelectedItemExtraSize**

setSelectedItemOffset

virtual void setSelectedItemOffset (int16_t offset)

Sets selected item offset, measured in pixels, from the edge of the widget.

The offset is the relative x coordinate if the **ScrollWheel** is horizontal, otherwise it is the relative y coordinate. If this value is zero, the selected item is placed at the very start of the widget.

Parameters:

offset The offset.

See also:

getSelectedItemOffset

Reimplements: **touchgfx::ScrollWheelBase::setSelectedItemOffset**

setSelectedItemPosition

```
virtual void setSelectedItemPosition ( int16_t offset ,
                                       int16_t extraSizeBefore ,
                                       int16_t extraSizeAfter ,
                                       int16_t marginBefore ,
                                       int16_t marginAfter
                                       )
```

Sets the selected item offset.

This is the distance from the beginning of the **ScrollWheel** measured in pixels. The distance before and after that should also be drawn using the center drawables - for example to extend area of emphasized elements - can also be specified. Further, if a gap is needed between the "normal" drawables and the center drawables - for example to give the illusion that that items disappear under a graphical element, only to appear in the center.

This is a combination of setSelectedItemOffset, setSelectedItemExtraSize and setSelectedItemMargin.

Parameters:

offset	The offset of the selected item.
extraSizeBefore	The extra size before the selected item.
extraSizeAfter	The extra size after the selected item.
marginBefore	The margin before the selected item.
marginAfter	The margin after the selected item.

See also:

setSelectedItemOffset, **setSelectedItemExtraSize**, **setSelectedItemMargin**

setWidth

```
virtual void setWidth ( int16_t width )
```

Sets the width of this drawable.

Parameters:

width	The new width.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.Also sets the width of the children.

Reimplements: **touchgfx::ScrollBase::setWidth**

Protected Functions Documentation

refreshDrawableListsLayout

void refreshDrawableListsLayout ( )

Refresh drawable lists layout.

Ensure that the three DrawableLists are places correctly and setup properly. This is typically done after the **ScrollWheelWithSelectionStyle** has been resized or the size of the selected item is changed.

setOffset

virtual void setOffset (int32_t offset)

Sets display offset of first item.

Parameters:

offset The offset.

Reimplements: **touchgfx::ScrollBase::setOffset**

Protected Attributes Documentation

centerDrawables

DrawableListItemsInterface * centerDrawables

The drawables at the center of the scroll wheel.

drawables

DrawableListItemsInterface * drawables

The drawables at the beginning and end of the scroll wheel.

drawablesInFirstList

int16_t drawablesInFirstList

List of drawables in firsts.

extraSizeAfterSelectedItem

int16_t extraSizeAfterSelectedItem

The distance after selected item.

extraSizeBeforeSelectedItem

int16_t extraSizeBeforeSelectedItem

The distance before selected item.

list1

DrawableList list1

The center list.

list2

DrawableList list2

The list for items not in the center.

marginAfterSelectedItem

int16_t marginAfterSelectedItem

The distance after selected item.

marginBeforeSelectedItem

int16_t marginBeforeSelectedItem

The distance before selected item.

originalUpdateCenterDrawableCallback

GenericCallback< DrawableListItemsInterface , *int16_t, int16_t* > originalUpdateCenterDrawableCallback

The original update center drawable callback.

originalUpdateDrawableCallback

GenericCallback< DrawableListItemsInterface , *int16_t, int16_t* > originalUpdateDrawableCallback

The original update drawable callback.

Version: 4.16

SDL2TouchController

TouchController for the simulator.

See: TouchController

Inherits from: TouchController

Public Functions

virtual void	**init**()
	Initializes touch controller.
virtual bool	**sampleTouch**(int32_t & x, int32_t & y)
	Checks whether the touch screen is being touched, and if so, what coordinates.

Additional inherited members

Public Functions inherited from TouchController

virtual	**~TouchController**()
	Finalizes an instance of the **TouchController** class.

Public Functions Documentation

init

virtual void init ( )

Initializes touch controller.

Reimplements: **touchgfx::TouchController::init**

sampleTouch

```
virtual bool sampleTouch ( int32_t & x ,
                           int32_t & y
                           )
```

Checks whether the touch screen is being touched, and if so, what coordinates.

Parameters:

x The x position of the touch.
y The y position of the touch.

Returns:

True if a touch has been detected, otherwise false.

Reimplements: **touchgfx::TouchController::sampleTouch**

Version: 4.16

SDLTouchController

TouchController for the simulator.

See: TouchController

Inherits from: TouchController

Public Functions

virtual void	**init**()
	Initializes touch controller.
virtual bool	**sampleTouch**(int32_t & x, int32_t & y)
	Checks whether the touch screen is being touched, and if so, what coordinates.

Additional inherited members

Public Functions inherited from TouchController

virtual	**~TouchController**()
	Finalizes an instance of the **TouchController** class.

Public Functions Documentation

init

virtual void init ( )

Initializes touch controller.

Reimplements: **touchgfx::TouchController::init**

sampleTouch

```
virtual bool sampleTouch ( int32_t & x ,
                           int32_t & y
                           )
```

Checks whether the touch screen is being touched, and if so, what coordinates.

Parameters:

x The x position of the touch.
y The y position of the touch.

Returns:

True if a touch has been detected, otherwise false.

Reimplements: **touchgfx::TouchController::sampleTouch**

Version: 4.16

Shape

Simple widget capable of drawing a fully filled shape. The shape can be scaled and rotated. The Shape class allows the user to draw any shape and allows the defined shape to be scaled, rotated and moved freely. Example uses could be the hands of a clock.

See: AbstractShape

Inherits from: AbstractShape, CanvasWidget, Widget, Drawable

Public Functions

virtual **CWRUtil::Q5**	**getCornerX**(int i) const
	Gets the x coordinate of a corner (a point) of the shape.
virtual **CWRUtil::Q5**	**getCornerY**(int i) const
	Gets the y coordinate of a corner (a point) of the shape.
virtual int	**getNumPoints**() const
	Gets number of points used to make up the shape.
virtual void	**setCorner**(int i, **CWRUtil::Q5** x, **CWRUtil::Q5** y)
	Sets one of the points (a corner) of the shape in **CWRUtil::Q5** format.

Protected Functions

virtual **CWRUtil::Q5**	**getCacheX**(int i) const
	Gets cached x coordinate of a point/corner.
virtual **CWRUtil::Q5**	**getCacheY**(int i) const
	Gets cached y coordinate of a point/corner.
virtual void	**setCache**(int i, **CWRUtil::Q5** x, **CWRUtil::Q5** y)
	Sets the cached coordinates of a given point/corner.

Additional inherited members

Public Classes inherited from AbstractShape

struct	**ShapePoint**
	Defines an alias for a single point.

Public Functions inherited from AbstractShape

	AbstractShape()
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const
	Draw canvas widget for the given invalidated area.
int	**getAngle**() const
	Gets the current angle of the abstractShape.
void	**getAngle**(T & angle)
	Gets the abstractShape's angle.
void	**getOrigin**(T & dx, T & dy) const
	Gets the position of the shapes (0,0).
void	**getScale**(T & x, T & y) const
	Gets the x scale and y scale of the shape as previously set using setScale.
void	**moveOrigin**(T x, T y)
	Sets the position of the shape's (0,0) in the widget.
void	**setAngle**(T angle)
	Sets the absolute angle to turn the **AbstractShape**.
void	**setOrigin**(T x, T y)
	Sets the position of the shape's (0,0) in the widget.
void	**setScale**(T newXScale, T newYScale)
	Scale the **AbstractShape** the given amounts in the x direction and the y direction.
void	**setScale**(T scale)
	Scale the **AbstractShape** the given amount in the x direction and the y direction.
void	**setShape**(const **ShapePoint**< T > * points)
	Sets a shape the struct Points.

void	**setShape**(**ShapePoint**< T > * points)
	Sets a shape the struct Points.
void	**updateAbstractShapeCache**()
	Updates the **AbstractShape** cache.
void	**updateAngle**(T angle)
	Sets the absolute angle to turn the **AbstractShape**.
void	**updateScale**(T newXScale, T newYScale)
	Scale the **AbstractShape** the given amount in the x direction and the y direction.

Protected Functions inherited from AbstractShape

virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.

Public Functions inherited from CanvasWidget

	CanvasWidget()
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draws the given invalidated area.
virtual bool	**drawCanvasWidget**(const **Rect** & invalidatedArea) const =0
	Draw canvas widget for the given invalidated area.
virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getMinimalRect**() const
	Gets minimal rectangle containing the shape drawn by this widget.
virtual **AbstractPainter** &	**getPainter**() const
	Gets the current painter for the **CanvasWidget**.
virtual **Rect**	**getSolidRect**() const
	Gets the largest solid (non-transparent) rectangle.
virtual void	**invalidate**() const

	Invalidates the area covered by this **CanvasWidget**.
void	**resetMaxRenderLines**()
	Resets the maximum render lines.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setPainter**(**AbstractPainter** & painter)
	Sets a painter for the **CanvasWidget**.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()

	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.

bool **visible**	
	True if this drawable should be drawn.

Public Functions Documentation

getCornerX

virtual CWRUtil::Q5 getCornerX (int i)

Gets the x coordinate of a corner (a point) of the shape.

Parameters:

i Zero-based index of the corner.

Returns:

The corner x coordinate in **CWRUtil::Q5** format.

Reimplements: **touchgfx::AbstractShape::getCornerX**

getCornerY

virtual CWRUtil::Q5 getCornerY (int i)

Gets the y coordinate of a corner (a point) of the shape.

Parameters:

i Zero-based index of the corner.

Returns:

The corner y coordinate in **CWRUtil::Q5** format.

Reimplements: **touchgfx::AbstractShape::getCornerY**

getNumPoints

virtual int getNumPoints ( ) const

Gets number of points used to make up the shape.

Returns:

The number of points.

Reimplements: **touchgfx::AbstractShape::getNumPoints**

setCorner

```
virtual void setCorner ( int             i ,
                         CWRUtil::Q5     x ,
                         CWRUtil::Q5     y
                         )
```

Sets one of the points (a corner) of the shape in **CWRUtil::Q5** format.

Parameters:

i Zero-based index of the corner.
x The x coordinate in **CWRUtil::Q5** format.
y The y coordinate in **CWRUtil::Q5** format.

NOTE

Remember to call updateAbstractShapeCache() after calling setCorner one or more times, to make sure that the cached outline of the shape is correct.

See also:

updateAbstractShapeCache

Reimplements: **touchgfx::AbstractShape::setCorner**

Protected Functions Documentation

getCacheX

```
virtual CWRUtil::Q5 getCacheX ( int i )
```

Gets cached x coordinate of a point/corner.

Parameters:

i Zero-based index of the point/corner.

Returns:

The cached x coordinate, or zero if nothing is cached for the given i.

Reimplements: touchgfx::AbstractShape::getCacheX

getCacheY

```
virtual CWRUtil::Q5 getCacheY ( int i )
```

Gets cached y coordinate of a point/corner.

Parameters:

i Zero-based index of the point/corner.

Returns:

The cached y coordinate, or zero if nothing is cached for the given i.

Reimplements: touchgfx::AbstractShape::getCacheY

setCache

```
virtual void setCache ( int          i ,
                        CWRUtil::Q5  x ,
                        CWRUtil::Q5  y
                        )
```

Sets the cached coordinates of a given point/corner.

The coordinates in the cache are the coordinates from the corners after rotation and scaling has been applied to the coordinate.

Parameters:

i Zero-based index of the corner.
x The x coordinate.
y The y coordinate.

Reimplements: touchgfx::AbstractShape::setCache

Version: 4.16

ShapePoint

Defines an alias for a single point. Users of the AbstractShape can chose to store the coordinates as int or float depending on the needs. This will help setting up the abstractShape very easily using setAbstractShape().

Template Parameters:

- **T** Generic type parameter, either int or float.

See: setShape

Public Attributes

T	**x**
	The x coordinate of the points.
T	**y**
	The y coordinate of the points.

Public Attributes Documentation

x

T x

The x coordinate of the points.

y

T y

The y coordinate of the points.

Version: 4.16

SlideMenu

SlideMenu is a menu that can expand and collapse at the touch of a button. The SlideMenu can expand in any of the four directions. Menu items can be added, just like items are added to a normal container.

The relative positions of the background and state change button is configurable as is the direction in which the SlideMenu expands and collapses. How much of the SlideMenu that is visible when collapsed can also be set with the. It is, of course, important that the state change button is accessible when collapsed. The SlideMenu will collapse after a given timeout is reached. The timer can be reset, for example when the user interacts with elements in the list.

Menu elements are added normally using the add() method and are positioned relative to the SlideMenu.

Inherits from: Container, Drawable

Public Types

enum	**ExpandDirection** { SOUTH, NORTH, EAST, WEST }
	Values that represent the expand directions.
enum	**State** { COLLAPSED, EXPANDED }
	Values that represent the **SlideMenu** states.

Public Functions

virtual void	**add**(**Drawable** & d)
	Adds a drawable to the container.
virtual void	**animateToState**(**SlideMenu::State** newState)
	Animate to the given expanded or collapsed state.
virtual uint16_t	**getAnimationDuration**() const
	Gets the animation duration.
virtual **EasingEquation**	**getAnimationEasingEquation**() const
	Gets the animation easing equation.
virtual int16_t	**getBackgroundX**() const
	Gets the background **Image** x coordinate.
virtual int16_t	**getBackgroundY**() const

	Gets the background **Image** y coordinate.
virtual **SlideMenu::ExpandDirection**	**getExpandDirection**() const
	Gets the expand direction.
virtual uint16_t	**getExpandedStateTimeout**() const
	Gets expanded state timeout.
virtual uint16_t	**getExpandedStateTimer**() const
	Gets the expanded state timer.
virtual int16_t	**getHiddenPixelsWhenExpanded**() const
	Gets the hidden pixels when expanded.
virtual **SlideMenu::State**	**getState**()
	Gets the current expanded or collapsed state.
virtual int16_t	**getStateChangeButtonX**() const
	Gets the state change button x coordinate.
virtual int16_t	**getStateChangeButtonY**() const
	Gets the state change button y coordinate.
virtual int16_t	**getVisiblePixelsWhenCollapsed**() const
	Gets the visible pixels when collapsed.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**remove**(**Drawable** & d)
	Removes the drawable from the container.
virtual void	**resetExpandedStateTimer**()
	Resets the expanded state timer.
virtual void	**setAnimationDuration**(uint16_t duration)
	Sets the animation duration.
virtual void	**setAnimationEasingEquation**(**EasingEquation** animationEasingEquation)
	Sets the animation easing equation.
virtual void	**setExpandDirection**(**SlideMenu::ExpandDirection** newExpandDirection)
	Sets the expand direction.

virtual void	**setExpandedStateTimeout**(uint16_t timeout)
	Sets the expanded state timeout in ticks.
virtual void	**setHiddenPixelsWhenExpanded**(int16_t hiddenPixels)
	Sets the amount of hidden pixels when expanded.
virtual void	**setState**(**SlideMenu::State** newState)
	Sets the state of the **SlideMenu**.
virtual void	**setStateChangedAnimationEndedCallback**(**GenericCallback**< const **SlideMenu** & > & callback)
	Set the state change animation ended callback.
virtual void	**setStateChangedCallback**(**GenericCallback**< const **SlideMenu** & > & callback)
	Set the state changed callback.
virtual void	**setup**(**SlideMenu::ExpandDirection** newExpandDirection, const **Bitmap** & backgroundBMP, const **Bitmap** & stateChangeButtonBMP, const **Bitmap** & stateChangeButtonPressedBMP)
	Setup the **SlideMenu** by positioning the stateChangeButton next to background image relative to the expand direction, and center it in the other dimension.
virtual void	**setup**(**SlideMenu::ExpandDirection** newExpandDirection, const **Bitmap** & backgroundBMP, const **Bitmap** & stateChangeButtonBMP, const **Bitmap** & stateChangeButtonPressedBMP, int16_t backgroundX, int16_t backgroundY, int16_t stateChangeButtonX, int16_t stateChangeButtonY)
	Setup method for the **SlideMenu**.
virtual void	**setup**(**SlideMenu::ExpandDirection** newExpandDirection, const **Bitmap** & backgroundBMP, int16_t backgroundX, int16_t backgroundY)
	Setup method for the **SlideMenu**.
virtual void	**setVisiblePixelsWhenCollapsed**(int16_t visiblePixels)
	Sets the amount of visible pixels when collapsed.
	SlideMenu()
virtual	**~SlideMenu**()

Protected Functions

void	**animationEndedHandler**(const **MoveAnimator**< **Container** > & container)
	Handler for the state change animation ended event.
virtual int16_t	**getCollapsedXCoordinate**()
	Gets the x coordinate for the collapsed state.
virtual int16_t	**getCollapsedYCoordinate**()
	Gets the y coordinate for the collapsed state.
virtual int16_t	**getExpandedXCoordinate**()
	Gets the x coordinate for the expanded state.
virtual int16_t	**getExpandedYCoordinate**()
	Gets the y coordinate for the expanded state.
void	**stateChangeButtonClickedHandler**(const **AbstractButton** & button)
	Handler for the state change button clicked event.

Protected Attributes

uint16_t	**animationDuration**
	The animation duration of the state change animation.
Callback< **SlideMenu**, const **MoveAnimator**< **Container** > & >	**animationEndedCallback**
	The local state changed animation ended callback.
EasingEquation	**animationEquation**
	The easing equation used for the state change animation.
Image	**background**
	The background of the **SlideMenu**.
SlideMenu::State	**currentState**
	The current state of the **SlideMenu**.
SlideMenu::ExpandDirection	**expandDirection**
	The expand direction of the **SlideMenu**.
uint16_t	**expandedStateTimeout**
	The expanded state timeout.
uint16_t	**expandedStateTimer**

	The timer that counts towards the expandedStateTimeout. If reached the **SlideMenu** will animate to COLLAPSED.
int16_t	**hiddenPixelsWhenExpanded**
	The number of hidden pixels when expanded.
MoveAnimator< **Container** >	**menuContainer**
	The container holding the actual menu items. This is the container that performs the state change animation.
Callback< **SlideMenu**, const **AbstractButton** & >	**onStateChangeButtonClicked**
	The local state changed button clicked callback.
Button	**stateChangeButton**
	The state change button that toggles the **SlideMenu** state.
GenericCallback< const **SlideMenu** & > *	**stateChangedAnimationEndedCallback**
	The public state changed animation ended callback.
GenericCallback< const **SlideMenu** & > *	**stateChangedCallback**
	The public state changed button clicked callback.
int16_t	**visiblePixelsWhenCollapsed**
	The number of visible pixels when collapsed.

Additional inherited members

Public Functions inherited from Container

	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.

virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0

Draw this drawable.

Drawable()

Initializes a new instance of the **Drawable** class.

void **drawToDynamicBitmap**(**BitmapId** id)

Render the **Drawable** object into a dynamic bitmap.

Rect **getAbsoluteRect**() const

Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.

virtual **Drawable** * **getFirstChild**()

Function for obtaining the first child of this drawable if any.

int16_t **getHeight**() const

Gets the height of this **Drawable**.

virtual void **getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0

Function for obtaining the the last child of this drawable that intersects with the specified point.

Drawable * **getNextSibling**()

Gets the next sibling node.

Drawable * **getParent**() const

Returns the parent node.

const **Rect** & **getRect**() const

Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.

virtual **Rect** **getSolidRect**() const =0

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

virtual **Rect** **getSolidRectAbsolute**()

Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.

virtual void **getVisibleRect**(**Rect** & rect) const

Function for finding the visible part of this drawable.

int16_t **getWidth**() const

Gets the width of this **Drawable**.

int16_t **getX**() const

Gets the x coordinate of this **Drawable**, relative to its parent.

int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)

	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**

	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

ExpandDirection

enum ExpandDirection

Values that represent the expand directions.

SOUTH	Menu expands downwards (**Towards** the south)
NORTH	Menu expands upwards (**Towards** the north)
EAST	Menu expands to the right (**Towards** the east)
WEST	Menu expands to the left (**Towards** the west)

State

enum State

Values that represent the **SlideMenu** states.

COLLAPSED	Menu is currently collapsed.
EXPANDED	Menu is currently expanded.

Public Functions Documentation

add

virtual void add (Drawable & d)

Adds a drawable to the container.

Make sure the x and y coordinates of the **Drawable** is correct relative to the **SlideMenu**.

Parameters:

d The drawable to add.

Reimplements: **touchgfx::Container::add**

animateToState

virtual void animateToState (SlideMenu::State newState)

Animate to the given expanded or collapsed state.

Parameters:

newState The new state of the **SlideMenu**.

See also:

setState, **getState**

getAnimationDuration

virtual uint16_t getAnimationDuration ( ) const

Gets the animation duration.

Returns:

The animation duration.

getAnimationEasingEquation

virtual EasingEquation getAnimationEasingEquation ( ) const

Gets the animation easing equation.

Returns:

The animation easing equation.

getBackgroundX

virtual int16_t getBackgroundX ( ) const

Gets the background **Image** x coordinate.

Returns:

The background **Image** x coordinate.

getBackgroundY

virtual int16_t getBackgroundY ( ) const

Gets the background **Image** y coordinate.

Returns:

The background **Image** y coordinate.

getExpandDirection

virtual SlideMenu::ExpandDirection getExpandDirection ( ) const

Gets the expand direction.

Returns:

The expand direction.

getExpandedStateTimeout

virtual uint16_t getExpandedStateTimeout ( ) const

Gets expanded state timeout.

Returns:

The expanded state timeout.

getExpandedStateTimer

virtual uint16_t getExpandedStateTimer ( ) const

Gets the expanded state timer.

Returns:

The expanded state timer.

See also:

resetExpandedStateTimer

getHiddenPixelsWhenExpanded

virtual int16_t getHiddenPixelsWhenExpanded ( ) const

Gets the hidden pixels when expanded.

Returns:

The hidden pixels when expanded.

getState

virtual SlideMenu::State getState ( )

Gets the current expanded or collapsed state.

Returns:

The current state.

See also:

setState, **animateToState**

getStateChangeButtonX

virtual int16_t getStateChangeButtonX ( ) const

Gets the state change button x coordinate.

Returns:

The state change button x coordinate.

getStateChangeButtonY

virtual int16_t getStateChangeButtonY ( ) const

Gets the state change button y coordinate.

Returns:

The state change button y coordinate.

getVisiblePixelsWhenCollapsed

virtual int16_t getVisiblePixelsWhenCollapsed ( ) const

Gets the visible pixels when collapsed.

Returns:

The visible pixels when collapsed.

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

remove

virtual void remove (Drawable & d)

Removes the drawable from the container.

Parameters:

d The drawable to remove.

Reimplements: **touchgfx::Container::remove**

resetExpandedStateTimer

virtual void resetExpandedStateTimer ( )

Resets the expanded state timer.

The **SlideMenu** will automatically animate to the COLLAPSED state after a number of ticks, as set with **setExpandedStateTimeout()**. This method resets this timer.

See also:

getExpandedStateTimer

setAnimationDuration

virtual void setAnimationDuration (uint16_t duration)

Sets the animation duration.

Parameters:

duration The animation duration.

setAnimationEasingEquation

virtual void setAnimationEasingEquation (EasingEquation animationEasingEquation)

Sets the animation easing equation.

Parameters:

animationEasingEquation The animation easing equation.

setExpandDirection

virtual void setExpandDirection (SlideMenu::ExpandDirection newExpandDirection)

Sets the expand direction.

Parameters:

newExpandDirection The new expand direction.

setExpandedStateTimeout

virtual void setExpandedStateTimeout (uint16_t timeout)

Sets the expanded state timeout in ticks.

The **SlideMenu** will animate to the COLLAPSED state when this number of ticks has been executed while the **SlideMenu** is in the EXPANDED state. The timer can be reset with the resetExpandedStateTimer method.

Parameters:

timeout The timeout in ticks.

setHiddenPixelsWhenExpanded

virtual void setHiddenPixelsWhenExpanded (int16_t hiddenPixels)

Sets the amount of hidden pixels when expanded.

Parameters:

hiddenPixels The hidden pixels.

setState

virtual void setState (SlideMenu::State newState)

Sets the state of the **SlideMenu**.

No animation is performed.

Parameters:

newState The new state of the **SlideMenu**.

See also:

animateToState, **getState**

setStateChangedAnimationEndedCallback

virtual void setStateChangedAnimationEndedCallback (GenericCallback< const SlideMenu & > & callback)

Set the state change animation ended callback.

This callback is called when a state change animation has ended.

Parameters:

callback The callback.

setStateChangedCallback

virtual void setStateChangedCallback (GenericCallback< const SlideMenu & > & callback)

Set the state changed callback.

This callback is called when the state change button is clicked.

Parameters:

callback The callback.

setup

```
virtual void setup ( SlideMenu::ExpandDirection newExpandDirection ,
                     const Bitmap &             backgroundBMP ,
                     const Bitmap &             stateChangeButtonBMP ,
                     const Bitmap &             stateChangeButtonPressedBMP
                     )
```

Setup the **SlideMenu** by positioning the stateChangeButton next to background image relative to the expand direction, and center it in the other dimension.

The width and height of the **SlideMenu** will be automatically set to span both elements. Default values are: expandedStateTimeout = 200, visiblePixelsWhenCollapsed = 0, hiddenPixelsWhenExpanded = 0, animationDuration = 10, animationEquation = cubicEaseInOut.

Parameters:

newExpandDirection	The new expand direction.
backgroundBMP	The background bitmap.
stateChangeButtonBMP	The state change button bitmap.
stateChangeButtonPressedBMP	The state change button pressed bitmap.

setup

```
virtual void setup ( SlideMenu::ExpandDirection newExpandDirection ,
                     const Bitmap &             backgroundBMP ,
                     const Bitmap &             stateChangeButtonBMP ,
                     const Bitmap &             stateChangeButtonPressedBMP ,
                     int16_t                    backgroundX ,
                     int16_t                    backgroundY ,
                     int16_t                    stateChangeButtonX ,
                     int16_t                    stateChangeButtonY
                     )
```

Setup method for the **SlideMenu**.

Positioning of the background image and the stateChangeButton is done by stating the X and Y coordinates for the elements (relative to the **SlideMenu**). The width and height of the **SlideMenu** will be automatically set to span both elements. Default values are: expandedStateTimeout = 200, visiblePixelsWhenCollapsed = 0, hiddenPixelsWhenExpanded = 0, animationDuration = 10, animationEquation = cubicEaseInOut.

Parameters:

newExpandDirection	The new expand direction.
backgroundBMP	The background bitmap.
stateChangeButtonBMP	The state change button bitmap.
stateChangeButtonPressedBMP	The state change button pressed bitmap.
backgroundX	The background x coordinate.
backgroundY	The background y coordinate.
stateChangeButtonX	The state change button x coordinate.
stateChangeButtonY	The state change button y coordinate.

setup

```
virtual void setup ( SlideMenu::ExpandDirection newExpandDirection ,
                     const Bitmap &             backgroundBMP ,
                     int16_t                    backgroundX ,
                     int16_t                    backgroundY
                     )
```

Setup method for the **SlideMenu**.

Positioning of the background is done by stating the X and Y coordinates for the element (relative to the **SlideMenu**). The width and height of the **SlideMenu** will be automatically set to the size of the background. Default values are: expandedStateTimeout = 200, visiblePixelsWhenCollapsed = 0, hiddenPixelsWhenExpanded = 0, animationDuration * = 10, animationEquation = cubicEaseInOut.

Parameters:

newExpandDirection	The new expand direction.
backgroundBMP	The background bitmap.
backgroundX	The background x coordinate.
backgroundY	The background y coordinate.

setVisiblePixelsWhenCollapsed

virtual void setVisiblePixelsWhenCollapsed (int16_t visiblePixels)

Sets the amount of visible pixels when collapsed.

Parameters:

visiblePixels	The visible pixels.

SlideMenu

SlideMenu ( )

~SlideMenu

virtual ~SlideMenu ( )

Protected Functions Documentation

animationEndedHandler

void animationEndedHandler (const MoveAnimator< Container > & container)

Handler for the state change animation ended event.

Parameters:

container	The menuContainer.

getCollapsedXCoordinate

virtual int16_t getCollapsedXCoordinate ( )

Gets the x coordinate for the collapsed state.

Returns:

The collapsed x coordinate.

getCollapsedYCoordinate

virtual int16_t getCollapsedYCoordinate ( )

Gets the y coordinate for the collapsed state.

Returns:

The collapsed y coordinate.

getExpandedXCoordinate

virtual int16_t getExpandedXCoordinate ( )

Gets the x coordinate for the expanded state.

Returns:

The expanded x coordinate.

getExpandedYCoordinate

virtual int16_t getExpandedYCoordinate ( )

Gets the y coordinate for the expanded state.

Returns:

The expanded y coordinate.

stateChangeButtonClickedHandler

void stateChangeButtonClickedHandler (const AbstractButton & button)

Handler for the state change button clicked event.

Parameters:

button The state change button.

Protected Attributes Documentation

animationDuration

uint16_t animationDuration

The animation duration of the state change animation.

animationEndedCallback

Callback< SlideMenu, const MoveAnimator< Container > & > animationEndedCallback

The local state changed animation ended callback.

animationEquation

EasingEquation animationEquation

The easing equation used for the state change animation.

background

Image background

The background of the **SlideMenu**.

currentState

SlideMenu::State currentState

The current state of the **SlideMenu**.

expandDirection

SlideMenu::ExpandDirection expandDirection

The expand direction of the **SlideMenu**.

expandedStateTimeout

uint16_t expandedStateTimeout

The expanded state timeout.

expandedStateTimer

uint16_t expandedStateTimer

The timer that counts towards the expandedStateTimeout. If reached the **SlideMenu** will animate to COLLAPSED.

hiddenPixelsWhenExpanded

int16_t hiddenPixelsWhenExpanded

The number of hidden pixels when expanded.

menuContainer

MoveAnimator< Container > menuContainer

The container holding the actual menu items. This is the container that performs the state change animation.

onStateChangeButtonClicked

Callback< SlideMenu, const AbstractButton & > onStateChangeButtonClicked

The local state changed button clicked callback.

stateChangeButton

Button stateChangeButton

The state change button that toggles the **SlideMenu** state.

stateChangedAnimationEndedCallback

GenericCallback< const SlideMenu & > * stateChangedAnimationEndedCallback

The public state changed animation ended callback.

stateChangedCallback

GenericCallback< const SlideMenu & > * stateChangedCallback

The public state changed button clicked callback.

visiblePixelsWhenCollapsed

int16_t visiblePixelsWhenCollapsed

The number of visible pixels when collapsed.

Version: 4.16

Slider

A slider is a graphical element with which the user may set a value by moving an indicator on a slider, or simply by clicking the slider. The slider can operate in horizontal or vertical mode. The slider has two bitmaps. One bitmap is used on one side of the indicator. The other is used on the other side. They can be used in indicating the part of the slider value range that is currently selected.

The slider operates on an integer value range that can be set by the user.

Inherits from: Container, Drawable

Protected Types

enum	**SliderOrientation** { HORIZONTAL, VERTICAL }
	Values that represent slider orientations.

Public Functions

virtual uint16_t	**getIndicatorMax**() const
	Gets indicator maximum previous set using setupHorizontalSlider() or setupVerticalSlider().
virtual uint16_t	**getIndicatorMin**() const
	Gets indicator minimum previously set using setupHorizontalSlider() or setupVerticalSlider().
virtual uint16_t	**getMaxValue**() const
	Gets the maximum value previously set using setValueRange().
virtual uint16_t	**getMinValue**() const
	Gets the minimum value previously set using setValueRange().
int	**getValue**()
	Gets the current value represented by the indicator.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.

virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
void	**setBitmaps**(const **Bitmap** & sliderBackground, const **Bitmap** & sliderBackgroundSelected, const **Bitmap** & indicator)
	Sets all the bitmaps for the **Slider**.
void	**setBitmaps**(const **BitmapId** sliderBackground, const **BitmapId** sliderBackgroundSelected, const **BitmapId** indicator)
	Sets all the bitmaps for the **Slider**.
void	**setNewValueCallback**(**GenericCallback**< const **Slider** &, int > & callback)
	Associates an action to be performed when the slider changes its value.
void	**setStartValueCallback**(**GenericCallback**< const **Slider** &, int > & callback)
	Associates an action to be performed when an interaction with the slider is initiated (click or drag).
void	**setStopValueCallback**(**GenericCallback**< const **Slider** &, int > & callback)
	Associates an action to be performed when an interaction with the slider ends (click or drag).
virtual void	**setupHorizontalSlider**(uint16_t backgroundX, uint16_t backgroundY, uint16_t indicatorY, uint16_t indicatorMinX, uint16_t indicatorMaxX)
	Sets up the slider in horizontal mode with the range going from the left to right.
virtual void	**setupVerticalSlider**(uint16_t backgroundX, uint16_t backgroundY, uint16_t indicatorX, uint16_t indicatorMinY, uint16_t indicatorMaxY)
	Sets up the slider in vertical mode with the range going from the bottom to top.
virtual void	**setValue**(int value)
	Places the indicator at the specified value relative to the specified value range.
virtual void	**setValueRange**(int minValue, int maxValue)
	Sets the value range of the slider.
virtual void	**setValueRange**(int minValue, int maxValue, int newValue)
	Sets the value range of the slider.
	Slider()

Protected Functions

virtual int	**getIndicatorPositionRangeSize**() const
	Gets the indicator position range, i.e.
virtual uint16_t	**getIndicatorRadius**() const
	Gets the indicator radius, which is half the size of the indicator.
virtual int	**getValueRangeSize**() const
	Gets the value range, i.e.
virtual int	**positionToValue**(int16_t position) const
	Translate a position (x coordinate in horizontal mode and y in vertical mode) in the indicator position range to the corresponding value in the value range.
virtual void	**updateIndicatorPosition**(int16_t position)
	Updates the indicator position described by position.
virtual int16_t	**valueToPosition**(int value) const
	Translate a value in the value range to the corresponding position in the indicator position range (x coordinate in horizontal mode and y in vertical mode).

Protected Attributes

Image	**background**
	The background image.
Image	**backgroundSelected**
	The backgroundSelected image.
Container	**backgroundSelectedViewPort**
	The backgroundSelected view port. Controls the visible part of the backgroundSelected image.
int	**currentValue**
	The current value represented by the slider.
Image	**indicator**
	The indicator image.
int16_t	**indicatorMaxPosition**
	The maximum position of the indicator (either x coordinate in horizontal mode or y coordinate in vertical mode)

int16_t	**indicatorMinPosition**
	The minimum position of the indicator (either x coordinate in horizontal mode or y coordinate in vertical mode)
GenericCallback< const **Slider** &, int > *	**newValueCallback**
	The new value callback (called when the indicator is moved)
SliderOrientation	**sliderOrientation**
	The selected slider orientation.
GenericCallback< const **Slider** &, int > *	**startValueCallback**
	The start value callback (called when an interaction with the indicator is initiated)
GenericCallback< const **Slider** &, int > *	**stopValueCallback**
	The stop value callback (called when an interaction with the indicator ends)
int	**valueRangeMax**
	The value range max.
int	**valueRangeMin**
	The value range min.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.

virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.

virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.

void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.

virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Protected Types Documentation

SliderOrientation

enum SliderOrientation

Values that represent slider orientations.

HORIZONTAL	The **Slider** can be moved horizontally between left and right.
VERTICAL	The **Slider** can be moved vertically between top and bottom.

Public Functions Documentation

getIndicatorMax

virtual uint16_t getIndicatorMax ( ) const

Gets indicator maximum previous set using setupHorizontalSlider() or setupVerticalSlider().

Returns:

The calculated indicator maximum.

See also:

setupHorizontalSlider, **setupVerticalSlider**, **getIndicatorMin**

getIndicatorMin

virtual uint16_t getIndicatorMin ( ) const

Gets indicator minimum previously set using setupHorizontalSlider() or setupVerticalSlider().

Returns:

The indicator minimum.

See also:

setupHorizontalSlider, **setupVerticalSlider**, **getIndicatorMax**

getMaxValue

virtual uint16_t getMaxValue ( ) const

Gets the maximum value previously set using setValueRange().

Returns:

The maximum value.

See also:

setValueRange, **getMinValue**

getMinValue

virtual uint16_t getMinValue ( ) const

Gets the minimum value previously set using setValueRange().

Returns:

The minimum value.

See also:

setValueRange, **getMaxValue**

getValue

int getValue ( )

Gets the current value represented by the indicator.

Returns:

The current value.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

setBitmaps

```
void setBitmaps ( const Bitmap & sliderBackground ,
                  const Bitmap & sliderBackgroundSelected ,
                  const Bitmap & indicator
                  )
```

Sets all the bitmaps for the **Slider**.

The **Slider** shows the sliderBackgroundSelected bitmap in the region of the **Slider** that is selected, that is the area to the left of the indicator for a horizontal **Slider** and below the indicator for a vertical **Slider**. The sliderBackground is shown on the other side of the **Slider**. To ignore this effect simply use the same bitmap for both the sliderBackground and the sliderBackgroundSelected.

Parameters:

sliderBackground	The slider background with the slider range unselected.
sliderBackgroundSelected	The slider background with the slider range selected.
indicator	The indicator.

setBitmaps

```
void setBitmaps ( const BitmapId sliderBackground ,
                  const BitmapId sliderBackgroundSelected ,
                  const BitmapId indicator
                  )
```

Sets all the bitmaps for the **Slider**.

The **Slider** shows the sliderBackgroundSelected bitmap in the region of the **Slider** that is selected, that is the area to the left of the indicator for a horizontal **Slider** and below the indicator for a vertical **Slider**. The sliderBackground is shown on the other side of the **Slider**. To ignore this effect simply use the same bitmap for both the sliderBackground and the sliderBackgroundSelected.

Parameters:

sliderBackground	The slider background with the slider range unselected.
sliderBackgroundSelected	The slider background with the slider range selected.
indicator	The indicator.

setNewValueCallback

```
void setNewValueCallback ( GenericCallback< const Slider &, int > & callback )
```

Associates an action to be performed when the slider changes its value.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **Slider** and the current value of the slider.

See also:

GenericCallback

setStartValueCallback

void setStartValueCallback (GenericCallback< const Slider &, int > & callback)

Associates an action to be performed when an interaction with the slider is initiated (click or drag).

Parameters:

callback The callback to be executed. The callback will be given a reference to the **Slider** and the current value of the slider at interaction start.

See also:

GenericCallback

setStopValueCallback

void setStopValueCallback (GenericCallback< const Slider &, int > & callback)

Associates an action to be performed when an interaction with the slider ends (click or drag).

Parameters:

callback The callback to be executed. The callback will be given a reference to the **Slider** and the current value of the slider at interaction end.

See also:

GenericCallback

setupHorizontalSlider

```
virtual void setupHorizontalSlider ( uint16_t backgroundX ,
                                     uint16_t backgroundY ,
                                     uint16_t indicatorY ,
                                     uint16_t indicatorMinX ,
                                     uint16_t indicatorMaxX
                                     )
```

Sets up the slider in horizontal mode with the range going from the left to right.

Places the backgrounds and the indicator inside the **Slider** container. It is possible to place the end points of the indicator outside the background image if it needs to go beyond the boundaries of the background. The width and height of the **Slider** will be adjusted appropriately so that both the background and the indicator will be fully visible in both the minimum and maximum indicator positions.

Calls **setValue()** with the current value (default 0) and triggers the newSliderValue callback.

Parameters:

backgroundX	The background x coordinate inside the slider.
backgroundY	The background y coordinate inside the slider.
indicatorY	The indicator y coordinate inside the slider.
indicatorMinX	The indicator minimum x coordinate inside the slider. This is the position used when the slider is at its minimum value. Must be less than indicatorMaxX.
indicatorMaxX	The indicator maximum x coordinate inside the slider. This is the position used when the slider is at its maximum value. Must be greater than indicatorMinX.

NOTE

The x and y position of the Slider will either be the left/top of the background or the left/top of the indicator in its minimum x coordinate.

setupVerticalSlider

```
virtual void setupVerticalSlider ( uint16_t backgroundX ,
                                   uint16_t backgroundY ,
                                   uint16_t indicatorX ,
                                   uint16_t indicatorMinY ,
                                   uint16_t indicatorMaxY
                                   )
```

Sets up the slider in vertical mode with the range going from the bottom to top.

Places the backgrounds and the indicator inside the **Slider** container. It is possible to place the end points of the indicator outside the background image if it needs to go beyond the boundaries of the background. The width and height of the **Slider** will be adjusted appropriately so that both the background and the indicator will be fully visible in both the minimum and maximum indicator positions.

Calls setValue with the current value (default 0) and triggers the newSliderValue callback.

Parameters:

backgroundX	The background x coordinate inside the slider.
backgroundY	The background y coordinate inside the slider.
indicatorX	The indicator x coordinate inside the slider.
indicatorMinY	The indicator minimum y coordinate inside the slider. This is the position used when the slider is at its maximum value. Must be less than indicatorMaxX.
indicatorMaxY	The indicator maximum y coordinate inside the slider. This is the position used when the slider is at its minimum value. Must be greater than indicatorMinX.

NOTE

The x and y position of the Slider will either be the left/top of the background or the left/top of the indicator in its minimum y coordinate.

setValue

virtual void setValue (int value)

Places the indicator at the specified value relative to the specified value range.

Values beyond the value range will be rounded to the min/max value in the value range.

Parameters:

value The value.

NOTE

The value update triggers a newSliderValue callback just as a drag or click does. If the value range is larger than the number of pixels specified for the indicator min and indicator max, some values will not be represented by the slider and thus is not possible to set with this method. In this case the value will be rounded to the nearest value that is represented in the current setting.

See also:

setValueRange

setValueRange

virtual void setValueRange (int minValue ,
int maxValue
)

Sets the value range of the slider.

Values accepted and returned by the slider will be in this range.

The slider will set its value to the current value or round to minValue or maxValue if the current value is outside the new range.

Parameters:

minValue	The minimum value. Must be less than maxValue.
maxValue	The maximum value. Must be greater than minValue.

NOTE

If the range is larger than the number of pixels specified for the indicator min and indicator max, some values will not be represented by the slider.

setValueRange

```
virtual void setValueRange ( int minValue ,
                             int maxValue ,
                             int newValue
                             )
```

Sets the value range of the slider.

Values accepted and returned by the slider will be in this range.

The slider will set its value to the specified new value.

Parameters:

minValue	The minimum value. Must be less than maxValue.
maxValue	The maximum value. Must be greater than minValue.
newValue	The new value.

NOTE

If the range is larger than the number of pixels specified for the indicator min and max some values will not be represented by the slider.

Slider

```
Slider (  )
```

Protected Functions Documentation

getIndicatorPositionRangeSize

virtual int getIndicatorPositionRangeSize ( ) const

Gets the indicator position range, i.e.

the difference between max and min for the position of the indicator.

Returns:

The indicator position range.

getIndicatorRadius

virtual uint16_t getIndicatorRadius ( ) const

Gets the indicator radius, which is half the size of the indicator.

Returns:

The the indicator radius.

getValueRangeSize

virtual int getValueRangeSize ( ) const

Gets the value range, i.e.

the difference between max and min for the value range.

Returns:

The value range.

positionToValue

virtual int positionToValue (int16_t position)

Translate a position (x coordinate in horizontal mode and y in vertical mode) in the indicator position range to the corresponding value in the value range.

Parameters:

position The position.

Returns:

The value that corresponds to the coordinate.

updateIndicatorPosition

virtual void updateIndicatorPosition (int16_t position)

Updates the indicator position described by position.

Calls the newSliderValueCallback with the new value.

Parameters:

position The position (x coordinate in horizontal mode and y coordinate in vertical mode).

valueToPosition

virtual int16_t valueToPosition (int value)

Translate a value in the value range to the corresponding position in the indicator position range (x coordinate in horizontal mode and y in vertical mode).

Parameters:

value The value.

Returns:

The coordinate that corresponds to the value.

Protected Attributes Documentation

background

Image background

The background image.

backgroundSelected

Image backgroundSelected

The backgroundSelected image.

backgroundSelectedViewPort

Container backgroundSelectedViewPort

The backgroundSelected view port. Controls the visible part of the backgroundSelected image.

currentValue

int currentValue

The current value represented by the slider.

indicator

Image indicator

The indicator image.

indicatorMaxPosition

int16_t indicatorMaxPosition

The maximum position of the indicator (either x coordinate in horizontal mode or y coordinate in vertical mode)

indicatorMinPosition

int16_t indicatorMinPosition

The minimum position of the indicator (either x coordinate in horizontal mode or y coordinate in vertical mode)

newValueCallback

GenericCallback< const Slider &, int > * newValueCallback

The new value callback (called when the indicator is moved)

sliderOrientation

SliderOrientation sliderOrientation

The selected slider orientation.

startValueCallback

GenericCallback< const Slider &, int > * startValueCallback

The start value callback (called when an interaction with the indicator is initiated)

stopValueCallback

GenericCallback< const Slider &, int > * stopValueCallback

The stop value callback (called when an interaction with the indicator ends)

valueRangeMax

int valueRangeMax

The value range max.

valueRangeMin

int valueRangeMin

The value range min.

Version: 4.16

SlideTransition

A Transition that slides from one screen to the next. It does so by moving a SnapShotWidget with a snapshot of the Screen transitioning away from, and by moving the contents of Screen transitioning to.

See: Transition

Inherits from: Transition

Public Functions

virtual void	**handleTickEvent**()
	Handles the tick event when transitioning.
virtual void	**init**()
	Initializes the transition.
	SlideTransition(const uint8_t transitionSteps =20)
	Initializes a new instance of the **SlideTransition** class.
virtual void	**tearDown**()
	Tears down the Animation.

Protected Functions

virtual void	**initMoveDrawable**(**Drawable** & d)
	Moves the **Drawable** to its initial position, just outside the actual display.
virtual void	**tickMoveDrawable**(**Drawable** & d)
	Moves the **Drawable**.

Protected Attributes

SnapshotWidget	**snapshot**

	The SnapshotWidget that is moved when transitioning.
SnapshotWidget *	**snapshotPtr**
	Pointer pointing to the snapshot used in this transition.The snapshot pointer.

Additional inherited members

Public Functions inherited from Transition

virtual void	**invalidate**()
	Invalidates the screen when starting the **Transition**.
bool	**isDone**() const
	Query if the transition is done transitioning.
virtual void	**setScreenContainer**(**Container** & cont)
	Sets the **ScreenContainer**.
	Transition()
	Initializes a new instance of the **Transition** class.
virtual	**~Transition**()
	Finalizes an instance of the **Transition** class.

Protected Attributes inherited from Transition

bool	**done**
	Flag that indicates when the transition is done. This should be set by implementing classes.
Container *	**screenContainer**
	The screen **Container** of the **Screen** transitioning to.

Public Functions Documentation

handleTickEvent

virtual void handleTickEvent ( )

Handles the tick event when transitioning.

It moves the contents of the **Screen**'s container and a **SnapshotWidget** with a snapshot of the previous **Screen**. The direction of the transition determines the direction the contents of the container and the **SnapshotWidget** moves.

Reimplements: **touchgfx::Transition::handleTickEvent**

init

virtual void init ( )

Initializes the transition.

Called after the constructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::init**

SlideTransition

SlideTransition (const uint8_t transitionSteps =20)

Initializes a new instance of the **SlideTransition** class.

Parameters:

transitionSteps (Optional) Number of steps (ticks) in the transition animation, default is 20.

tearDown

virtual void tearDown ( )

Tears down the Animation.

Called before the destructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::tearDown**

Protected Functions Documentation

initMoveDrawable

virtual void initMoveDrawable (Drawable & d)

Moves the **Drawable** to its initial position, just outside the actual display.

Parameters:

d The **Drawable** to move.

tickMoveDrawable

virtual void tickMoveDrawable (Drawable & d)

Moves the **Drawable**.

Parameters:

d The **Drawable** to move.

Protected Attributes Documentation

snapshot

SnapshotWidget snapshot

The SnapshotWidget that is moved when transitioning.

snapshotPtr

SnapshotWidget * snapshotPtr

Pointer pointing to the snapshot used in this transition.The snapshot pointer.

Version: 4.16

Snapper

A mix-in that will make class T draggable and able to snap to a position when a drag operation has ended. The mix-in is able to perform callbacks when the snapper gets dragged and when the Snapper snaps to its snap position.

Template Parameters:

- **T** specifies the type to enable the Snap behavior to.

See: Draggable<T>

Inherits from: touchgfx::Draggable< T >, T

Public Functions

virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Handles the click events when the **Snapper** is clicked.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Called when dragging the **Draggable** object.
void	**setDragAction**(**GenericCallback**< const **DragEvent** & > & callback)
	Associates an action to be performed when the **Snapper** is dragged.
void	**setSnappedAction**(**GenericCallback**<> & callback)
	Associates an action to be performed when the **Snapper** is snapped.
void	**setSnapPosition**(int16_t x, int16_t y)
	Sets the position the **Snapper** should snap to.
	Snapper()

Additional inherited members

Public Functions inherited from touchgfx::Draggable< T >

Draggable()

Initializes a new instance of the **Draggable** class.

Public Functions Documentation

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Handles the click events when the **Snapper** is clicked.

It saves its current position as the snap position if the **Snapper** is pressed. This happens when the drag operation starts.

The snapper will then move to the snap position when the click is released. This happens when the drag operation ends.

Parameters:

evt The click event.

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Called when dragging the **Draggable** object.

The object is moved according to the drag event.

Parameters:

evt The drag event.

Reimplements: **touchgfx::Draggable::handleDragEvent**

setDragAction

void setDragAction (GenericCallback< const DragEvent & > & callback)

Associates an action to be performed when the **Snapper** is dragged.

Parameters:

callback The callback will be executed with the **DragEvent**.

See also:

GenericCallback

setSnappedAction

```
void setSnappedAction ( GenericCallback<> & callback )
```

Associates an action to be performed when the **Snapper** is snapped.

Parameters:

callback The callback to be executed on snap.

See also:

GenericCallback

setSnapPosition

```
void setSnapPosition ( int16_t x ,
                       int16_t y
                       )
```

Sets the position the **Snapper** should snap to.

This position will be overridden with the Snappers current position when the **Snapper** is pressed.

Parameters:

x The x coordinate.
y The y coordinate.

Snapper

```
Snapper (  )
```

Version: 4.16

SnapshotWidget

A widget that is able to make a snapshot of the area the SnapshotWidget covers into either a Bitmap or into animation storage (if this available). Once the snapshot has been taken using SnapshowWidget::makeSnapshot(), the SnapshotWidget will show the captured snapshot when it is subsequently drawn.

Inherits from: Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**makeSnapshot**()
	Makes a snapshot of the area the **SnapshotWidget** currently covers.
virtual void	**makeSnapshot**(const **BitmapId** bmp)
	Makes a snapshot of the area the **SnapshotWidget** currently covers.
void	**setAlpha**(const uint8_t newAlpha)
	Sets the opacity (alpha value).
	SnapshotWidget()

Protected Attributes

uint8_t	**alpha**
	The alpha with which to draw this snapshot.
BitmapId	**bitmapId**

	BitmapId where copy is stored s copied to.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const

	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const

	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

makeSnapshot

virtual void makeSnapshot ( )

Makes a snapshot of the area the **SnapshotWidget** currently covers.

This area is defined by setting the dimensions and the position of the **SnapshotWidget**. The snapshot is stored in Animation Storage.

See also:

setPosition

makeSnapshot

virtual void makeSnapshot (const BitmapId bmp)

Makes a snapshot of the area the **SnapshotWidget** currently covers.

This area is defined by setting the dimensions and the position of the **SnapshotWidget**. The snapshot is stored in the provided dynamic bitmap. The format of the **Bitmap** must match the format of the display.

Parameters:

bmp The target dynamic bitmap.

setAlpha

void setAlpha (const uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

SnapshotWidget

```
SnapshotWidget ( )
```

Protected Attributes Documentation

alpha

uint8_t alpha

The alpha with which to draw this snapshot.

bitmapId

BitmapId bitmapId

BitmapId where copy is stored s copied to.

Version: 4.16

StringVisuals

The visual elements when writing a string.

Public Functions

StringVisuals()

Initializes a new instance of the **LCD** class.

StringVisuals(const **Font** * font, **colortype** color, uint8_t alpha, **Alignment** alignment, int16_t linespace, **TextRotation** rotation, **TextDirection** textDirection, uint8_t indentation, **WideTextAction** wideTextAction =**WIDE_TEXT_NONE**)

Construct a StringVisual object for rendering text.

Public Attributes

Alignment	**alignment**
	The alignment to use. Default is LEFT.
uint8_t	**alpha**
	8-bit alpha value. Default is 255 (solid).
colortype	**color**
	RGB color value. Default is 0 (black).
const **Font** *	**font**
	The font to use.
uint8_t	**indentation**
	Indentation of text inside rectangle. Text will start this far from the left/right edge.
int16_t	**linespace**
	Line space in pixels for multiline strings. Default is 0.
TextRotation	**rotation**
	Orientation (rotation) of the text. Default is TEXT_ROTATE_0.
TextDirection	**textDirection**

	The direction to use. Default is LTR.
WideTextAction	**wideTextAction**
	What to do with wide text lines.

Public Functions Documentation

StringVisuals

```
StringVisuals ( )
```

Initializes a new instance of the **LCD** class.

StringVisuals

```
StringVisuals ( const Font *    font ,
                colortype       color ,
                uint8_t         alpha ,
                Alignment       alignment ,
                int16_t         linespace ,
                TextRotation    rotation ,
                TextDirection   textDirection ,
                uint8_t         indentation ,
                WideTextAction  wideTextAction =WIDE_TEXT_NONE
                )
```

Construct a StringVisual object for rendering text.

Parameters:

font	The **Font** with which to draw the text.
color	The color with which to draw the text.
alpha	Alpha blending. Default value is 255 (solid)
alignment	How to align the text.
linespace	**Line** space in pixels between each line, in case the text contains newline characters.
rotation	How to rotate the text.
textDirection	The text direction.
indentation	The indentation of the text from the left and right of the text area rectangle.
wideTextAction	(Optional) What to do with lines longer than the width of the **TextArea**.

Public Attributes Documentation

alignment

Alignment alignment

The alignment to use. Default is LEFT.

alpha

uint8_t alpha

8-bit alpha value. Default is 255 (solid).

color

colortype color

RGB color value. Default is 0 (black).

font

const Font * font

The font to use.

indentation

uint8_t indentation

Indentation of text inside rectangle. Text will start this far from the left/right edge.

linespace

int16_t linespace

Line space in pixels for multiline strings. Default is 0.

rotation

TextRotation rotation

Orientation (rotation) of the text. Default is TEXT_ROTATE_0.

textDirection

TextDirection textDirection

The direction to use. Default is LTR.

wideTextAction

WideTextAction wideTextAction

What to do with wide text lines.

Version: 4.16

SwipeContainer

A SwipeContainer is a Container with a horizontally laid out list of identically sized Drawables. The bottom of the SwipeContainer shows a page indicator to indicate the position in the horizontal list of items in the SwipeContainer.

See: ListLayout

Inherits from: Container, Drawable

Public Functions

virtual void	**add**(**Drawable** & page)
	Adds a page to the container.
uint8_t	**getNumberOfPages**()
	Gets number of pages.
uint8_t	**getSelectedPage**() const
	Gets the currently selected page.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**remove**(**Drawable** & page)
	Removes the page from the container.
void	**setEndSwipeElasticWidth**(uint16_t width)
	When dragging either one of the end pages a part of the background will become visible until the user stop dragging and the end page swipes back to its position.

void	**setPageIndicatorBitmaps**(const **Bitmap** & normalPage, const **Bitmap** & highlightedPage)
	Sets the bitmaps that are used by the page indicator.
void	**setPageIndicatorXY**(int16_t x, int16_t y)
	Sets the x and y position of the page indicator.
void	**setPageIndicatorXYWithCenteredX**(int16_t x, int16_t y)
	Sets the x and y position of the page indicator.
void	**setSelectedPage**(uint8_t pageIndex)
	Sets the selected page.
virtual void	**setSwipeCutoff**(uint16_t cutoff)
	Set the swipe cutoff which indicates how far you should drag a page before it results in a page change.
	SwipeContainer()
virtual	**~SwipeContainer**()

Additional inherited members

Public Functions inherited from Container

	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)

	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.

void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const

	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.

bool **visible**	
	True if this drawable should be drawn.

Public Functions Documentation

add

virtual void add (Drawable & page)

Adds a page to the container.

Parameters:

page The page to add.

NOTE

All pages must have the same width and height.

Reimplements: **touchgfx::Container::add**

getNumberOfPages

uint8_t getNumberOfPages ( )

Gets number of pages.

Returns:

The number of pages.

getSelectedPage

uint8_t getSelectedPage ( ) const

Gets the currently selected page.

Returns:

Zero-based index of the current page. Rage from 0 to numberOfPages-1.

See also:

setSelectedPage

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & evt)

Defines the event handler interface for GestureEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **GestureEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleGestureEvent**

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

remove

virtual void remove (Drawable & page)

Removes the page from the container.

Parameters:

page The page to remove.

NOTE

This is safe to call even if page is not a page (in which case nothing happens).

Reimplements: **touchgfx::Container::remove**

setEndSwipeElasticWidth

void setEndSwipeElasticWidth (uint16_t width)

When dragging either one of the end pages a part of the background will become visible until the user stop dragging and the end page swipes back to its position.

The width of this area is set by this method.

Parameters:

width The width in pixels.

setPageIndicatorBitmaps

void setPageIndicatorBitmaps (const Bitmap & normalPage ,
const Bitmap & highlightedPage

```
                                    )
```

Sets the bitmaps that are used by the page indicator.

The bitmap for the normal page is repeated side-by-side and the bitmap for a highlighted page is put in the proper position.

Parameters:

normalPage	The normal page.
highlightedPage	The highlighted page.

setPageIndicatorXY

```
void setPageIndicatorXY ( int16_t x ,
                          int16_t y
                          )
```

Sets the x and y position of the page indicator.

Parameters:

x	The x coordinate.
y	The y coordinate.

See also:

setPageIndicatorXYWithCenteredX

setPageIndicatorXYWithCenteredX

```
void setPageIndicatorXYWithCenteredX ( int16_t x ,
                                       int16_t y
                                       )
```

Sets the x and y position of the page indicator.

The value specified as x will be the center coordinate of the page indicators.

Parameters:

x	The center x coordinate.
y	The y coordinate.

NOTE

This method should not be used until all pages have been added, the setPageIndicatorBitmaps() has been called and the page indicator therefore has the correct width.

setSelectedPage

void setSelectedPage (uint8_t pageIndex)

Sets the selected page.

Parameters:

pageIndex Zero-based index of the page. Range from 0 to numberOfPages-1.

See also:

getSelectedPage

setSwipeCutoff

virtual void setSwipeCutoff (uint16_t cutoff)

Set the swipe cutoff which indicates how far you should drag a page before it results in a page change.

Parameters:

cutoff The cutoff in pixels.

SwipeContainer

SwipeContainer ( )

~SwipeContainer

virtual ~SwipeContainer ( )

Version: 4.16

TextArea

This widget is capable of showing a text area on the screen. The text must be a predefined TypedText in the text sheet in the assets folder. In order to display a dynamic text, use TextAreaWithOneWildcard or TextAreaWithTwoWildcards.

See: TypedText, TextAreaWithOneWildcard, TextAreaWithTwoWildcards

Note: A TextArea just holds a pointer to the text displayed. The developer must ensure that the pointer remains valid when drawing.

Inherits from: Widget, Drawable

Inherited by: TextAreaWithOneWildcard, TextAreaWithTwoWildcards

Public Functions

virtual int16_t	**calculateTextHeight**(const **Unicode::UnicodeChar** * format, ...) const
	Gets the total height needed by the text.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
FORCE_INLINE_FUNCTION **colortype**	**getColor**() const
	Gets the color of the text.
FORCE_INLINE_FUNCTION uint8_t	**getIndentation**()
	Gets the indentation of text inside the **TextArea**.
FORCE_INLINE_FUNCTION int16_t	**getLinespacing**() const
	Gets the line spacing of the **TextArea**.
TextRotation	**getRotation**() const
	Gets rotation of the text in the **TextArea**.
virtual **Rect**	**getSolidRect**() const

	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual int16_t	**getTextHeight**()
	Gets the total height needed by the text, taking number of lines and line spacing into consideration.
virtual uint16_t	**getTextWidth**() const
	Gets the width in pixels of the current associated text in the current selected language.
TypedText	**getTypedText**() const
	Gets the TypedText of the text area.
WideTextAction	**getWideTextAction**() const
	Gets wide text action previously set using setWideTextAction.
void	**resizeHeightToCurrentText**()
	Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.
void	**resizeHeightToCurrentTextWithRotation**()
	Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.
void	**resizeToCurrentText**()
	Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language.
void	**resizeToCurrentTextWithAlignment**()
	Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language, and for centered and right aligned text, the position of the **TextArea** widget is also updated to keep the text in the same position on the display.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBaselineY**(int16_t baselineY)
	Adjusts the **TextArea** y coordinate so the text will have its baseline at the specified value.
FORCE_INLINE_FUNCTION void	**setColor**(**colortype** color)

	Sets the color of the text.
FORCE_INLINE_FUNCTION void	**setIndentation**(uint8_t indent)
	Sets the indentation for the text.
FORCE_INLINE_FUNCTION void	**setLinespacing**(int16_t space)
	Sets the line spacing of the **TextArea**.
void	**setRotation**(const **TextRotation** rotation)
	Sets rotation of the text in the **TextArea**.
void	**setTypedText**(**TypedText** t)
	Sets the TypedText of the text area.
void	**setWideTextAction**(**WideTextAction** action)
	Defines what to do if a line of text is wider than the text area.
virtual void	**setXBaselineY**(int16_t x, int16_t baselineY)
	Adjusts the **TextArea** x and y coordinates so the text will have its baseline at the specified y value.
	TextArea()

Protected Attributes

uint8_t	**alpha**
	The alpha to use.
colortype	**color**
	The color to use for the text.
uint8_t	**indentation**
	The indentation of the text inside the text area.
int16_t	**linespace**
	The extra space between lines of text, measured in pixels.
TextRotation	**rotation**
	The text rotation to use in steps of 90 degrees.
TypedText	**typedText**

	The TypedText to display.
WideTextAction	**wideTextAction**
	What to do if the lines of text are wider than the text area.

Additional inherited members

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()

	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const

Gets whether this **Drawable** receives touch events or not.

bool **isVisible**() const

Gets whether this **Drawable** is visible.

virtual void **moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

calculateTextHeight

virtual int16_t calculateTextHeight (const Unicode::UnicodeChar * format , const ... const) const

Gets the total height needed by the text.

Determined by number of lines and linespace. The number of parameters passed after the format, must match the number of wildcards in the **TypedText**.

Parameters:

format The text containing <placeholder> wildcards.
... Variable arguments providing additional information.

Returns:

the total height needed by the text.

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

Reimplemented by: **touchgfx::TextAreaWithOneWildcard::draw**, **touchgfx::TextAreaWithTwoWildcards::draw**

getAlpha

uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getColor

FORCE_INLINE_FUNCTION colortype getColor ( ) const

Gets the color of the text.

If no color has been set, the default color, black, is returned.

Returns:

The color to used for drawing the text.

getIndentation

FORCE_INLINE_FUNCTION uint8_t getIndentation ( )

Gets the indentation of text inside the **TextArea**.

Returns:

The indentation.

See also:

setIndentation

getLinespacing

FORCE_INLINE_FUNCTION int16_t getLinespacing ( ) const

Gets the line spacing of the **TextArea**.

If no line spacing has been set, the line spacing is 0.

Returns:

The line spacing.

See also:

setLinespacing

getRotation

TextRotation getRotation ( ) const

Gets rotation of the text in the **TextArea**.

Returns:

The rotation of the text.

See also:

setRotation

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

getTextHeight

virtual int16_t getTextHeight ( )

Gets the total height needed by the text, taking number of lines and line spacing into consideration.

Returns:

the total height needed by the text.

Reimplemented by: **touchgfx::TextAreaWithOneWildcard::getTextHeight**, **touchgfx::TextAreaWithTwoWildcards::getTextHeight**

getTextWidth

virtual uint16_t getTextWidth ( ) const

Gets the width in pixels of the current associated text in the current selected language.

In case of multi-lined text the width of the widest line is returned.

Returns:

The width in pixels of the current text.

Reimplemented by: **touchgfx::TextAreaWithOneWildcard::getTextWidth**, **touchgfx::TextAreaWithTwoWildcards::getTextWidth**

getTypedText

TypedText getTypedText ( ) const

Gets the TypedText of the text area.

Returns:

The currently used **TypedText**.

getWideTextAction

WideTextAction getWideTextAction ( ) const

Gets wide text action previously set using setWideTextAction.

Returns:

current WideTextAction setting.

See also:

setWideTextAction, **WideTextAction**

resizeHeightToCurrentText

void resizeHeightToCurrentText ( )

Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.

This is especially useful for texts with WordWrap enabled.

NOTE

If the current text rotation is either 90 or 270 degrees, the width of the text area will be set and not the height, as the text is rotated. If the current text is rotated, the x/y coordinate is not updated, which means that the text will be repositioned on the display.

See also:

resizeToCurrentText, **setWideTextAction**, **setRotation**, **resizeHeightToCurrentTextWithRotation**

resizeHeightToCurrentTextWithRotation

void resizeHeightToCurrentTextWithRotation ( )

Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.

This is especially useful for texts with WordWrap enabled.

NOTE

If the current text rotation is either 90 or 270 degrees, the width of the text area will be set and not the height, as the text is rotated. Also, the x or y coordinates will be updated.

See also:

resizeToCurrentText, **setWideTextAction**, **setRotation**, **resizeHeightToCurrentText**

resizeToCurrentText

void resizeToCurrentText ( )

Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language.

If WordWrap is turned on for the **TextArea**, the height might be set to an unexpected value, as only manually insert line breaks in the text will be respected - use **resizeHeightToCurrentText()** to keep

the width of the **TextArea** and therefore retain word wrapping.

If the text is centered or right aligned, calling **resizeToCurrentText()** will actually move the text on the screen, as the x and y coordinates of the **TextArea** widget is not changed. To simply minimize the size of the **TextArea** but keep the **TypedText** in the same position on the screen, use **resizeToCurrentTextWithAlignment()**. This is also the case if the text is rotated, e.g. 180 degrees.

> **NOTE**
>
> **If the current text rotation is either 90 or 270 degrees, the width of the text area will be set to the height of the text and vice versa, as the text is rotated.**

See also:

setRotation, **resizeHeightToCurrentText**

resizeToCurrentTextWithAlignment

void resizeToCurrentTextWithAlignment ( )

Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language, and for centered and right aligned text, the position of the **TextArea** widget is also updated to keep the text in the same position on the display.

Text that is rotated is also handled properly.

> **NOTE**
>
> **If the current text rotation is either 90 or 270 degrees, the width of the text area will be set to the height of the text and vice versa, as the text is rotated.**

See also:

setRotation, **resizeHeightToCurrentText**

setAlpha

void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setBaselineY

virtual void setBaselineY (int16_t baselineY)

Adjusts the **TextArea** y coordinate so the text will have its baseline at the specified value.

The placements is relative to the specified **TypedText** so if the **TypedText** is changed, you have to set the baseline again.

Parameters:

baselineY The y coordinate of the baseline of the text.

NOTE

setTypedText() must be called prior to setting the baseline.

setColor

FORCE_INLINE_FUNCTION void setColor (colortype color)

Sets the color of the text.

If no color is set, the default color (black) is used.

Parameters:

color The color to use.

setIndentation

FORCE_INLINE_FUNCTION void setIndentation (uint8_t indent)

Sets the indentation for the text.

This can be very useful when a font is an italic font where letters such as "j" and "g" extend a lot to the left under the previous character(s). if a line starts with a "j" or "g" this letter would either have to be pushed to the right to be able to see all of it, e.g. using spaces (which would ruin a multi line text which is left aligned) - or by clipping the first letter (which could ruin the nice graphics). The solution is to change

```
textarea.setPosition(50, 50, 100, 100);
```

to

```
textarea.setPosition(45, 50, 110, 100);
textarea.setIndentation(5);
```

Characters that do not extend to the left under the previous characters will be drawn in the same position in either case, but "j" and "g" will be aligned with other lines.

The function **Font::getMaxPixelsLeft()** will give you the maximum number of pixels any glyph in the font extends to the left.

Parameters:

indent The indentation from left (when left aligned text) and right (when right aligned text).

See also:

Font::getMaxPixelsLeft

setLinespacing

FORCE_INLINE_FUNCTION void setLinespacing (int16_t space)

Sets the line spacing of the **TextArea**.

Setting a larger value will increase the space between lines. It is possible to set a negative value to have lines (partially) overlap. Default line spacing, if not set, is 0.

Parameters:

space The line spacing of use in the **TextArea**.

See also:

getLinespacing

setRotation

void setRotation (const TextRotation rotation)

Sets rotation of the text in the **TextArea**.

The value TEXT_ROTATE_0 is the default for normal text. The value TEXT_ROTATE_90 will rotate the text clockwise, thus writing from the top of the display and down. Similarly TEXT_ROTATE_180 and TEXT_ROTATE_270 will each rotate the text further 90 degrees clockwise.

Parameters:

rotation The rotation of the text.

setTypedText

void setTypedText (TypedText t)

Sets the TypedText of the text area.

If no prior size has been set, the **TextArea** will be resized to fit the new **TypedText**.

Parameters:

t The **TypedText** for this widget to display.

See also:

resizeToCurrentText

setWideTextAction

void setWideTextAction (WideTextAction action)

Defines what to do if a line of text is wider than the text area.

Default action is **WIDE_TEXT_NONE** which means that text lines are only broken if there is a manually inserted newline in the text.

If wrapping is enabled and the text would occupy more lines than the size of the **TextArea**, the end of the last line will get an ellipsis (often) to signal that some text is missing. The character used for ellipsis is taken from the text spreadsheet.

Parameters:

action The action to perform for wide lines of text.

See also:

WideTextAction, **getWideTextAction**, **resizeHeightToCurrentText**

setXBaselineY

```
virtual void setXBaselineY ( int16_t x ,
                             int16_t baselineY
                             )
```

Adjusts the **TextArea** x and y coordinates so the text will have its baseline at the specified y value.

The placements is relative to the specified **TypedText** so if the **TypedText** is changed you have to set the baseline again. The specified x coordinate will be used as the x coordinate of the **TextArea**.

Parameters:

x	The x coordinate of the **TextArea**.
baselineY	The y coordinate of the baseline of the text.

NOTE

setTypedText() must be called prior to setting the baseline.

TextArea

```
TextArea ( )
```

Protected Attributes Documentation

alpha

uint8_t alpha

The alpha to use.

color

colortype color

The color to use for the text.

indentation

uint8_t indentation

The indentation of the text inside the text area.

linespace

int16_t linespace

The extra space between lines of text, measured in pixels.

rotation

TextRotation rotation

The text rotation to use in steps of 90 degrees.

typedText

TypedText typedText

The TypedText to display.

wideTextAction

WideTextAction wideTextAction

What to do if the lines of text are wider than the text area.

Version: 4.16

TextAreaWithOneWildcard

TextArea with one wildcard. The format string (i.e. the TypedText set in setTypedText()) is expected to contain a wildcard <placeholder> from the text.

Note: the text converter tool converts the <...> to ascii value 2 which is then being replaced by a wildcard text.

Inherits from: TextArea, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual int16_t	**getTextHeight**()
	Gets the total height needed by the text, taking number of lines and line spacing into consideration.
virtual uint16_t	**getTextWidth**() const
	Gets the width in pixels of the current associated text in the current selected language.
const **Unicode::UnicodeChar** *	**getWildcard**() const
	Gets the wildcard used in the TypedText as previously set using setWildcard().
void	**setWildcard**(const **Unicode::UnicodeChar** * value)
	Sets the wildcard used in the TypedText where <placeholder> is placed.
	TextAreaWithOneWildcard()

Protected Attributes

const **Unicode::UnicodeChar** *	**wildcard**
	Pointer to the wildcard string. Must be null-terminated.

Additional inherited members

Public Functions inherited from TextArea

virtual int16_t	**calculateTextHeight**(const **Unicode::UnicodeChar** * format, ...) const
	Gets the total height needed by the text.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
FORCE_INLINE_FUNCTION **colortype**	**getColor**() const
	Gets the color of the text.
FORCE_INLINE_FUNCTION uint8_t	**getIndentation**()
	Gets the indentation of text inside the **TextArea**.
FORCE_INLINE_FUNCTION int16_t	**getLinespacing**() const
	Gets the line spacing of the **TextArea**.
TextRotation	**getRotation**() const
	Gets rotation of the text in the **TextArea**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
TypedText	**getTypedText**() const
	Gets the TypedText of the text area.
WideTextAction	**getWideTextAction**() const
	Gets wide text action previously set using setWideTextAction.
void	**resizeHeightToCurrentText**()
	Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.
void	**resizeHeightToCurrentTextWithRotation**()
	Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.
void	**resizeToCurrentText**()

	Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language.
void	**resizeToCurrentTextWithAlignment**()
	Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language, and for centered and right aligned text, the position of the **TextArea** widget is also updated to keep the text in the same position on the display.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBaselineY**(int16_t baselineY)
	Adjusts the **TextArea** y coordinate so the text will have its baseline at the specified value.
FORCE_INLINE_FUNCTION void	**setColor**(**colortype** color)
	Sets the color of the text.
FORCE_INLINE_FUNCTION void	**setIndentation**(uint8_t indent)
	Sets the indentation for the text.
FORCE_INLINE_FUNCTION void	**setLinespacing**(int16_t space)
	Sets the line spacing of the **TextArea**.
void	**setRotation**(const **TextRotation** rotation)
	Sets rotation of the text in the **TextArea**.
void	**setTypedText**(**TypedText** t)
	Sets the TypedText of the text area.
void	**setWideTextAction**(**WideTextAction** action)
	Defines what to do if a line of text is wider than the text area.
virtual void	**setXBaselineY**(int16_t x, int16_t baselineY)
	Adjusts the **TextArea** x and y coordinates so the text will have its baseline at the specified y value.
	TextArea()

Protected Attributes inherited from TextArea

uint8_t	**alpha**
	The alpha to use.
colortype	**color**
	The color to use for the text.
uint8_t	**indentation**
	The indentation of the text inside the text area.
int16_t	**linespace**
	The extra space between lines of text, measured in pixels.
TextRotation	**rotation**
	The text rotation to use in steps of 90 degrees.
TypedText	**typedText**
	The TypedText to display.
WideTextAction	**wideTextAction**
	What to do if the lines of text are wider than the text area.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.

virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)

	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**

	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::TextArea::draw**

getTextHeight

virtual int16_t getTextHeight ( )

Gets the total height needed by the text, taking number of lines and line spacing into consideration.

Returns:

the total height needed by the text.

Reimplements: **touchgfx::TextArea::getTextHeight**

getTextWidth

virtual uint16_t getTextWidth ( ) const

Gets the width in pixels of the current associated text in the current selected language.

In case of multi-lined text the width of the widest line is returned.

Returns:

The width in pixels of the current text.

Reimplements: **touchgfx::TextArea::getTextWidth**

getWildcard

const Unicode::UnicodeChar * getWildcard ( ) const

Gets the wildcard used in the TypedText as previously set using setWildcard().

Returns:

The wildcard used in the text.

setWildcard

void setWildcard (const Unicode::UnicodeChar * value)

Sets the wildcard used in the TypedText where <placeholder> is placed.

Wildcard string must be a null-terminated UnicodeChar array.

Parameters:

value A pointer to the UnicodeChar to set the wildcard to.

NOTE

The pointer passed is saved, and must be accessible whenever TextAreaWithOneWildcard may need it.

TextAreaWithOneWildcard

TextAreaWithOneWildcard ( )

Protected Attributes Documentation

wildcard

const Unicode::UnicodeChar * wildcard

Pointer to the wildcard string. Must be null-terminated.

Version: 4.16

TextAreaWithTwoWildcards

TextArea with two wildcards. The format string (i.e. the TypedText set in setTypedText()) is expected to contain two wildcards <placeholders> from the text.

Note: the text converter tool converts the <...> to ascii value 2 which is what is being replaced by a wildcard text.

Inherits from: TextArea, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual int16_t	**getTextHeight**()
	Gets the total height needed by the text, taking number of lines and line spacing into consideration.
virtual uint16_t	**getTextWidth**() const
	Gets the width in pixels of the current associated text in the current selected language.
const **Unicode::UnicodeChar** *	**getWildcard1**() const
	Gets the first wildcard used in the TypedText as previously set using setWildcard1().
const **Unicode::UnicodeChar** *	**getWildcard2**() const
	Gets the second wildcard used in the TypedText as previously set using setWildcard1().
void	**setWildcard1**(const **Unicode::UnicodeChar** * value)
	Sets the wildcard used in the TypedText where first <placeholder> is placed.
void	**setWildcard2**(const **Unicode::UnicodeChar** * value)
	Sets the wildcard used in the TypedText where second <placeholder> is placed.
	TextAreaWithTwoWildcards()

Protected Attributes

const **Unicode::UnicodeChar** *	**wc1**
	Pointer to the first wildcard string. Must be null-terminated.
const **Unicode::UnicodeChar** *	**wc2**
	Pointer to the second wildcard string. Must be null-terminated.

Additional inherited members

Public Functions inherited from TextArea

virtual int16_t	**calculateTextHeight**(const **Unicode::UnicodeChar** * format, ...) const
	Gets the total height needed by the text.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
FORCE_INLINE_FUNCTION **colortype**	**getColor**() const
	Gets the color of the text.
FORCE_INLINE_FUNCTION uint8_t	**getIndentation**()
	Gets the indentation of text inside the **TextArea**.
FORCE_INLINE_FUNCTION int16_t	**getLinespacing**() const
	Gets the line spacing of the **TextArea**.
TextRotation	**getRotation**() const
	Gets rotation of the text in the **TextArea**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
TypedText	**getTypedText**() const
	Gets the TypedText of the text area.
WideTextAction	**getWideTextAction**() const
	Gets wide text action previously set using setWideTextAction.

void	**resizeHeightToCurrentText**()
	Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.
void	**resizeHeightToCurrentTextWithRotation**()
	Sets the height of the **TextArea** to match the height of the current associated text for the current selected language.
void	**resizeToCurrentText**()
	Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language.
void	**resizeToCurrentTextWithAlignment**()
	Sets the dimensions of the **TextArea** to match the width and height of the current associated text for the current selected language, and for centered and right aligned text, the position of the **TextArea** widget is also updated to keep the text in the same position on the display.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setBaselineY**(int16_t baselineY)
	Adjusts the **TextArea** y coordinate so the text will have its baseline at the specified value.
FORCE_INLINE_FUNCTION void	**setColor**(**colortype** color)
	Sets the color of the text.
FORCE_INLINE_FUNCTION void	**setIndentation**(uint8_t indent)
	Sets the indentation for the text.
FORCE_INLINE_FUNCTION void	**setLinespacing**(int16_t space)
	Sets the line spacing of the **TextArea**.
void	**setRotation**(const **TextRotation** rotation)
	Sets rotation of the text in the **TextArea**.
void	**setTypedText**(**TypedText** t)
	Sets the TypedText of the text area.
void	**setWideTextAction**(**WideTextAction** action)
	Defines what to do if a line of text is wider than the text area.

virtual void	**setXBaselineY**(int16_t x, int16_t baselineY)
	Adjusts the **TextArea** x and y coordinates so the text will have its baseline at the specified y value.
	TextArea()

Protected Attributes inherited from TextArea

uint8_t	**alpha**
	The alpha to use.
colortype	**color**
	The color to use for the text.
uint8_t	**indentation**
	The indentation of the text inside the text area.
int16_t	**linespace**
	The extra space between lines of text, measured in pixels.
TextRotation	**rotation**
	The text rotation to use in steps of 90 degrees.
TypedText	**typedText**
	The TypedText to display.
WideTextAction	**wideTextAction**
	What to do if the lines of text are wider than the text area.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()

	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.

int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.

void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.

virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::TextArea::draw**

getTextHeight

virtual int16_t getTextHeight ( )

Gets the total height needed by the text, taking number of lines and line spacing into consideration.

Returns:

the total height needed by the text.

Reimplements: **touchgfx::TextArea::getTextHeight**

getTextWidth

virtual uint16_t getTextWidth ( ) const

Gets the width in pixels of the current associated text in the current selected language.

In case of multi-lined text the width of the widest line is returned.

Returns:

The width in pixels of the current text.

Reimplements: **touchgfx::TextArea::getTextWidth**

getWildcard1

const Unicode::UnicodeChar * getWildcard1 ( ) const

Gets the first wildcard used in the TypedText as previously set using setWildcard1().

Returns:

The first wildcard used in the text.

getWildcard2

const Unicode::UnicodeChar * getWildcard2 ( ) const

Gets the second wildcard used in the TypedText as previously set using setWildcard1().

Returns:

The second wildcard used in the text.

setWildcard1

void setWildcard1 (const Unicode::UnicodeChar * value)

Sets the wildcard used in the TypedText where first <placeholder> is placed.

Wildcard string must be a null-terminated UnicodeChar array.

Parameters:

value A pointer to the UnicodeChar to set the wildcard to.

NOTE

The pointer passed is saved, and must be accessible whenever TextAreaWithTwoWildcard may need it.

setWildcard2

void setWildcard2 (const Unicode::UnicodeChar * value)

Sets the wildcard used in the TypedText where second <placeholder> is placed.

Wildcard string must be a null-terminated UnicodeChar array.

Parameters:

value A pointer to the UnicodeChar to set the wildcard to.

NOTE

The pointer passed is saved, and must be accessible whenever TextAreaWithTwoWildcard may need it.

TextAreaWithTwoWildcards

TextAreaWithTwoWildcards ( )

Protected Attributes Documentation

wc1

const Unicode::UnicodeChar * wc1

Pointer to the first wildcard string. Must be null-terminated.

wc2

const Unicode::UnicodeChar * wc2

Pointer to the second wildcard string. Must be null-terminated.

Version: 4.16

TextButtonStyle

A text button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show a text in one of two colors depending on the state of the button (pressed or released).

The TextButtonStyle does not set the size of the enclosing container (normally AbstractButtonContainer). The size must be set manually.

To get a background behind the text, use TextButtonStyle together with e.g. ImageButtonStyle: TextButtonStyle<ImageButtonStyle<ClickButtonTrigger> > myButton;

The position of the text can be adjusted with setTextXY (default is centered).

See: AbstractButtonContainer

Inherits from: T

Public Functions

void	**setText**(**TypedText** t)
	Sets a text.
void	**setTextColors**(**colortype** newColorReleased, **colortype** newColorPressed)
	Sets text colors.
void	**setTextPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets text position.
void	**setTextRotation**(**TextRotation** rotation)
	Sets text rotation.
void	**setTextX**(int16_t x)
	Sets text x coordinate.
void	**setTextXY**(int16_t x, int16_t y)
	Sets text x and y.
void	**setTextY**(int16_t y)
	Sets text y coordinate.

	TextButtonStyle()

Protected Functions

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

colortype	**colorPressed**
	The color pressed.
colortype	**colorReleased**
	The color released.
TextArea	**text**
	The text.

Public Functions Documentation

setText

void setText (TypedText t)

Sets a text.

Parameters:

t A **TypedText** to process.

setTextColors

void setTextColors (colortype newColorReleased ,

```
                colortype newColorPressed
            )
```

Sets text colors.

Parameters:

newColorReleased	The new color released.
newColorPressed	The new color pressed.

setTextPosition

```
void setTextPosition ( int16_t x ,
                       int16_t y ,
                       int16_t width ,
                       int16_t height
                       )
```

Sets text position.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the text.
height	The height of the text.

setTextRotation

```
void setTextRotation ( TextRotation rotation )
```

Sets text rotation.

Parameters:

rotation	The rotation.

setTextX

```
void setTextX ( int16_t x )
```

Sets text x coordinate.

Parameters:

x	The x coordinate.

setTextXY

```
void setTextXY ( int16_t x ,
                 int16_t y
                 )
```

Sets text x and y.

Parameters:

x The x coordinate.
y The y coordinate.

setTextY

```
void setTextY ( int16_t y )
```

Sets text y coordinate.

Parameters:

y The y coordinate.

TextButtonStyle

```
TextButtonStyle (  )
```

Protected Functions Documentation

handleAlphaUpdated

```
virtual void handleAlphaUpdated (  )
```

Handles what should happen when the alpha is updated.

handlePressedUpdated

```
virtual void handlePressedUpdated (  )
```

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

colorPressed

colortype colorPressed

The color pressed.

colorReleased

colortype colorReleased

The color released.

text

TextArea text

The text.

Version: 4.16

TextProgress

A text progress will display progress as a number with a given number of decimals.

Note: The implementation does not use floating point variables to calculate the progress.

Inherits from: AbstractProgressIndicator, Container, Drawable

Public Functions

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual **colortype**	**getColor**() const
	Gets the color of the text in the used text area.
virtual uint16_t	**getNumberOfDecimals**() const
	Gets number of decimals.
virtual **TypedText**	**getTypedText**() const
	Gets the typed text.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setColor**(**colortype** color)
	Sets the color of the text in the used text area.
virtual void	**setNumberOfDecimals**(uint16_t numberOfDecimals)
	Sets number of decimals when displaying progress.
virtual void	**setProgressIndicatorPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the position and dimensions of the text progress indicator.
virtual void	**setTypedText**(const **TypedText** & t)
	Sets the typed text.
virtual void	**setValue**(int value)
	Sets the new value for the progress indicator.

	TextProgress()

Protected Attributes

uint16_t	**decimals**
	The number of decimals.
TextAreaWithOneWildcard	**textArea**
	The text area.
Unicode::UnicodeChar	**textBuffer**
	Room for 100.0000.

Additional inherited members

Public Functions inherited from AbstractProgressIndicator

	AbstractProgressIndicator()
	Initializes a new instance of the **AbstractProgressIndicator** class with a default range 0-100.
virtual uint16_t	**getProgress**(uint16_t range =100) const
	Gets the current progress based on the range set by setRange() and the value set by setValue().
virtual int16_t	**getProgressIndicatorHeight**() const
	Gets progress indicator height.
virtual int16_t	**getProgressIndicatorWidth**() const
	Gets progress indicator width.
virtual int16_t	**getProgressIndicatorX**() const
	Gets progress indicator x coordinate.
virtual int16_t	**getProgressIndicatorY**() const
	Gets progress indicator y coordinate.
virtual void	**getRange**(int & min, int & max) const
	Gets the range set by setRange().

virtual void	**getRange**(int & min, int & max, uint16_t & steps) const
	Gets the range set by setRange().
virtual void	**getRange**(int & min, int & max, uint16_t & steps, uint16_t & minStep) const
	Gets the range set by setRange().
virtual int	**getValue**() const
	Gets the current value set by setValue().
virtual void	**handleTickEvent**()
	Called periodically by the framework if the Drawable instance has subscribed to timer ticks.
virtual void	**setBackground**(const **Bitmap** & bitmapBackground)
	Sets the background image.
virtual void	**setEasingEquation**(**EasingEquation** easingEquation)
	Sets easing equation to be used in updateValue.
virtual void	**setRange**(int min, int max, uint16_t steps =0, uint16_t minStep =0)
	Sets the range for the progress indicator.
void	**setValueSetAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered every time a new value is assigned to the progress indicator.
void	**setValueUpdatedAction**(**GenericCallback**< const **AbstractProgressIndicator** & > & callback)
	Sets callback that will be triggered when updateValue has finished animating to the final value.
virtual void	**updateValue**(int value, uint16_t duration)
	Update the current value in the range (min..max) set by **setRange()**.

Protected Attributes inherited from AbstractProgressIndicator

int	**animationDuration**
	Duration of the animation.
int	**animationEndValue**
	The animation end value.

int	**animationStartValue**
	The animation start value.
int	**animationStep**
	The current animation step.
Image	**background**
	The background image.
int	**currentValue**
	The current value.
EasingEquation	**equation**
	The equation used in updateValue()
Container	**progressIndicatorContainer**
	The container that holds the actual progress indicator.
int	**rangeMax**
	The range maximum.
int	**rangeMin**
	The range minimum.
uint16_t	**rangeSteps**
	The range steps.
uint16_t	**rangeStepsMin**
	The range steps minimum.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueSetCallback**
	New value assigned Callback.
GenericCallback< const **AbstractProgressIndicator** & > *	**valueUpdatedCallback**
	Animation ended Callback.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.

	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const

	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const

	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.

void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getColor

virtual colortype getColor ( ) const

Gets the color of the text in the used text area.

Returns:

The color.

getNumberOfDecimals

virtual uint16_t getNumberOfDecimals ( ) const

Gets number of decimals.

Returns:

The number of decimals.

See also:

setNumberOfDecimals

getTypedText

virtual TypedText getTypedText ( ) const

Gets the typed text.

Returns:

The typed text.

See also:

setTypedText

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setColor

virtual void setColor (colortype color)

Sets the color of the text in the used text area.

Parameters:

color The color.

See also:

getColor, **TextArea::setColor**

setNumberOfDecimals

virtual void setNumberOfDecimals (uint16_t numberOfDecimals)

Sets number of decimals when displaying progress.

Parameters:

numberOfDecimals Number of decimals. Only up to two decimals is supported.

See also:

getNumberOfDecimals

setProgressIndicatorPosition

```
virtual void setProgressIndicatorPosition ( int16_t x ,
                                            int16_t y ,
                                            int16_t width ,
                                            int16_t height
                                            )
```

Sets the position and dimensions of the text progress indicator.

Sets the position and dimensions of the text progress indicator relative to the background image.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the text progress indicator.
height	The height of the text progress indicator.

Reimplements: **touchgfx::AbstractProgressIndicator::setProgressIndicatorPosition**

setTypedText

```
virtual void setTypedText ( const TypedText & t )
```

Sets the typed text.

The text should have exactly one wildcard and could for example look like this: "<progress>\%".

Parameters:

t	The **TypedText** to process.

See also:

getTypedText

setValue

```
virtual void setValue ( int value )
```

Sets the new value for the progress indicator.

Parameters:

value	The value.

Reimplements: **touchgfx::AbstractProgressIndicator::setValue**

TextProgress

```
TextProgress ( )
```

Protected Attributes Documentation

decimals

uint16_t decimals

The number of decimals.

textArea

TextAreaWithOneWildcard textArea

The text area.

textBuffer

Unicode::UnicodeChar textBuffer

Room for 100.0000.

Version: 4.16

TextProvider

The TextProvider is used in drawing basic strings and strings with one or two wildcards. The TextProvider enables wildcard expansion of the string at the time it is written to the LCD.

Wildcards specified as <placeholder> are converted to Unicode value 2 by the text converter tool, and the placeholders are automatically expanded with the specified wildcard buffers at runtime.

Public Functions

bool	**endOfString**()
	Tells if the end of the string has been reached.
Unicode::UnicodeChar	**getNextChar**()
	Gets the next character.
Unicode::UnicodeChar	**getNextLigature**(**TextDirection** direction)
	Gets the next ligature.
Unicode::UnicodeChar	**getNextLigature**(**TextDirection** direction, const **Font** *font, const* ***GlyphNode*** & glyph)
	Gets the next ligature.
Unicode::UnicodeChar	**getNextLigature**(**TextDirection** direction, const **Font** *font, const* ***GlyphNode*** & glyph, const uint8_t *& pixelData, uint8_t & bitsPerPixel)
	Gets the next ligature.
void	**initialize**(const **Unicode::UnicodeChar** *stringFormat, const uint16_t* gsubTable =0, ...)
	Initializes the **TextProvider**.
void	**initialize**(const **Unicode::UnicodeChar** *stringFormat, va_list pArg, const uint16_t* gsubTable =0)
	Initializes the **TextProvider**.
	TextProvider()
	Initializes a new instance of the **TextProvider** class.

Public Attributes

const uint32_t	**MAX_32BIT_INTEGER_DIGITS**
	Max number of digits used for the text representation of a 32 bit integer.

Public Functions Documentation

endOfString

bool endOfString ( )

Tells if the end of the string has been reached.

Returns:

True if the end of the string has been reached, false if not.

See also:

TextProvider::getNextLigature()

getNextChar

Unicode::UnicodeChar getNextChar ( )

Gets the next character.

For Arabic and Thai, it is important to use the getNextLigature instead.

Returns:

The next character of the expanded string or 0 if end of string is reached.

See also:

TextProvider::getNextLigature

getNextLigature

Unicode::UnicodeChar getNextLigature (TextDirection direction)

Gets the next ligature.

For most languages this is simply the next **Unicode** character from the buffer, but e.g. Arabic has different ligatures for each character. Thai character placement might also depend on previous characters. It is recommended to use getNextLigature with font and glyph parameters to ensure coming glyphs in a text are placed correctly.

Parameters:

direction	The direction.

Returns:

The next character of the expanded string or 0 if end of string is reached.

NOTE

Functions getNextLigature() and getNextChar() will advance through the same buffer and mixing the use of those functions is not recommended and may cause undesired results. Instead create two TextProviders and user getNextChar() on one and getNextLigature() on the other.

See also:

TextProvider::getNextChar

getNextLigature

```
Unicode::UnicodeChar getNextLigature ( TextDirection       direction ,
                                       const Font *        font ,
                                       const GlyphNode *&  glyph
                                       )
```

Gets the next ligature.

For most languages this is simply the next **Unicode** character from the buffer, but e.g. Arabic has different ligatures for each character.

Also gets a glyph for the ligature in a font. For non-Thai Unicodes, this is identical to using **Font::getGlyph()**, but for Thai characters where diacritics glyphs are not always placed at the same relative position, an adjusted **GlyphNode** will be generated with correct relative X/Y coordinates.

Parameters:

direction	The direction.
font	The font.
glyph	The glyph.

Returns:

The next character of the expanded string or 0 if end of string i reached.

NOTE

Functions **getNextLigature()** and **getNextChar()** will advance through the same buffer and mixing the use of those functions is not recommended and may cause undesired results. Instead create two TextProviders and user **getNextChar()** on one and **getNextLigature()** on the other.

See also:

TextProvider::getNextChar, **Font::getGlyph**

getNextLigature

```
Unicode::UnicodeChar getNextLigature ( TextDirection          direction ,
                                       const Font *           font ,
                                       const GlyphNode *&     glyph ,
                                       const uint8_t *&       pixelData ,
                                       uint8_t &              bitsPerPixel
                                       )
```

Gets the next ligature.

For most languages this is simply the next **Unicode** character from the buffer, but e.g. Arabic has different ligatures for each character.

Also gets a glyph for the ligature in a font. For non-Thai Unicodes, this is identical to using **Font::getGlyph()**, but for Thai characters where diacritics glyphs are not always placed at the same relative position, an adjusted **GlyphNode** will be generated with correct relative X/Y coordinates.

Furthermore a pointer to the glyph data and the bit depth of the font are returned in parameters.

Parameters:

direction	The direction.
font	The font.
glyph	The glyph.
pixelData	Information describing the pixel.
bitsPerPixel	The bits per pixel.

Returns:

The next character of the expanded string or 0 if end of string is reached.

NOTE

Functions **getNextLigature()** and **getNextChar()** will advance through the same buffer and mixing the use of those functions is not recommended and may cause undesired results. Instead create two TextProviders and user **getNextChar()** on one and **getNextLigature()** on the other.

See also:

TextProvider::getNextChar, **Font::getGlyph**

initialize

```
void initialize ( const Unicode::UnicodeChar * stringFormat ,
                  const uint16_t *             gsubTable =0,
                  ...
                  )
```

Initializes the **TextProvider**.

Each '\2' character in the format is replaced by one UnicodeChar* argument from pArg.

Parameters:

stringFormat	The string to format.
gsubTable	(Optional) Pointer to GSUB table with **Unicode** substitution rules.
...	Variable arguments providing additional information.

initialize

```
void initialize ( const Unicode::UnicodeChar * stringFormat ,
                  va_list                      pArg ,
                  const uint16_t *             gsubTable =0
                  )
```

Initializes the **TextProvider**.

Each '\2' character in the format is replaced by one UnicodeChar* argument from pArg.

Parameters:

stringFormat	The string to format.
pArg	Format arguments in the form of a va_list.
gsubTable	(Optional) Pointer to GSUB table with **Unicode** substitution rules.

TextProvider

```
TextProvider (  )
```

Initializes a new instance of the **TextProvider** class.

NOTE

The user must call **initialize()** before characters can be provided.

Public Attributes Documentation

MAX_32BIT_INTEGER_DIGITS

const uint32_t MAX_32BIT_INTEGER_DIGITS = 33U

Max number of digits used for the text representation of a 32 bit integer.

Version: 4.16

Texts

Class for setting language and getting texts. The language set will determine which texts will be used in the application.

Public Functions

LanguageId	**getLanguage**()
	Gets the current language.
void	**setLanguage**(**LanguageId** id)
	Sets the current language for texts.
void	**setTranslation**(**LanguageId** id, const void * translation)
	Adds or replaces a translation.
const **Unicode::UnicodeChar** *	**getText**(**TypedTextId** id) const
	Get text in the set language.

Public Functions Documentation

getLanguage

static LanguageId getLanguage ( )

Gets the current language.

Returns:

The id of the language.

setLanguage

static void setLanguage (LanguageId id)

Sets the current language for texts.

Parameters:

id The id of the language.

setTranslation

```
static void setTranslation ( LanguageId id ,
                             const void * translation
                             )
```

Adds or replaces a translation.

This function allows an application to add a translation at runtime.

Parameters:

id	The id of the language to add or replace.
translation	A pointer to the translation in flash or RAM.

getText

```
const Unicode::UnicodeChar * getText ( TypedTextId id )
```

Get text in the set language.

Parameters:

id The id of the text to lookup.

Returns:

The text.

See also:

setLanguage

Version: 4.16

TextureMapper

The TextureMapper widget displays a transformed image. It can be used to generate effects where an image should be rotated in two or three dimensions.

The image can be freely scaled and rotated in three dimensions. The scaling and rotation is done around the adjustable origin. A virtual camera is applied to the rendered image yielding a perspective impression. The amount of perspective impression can be adjusted. The transformed image is clipped according to the dimensions of the TextureMapper widget. In order to make the image fully visible the TextureMapper should be large enough to accommodate the transformed image, which may be larger than the raw image.

See: Widget

Note:

- The drawing of this widget is not trivial and typically has a significant performance penalty. The number of pixels drawn, the presence of global alpha or per pixel alpha inflicts the computation and should be considered.
- This widget does not support 1 bit per pixel color depth.

Inherits from: Image, Widget, Drawable

Inherited by: AnimationTextureMapper

Public Types

enum	**RenderingAlgorithm** { NEAREST_NEIGHBOR, BILINEAR_INTERPOLATION }
	Rendering algorithm to use when scaling the bitmap.

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual float	**getBitmapPositionX**() const
	Gets bitmap position x coordinate.

virtual float	**getBitmapPositionY**() const
	Gets bitmap position y coordinate.
virtual float	**getCameraDistance**() const
	Gets camera distance.
virtual float	**getCameraX**() const
	Gets camera x coordinate.
virtual float	**getCameraY**() const
	Gets camera y coordinate.
virtual float	**getOrigoX**() const
	Gets transformation origo x coordinate.
virtual float	**getOrigoY**() const
	Gets transformation origo y coordinate.
virtual float	**getOrigoZ**() const
	Gets transformation origo z coordinate.
virtual **RenderingAlgorithm**	**getRenderingAlgorithm**() const
	Gets the algorithm used when rendering.
virtual float	**getScale**() const
	Gets the scale of the image.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual float	**getX0**() const
	Get the x coordinate of the top left corner of the transformed bitmap.
virtual float	**getX1**() const
	Get the x coordinate of the top right corner of the transformed bitmap.
virtual float	**getX2**() const
	Get the x coordinate of the bottom right of the transformed bitmap.
virtual float	**getX3**() const
	Get the x coordinate of the bottom left corner of the transformed bitmap.

virtual float	**getXAngle**() const
	Get the x angle.
virtual float	**getY0**() const
	Get the y coordinate of the top left corner of the transformed bitmap.
virtual float	**getY1**() const
	Get the y coordinate of the top right corner of the transformed bitmap.
virtual float	**getY2**() const
	Get the y coordinate of the bottom right corner of the transformed bitmap.
virtual float	**getY3**() const
	Get the y coordinate of the bottom left corner of the transformed bitmap.
virtual float	**getYAngle**() const
	Get the y angle.
virtual float	**getZ0**() const
	Get the z coordinate of the top left corner of the transformed bitmap.
virtual float	**getZ1**() const
	Get the z coordinate of the top right corner of the transformed bitmap.
virtual float	**getZ2**() const
	Get the z coordinate of the bottom right corner of the transformed bitmap.
virtual float	**getZ3**() const
	Get the z coordinate of the bottom left corner of the transformed bitmap.
virtual float	**getZAngle**() const
	Get the z angle.
void	**invalidateBoundingRect**() const
	Invalidate the bounding rectangle of the transformed bitmap.
virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this **TextureMapper** and updates the width and height of this widget to match those of the **Bitmap**.

virtual void	**setBitmapPosition**(float x, float y)
	Sets the position of the bitmap within the **TextureMapper**.
virtual void	**setBitmapPosition**(int x, int y)
	Sets the position of the bitmap within the **TextureMapper**.
virtual void	**setCamera**(float x, float y)
	Sets the camera coordinate.
virtual void	**setCameraDistance**(float d)
	Sets camera distance.
virtual void	**setOrigo**(float x, float y)
	Sets the transformation origo (center) in two dimensions.
virtual void	**setOrigo**(float x, float y, float z)
	Sets the transformation origo (center).
virtual void	**setRenderingAlgorithm**(**RenderingAlgorithm** algorithm)
	Sets the render algorithm to be used.
virtual void	**setScale**(float scale)
	Sets the scale of the image.
	TextureMapper(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **TextureMapper** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
virtual void	**updateAngles**(float newXAngle, float newYAngle, float newZAngle)
	Updates the angles of the image.
virtual void	**updateXAngle**(float newXAngle)
	Updates the x angle.
virtual void	**updateYAngle**(float newYAngle)
	Updates the y angle.
virtual void	**updateZAngle**(float newZAngle)
	Updates the z angle.

Protected Functions

void	**applyTransformation**()
	Transform the bitmap using the supplied origo, scale, rotation and camera.
void	**drawTriangle**(const **Rect** & invalidatedArea, uint16_t *fb, const float* triangleXs, const float *triangleYs, const float* triangleZs, const float *triangleUs, const float* triangleVs) const
	The **TextureMapper** will draw the transformed bitmap by drawing two triangles.
Rect	**getBoundingRect**() const
	Gets bounding rectangle of the transformed bitmap.
RenderingVariant	**lookupRenderVariant**() const
	Returns the rendering variant based on the bitmap format, alpha value and rendering algorithm.

Protected Attributes

float	**cameraDistance**
	The camera distance.
RenderingAlgorithm	**currentRenderingAlgorithm**
	The current rendering algorithm.
float	**imageX0**
	The coordinate for the image points.
float	**imageX1**
	The coordinate for the image points.
float	**imageX2**
	The coordinate for the image points.
float	**imageX3**
	The coordinate for the image points.
float	**imageY0**
	The coordinate for the image points.
float	**imageY1**
	The coordinate for the image points.
float	**imageY2**

The coordinate for the image points.

float **imageY3**

The coordinate for the image points.

float **imageZ0**

The coordinate for the image points.

float **imageZ1**

The coordinate for the image points.

float **imageZ2**

The coordinate for the image points.

float **imageZ3**

The coordinate for the image points.

float **scale**

The scale.

uint16_t **subDivisionSize**

The size of the affine sub divisions.

float **xAngle**

The angle x.

float **xBitmapPosition**

The bitmap position x.

float **xCamera**

The camera x coordinate.

float **xOrigo**

The origo x coordinate.

float **yAngle**

The angle y.

float **yBitmapPosition**

The bitmap position y.

float **yCamera**

The camera y coordinate.

float	**yOrigo**
	The origo y coordinate.
float	**zAngle**
	The angle z.
float	**zOrigo**
	The origo z coordinate.
const int	**MINIMAL_CAMERA_DISTANCE**
	The minimal camera distance.

Additional inherited members

Public Functions inherited from Image

uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getBitmap**() const
	Gets the **Bitmap** currently assigned to the **Image** widget.
BitmapId	**getBitmapId**() const
	Gets the BitmapId currently assigned to the **Image** widget.
	Image(const **Bitmap** & bitmap =**Bitmap**())
	Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

Protected Attributes inherited from Image

uint8_t	**alpha**
	The Alpha for this image.
Bitmap	**bitmap**
	The **Bitmap** to display.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()

	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)

	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.

virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

RenderingAlgorithm

enum RenderingAlgorithm

Rendering algorithm to use when scaling the bitmap.

NEAREST_NEIGHBOR	Fast but not a very good image quality. Good for fast animations.
BILINEAR_INTERPOLATION	Slower but better image quality. Good for static representation of a scaled image.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Image::draw**

getBitmapPositionX

virtual float getBitmapPositionX ( ) const

Gets bitmap position x coordinate.

Returns:

The bitmap position x coordinate.

See also:

setBitmapPosition

getBitmapPositionY

virtual float getBitmapPositionY ( ) const

Gets bitmap position y coordinate.

Returns:

The bitmap position y coordinate.

See also:

setBitmapPosition

getCameraDistance

virtual float getCameraDistance ( ) const

Gets camera distance.

Returns:

The camera distance.

See also:

setCameraDistance

getCameraX

virtual float getCameraX ( ) const

Gets camera x coordinate.

Returns:

The camera x coordinate.

See also:

setCamera

getCameraY

virtual float getCameraY ( ) const

Gets camera y coordinate.

Returns:

The camera y coordinate.

See also:

setCamera

getOrigoX

virtual float getOrigoX ( ) const

Gets transformation origo x coordinate.

Returns:

The transformation origo x coordinate.

See also:

setOrigo

getOrigoY

virtual float getOrigoY ( ) const

Gets transformation origo y coordinate.

Returns:

The transformation origo y coordinate.

See also:

setOrigo

getOrigoZ

virtual float getOrigoZ ( ) const

Gets transformation origo z coordinate.

Returns:

The transformation origo z coordinate.

See also:

setOrigo

getRenderingAlgorithm

virtual RenderingAlgorithm getRenderingAlgorithm ( ) const

Gets the algorithm used when rendering.

Returns:

The algorithm used when rendering.

getScale

virtual float getScale ( ) const

Gets the scale of the image.

Returns:

The scale.

See also:

setScale

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Image::getSolidRect**

getX0

virtual float getX0 ( ) const

Get the x coordinate of the top left corner of the transformed bitmap.

Returns:

The X0 coordinate.

getX1

virtual float getX1 ( ) const

Get the x coordinate of the top right corner of the transformed bitmap.

Returns:

The X1 coordinate.

getX2

virtual float getX2 ( ) const

Get the x coordinate of the bottom right of the transformed bitmap.

Returns:

The X2 coordinate.

getX3

virtual float getX3 ( ) const

Get the x coordinate of the bottom left corner of the transformed bitmap.

Returns:

The X3 coordinate.

getXAngle

virtual float getXAngle ( ) const

Get the x angle.

Returns:

The x angle.

See also:

updateXAngle

getY0

virtual float getY0 ( ) const

Get the y coordinate of the top left corner of the transformed bitmap.

Returns:

The Y0 coordinate.

getY1

virtual float getY1 ( ) const

Get the y coordinate of the top right corner of the transformed bitmap.

Returns:

The Y1 coordinate.

getY2

virtual float getY2 ( ) const

Get the y coordinate of the bottom right corner of the transformed bitmap.

Returns:

The Y2 coordinate.

getY3

virtual float getY3 ( ) const

Get the y coordinate of the bottom left corner of the transformed bitmap.

Returns:

The Y3 coordinate.

getYAngle

virtual float getYAngle ( ) const

Get the y angle.

Returns:

The y angle.

See also:

updateYAngle

getZ0

virtual float getZ0 ( ) const

Get the z coordinate of the top left corner of the transformed bitmap.

Returns:

The Z0 coordinate.

getZ1

virtual float getZ1 ( ) const

Get the z coordinate of the top right corner of the transformed bitmap.

Returns:

The Z1 coordinate.

getZ2

virtual float getZ2 ( ) const

Get the z coordinate of the bottom right corner of the transformed bitmap.

Returns:

The Z2 coordinate.

getZ3

virtual float getZ3 ( ) const

Get the z coordinate of the bottom left corner of the transformed bitmap.

Returns:

The Z3 coordinate.

getZAngle

virtual float getZAngle ( ) const

Get the z angle.

Returns:

The z angle.

See also:

updateZAngle

invalidateBoundingRect

void invalidateBoundingRect ( ) const

Invalidate the bounding rectangle of the transformed bitmap.

See also:

getBoundingRect

setBitmap

virtual void setBitmap (const Bitmap & bitmap)

Sets the bitmap for this **TextureMapper** and updates the width and height of this widget to match those of the **Bitmap**.

Parameters:

bitmap The bitmap instance.

NOTE

The user code must call invalidate() in order to update the image on the display.

Reimplements: **touchgfx::Image::setBitmap**

setBitmapPosition

virtual void setBitmapPosition (float x ,
float y

```
)
```

Sets the position of the bitmap within the **TextureMapper**.

The bitmap is clipped with respect to the dimensions of the **TextureMapper** widget.

Parameters:

x The x coordinate.
y The y coordinate.

See also:

getBitmapPositionX, **getBitmapPositionY**

setBitmapPosition

```
virtual void setBitmapPosition ( int x ,
                                 int y
                                 )
```

Sets the position of the bitmap within the **TextureMapper**.

The bitmap is clipped with respect to the dimensions of the **TextureMapper** widget.

Parameters:

x The x coordinate.
y The y coordinate.

See also:

getBitmapPositionX, **getBitmapPositionY**

setCamera

```
virtual void setCamera ( float x ,
                         float y
                         )
```

Sets the camera coordinate.

Parameters:

x The x coordinate for the camera.
y The y coordinate for the camera.

See also:

getCameraX, **getCameraY**

setCameraDistance

virtual void setCameraDistance (float d)

Sets camera distance.

If the given value is below **TextureMapper::MINIMAL_CAMERA_DISTANCE**, it will be set to **TextureMapper::MINIMAL_CAMERA_DISTANCE**.

Parameters:

d The new camera distance.

See also:

getCameraDistance

setOrigo

virtual void setOrigo (float x ,
float y
)

Sets the transformation origo (center) in two dimensions.

Leaves the z coordinate untouched.

Parameters:

x The x coordinate.
y The y coordinate.

See also:

getOrigoX, **getOrigoY**

setOrigo

virtual void setOrigo (float x ,
float y ,
float z
)

Sets the transformation origo (center).

Parameters:

x The x coordinate.
y The y coordinate.
z The z coordinate.

See also:

getOrigoX, **getOrigoY**, **getOrigoZ**

setRenderingAlgorithm

virtual void setRenderingAlgorithm (RenderingAlgorithm algorithm)

Sets the render algorithm to be used.

Default setting is NEAREST_NEIGHBOR.

Parameters:

algorithm The algorithm to use when rendering.

setScale

virtual void setScale (float scale)

Sets the scale of the image.

Parameters:

scale The new scale value.

See also:

getScale

TextureMapper

TextureMapper (const Bitmap & bitmap =Bitmap())

Constructs a new **TextureMapper** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

If a **Bitmap** is passed to the constructor, the width and height of this widget is set to those of the bitmap.

Parameters:

bitmap (Optional) The bitmap to display.

See also:

setBitmap

updateAngles

virtual void updateAngles (float newXAngle ,
float newYAngle ,
float newZAngle
)

Updates the angles of the image.

The area covered by the image before and after changing the angles is invalidated, which is the smallest required rectangle.

Parameters:

newXAngle The new x Angle.
newYAngle The new y Angle.
newZAngle The new x Angle.

See also:

updateXAngle, **updateYAngle**, **updateZAngle**, **getXAngle**, **getYAngle**, **getZAngle**

updateXAngle

virtual void updateXAngle (float newXAngle)

Updates the x angle.

Parameters:

newXAngle The new x angle.

See also:

updateAngles, **getXAngle**

updateYAngle

virtual void updateYAngle (float newYAngle)

Updates the y angle.

Parameters:

newYAngle The new y angle.

See also:

updateAngles, **getYAngle**

updateZAngle

virtual void updateZAngle (float newZAngle)

Updates the z angle.

Parameters:

newZAngle The new z angle.

See also:

updateAngles, **getZAngle**

Protected Functions Documentation

applyTransformation

void applyTransformation ( )

Transform the bitmap using the supplied origo, scale, rotation and camera.

This method is called by all the methods that manipulate origo, scale, rotation and camera.

drawTriangle

```
void drawTriangle ( const Rect & invalidatedArea ,
                    uint16_t *    fb ,
                    const float * triangleXs ,
                    const float * triangleYs ,
                    const float * triangleZs ,
                    const float * triangleUs ,
                    const float * triangleVs
                    ) const
```

The **TextureMapper** will draw the transformed bitmap by drawing two triangles.

One triangle is created from the points 0,1,2 and the other triangle from the points 1,2,3. The triangle is drawn using the x,y,z values from each point along with the u,v coordinates in the bitmap associated with each point.

Parameters:

invalidatedArea	The invalidated area.
fb	The framebuffer.
triangleXs	The triangle xs.
triangleYs	The triangle ys.
triangleZs	The triangle zs.
triangleUs	The triangle us.
triangleVs	The triangle vs.

getBoundingRect

Rect getBoundingRect ( ) const

Gets bounding rectangle of the transformed bitmap.

This is the smallest possible rectangle which covers the image of the bitmap after applying scale and rotation.

Returns:

The bounding rectangle.

lookupRenderVariant

RenderingVariant lookupRenderVariant ( ) const

Returns the rendering variant based on the bitmap format, alpha value and rendering algorithm.

Returns:

The RenderingVariant.

Protected Attributes Documentation

cameraDistance

float cameraDistance

The camera distance.

currentRenderingAlgorithm

RenderingAlgorithm currentRenderingAlgorithm

The current rendering algorithm.

imageX0

float imageX0

The coordinate for the image points.

imageX1

float imageX1

The coordinate for the image points.

imageX2

float imageX2

The coordinate for the image points.

imageX3

float imageX3

The coordinate for the image points.

imageY0

float imageY0

The coordinate for the image points.

imageY1

float imageY1

The coordinate for the image points.

imageY2

float imageY2

The coordinate for the image points.

imageY3

float imageY3

The coordinate for the image points.

imageZ0

float imageZ0

The coordinate for the image points.

imageZ1

float imageZ1

The coordinate for the image points.

imageZ2

float imageZ2

The coordinate for the image points.

imageZ3

float imageZ3

The coordinate for the image points.

scale

float scale

The scale.

subDivisionSize

uint16_t subDivisionSize

The size of the affine sub divisions.

xAngle

float xAngle

The angle x.

xBitmapPosition

float xBitmapPosition

The bitmap position x.

xCamera

float xCamera

The camera x coordinate.

xOrigo

float xOrigo

The origo x coordinate.

yAngle

float yAngle

The angle y.

yBitmapPosition

float yBitmapPosition

The bitmap position y.

yCamera

float yCamera

The camera y coordinate.

yOrigo

float yOrigo

The origo y coordinate.

zAngle

float zAngle

The angle z.

zOrigo

float zOrigo

The origo z coordinate.

MINIMAL_CAMERA_DISTANCE

const int MINIMAL_CAMERA_DISTANCE = 1

The minimal camera distance.

Version: 4.16

TextureSurface

A texture source. Contains a pointer to the data and the width and height of the texture. The alpha channel is used in 565 rendering with alpha. The stride is the width used when moving to the next line of the texture.

Public Attributes

const uint16_t *	**data**
	The pixel bits or indexes for color in CLUT entries.
const uint8_t *	**extraData**
	The alpha channel or clut data.
int32_t	**height**
	The height.
int32_t	**stride**
	The stride.
int32_t	**width**
	The width.

Public Attributes Documentation

data

const uint16_t * data

The pixel bits or indexes for color in CLUT entries.

extraData

const uint8_t * extraData

The alpha channel or clut data.

height

int32_t height

The height.

stride

int32_t stride

The stride.

width

int32_t width

The width.

Version: 4.16

TiledImage

Simple widget capable of showing a bitmap tiled indefinitely horizontally and vertically. This means that when the TiledImageWidget is larger than the provided Bitmap, the Bitmap is repeated over and over horizontally and vertically. The bitmap can be alpha- blended with the background and have areas of transparency.

Inherits from: Image, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**getOffset**(int16_t & x, int16_t & y)
	Gets the offset into the bitmap where the tile drawing should start.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual int16_t	**getXOffset**()
	Get x offset.
virtual int16_t	**getYOffset**()
	Get y coordinate offset.
virtual void	**setBitmap**(const **Bitmap** & bitmap)
	Sets the bitmap for this **Image** and updates the width and height of this widget to match those of the **Bitmap**.
virtual void	**setOffset**(int16_t x, int16_t y)
	Sets an offset into the bitmap where the tile drawing should start.
virtual void	**setXOffset**(int16_t x)
	Sets x offset into the bitmap where the tile drawing should start.
virtual void	**setYOffset**(int16_t y)
	Sets y offset into the bitmap where the tile drawing should start.
	TiledImage(const **Bitmap** & bmp =**Bitmap**())

Constructs a new **TiledImage** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

Protected Attributes

int16_t **xOffset**

The X offset into the bitmap to start drawing in range 0..bitmap.width-1.

int16_t **yOffset**

The Y offset into the bitmap to start drawing in range 0..bitmap.height-1.

Additional inherited members

Public Functions inherited from Image

uint8_t **getAlpha**() const

Gets the current alpha value of the widget.

Bitmap **getBitmap**() const

Gets the **Bitmap** currently assigned to the **Image** widget.

BitmapId **getBitmapId**() const

Gets the BitmapId currently assigned to the **Image** widget.

Image(const **Bitmap** & bitmap =**Bitmap**())

Constructs a new **Image** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

void **setAlpha**(uint8_t newAlpha)

Sets the opacity (alpha value).

Protected Attributes inherited from Image

uint8_t **alpha**

The Alpha for this image.

Bitmap **bitmap**

The **Bitmap** to display.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.

virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.

virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)

	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Image::draw**

getOffset

```
virtual void getOffset ( int16_t & x ,
                         int16_t & y
                         )
```

Gets the offset into the bitmap where the tile drawing should start.

Please note that the offsets set using setOffset have been normalized so that x is in the range 0 to bitmap width - 1, and y is in the range 0 to bitmap height - 1.

Parameters:

x The x offset.
y The y offset.

See also:

getXOffset, **getYOffset**

getSolidRect

```
virtual Rect getSolidRect (  ) const
```

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Image::getSolidRect**

getXOffset

virtual int16_t getXOffset ( )

Get x offset.

This is the value set using **setXOffset()** (or **setOffset()**) normalized to be in the range 0 to bitmap width - 1.

Returns:

The x offset.

See also:

getYOffset, **getOffset**

getYOffset

virtual int16_t getYOffset ( )

Get y coordinate offset.

This is the value set using **setYOffset()** (or **setOffset()**) normalized to be in the range 0 to bitmap height - 1.

Returns:

The y offset.

See also:

getXOffset, **getOffset**

setBitmap

virtual void setBitmap (const Bitmap & bitmap)

Sets the bitmap for this **Image** and updates the width and height of this widget to match those of the **Bitmap**.

Parameters:

bitmap The bitmap instance.

NOTE

The user code must call **invalidate()** in order to update the image on the display.

Reimplements: **touchgfx::Image::setBitmap**

setOffset

```
virtual void setOffset ( int16_t x ,
                         int16_t y
                         )
```

Sets an offset into the bitmap where the tile drawing should start.

By default the first image is aligned along the top and left, i.e. offset at (0, 0).

Parameters:

x The x coordinate offset.
y The y coordinate offset.

See also:

setXOffset, **setYOffset**

setXOffset

```
virtual void setXOffset ( int16_t x )
```

Sets x offset into the bitmap where the tile drawing should start.

Setting the x offset to 1 will push all images one pixel to the left.

Parameters:

x The x offset.

See also:

setYOffset, **setOffset**

setYOffset

```
virtual void setYOffset ( int16_t y )
```

Sets y offset into the bitmap where the tile drawing should start.

Setting the y offset to 1 will push all images one pixel up.

Parameters:

y The y offset.

See also:

setXOffset, **setOffset**

TiledImage

TiledImage (const Bitmap & bmp =Bitmap())

Constructs a new **TiledImage** with a default alpha value of 255 (solid) and a default **Bitmap** (undefined) if none is specified.

If a **Bitmap** is passed to the constructor, the width and height of this widget is set to those of the bitmap.

Parameters:

bmp (Optional) The bitmap to display.

See also:

setBitmap

Protected Attributes Documentation

xOffset

int16_t xOffset

The X offset into the bitmap to start drawing in range 0..bitmap.width-1.

yOffset

int16_t yOffset

The Y offset into the bitmap to start drawing in range 0..bitmap.height-1.

Version: 4.16

TiledImageButtonStyle

A tiled image button style. An tiled image button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show one of two tiled images depending on the state of the button (pressed or released).

The TiledImageButtonStyle does not set the size of the enclosing container (normally AbstractButtonContainer) to the size of the pressed Bitmap. This can be overridden by calling setWidth/setHeight after setting the bitmaps.

Template Parameters:

- **T** Generic type parameter. Typically a AbstractButtonContainer subclass.

See: AbstractButtonContainer

Inherits from: T

Public Functions

virtual void	**setHeight**(int16_t height)
	Sets height.
virtual void	**setTileBitmaps**(const **Bitmap** & bmpReleased, const **Bitmap** & bmpPressed)
	Sets tile bitmaps.
virtual void	**setTileOffset**(int16_t x, int16_t y)
	Sets an offset into the bitmap where the tile drawing should start.
virtual void	**setWidth**(int16_t width)
	Sets width.
	TiledImageButtonStyle()

Protected Functions

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.

virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes

Bitmap	**downTile**
	The image to display when button is pressed.
TiledImage	**tiledImage**
	The tiled image.
Bitmap	**upTile**
	The image to display when button is released.

Public Functions Documentation

setHeight

virtual void setHeight (int16_t height)

Sets height.

Parameters:

height The height.

setTileBitmaps

virtual void setTileBitmaps (const Bitmap & bmpReleased ,
const Bitmap & bmpPressed
)

Sets tile bitmaps.

Parameters:

bmpReleased The bitmap released.
bmpPressed The bitmap pressed.

setTileOffset

virtual void setTileOffset (int16_t x ,
int16_t y
)

Sets an offset into the bitmap where the tile drawing should start.

Parameters:

x The x coordinate offset.
y The y coordinate offset.

See also:

TiledImage::setOffset

setWidth

virtual void setWidth (int16_t width)

Sets width.

Parameters:

width The width.

TiledImageButtonStyle

TiledImageButtonStyle ( )

Protected Functions Documentation

handleAlphaUpdated

virtual void handleAlphaUpdated ( )

Handles what should happen when the alpha is updated.

handlePressedUpdated

virtual void handlePressedUpdated ( )

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

downTile

Bitmap downTile

The image to display when button is pressed.

tiledImage

TiledImage tiledImage

The tiled image.

upTile

Bitmap upTile

The image to display when button is released.

Version: 4.16

ToggleButton

A ToggleButton is a Button specialization that swaps the two bitmaps when clicked, such that the previous "pressed" bitmap, now becomes the one displayed when button is not pressed. This can by used to give the effect of a button that can be pressed in and when it is subsequently pressed, it will pop back out.

Inherits from: Button, AbstractButton, Widget, Drawable

Public Functions

void	**forceState**(bool activeState)
	Allows the **ToggleButton** to be forced into either the pressed state, or the normal state.
bool	**getState**() const
	Gets the state of the **ToggleButton** as set with forceState.
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
virtual void	**setBitmaps**(const **Bitmap** & bitmapReleased, const **Bitmap** & bitmapPressed)
	Sets the two bitmaps used by this button.

Protected Attributes

Bitmap	**originalPressed**
	Contains the bitmap that was originally being displayed when button is pressed.

Additional inherited members

Public Functions inherited from Button

	Button()
virtual void	**draw**(const **Rect** & invalidatedArea) const

	Draw this drawable.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
Bitmap	**getCurrentlyDisplayedBitmap**() const
	Gets currently displayed bitmap.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).

Protected Attributes inherited from Button

uint8_t	**alpha**
	The current alpha value. 255=solid, 0=invisible.
Bitmap	**down**
	The image to display when button is pressed.
Bitmap	**up**
	The image to display when button is released (normal state).

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.

Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.

bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)

	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

forceState

void forceState (bool activeState)

Allows the **ToggleButton** to be forced into either the pressed state, or the normal state.

In the pressed state, the **Button** will always be shown as pressed down (and shown as released when the user presses it). In the normal state, the **Button** will be show as released or pressed depending on its actual state.

Parameters:

activeState If true, swap the images for released and pressed. If false display the **Button** normally.

getState

bool getState ( ) const

Gets the state of the **ToggleButton** as set with forceState.

Returns:

True if the button has been toggled, i.e. the pressed state is shown when the button is not pressed.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Updates the current state of the button.

The state can be either pressed or released, and if the new state is different from the current state, the button is also invalidated to force a redraw.

If the button state is changed from **ClickEvent::PRESSED** to **ClickEvent::RELEASED**, the associated action (if any) is also executed.

Parameters:

event Information about the click.

Reimplements: **touchgfx::AbstractButton::handleClickEvent**

setBitmaps

```
virtual void setBitmaps ( const Bitmap & bitmapReleased ,
                          const Bitmap & bitmapPressed
                          )
```

Sets the two bitmaps used by this button.

One bitmap for the released (normal) state and one bitmap for the pressed state. The images are expected to be of the same dimensions, and the **Button** is resized to the dimensions of the pressed **Bitmap**.

Parameters:

bitmapReleased **Bitmap** to use when button is released.
bitmapPressed **Bitmap** to use when button is pressed.

NOTE

It is assumed that the dimensions of the bitmaps are the same. Unexpected (visual) behavior may be observed if the bitmaps are of different sizes. The user code must call invalidate() in order to update the button on the display.

Reimplements: **touchgfx::Button::setBitmaps**

Protected Attributes Documentation

originalPressed

Bitmap originalPressed

Contains the bitmap that was originally being displayed when button is pressed.

Version: 4.16

ToggleButtonTrigger

A toggle button trigger. This trigger will create a button that reacts on clicks. This means it will call the set action when it gets a touch released event, just like a ClickButtonTrigger. The difference being that a ToggleButtonTrigger will stay in pressed state until it is clicked again.

The ToggleButtonTrigger can be combined with one or more of the ButtonStyle classes to create a fully functional button.

Inherits from: AbstractButtonContainer, Container, Drawable

Public Functions

void	**forceState**(bool activeState)
	Allows the button to be forced into either the pressed state, or the normal state.
bool	**getToggleCanceled**()
	Gets toggle canceled.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
void	**setToggleCanceled**(bool isToggleCanceled)
	Sets toggle canceled.
	ToggleButtonTrigger()

Protected Attributes

bool	**toggleCanceled**
	True if toggle canceled.

Additional inherited members

Public Functions inherited from AbstractButtonContainer

	AbstractButtonContainer()
virtual void	**executeAction**() Executes the previously set action.
uint8_t	**getAlpha**() const Gets the current alpha value of the widget.
bool	**getPressed**() Gets the pressed state.
void	**setAction**(**GenericCallback**< const **AbstractButtonContainer** & > & callback) Sets an action callback to be executed by the subclass of AbstractContainerButton.
void	**setAlpha**(uint8_t newAlpha) Sets the opacity (alpha value).
void	**setPressed**(bool isPressed) Sets the pressed state to the given state.

Protected Functions inherited from AbstractButtonContainer

virtual void	**handleAlphaUpdated**() Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**() Handles what should happen when the pressed state is updated.

Protected Attributes inherited from AbstractButtonContainer

GenericCallback< const **AbstractButtonContainer** & > *	**action** The action to be executed.
uint8_t	**alpha** The current alpha value. 255 denotes solid, 0 denotes completely invisible.
bool	**pressed** True if pressed.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const

	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()

	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const

Gets whether this **Drawable** receives touch events or not.

bool **isVisible**() const

Gets whether this **Drawable** is visible.

virtual void **moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

virtual void **setHeight**(int16_t height)

Sets the height of this drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

virtual void **setWidth**(int16_t width)

Sets the width of this drawable.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

forceState

void forceState (bool activeState)

Allows the button to be forced into either the pressed state, or the normal state.

In the pressed state, the button will always be shown as pressed down (and shown as released when the user presses it). In the normal state, the button will be show as released or pressed depending on its actual state.

Parameters:

activeState If true, swap the images for released and pressed. If false display the button normally.

getToggleCanceled

bool getToggleCanceled ( )

Gets toggle canceled.

Returns:

True if it succeeds, false if it fails.

handleClickEvent

virtual void handleClickEvent (const ClickEvent & evt)

Defines the event handler interface for ClickEvents.

The default implementation ignores the event. The event is only received if the **Drawable** is touchable and visible.

Parameters:

evt The **ClickEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleClickEvent**

setToggleCanceled

void setToggleCanceled (bool isToggleCanceled)

Sets toggle canceled.

Parameters:

isToggleCanceled True if is toggle canceled, false if not.

ToggleButtonTrigger

ToggleButtonTrigger ( )

Protected Attributes Documentation

toggleCanceled

bool toggleCanceled

True if toggle canceled.

Version: 4.16

TouchArea

Invisible widget used to capture touch events. The TouchArea consumes drag events without the widget it self moving.

Inherits from: AbstractButton, Widget, Drawable

Public Functions

virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Updates the current state of the button.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
void	**setPressedAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action to be performed when the **TouchArea** is pressed.
	TouchArea()

Protected Attributes

GenericCallback< const **AbstractButton** & > *	**pressedAction**
	The action to perform when the **TouchArea** is clicked.

Additional inherited members

Public Functions inherited from AbstractButton

	AbstractButton()
	Sets this Widget touchable so the user can interact with buttons.
virtual void	**executeAction**()
	Executes the previously set action.
virtual bool	**getPressedState**() const
	Function to determine if the **AbstractButton** is currently pressed.
void	**setAction**(**GenericCallback**< const **AbstractButton** & > & callback)
	Associates an action with the button.

Protected Attributes inherited from AbstractButton

GenericCallback< const **AbstractButton** & > *	**action**
	The callback to be executed when this **AbstractButton** is clicked.
bool	**pressed**
	Is the button pressed or released? True if pressed.

Public Functions inherited from Widget

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const

	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**

True if this drawable should be drawn.

Public Functions Documentation

draw

virtual void draw (const Rect & invalidatedArea)

Draw this drawable.

It is a requirement that the draw implementation does not draw outside the region specified by invalidatedArea.

Parameters:

invalidatedArea	The sub-region of this drawable that needs to be redrawn, expressed in coordinates relative to its parent (e.g. for a complete redraw, invalidatedArea will be (0, 0, width, height).

Reimplements: **touchgfx::Drawable::draw**

getSolidRect

virtual Rect getSolidRect ( ) const

Get (the largest possible) rectangle that is guaranteed to be solid (opaque).

This information is important, as any **Drawable** underneath the solid area does not need to be drawn.

Returns:

The solid rectangle part of the **Drawable**.

NOTE

The rectangle returned must be relative to upper left corner of the Drawable, meaning that a completely solid widget should return the full size Rect(0, 0, getWidth(), getHeight()). If no area can be guaranteed to be solid, an empty Rect(0, 0, 0, 0) must be returned. Failing to return the correct rectangle may result in errors on the display.

Reimplements: **touchgfx::Drawable::getSolidRect**

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Updates the current state of the button.

The state can be either pressed or released, and if the new state is different from the current state, the button is also invalidated to force a redraw.

If the button state is changed from **ClickEvent::PRESSED** to **ClickEvent::RELEASED**, the associated action (if any) is also executed.

Parameters:

event Information about the click.

Reimplements: **touchgfx::AbstractButton::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & evt)

Defines the event handler interface for DragEvents.

The default implementation ignores the event. The event is only received if the drawable is touchable and visible.

Parameters:

evt The **DragEvent** received from the **HAL**.

Reimplements: **touchgfx::Drawable::handleDragEvent**

setPressedAction

void setPressedAction (GenericCallback< const AbstractButton & > & callback)

Associates an action to be performed when the **TouchArea** is pressed.

Parameters:

callback The callback is given a reference to this touch area.

TouchArea

TouchArea ( )

Protected Attributes Documentation

pressedAction

GenericCallback< const AbstractButton & > * pressedAction

The action to perform when the **TouchArea** is clicked.

Version: 4.16

TouchButtonTrigger

A touch button trigger. This trigger will create a button that reacts on touches. This means it will call the set action when it gets a touch pressed event. The TouchButtonTrigger can be combined with one or more of the ButtonStyle classes to create a fully functional button.

See: ClickButtonTrigger

Inherits from: AbstractButtonContainer, Container, Drawable

Public Functions

virtual void	**handleClickEvent**(const **ClickEvent** & event)
	Handles a ClickAvent.

Additional inherited members

Public Functions inherited from AbstractButtonContainer

	AbstractButtonContainer()
virtual void	**executeAction**()
	Executes the previously set action.
uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
bool	**getPressed**()
	Gets the pressed state.
void	**setAction**(**GenericCallback**< const **AbstractButtonContainer** & > & callback)
	Sets an action callback to be executed by the subclass of AbstractContainerButton.
void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
void	**setPressed**(bool isPressed)

	Sets the pressed state to the given state.

Protected Functions inherited from AbstractButtonContainer

virtual void	**handleAlphaUpdated**()
	Handles what should happen when the alpha is updated.
virtual void	**handlePressedUpdated**()
	Handles what should happen when the pressed state is updated.

Protected Attributes inherited from AbstractButtonContainer

GenericCallback< const **AbstractButtonContainer** & > *	**action**
	The action to be executed.
uint8_t	**alpha**
	The current alpha value. 255 denotes solid, 0 denotes completely invisible.
bool	**pressed**
	True if pressed.

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()

	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const
	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()

	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.

int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.

void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.

virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

Handles a ClickAvent.

The action callback is called when the **ClickButtonTrigger** receives a **ClickEvent::PRESSED** event. Function **setPressed()** will be called with the new button state.

Parameters:

event The click event.

See also:

setAction, **setPressed**, **getPressed**

Reimplements: **touchgfx::Drawable::handleClickEvent**

Version: 4.16

TouchCalibration

Calibrates a touch coordinate. Class TouchCalibraiton is responsible for translating coordinates (Point) based on matrix of calibration values.

Public Functions

	TouchCalibration()
void	**setCalibrationMatrix**(const **Point** *ref*, *const* ***Point*** scr)
	Initializes the calibration matrix based on reference and measured values.
void	**translatePoint**(**Point** & p)
	Translates the specified point using the matrix.

Public Functions Documentation

TouchCalibration

```
TouchCalibration ( )
```

setCalibrationMatrix

```
static void setCalibrationMatrix ( const Point * ref ,
                                   const Point * scr
                                   )
```

Initializes the calibration matrix based on reference and measured values.

Parameters:

ref Pointer to array of three reference points.
scr Pointer to array of three measured points.

translatePoint

static void translatePoint (Point & p)

Translates the specified point using the matrix.

If matrix has not been initialized, p is not modified.

Parameters:

p The point to translate.

Version: 4.16

TouchController

Basic Touch Controller interface.

Inherited by: I2CTouchController, NoTouchController, SDL2TouchController, SDLTouchController

Public Functions

virtual void	**init**() =0
	Initializes touch controller.
virtual bool	**sampleTouch**(int32_t & x, int32_t & y) =0
	Checks whether the touch screen is being touched, and if so, what coordinates.
virtual	**~TouchController**()
	Finalizes an instance of the **TouchController** class.

Public Functions Documentation

init

virtual void init ( ) =0

Initializes touch controller.

Reimplemented by: **touchgfx::NoTouchController::init**, **touchgfx::SDL2TouchController::init**, **touchgfx::SDLTouchController::init**, **touchgfx::I2CTouchController::init**

sampleTouch

virtual bool sampleTouch (int32_t & x , =0
int32_t & y =0
) =0

Checks whether the touch screen is being touched, and if so, what coordinates.

Parameters:

x The x position of the touch.

y The y position of the touch.

Returns:

True if a touch has been detected, otherwise false.

Reimplemented by: **touchgfx::NoTouchController::sampleTouch**, **touchgfx::SDL2TouchController::sampleTouch**, **touchgfx::SDLTouchController::sampleTouch**, **touchgfx::I2CTouchController::sampleTouch**

~TouchController

virtual ~TouchController ( )

Finalizes an instance of the **TouchController** class.

Version: 4.16

Transition

The Transition class is the base class for Transitions. Implementations of Transition defines what happens when transitioning between Screens, which typically involves visual effects. An example of a transition implementation can be seen in example custom_transition_example. The most basic transition is the NoTransition class that does a transition without any visual effects.

See: NoTransition, SlideTransition

Inherited by: BlockTransition, CoverTransition< templateDirection >, NoTransition, SlideTransition< templateDirection >, WipeTransition< templateDirection >

Public Functions

virtual void	**handleTickEvent**()
	Called for every tick when transitioning.
virtual void	**init**()
	Initializes the transition.
virtual void	**invalidate**()
	Invalidates the screen when starting the **Transition**.
bool	**isDone**() const
	Query if the transition is done transitioning.
virtual void	**setScreenContainer**(**Container** & cont)
	Sets the **ScreenContainer**.
virtual void	**tearDown**()
	Tears down the Animation.
	Transition()
	Initializes a new instance of the **Transition** class.
virtual	**~Transition**()
	Finalizes an instance of the **Transition** class.

Protected Attributes

bool	**done**
	Flag that indicates when the transition is done. This should be set by implementing classes.
Container *	**screenContainer**
	The screen **Container** of the **Screen** transitioning to.

Public Functions Documentation

handleTickEvent

virtual void handleTickEvent ( )

Called for every tick when transitioning.

Reimplemented by: **touchgfx::BlockTransition::handleTickEvent**, **touchgfx::CoverTransition::handleTickEvent**, **touchgfx::NoTransition::handleTickEvent**, **touchgfx::SlideTransition::handleTickEvent**, **touchgfx::WipeTransition::handleTickEvent**

init

virtual void init ( )

Initializes the transition.

Called after the constructor is called, when the application changes the transition.

Reimplemented by: **touchgfx::BlockTransition::init**, **touchgfx::CoverTransition::init**, **touchgfx::SlideTransition::init**, **touchgfx::WipeTransition::init**

invalidate

virtual void invalidate ( )

Invalidates the screen when starting the **Transition**.

Default is to invalidate the whole screen. Subclasses can do partial invalidation.

Reimplemented by: **touchgfx::BlockTransition::invalidate**, **touchgfx::WipeTransition::invalidate**

isDone

bool isDone ( ) const

Query if the transition is done transitioning.

It is the responsibility of the inheriting class to set the underlying done flag once the transition has been completed.

Returns:

True if the transition is done, false otherwise.

setScreenContainer

virtual void setScreenContainer (Container & cont)

Sets the **ScreenContainer**.

Is used by **Screen** to enable the transition to access the **Container**.

Parameters:

cont The **Container** the transition should have access to.

tearDown

virtual void tearDown ( )

Tears down the Animation.

Called before the destructor is called, when the application changes the transition.

Reimplemented by: **touchgfx::BlockTransition::tearDown**, **touchgfx::CoverTransition::tearDown**, **touchgfx::SlideTransition::tearDown**, **touchgfx::WipeTransition::tearDown**

Transition

Transition ( )

Initializes a new instance of the **Transition** class.

~Transition

virtual ~Transition ( )

Finalizes an instance of the **Transition** class.

Protected Attributes Documentation

done

bool done

Flag that indicates when the transition is done. This should be set by implementing classes.

screenContainer

Container * screenContainer

The screen **Container** of the **Screen** transitioning to.

Version: 4.16

TwoWildcardTextButtonStyle

A wildcard text button style. An wildcard text button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show a text with a wildcard in one of two colors depending on the state of the button (pressed or released).

The TwoWildcardTextButtonStyle does not set the size of the enclosing container (normally AbstractButtonContainer). The size must be set manually.

To get a background behind the text, use TwoWildcardTextButtonStyle together with e.g. ImageButtonStyle:

```
TwoWildcardTextButtonStyle<ImageButtonStyle<ClickButtonTrigger> > myButton;
```

The position of the text can be adjusted with setTwoWildcardTextXY (default is centered).

Template Parameters:

- **T** Generic type parameter. Typically a AbstractButtonContainer subclass.

See: AbstractButtonContainer

Inherits from: T

Public Functions

void	**setTwoWildcardText**(**TypedText** t)
	Sets wildcard text.
void	**setTwoWildcardTextColors**(**colortype** newColorReleased, **colortype** newColorPressed)
	Sets wild card text colors.
void	**setTwoWildcardTextPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets text position and dimensions.
void	**setTwoWildcardTextRotation**(**TextRotation** rotation)
	Sets wildcard text rotation.
void	**setTwoWildcardTextX**(int16_t x)

Sets wildcard text x coordinate.

void **setTwoWildcardTextXY**(int16_t x, int16_t y)

Sets wildcard text position.

void **setTwoWildcardTextY**(int16_t y)

Sets wildcard text y coordinate.

void **setWildcardTextBuffer1**(const **Unicode::UnicodeChar** * value)

Sets the first wildcard in the text.

void **setWildcardTextBuffer2**(const **Unicode::UnicodeChar** * value)

Sets the second wildcard in the text.

TwoWildcardTextButtonStyle()

Protected Functions

virtual void **handleAlphaUpdated**()

Handles what should happen when the alpha is updated.

virtual void **handlePressedUpdated**()

Handles what should happen when the pressed state is updated.

Protected Attributes

colortype **colorPressed**

The color pressed.

colortype **colorReleased**

The color released.

TextAreaWithTwoWildcards **twoWildcardText**

The wildcard text.

Public Functions Documentation

setTwoWildcardText

```
void setTwoWildcardText ( TypedText t )
```

Sets wildcard text.

Parameters:

t	A **TypedText** to process.

setTwoWildcardTextColors

```
void setTwoWildcardTextColors ( colortype newColorReleased ,
                                colortype newColorPressed
                                )
```

Sets wild card text colors.

Parameters:

newColorReleased	The new color released.
newColorPressed	The new color pressed.

setTwoWildcardTextPosition

```
void setTwoWildcardTextPosition ( int16_t x ,
                                  int16_t y ,
                                  int16_t width ,
                                  int16_t height
                                  )
```

Sets text position and dimensions.

Parameters:

x	The x coordinate.
y	The y coordinate.
width	The width of the text.
height	The height of the text.

setTwoWildcardTextRotation

```
void setTwoWildcardTextRotation ( TextRotation rotation )
```

Sets wildcard text rotation.

Parameters:

rotation The rotation.

setTwoWildcardTextX

void setTwoWildcardTextX (int16_t x)

Sets wildcard text x coordinate.

Parameters:

x The x coordinate.

setTwoWildcardTextXY

void setTwoWildcardTextXY (int16_t x ,
int16_t y
)

Sets wildcard text position.

Parameters:

x The x coordinate.
y The y coordinate.

setTwoWildcardTextY

void setTwoWildcardTextY (int16_t y)

Sets wildcard text y coordinate.

Parameters:

y The y coordinate.

setWildcardTextBuffer1

void setWildcardTextBuffer1 (const Unicode::UnicodeChar * value)

Sets the first wildcard in the text.

Must be a null-terminated UnicodeChar array.

Parameters:

value A pointer to the UnicodeChar to set the wildcard to.

setWildcardTextBuffer2

void setWildcardTextBuffer2 (const Unicode::UnicodeChar * value)

Sets the second wildcard in the text.

Must be a null-terminated UnicodeChar array.

Parameters:

value A pointer to the UnicodeChar to set the wildcard to.

TwoWildcardTextButtonStyle

TwoWildcardTextButtonStyle ( )

Protected Functions Documentation

handleAlphaUpdated

virtual void handleAlphaUpdated ( )

Handles what should happen when the alpha is updated.

handlePressedUpdated

virtual void handlePressedUpdated ( )

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

colorPressed

colortype colorPressed

The color pressed.

colorReleased

colortype colorReleased

The color released.

twoWildcardText

TextAreaWithTwoWildcards twoWildcardText

The wildcard text.

Version: 4.16

TypedText

TypedText represents text (for characters) and typography (for font and alignment). TypedText provides methods for adjusting the text, font and alignment.

See: TextArea

Public Classes

struct	**TypedTextData**
	The data structure for typed texts.

Public Functions

FORCE_INLINE_FUNCTION **Alignment**	**getAlignment**() const
	Gets the alignment associated with this **TypedText**.
FORCE_INLINE_FUNCTION const **Font** *	**getFont**() const
	Gets the font associated with this **TypedText**.
FORCE_INLINE_FUNCTION **FontId**	**getFontId**() const
	Gets the font ID associated with this **TypedText**.
FORCE_INLINE_FUNCTION **TypedTextId**	**getId**() const
	Gets the id of the typed text.
FORCE_INLINE_FUNCTION const **Unicode::UnicodeChar** *	**getText**() const
	Gets the text associated with this **TypedText**.
FORCE_INLINE_FUNCTION **TextDirection**	**getTextDirection**() const
	Gets the text direction associated with this **TypedText**.
FORCE_INLINE_FUNCTION bool	**hasValidId**() const

	Has the **TypedText** been set to a proper value?
	TypedText(const **TypedTextId** id =**TYPED_TEXT_INVALID**)
	Initializes a new instance of the **TypedText** class.
virtual	**~TypedText**()
	Finalizes an instance of the **TypedText** class.
void	**registerTexts**(const **Texts** * t)
	Registers an array of texts.
void	**registerTypedTextDatabase**(const **TypedTextData** *data, const **Font*** const * f, const uint16_t n)
	Registers an array of typed texts.

Public Functions Documentation

getAlignment

FORCE_INLINE_FUNCTION Alignment getAlignment ( ) const

Gets the alignment associated with this **TypedText**.

Returns:

The alignment.

getFont

FORCE_INLINE_FUNCTION const Font * getFont ( ) const

Gets the font associated with this **TypedText**.

Returns:

The font.

getFontId

FORCE_INLINE_FUNCTION FontId getFontId ( ) const

Gets the font ID associated with this **TypedText**.

Returns:

The font.

getId

FORCE_INLINE_FUNCTION TypedTextId getId ( ) const

Gets the id of the typed text.

Returns:

The id.

getText

FORCE_INLINE_FUNCTION const Unicode::UnicodeChar * getText ( ) const

Gets the text associated with this **TypedText**.

Returns:

The text.

getTextDirection

FORCE_INLINE_FUNCTION TextDirection getTextDirection ( ) const

Gets the text direction associated with this **TypedText**.

Returns:

The alignment.

hasValidId

FORCE_INLINE_FUNCTION bool hasValidId ( ) const

Has the **TypedText** been set to a proper value?

Returns:

Is the id valid.

TypedText

```
explicit TypedText ( const TypedTextId id =TYPED_TEXT_INVALID )
```

Initializes a new instance of the **TypedText** class.

Parameters:

id (Optional) The identifier.

~TypedText

```
virtual ~TypedText (  )
```

Finalizes an instance of the **TypedText** class.

registerTexts

```
static void registerTexts ( const Texts * t )
```

Registers an array of texts.

This function is called automatically from **touchgfx_generic_init()**. Should not be called under normal circumstances.

Parameters:

t The array of texts.

registerTypedTextDatabase

```
static void registerTypedTextDatabase ( const TypedTextData * data ,
                                        const Font const       f ,
                                        const uint16_t         n
                                        )
```

Registers an array of typed texts.

All typed text instances are bound to this database. This function is called automatically when setting a new language. Use **Texts::setLanguage()** instead of calling this function directly.

Parameters:

data	A reference to the **TypedTextData** storage array.
f	The fonts associated with the array.
n	The number of typed texts in the array.

Version: 4.16

TypedTextData

The data structure for typed texts.

Public Attributes

const **Alignment**	**alignment**
	The alignment of the typed text (LEFT,CENTER,RIGHT)
const **TextDirection**	**direction**
	The text direction (LTR,RTL,...) of the typed text.
const unsigned char	**fontIdx**
	The ID of the font associated with the typed text.

Public Attributes Documentation

alignment

const Alignment alignment

The alignment of the typed text (LEFT,CENTER,RIGHT)

direction

const TextDirection direction

The text direction (LTR,RTL,...) of the typed text.

fontIdx

const unsigned char fontIdx

The ID of the font associated with the typed text.

Version: 4.16

UIEventListener

This class declares a handler interface for user interface events, i.e. events generated by the users interaction with the device. With the exception of the system timer tick, all other system events, which are not related to the user interface device peripherals (display, keys etc.) are not part of this interface.

Inherited by: Application

Public Functions

virtual void	**handleClickEvent**(const **ClickEvent** & event)
	This handler is invoked when a mouse click or display touch event has been detected by the system.
virtual void	**handleDragEvent**(const **DragEvent** & event)
	This handler is invoked when a drag event has been detected by the system.
virtual void	**handleGestureEvent**(const **GestureEvent** & event)
	This handler is invoked when a gesture event has been detected by the system.
virtual void	**handleKeyEvent**(uint8_t c)
	This handler is invoked when a key (or button) event has been detected by the system.
virtual void	**handlePendingScreenTransition**()
	This handler is invoked when a change screen event is pending.
virtual void	**handleTickEvent**()
	This handler is invoked when a system tick event has been generated.
virtual	**~UIEventListener**()
	Finalizes an instance of the **UIEventListener** class.

Public Functions Documentation

handleClickEvent

virtual void handleClickEvent (const ClickEvent & event)

This handler is invoked when a mouse click or display touch event has been detected by the system.

Parameters:

event The event data.

Reimplemented by: **touchgfx::Application::handleClickEvent**

handleDragEvent

virtual void handleDragEvent (const DragEvent & event)

This handler is invoked when a drag event has been detected by the system.

Parameters:

event The event data.

Reimplemented by: **touchgfx::Application::handleDragEvent**

handleGestureEvent

virtual void handleGestureEvent (const GestureEvent & event)

This handler is invoked when a gesture event has been detected by the system.

Parameters:

event The event data.

Reimplemented by: **touchgfx::Application::handleGestureEvent**

handleKeyEvent

virtual void handleKeyEvent (uint8_t c)

This handler is invoked when a key (or button) event has been detected by the system.

Parameters:

c The key or button pressed.

Reimplemented by: **touchgfx::Application::handleKeyEvent**

handlePendingScreenTransition

virtual void handlePendingScreenTransition ( )

This handler is invoked when a change screen event is pending.

Reimplemented by: **touchgfx::MVPApplication::handlePendingScreenTransition**, **touchgfx::Application::handlePendingScreenTransition**

handleTickEvent

virtual void handleTickEvent ( )

This handler is invoked when a system tick event has been generated.

The system tick period is configured in the **HAL**.

Reimplemented by: **touchgfx::Application::handleTickEvent**

~UIEventListener

virtual ~UIEventListener ( )

Finalizes an instance of the **UIEventListener** class.

Version: 4.16

Unicode

This class provides simple helper functions for working with strings which are stored as a null-terminated array of 16-bit characters.

Public Types

typedef uint16_t	**UnicodeChar**
	Use the UnicodeChar typename when referring to characters in a string.

Public Functions

int	**atoi**(const **UnicodeChar** * s)
	String to integer conversion.
uint16_t	**fromUTF8**(const uint8_t *utf8*, ***UnicodeChar*** dst, uint16_t maxchars)
	Convert a string from UTF8 to **Unicode**.
void	**itoa**(int32_t value, **UnicodeChar** * buffer, uint16_t bufferSize, int radix)
	Integer to ASCII conversion.
UnicodeChar *	**snprintf**(**UnicodeChar** *dst, uint16_t dstSize, const char* format, ...)
	Formats a string and adds null termination.
UnicodeChar *	**snprintf**(**UnicodeChar** *dst, uint16_t dstSize, const* ***UnicodeChar*** format, ...)
	Formats a string and adds null termination.
UnicodeChar *	**snprintfFloat**(**UnicodeChar** *dst, uint16_t dstSize, const char* format, const float value)
	Variant of snprintfFloats() for exactly one float only.
UnicodeChar *	**snprintfFloat**(**UnicodeChar** *dst, uint16_t dstSize, const* ***UnicodeChar*** format, const float value)
	Variant of snprintfFloats() for exactly one float only.
UnicodeChar *	**snprintfFloats**(**UnicodeChar** *dst, uint16_t dstSize, const char* format, const float * values)
	Variant of snprintf for floats only.

UnicodeChar *	**snprintfFloats**(**UnicodeChar** *dst, uint16_t dstSize, const* ***UnicodeChar*** format, const float * values)
	Variant of snprintf for floats only.
uint16_t	**strlen**(const char * str)
	Gets the length of a null-terminated string.
uint16_t	**strlen**(const **UnicodeChar** * str)
	Gets the length of a null-terminated **Unicode** string.
int	**strncmp**(const **UnicodeChar** *RESTRICT str1, const* ***UnicodeChar*** RESTRICT str2, uint16_t maxchars)
	Compares up to maxchars characters in two strings.
int	**strncmp_ignore_whitespace**(const **UnicodeChar** *RESTRICT str1, const* ***UnicodeChar*** RESTRICT str2, uint16_t maxchars)
	Like strncmp except that ignore any whitespaces in the two strings.
uint16_t	**strncpy**(**UnicodeChar** *RESTRICT dst, const char* RESTRICT src, uint16_t maxchars)
	Copy a string to a destination buffer, char to UnicodeChar version.
uint16_t	**strncpy**(**UnicodeChar** *RESTRICT dst, const* ***UnicodeChar*** RESTRICT src, uint16_t maxchars)
	Copy a string to a destination buffer, UnicodeChar to UnicodeChar version.
uint16_t	**toUTF8**(const **UnicodeChar** *unicode, uint8_t* utf8, uint16_t maxbytes)
	Converts a string from **Unicode** to UTF8.
void	**utoa**(uint32_t value, **UnicodeChar** * buffer, uint16_t bufferSize, int radix)
	Integer to ASCII conversion.
UnicodeChar *	**vsnprintf**(**UnicodeChar** *dst, uint16_t dstSize, const char* format, va_list pArg)
	Variant of snprintf.
UnicodeChar *	**vsnprintf**(**UnicodeChar** *dst, uint16_t dstSize, const* ***UnicodeChar*** format, va_list pArg)
	Variant of snprintf.

Public Types Documentation

UnicodeChar

typedef uint16_t UnicodeChar

Use the UnicodeChar typename when referring to characters in a string.

Public Functions Documentation

atoi

static int atoi (const UnicodeChar * s)

String to integer conversion.

Starts conversion at the start of the string. Running digits from here are converted. Only radix 10 supported.

Parameters:

s Radix 10, null-terminated **Unicode** string to convert.

Returns:

The converted integer value of the string, 0 if the string does not start with a digit.

fromUTF8

```
static uint16_t fromUTF8 ( const uint8_t * utf8 ,
                           UnicodeChar * dst ,
                           uint16_t maxchars
                           )
```

Convert a string from UTF8 to **Unicode**.

The conversion stops if there is no more room in the destination or if the terminating zero character has been converted.

Parameters:

utf8	The UTF8 string.
dst	The destination buffer for the converted string.
maxchars	The maximum number of chars that the dst array can hold.

Returns:

The number of characters successfully converted from UTF8 to **Unicode** including the terminating zero.

itoa

```
static void itoa ( int32_t          value ,
                   UnicodeChar * buffer ,
                   uint16_t         bufferSize ,
                   int              radix
                   )
```

Integer to ASCII conversion.

Supports radix 2 to radix 36.

Parameters:

value	to convert.
buffer	to place result in.
bufferSize	Size of buffer (number of UnicodeChar's).
radix	to use (8 for octal, 10 for decimal, 16 for hex)

snprintf

```
static UnicodeChar * snprintf ( UnicodeChar * dst ,
                                uint16_t         dstSize ,
                                const char *     format ,
                                ...
                                )
```

Formats a string and adds null termination.

The string is formatted like when the standard printf is used.

Support formats: %c (element type: char), %s (element type: null-terminated UnicodeChar list), %u, %i, %d, %o, %x (all these are integers formatted in radix 10, 10, 10, 8, 16 respectively).

The number formats (%u, %i, %d, %o and %x) all support

```
\%[flags][width][.precision]X
```

Where flags can be:

- '-': left justify the field (see width).
- '+': force sign.
- ' ': insert space if value is positive.
- '0': left pad with zeros instead of spaces (see width). Where width is the desired width of the output. If the value is larger, more characters may be generated, but not more than the

parameter dstSize. If width is '*' the actual width is read from the parameters passed to this function.

Where precision is the number of number of digits after the decimal point, default is

1. Use "\%.f" to not generate any numbers after the decimal point. If precision is '*' the actual precision is read from the parameters passed to this function.

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string.
...	The values to insert in the format string.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

NOTE

%f is not supported by this function because floats are converted to doubles when given as parameters in a variable argument list (va_list). Use snprintfFloat or snprintfFloats instead.

See also:

snprintfFloat, **snprintfFloats snprintfFloat**, **snprintfFloats**

snprintf

```
static UnicodeChar * snprintf ( UnicodeChar *       dst ,
                                uint16_t            dstSize ,
                                const UnicodeChar * format ,
                                ...
                                )
```

Formats a string and adds null termination.

The string is formatted like when the standard printf is used.

Support formats: %c (element type: char), %s (element type: null-terminated UnicodeChar list), %u, %i, %d, %o, %x (all these are integers formatted in radix 10, 10, 10, 8, 16 respectively).

The number formats (%u, %i, %d, %o and %x) all support %[0][length]X to specify the size of the generated field (length) and whether the number should be prefixed with zeros (or blanks).

Parameters:

dst	Buffer for the formatted string.

dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string.
...	The values to insert in the format string.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

NOTE

%f is not supported by this function because floats are converted to doubles when given as parameters in a variable argument list (va_list). Use snprintfFloat or snprintfFloats instead.

See also:

snprintfFloat, **snprintfFloats**

snprintfFloat

```
static UnicodeChar * snprintfFloat ( UnicodeChar * dst ,
                                     uint16_t      dstSize ,
                                     const char *  format ,
                                     const float   value
                                     )
```

Variant of snprintfFloats() for exactly one float only.

The number format supports only one %f with flags/modifiers. The following is supported:

```
\%[flags][width][.precision]f
```

Where flags can be:

- '-': left justify the field (see width).
- '+': force sign.
- ' ': insert space if value is positive.
- '#': insert decimal point even if there are not decimals.
- '0': left pad with zeros instead of spaces (see width). Where width is the desired width of the output. If the value is larger, more characters may be generated, but not more than the parameter dstSize.

Where precision is the number of number of digits after the decimal point, default is "3". Use "\%.f" to not generate any numbers after the decimal point.

```
Unicode::UnicodeChar buffer[20];
Unicode::snprintfFloat(buffer, 20, "%6.4f", 3.14159f);
// buffer="3.1416"
Unicode::snprintfFloat(buffer, 20, "%#6.f", 3.14159f);
// buffer="    3."
Unicode::snprintfFloat(buffer, 20, "%6f", 3.14159f);
// buffer=" 3.142"
Unicode::snprintfFloat(buffer, 20, "%+06.f", 3.14159f);
// buffer="+00003"
```

Filename: .cpp

If more control over the output is needed, see snprintfFloats which can have more than a single "\%f" in the string and also supports "*" in place of a number.

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string containing exactly on occurrence of f.
value	The floating point value to insert for f.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

See also:

snprintf, **snprintfFloats**

snprintfFloat

```
static UnicodeChar * snprintfFloat ( UnicodeChar *       dst ,
                                     uint16_t            dstSize ,
                                     const UnicodeChar * format ,
                                     const float         value
                                     )
```

Variant of snprintfFloats() for exactly one float only.

The number format supports only one %f with flags/modifiers. The following is supported:

```
\%[flags][width][.precision]f
```

Where flags can be:

- '-': left justify the field (see width).

- '+': force sign.
- ' ': insert space if value is positive.
- '#': insert decimal point even if there are not decimals.
- '0': left pad with zeros instead of spaces (see width). Where width is the desired width of the output. If the value is larger, more characters may be generated, but not more than the parameter dstSize.

Where precision is the number of number of digits after the decimal point, default is "3". Use "\%.f" to not generate any numbers after the decimal point.

```
Unicode::UnicodeChar buffer[20];
Unicode::snprintfFloat(buffer, 20, "%6.4f", 3.14159f);
// buffer="3.1416"
Unicode::snprintfFloat(buffer, 20, "%#6.f", 3.14159f);
// buffer="    3."
Unicode::snprintfFloat(buffer, 20, "%6f", 3.14159f);
// buffer=" 3.142"
Unicode::snprintfFloat(buffer, 20, "%+06.f", 3.14159f);
// buffer="+00003"
```

Filename: .cpp

If more control over the output is needed, see snprintfFloats which can have more than a single "\%f" in the string and also supports "*" in place of a number.

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string containing exactly on occurrence of f.
value	The floating point value to insert for f.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

See also:

snprintf, **snprintfFloats**

snprintfFloats

```
static UnicodeChar * snprintfFloats ( UnicodeChar *  dst ,
                                      uint16_t       dstSize ,
                                      const char *   format ,
                                      const float *  values
```

)

Variant of snprintf for floats only.

The format supports several %f with flags/modifiers. The following is supported:

```
\%[flags][width][.precision]f
```

Where flags can be:

- '-': left justify the field (see width).
- '+': force sign.
- ' ': insert space if value is positive
- '#': insert decimal point even if there are not decimals
- '0': left pad with zeros instead of spaces (see width) Where width is the desired width of the output. If the value is larger, more characters may be generated, but not more than the parameter dstSize. If width is '*' the actual width is read from the list of values passed to this function.

Where precision is the number of number of digits after the decimal point, default is

1. Use "\%.f" to not generate any numbers after the decimal point. If precision is '*' the actual precision is read from the list of values passed to this function.

```
float param1[3] = { 6.0f, 4.0f, 3.14159f };
Unicode::snprintfFloats(buffer, 20, "%*.*f", param1);
// buffer="3.1416" float param2[2] = { 3.14159f, -123.4f };
Unicode::snprintfFloats(buffer, 20, "%f %f", param2);
// buffer="3.142 -123.400"
```

Filename: .cpp

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string containing f's.
values	The floating point values to insert for f. The number of elements in the array must match the number of f's in the format string.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

See also:

snprintf, **snprintfFloat**

snprintfFloats

```
static UnicodeChar * snprintfFloats ( UnicodeChar *       dst ,
                                      uint16_t            dstSize ,
                                      const UnicodeChar * format ,
                                      const float *       values
                                      )
```

Variant of snprintf for floats only.

The format supports several %f with flags/modifiers. The following is supported:

```
\%[flags][width][.precision]f
```

Where flags can be:

- '-': left justify the field (see width).
- '+': force sign.
- ' ': insert space if value is positive
- '#': insert decimal point even if there are not decimals
- '0': left pad with zeros instead of spaces (see width) Where width is the desired width of the output. If the value is larger, more characters may be generated, but not more than the parameter dstSize. If width is '*' the actual width is read from the list of values passed to this function.

Where precision is the number of number of digits after the decimal point, default is

1. Use "\%.f" to not generate any numbers after the decimal point. If precision is '*' the actual precision is read from the list of values passed to this function.

```
float param1[3] = { 6.0f, 4.0f, 3.14159f };
Unicode::snprintfFloats(buffer, 20, "%*.*f", param1);
// buffer="3.1416" float param2[2] = { 3.14159f, -123.4f };
Unicode::snprintfFloats(buffer, 20, "%f %f", param2);
// buffer="3.142 -123.400"
```

Filename: .cpp

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string containing f's.

values The floating point values to insert for f. The number of elements in the array must match the number of f's in the format string.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

See also:

snprintf, **snprintfFloat**

strlen

static uint16_t strlen (const char * str)

Gets the length of a null-terminated string.

Parameters:

str The string.

Returns:

Length of string.

strlen

static uint16_t strlen (const UnicodeChar * str)

Gets the length of a null-terminated **Unicode** string.

Parameters:

str The string in question.

Returns:

Length of string.

strncmp

```
static int strncmp ( const UnicodeChar *RESTRICT str1 ,
                     const UnicodeChar *RESTRICT str2 ,
                     uint16_t                    maxchars
                     )
```

Compares up to maxchars characters in two strings.

One character from each buffer is compared, one at a time until the characters differ, until a terminating null- character is reached, or until maxchars characters match in both strings, whichever happens first.

Parameters:

str1	The first string.
str2	The second string.
maxchars	The maximum number of chars to compare.

Returns:

Returns an integral value indicating the relationship between the strings: A zero value indicates that the characters compared in both strings are all equal. A value greater than zero indicates that the first character that does not match has a greater value in str1 than in str2; And a value less than zero indicates the opposite.

strncmp_ignore_whitespace

```
static int strncmp_ignore_whitespace ( const UnicodeChar *RESTRICT str1 ,
                                       const UnicodeChar *RESTRICT str2 ,
                                       uint16_t                    maxchars
                                       )
```

Like strncmp except that ignore any whitespaces in the two strings.

Parameters:

str1	The first string.
str2	The second string.
maxchars	The maximum number of chars to compare.

Returns:

Returns an integral value indicating the relationship between the strings: A zero value indicates that the characters compared in both strings are all equal. A value greater than zero indicates that the first character that does not match has a greater value in str1 than in str2; And a value less than zero indicates the opposite.

strncpy

```
static uint16_t strncpy ( UnicodeChar *RESTRICT dst ,
                          const char *RESTRICT    src ,
                          uint16_t                maxchars
                          )
```

Copy a string to a destination buffer, char to UnicodeChar version.

Stops after copying maxchars **Unicode** characters or after copying the ending null-termination UnicodeChar.

Parameters:

dst	The destination buffer. Must have a size of at least maxchars.
src	The source string.
maxchars	Maximum number of chars to copy.

Returns:

The number of characters copied (excluding null-termination if encountered)

NOTE

If there is no null-termination among the first n UnicodeChars of src, the string placed in destination will NOT be null-terminated!

strncpy

```
static uint16_t strncpy ( UnicodeChar *RESTRICT       dst ,
                          const UnicodeChar *RESTRICT src ,
                          uint16_t                    maxchars
                          )
```

Copy a string to a destination buffer, UnicodeChar to UnicodeChar version.

Stops after copying maxchars **Unicode** characters or after copying the ending zero termination UnicodeChar.

Parameters:

dst	The destination buffer. Must have a size of at least maxchars.
src	The source string.
maxchars	Maximum number of UnicodeChars to copy.

Returns:

The number of characters copied (excluding null-termination if encountered)

NOTE

If there is no null-termination among the first n UnicodeChars of src, the string placed in destination will NOT be null-terminated!

toUTF8

```
static uint16_t toUTF8 ( const UnicodeChar * unicode ,
                         uint8_t *           utf8 ,
                         uint16_t            maxbytes
                         )
```

Converts a string from **Unicode** to UTF8.

The conversion stops if there is no more room in the destination or if the terminating zero character has been converted. U+10000 through U+10FFFF are skipped.

Parameters:

unicode	The **Unicode** string.
utf8	The destination buffer for the converted string.
maxbytes	The maximum number of bytes that the UTF8 array can hold.

Returns:

The number of characters successfully converted from **Unicode** to UTF8 including the terminating zero.

utoa

```
static void utoa ( uint32_t        value ,
                   UnicodeChar * buffer ,
                   uint16_t        bufferSize ,
                   int             radix
                   )
```

Integer to ASCII conversion.

Supports radix 2 to radix 36.

Parameters:

value	to convert.
buffer	to place result in.
bufferSize	Size of buffer (number of UnicodeChar's).
radix	to use (8 for octal, 10 for decimal, 16 for hex)

vsnprintf

```
static UnicodeChar * vsnprintf ( UnicodeChar * dst ,
                                 uint16_t      dstSize ,
```

```
                                const char *    format ,
                                va_list         pArg
                                )
```

Variant of snprintf.

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string.
pArg	The values to insert in the format string.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

See also:

snprintf

vsnprintf

```
static UnicodeChar * vsnprintf ( UnicodeChar *          dst ,
                                 uint16_t               dstSize ,
                                 const UnicodeChar *    format ,
                                 va_list                pArg
                                 )
```

Variant of snprintf.

Parameters:

dst	Buffer for the formatted string.
dstSize	Size of the dst buffer measured by number of UnicodeChars the buffer can hold.
format	The format string.
pArg	The values to insert in the format string.

Returns:

pointer to the first element in the buffer where the formatted string is placed.

See also:

snprintf

Version: 4.16

Vector

A very simple container class using pre-allocated memory.

Template Parameters:

- **T** The type of objects this container works on.
- **capacity** The maximum number of objects this container can store.

Public Functions

void	**add**(T e)
	Adds an element to the **Vector** if the **Vector** is not full.
void	**clear**()
	Clears the contents of the container.
bool	**contains**(T elem)
	Checks if the **Vector** contains an element.
bool	**isEmpty**() const
	Query if this object is empty.
uint16_t	**maxCapacity**() const
	Query the maximum capacity of the vector.
T &	**operator[]**(uint16_t idx)
	Index operator.
const T &	**operator[]**(uint16_t idx) const
	Const version of the index operator.
void	**quickRemoveAt**(uint16_t index)
	Removes an element at the specified index of the **Vector**.
void	**remove**(T e)
	Removes an element from the **Vector** if found in the **Vector**.
T	**removeAt**(uint16_t index)

	Removes an element at the specified index of the **Vector**.
void	**reverse**()
	Reverses the ordering of the elements in the **Vector**.
uint16_t	**size**() const
	Gets the current size of the **Vector** which is the number of elements contained in the **Vector**.
	Vector()
	Default constructor.

Public Functions Documentation

add

void add (T e)

Adds an element to the **Vector** if the **Vector** is not full.

Adds an element to the **Vector** if the **Vector** is not full. Does nothing if the **Vector** is full.

Parameters:

e The element to add to the **Vector**.

clear

void clear ( )

Clears the contents of the container.

It does not destruct any of the elements in the **Vector**.

contains

bool contains (T elem)

Checks if the **Vector** contains an element.

The == operator of the element is used when comparing it with the elements in the **Vector**.

Parameters:

elem The element.

Returns:

true if the **Vector** contains the element, false otherwise.

isEmpty

bool isEmpty ( ) const

Query if this object is empty.

Returns:

true if the **Vector** contains no elements.

maxCapacity

uint16_t maxCapacity ( ) const

Query the maximum capacity of the vector.

Returns:

The capacity the **Vector** was initialized with.

operator[]

T & operator[] (uint16_t idx)

Index operator.

Parameters:

idx The index of the element to obtain.

Returns:

A reference to the element placed at index idx.

operator[]

const T & operator[] (uint16_t idx)

Const version of the index operator.

Parameters:

idx The index of the element to obtain.

Returns:

A const reference to the element placed at index idx.

quickRemoveAt

void quickRemoveAt (uint16_t index)

Removes an element at the specified index of the **Vector**.

The last element in the list is moved to the position where the element is removed.

Parameters:

index The index to remove.

remove

void remove (T e)

Removes an element from the **Vector** if found in the **Vector**.

Does nothing if the element is not found in the **Vector**. The == operator of the element is used when comparing it with the elements in the **Vector**.

Parameters:

e The element to remove from the **Vector**.

removeAt

T removeAt (uint16_t index)

Removes an element at the specified index of the **Vector**.

Will "bubble-down" any remaining elements after the specified index.

Parameters:

index The index to remove.

Returns:

The value of the removed element.

reverse

void reverse ( )

Reverses the ordering of the elements in the **Vector**.

size

uint16_t size ( ) const

Gets the current size of the **Vector** which is the number of elements contained in the **Vector**.

Gets the current size of the **Vector** which is the number of elements contained in the **Vector**.

Returns:

The size of the **Vector**.

Vector

Vector ( )

Default constructor.

Constructs an empty vector.

Version: 4.16

Vector4

This class represents a homogeneous 3D vector.

See: Quadruple

Inherits from: Quadruple

Public Functions

FORCE_INLINE_FUNCTION **Vector4**	**crossProduct**(const **Vector4** & operand)
	Cross product.
FORCE_INLINE_FUNCTION	**Vector4**()
	Initializes a new instance of the **Vector4** class.
FORCE_INLINE_FUNCTION	**Vector4**(float x, float y, float z)
	Initializes a new instance of the **Vector4** class.

Additional inherited members

Public Functions inherited from Quadruple

FORCE_INLINE_FUNCTION float	**getElement**(int row) const
	Gets an element.
FORCE_INLINE_FUNCTION float	**getW**() const
	Get w coordinate.
FORCE_INLINE_FUNCTION float	**getX**() const
	Get x coordinate.
FORCE_INLINE_FUNCTION float	**getY**() const
	Get y coordinate.
FORCE_INLINE_FUNCTION float	**getZ**() const
	Get z coordinate.

FORCE_INLINE_FUNCTION void	**setElement**(int row, float value)
	Sets an element.
FORCE_INLINE_FUNCTION void	**setW**(float value)
	Sets a w coordinate.
FORCE_INLINE_FUNCTION void	**setX**(float value)
	Sets an x coordinate.
FORCE_INLINE_FUNCTION void	**setY**(float value)
	Sets a y coordinate.
FORCE_INLINE_FUNCTION void	**setZ**(float value)
	Sets a z coordinate.

Protected Functions inherited from Quadruple

FORCE_INLINE_FUNCTION	**Quadruple**()
	Initializes a new instance of the **Quadruple** class.
FORCE_INLINE_FUNCTION	**Quadruple**(float x, float y, float z, float w)
	Initializes a new instance of the **Quadruple** class.

Protected Attributes inherited from Quadruple

float	**elements**
	The elements[4].

Public Functions Documentation

crossProduct

FORCE_INLINE_FUNCTION Vector4 crossProduct (const Vector4 & operand)

Cross product.

Parameters:

operand The second operand.

Returns:

The result of the operation.

Vector4

```
FORCE_INLINE_FUNCTION Vector4 (  )
```

Initializes a new instance of the **Vector4** class.

Vector4

```
FORCE_INLINE_FUNCTION Vector4 ( float x ,
                                float y ,
                                float z
                                )
```

Initializes a new instance of the **Vector4** class.

Parameters:

x The x coordinate.
y The y coordinate.
z The z coordinate.

Version: 4.16

View

This is a generic touchgfx::Screen specialization for normal applications. It provides a link to the Presenter class.

Template Parameters:

- **T** The type of Presenter associated with this view.

See: Screen

Note: All views in the application must be a subclass of this type.

Inherits from: Screen

Public Functions

void	**bind**(T & presenter)
	Binds an instance of a specific **Presenter** type (subclass) to the **View** instance.
	View()

Protected Attributes

T *	**presenter**
	Pointer to the **Presenter** associated with this view.

Additional inherited members

Public Functions inherited from Screen

virtual void	**afterTransition**()
	Called by **Application::handleTickEvent()** when the transition to the screen is done.
void	**bindTransition**(**Transition** & trans)

	Enables the transition to access the containers.
void	**draw**()
	Tells the screen to draw its entire area.
virtual void	**draw**(**Rect** & rect)
	Tell the screen to draw the specified area.
Container &	**getRootContainer**()
	Obtain a reference to the root container of this screen.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Traverse the drawables in reverse z-order and notify them of a click event.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Traverse the drawables in reverse z-order and notify them of a drag event.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Handle gestures.
virtual void	**handleKeyEvent**(uint8_t key)
	Called by the **Application** on the reception of a "key", the meaning of which is platform/application specific.
virtual void	**handleTickEvent**()
	Called by the **Application** on the current screen with a frequency of **Application::TICK_INTERVAL_MS**.
void	**JSMOC**(const **Rect** & invalidatedArea, **Drawable** * widgetToDraw)
	Recursive JSMOC function.
	Screen()
	Initializes a new instance of the **Screen** class.
virtual void	**setupScreen**()
	Called by **Application::switchScreen()** when this screen is going to be displayed.
void	**startSMOC**(const **Rect** & invalidatedArea)
	Starts a JSMOC run, analyzing what parts of what widgets should be redrawn.
virtual void	**tearDownScreen**()
	Called by **Application::switchScreen()** when this screen will no longer be displayed.
bool	**usingSMOC**() const

	Determines if using JSMOC.
virtual	**~Screen**()
	Finalizes an instance of the **Screen** class.

Protected Functions inherited from Screen

void	**add**(**Drawable** & d)
	Add a drawable to the content container.
void	**remove**(**Drawable** & d)
	Removes a drawable from the content container.
void	**useSMOCDrawing**(bool enabled)
	Determines whether to use JSMOC or painter's algorithm for drawing.

Protected Attributes inherited from Screen

Container	**container**
	The container contains the contents of the screen.
Drawable *	**focus**
	The drawable currently in focus (set when DOWN_PRESSED is received).

Public Functions Documentation

bind

void bind (T & presenter)

Binds an instance of a specific **Presenter** type (subclass) to the **View** instance.

This function is called automatically when a new presenter/view pair is activated.

Parameters:

presenter The specific **Presenter** to be associated with the **View**.

View

```
View ( )
```

Protected Attributes Documentation

presenter

T * presenter

Pointer to the **Presenter** associated with this view.

Version: 4.16

Widget

A Widget is an element which can be displayed (drawn) in the framebuffer. Hence a Widget is a subclass of Drawable. It implements getLastChild(), but leaves the implementation of draw() and getSolidRect() to subclasses of Widget, so it is still an abstract class.

If a Widget contains more than one logical element, consider implementing several subclasses of Widget and create a Container with the Widgets.

See: Drawable

Inherits from: Drawable

Inherited by: AbstractButton, Box, CanvasWidget, Image, PixelDataWidget, SnapshotWidget, TextArea, WipeTransition< templateDirection >::FullSolidRect

Public Functions

virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Additional inherited members

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)

	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
Drawable *	**getNextSibling**()
	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)

	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const
	Gets whether this **Drawable** receives touch events or not.
bool	**isVisible**() const
	Gets whether this **Drawable** is visible.
virtual void	**moveRelative**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**moveTo**(int16_t x, int16_t y)
	Moves the drawable.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
void	**setPosition**(const **Drawable** & drawable)
	Sets the position of the **Drawable** to the same as the given **Drawable**.
void	**setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets the size and position of this **Drawable**, relative to its parent.
void	**setTouchable**(bool touch)
	Controls whether this **Drawable** receives touch events or not.
void	**setVisible**(bool vis)
	Controls whether this **Drawable** should be visible.
virtual void	**setWidth**(int16_t width)

	Sets the width of this drawable.
void	**setWidthHeight**(const **Bitmap** & bitmap)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Drawable** & drawable)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(const **Rect** & rect)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
void	**setWidthHeight**(int16_t width, int16_t height)
	Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).
virtual void	**setX**(int16_t x)
	Sets the x coordinate of this **Drawable**, relative to its parent.
void	**setXY**(const **Drawable** & drawable)
	Sets the x and y coordinates of this **Drawable**.
void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**

	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Functions Documentation

getLastChild

```
virtual void getLastChild ( int16_t       x ,
                            int16_t       y ,
                            Drawable ** last
                            )
```

Since a **Widget** is only one **Drawable**, Widget::getLastChild simply yields itself as result, but only if the **Widget** isVisible and isTouchable.

Parameters:

x	Not used since this **Widget** is the only "child".
y	Not used since this **Widget** is the only "child".
last	Result, the address of the actual instance of the **Widget**.

Reimplements: **touchgfx::Drawable::getLastChild**

Version: 4.16

WildcardTextButtonStyle

A wildcard text button style. An wildcard text button style. This class is supposed to be used with one of the ButtonTrigger classes to create a functional button. This class will show a text with a wildcard in one of two colors depending on the state of the button (pressed or released).

The WildcardTextButtonStyle does not set the size of the enclosing container (normally AbstractButtonContainer). The size must be set manually.

To get a background behind the text, use WildcardTextButtonStyle together with e.g. ImageButtonStyle:

```
WildcardTextButtonStyle<ImageButtonStyle<ClickButtonTrigger> > myButton;
```

The position of the text can be adjusted with setTextXY (default is centered).

Template Parameters:

- **T** Generic type parameter. Typically a AbstractButtonContainer subclass.

See: AbstractButtonContainer

Inherits from: T

Public Functions

void	**setWildcardText**(**TypedText** t)
	Sets wildcard text.
void	**setWildcardTextBuffer**(const **Unicode::UnicodeChar** * buffer)
	Sets wildcard text buffer.
void	**setWildcardTextColors**(**colortype** newColorReleased, **colortype** newColorPressed)
	Sets wild card text colors.
void	**setWildcardTextPosition**(int16_t x, int16_t y, int16_t width, int16_t height)
	Sets text position and dimensions.
void	**setWildcardTextRotation**(**TextRotation** rotation)

Sets wildcard text rotation.

void **setWildcardTextX**(int16_t x)

Sets wildcard text x coordinate.

void **setWildcardTextXY**(int16_t x, int16_t y)

Sets wildcard text position.

void **setWildcardTextY**(int16_t y)

Sets wildcard text y coordinate.

WildcardTextButtonStyle()

Protected Functions

virtual void **handleAlphaUpdated**()

Handles what should happen when the alpha is updated.

virtual void **handlePressedUpdated**()

Handles what should happen when the pressed state is updated.

Protected Attributes

colortype **colorPressed**

The color pressed.

colortype **colorReleased**

The color released.

TextAreaWithOneWildcard **wildcardText**

The wildcard text.

Public Functions Documentation

setWildcardText

void setWildcardText (TypedText t)

Sets wildcard text.

Parameters:

t A **TypedText** to process.

setWildcardTextBuffer

void setWildcardTextBuffer (const Unicode::UnicodeChar * buffer)

Sets wildcard text buffer.

Parameters:

buffer If non-null, the buffer.

setWildcardTextColors

void setWildcardTextColors (colortype newColorReleased ,
colortype newColorPressed
)

Sets wild card text colors.

Parameters:

newColorReleased The new color released.
newColorPressed The new color pressed.

setWildcardTextPosition

void setWildcardTextPosition (int16_t x ,
int16_t y ,
int16_t width ,
int16_t height
)

Sets text position and dimensions.

Parameters:

x The x coordinate.
y The y coordinate.
width The width of the text.
height The height of the text.

setWildcardTextRotation

void setWildcardTextRotation (TextRotation rotation)

Sets wildcard text rotation.

Parameters:

rotation The rotation.

setWildcardTextX

void setWildcardTextX (int16_t x)

Sets wildcard text x coordinate.

Parameters:

x The x coordinate.

setWildcardTextXY

void setWildcardTextXY (int16_t x ,
int16_t y
)

Sets wildcard text position.

Parameters:

x The x coordinate.
y The y coordinate.

setWildcardTextY

void setWildcardTextY (int16_t y)

Sets wildcard text y coordinate.

Parameters:

y The y coordinate.

WildcardTextButtonStyle

WildcardTextButtonStyle ( )

Protected Functions Documentation

handleAlphaUpdated

virtual void handleAlphaUpdated ( )

Handles what should happen when the alpha is updated.

handlePressedUpdated

virtual void handlePressedUpdated ( )

Handles what should happen when the pressed state is updated.

Protected Attributes Documentation

colorPressed

colortype colorPressed

The color pressed.

colorReleased

colortype colorReleased

The color released.

wildcardText

TextAreaWithOneWildcard wildcardText

The wildcard text.

Version: 4.16

WipeTransition

A Transition that expands the new screen over the previous from the given direction. This transition only draws the pixels in the framebuffer once, and never moves any pixels. It is therefore very usefull on MCUs with limited performance.

Inherits from: Transition

Public Classes

class	**FullSolidRect**
	A **Widget** that reports solid and but does not draw anything.

Public Functions

virtual void	**handleTickEvent**()
	Handles the tick event when transitioning.
virtual void	**init**()
	Initializes the transition.
virtual void	**invalidate**()
	Wipe transition does not require an invalidation.
virtual void	**tearDown**()
	Tears down the Animation.
	WipeTransition(const uint8_t transitionSteps =20)
	Initializes a new instance of the **WipeTransition** class.

Additional inherited members

Public Functions inherited from Transition

bool	**isDone**() const

	Query if the transition is done transitioning.
virtual void	**setScreenContainer**(**Container** & cont)
	Sets the **ScreenContainer**.
	Transition()
	Initializes a new instance of the **Transition** class.
virtual	**~Transition**()
	Finalizes an instance of the **Transition** class.

Protected Attributes inherited from Transition

bool	**done**
	Flag that indicates when the transition is done. This should be set by implementing classes.
Container *	**screenContainer**
	The screen **Container** of the **Screen** transitioning to.

Public Functions Documentation

handleTickEvent

virtual void handleTickEvent ( )

Handles the tick event when transitioning.

It uncovers and invalidates increasing parts of the new screen elements.

Reimplements: **touchgfx::Transition::handleTickEvent**

init

virtual void init ( )

Initializes the transition.

Called after the constructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::init**

invalidate

virtual void invalidate ( )

Wipe transition does not require an invalidation.

Invalidation is handled by the class. Do no invalidation initially.

Reimplements: **touchgfx::Transition::invalidate**

tearDown

virtual void tearDown ( )

Tears down the Animation.

Called before the destructor is called, when the application changes the transition.

Reimplements: **touchgfx::Transition::tearDown**

WipeTransition

WipeTransition (const uint8_t transitionSteps =20)

Initializes a new instance of the **WipeTransition** class.

Parameters:

transitionSteps (Optional) Number of steps in the transition animation.

Version: 4.16

ZoomAnimationImage

Class for optimizing and wrapping move and zoom operations on a ScalableImage. The ZoomAnimationImage takes two bitmaps representing the same image but at a small and a large resolution. These bitmaps should be the sizes that are used when not animating the image. The ZoomAnimationImage will use an Image for displaying the Bitmap when its width and height matches either of them. When it does not match the size of one of the bitmaps, it will use a ScalableImage instead. The main idea is that the supplied bitmaps should be the end points of the zoom animation so that it ends up using an Image when not animating. This is, however, not a requirement. You can animate from and to sizes that are not equal the sizes of the bitmaps. The result is a container that has the high performance of an ordinary image when the size matches the pre-rendered bitmaps. Moreover it supplies easy to use animation functions that lets you zoom and move the image.

Note: Since this container uses the ScalableImage it has the same restrictions as a ScaleableImage, i.e. 1bpp is not supported.

Inherits from: Container, Drawable

Public Types

enum	**ZoomMode** { FIXED_CENTER, FIXED_LEFT, FIXED_RIGHT, FIXED_TOP, FIXED_BOTTOM, FIXED_LEFT_AND_TOP, FIXED_RIGHT_AND_TOP, FIXED_LEFT_AND_BOTTOM, FIXED_RIGHT_AND_BOTTOM }
	A ZoomMode describes in which direction the image will grow/shrink when do a zoom animation.

Protected Types

enum	**States** { ANIMATE_ZOOM, ANIMATE_ZOOM_AND_MOVE, NO_ANIMATION }
	Animation states.

Public Functions

void	**cancelZoomAnimation**()
	Cancel zoom animation.

virtual uint8_t	**getAlpha**() const
	Gets the current alpha value of the widget.
virtual uint16_t	**getAnimationDelay**() const
	Gets the current animation delay.
Bitmap	**getLargeBitmap**() const
	Gets the large bitmap.
virtual **ScalableImage::ScalingAlgorithm**	**getScalingMode**()
	Gets the scaling algorithm of the **ScalableImage**.
Bitmap	**getSmallBitmap**() const
	Gets the small bitmap.
virtual void	**handleTickEvent**()
	Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.
bool	**isZoomAnimationRunning**() const
	Is there currently an animation running.
virtual void	**setAlpha**(uint8_t newAlpha)
	Sets the opacity (alpha value).
virtual void	**setAnimationDelay**(uint16_t delay)
	Sets a delay on animations done by the **ZoomAnimationImage**.
void	**setAnimationEndedCallback**(**GenericCallback**< const **ZoomAnimationImage** & > & callback)
	Associates an action to be performed when the animation ends.
void	**setBitmaps**(const **Bitmap** & smallBitmap, const **Bitmap** & largeBitmap)
	Initializes the bitmap of the image to be used.
virtual void	**setHeight**(int16_t height)
	Sets the height of this drawable.
virtual void	**setScalingMode**(**ScalableImage::ScalingAlgorithm** mode)
	Sets the algorithm to be used.

virtual void	**setWidth**(int16_t width)
	Sets the width of this drawable.
void	**startZoomAndMoveAnimation**(int16_t endX, int16_t endY, int16_t endWidth, int16_t endHeight, uint16_t duration, **ZoomMode** zoomMode =**FIXED_LEFT_AND_TOP**, **EasingEquation** xProgressionEquation =&**EasingEquations::linearEaseNone**, **EasingEquation** yProgressionEquation =&**EasingEquations::linearEaseNone**, **EasingEquation** widthProgressionEquation =&**EasingEquations::linearEaseNone**, **EasingEquation** heightProgressionEquation =&**EasingEquations::linearEaseNone**)
	Setup and starts the zoom and move animation.
void	**startZoomAnimation**(int16_t endWidth, int16_t endHeight, uint16_t duration, **ZoomMode** zoomMode =**FIXED_LEFT_AND_TOP**, **EasingEquation** widthProgressionEquation =&**EasingEquations::linearEaseNone**, **EasingEquation** heightProgressionEquation =&**EasingEquations::linearEaseNone**)
	Setup and starts the zoom animation.
	ZoomAnimationImage()

Protected Functions

virtual void	**setCurrentState**(**States** state)
	Sets the current animation state and reset the animation counter.
void	**startTimerAndSetParameters**(int16_t endWidth, int16_t endHeight, uint16_t duration, **ZoomMode** zoomMode, **EasingEquation** widthProgressionEquation, **EasingEquation** heightProgressionEquation)
	Starts timer and set parameters.
virtual void	**updateRenderingMethod**()
	Chooses the optimal rendering of the image given the current width and height.
virtual void	**updateZoomAnimationDeltaXY**()

Calculates the change in X and Y caused by the zoom animation given the current **ZoomMode**.

Protected Attributes

uint32_t	**animationCounter**
	The progress counter for the animation.
uint16_t	**animationDuration**
	Duration of the animation.
GenericCallback< const **ZoomAnimationImage** & > *	**animationEndedAction**
	The animation ended action.
States	**currentState**
	The current animation state.
ZoomMode	**currentZoomMode**
	The ZoomMode to use by the animation.
Image	**image**
	The image for displaying the bitmap when the width/height is equal one of the bitmaps.
Bitmap	**largeBmp**
	The bitmap representing the large image.
int16_t	**moveAnimationEndX**
	The move animation end x coordinate.
int16_t	**moveAnimationEndY**
	The move animation end y coordinate.
EasingEquation	**moveAnimationXEquation**
	The move animation x coordinate equation.
EasingEquation	**moveAnimationYEquation**
	The move animation y coordinate equation.
ScalableImage	**scalableImage**

	The scalable image for displaying the bitmap when the width/height is not equal one of the bitmaps.
Bitmap	**smallBmp**
	The bitmap representing the small image.
uint16_t	**zoomAnimationDelay**
	A delay that is applied before animation start. Expressed in ticks.
int16_t	**zoomAnimationDeltaX**
	The zoom animation delta x.
int16_t	**zoomAnimationDeltaY**
	The zoom animation delta y.
int16_t	**zoomAnimationEndHeight**
	Height of the zoom animation end.
int16_t	**zoomAnimationEndWidth**
	Width of the zoom animation end.
EasingEquation	**zoomAnimationHeightEquation**
	The zoom animation height equation.
int16_t	**zoomAnimationStartHeight**
	Height of the zoom animation start.
int16_t	**zoomAnimationStartWidth**
	Width of the zoom animation start.
int16_t	**zoomAnimationStartX**
	The zoom animation start x coordinate.
int16_t	**zoomAnimationStartY**
	The zoom animation start y coordinate.
EasingEquation	**zoomAnimationWidthEquation**
	The zoom animation width equation.

Additional inherited members

Public Functions inherited from Container

virtual void	**add**(**Drawable** & d)
	Adds a Drawable instance as child to this **Container**.
	Container()
virtual bool	**contains**(const **Drawable** & d)
	Query if a given Drawable has been added directly to this **Container**.
virtual void	**draw**(const **Rect** & invalidatedArea) const
	Draw this drawable.
virtual void	**forEachChild**(**GenericCallback**< **Drawable** & > * function)
	Executes the specified callback function for each child in the **Container**.
virtual **Drawable** *	**getFirstChild**()
	Obtain a pointer to the first child of this container.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last)
	Gets the last child in the list of children in this **Container**.
virtual **Rect**	**getSolidRect**() const
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual void	**insert**(**Drawable** * previous, **Drawable** & d)
	Inserts a Drawable after a specific child node.
virtual void	**remove**(**Drawable** & d)
	Removes a Drawable from the container by removing it from the linked list of children.
virtual void	**removeAll**()
	Removes all children in the **Container** by resetting their parent and sibling pointers.
virtual void	**unlink**()
	Removes all children by unlinking the first child.

Protected Functions inherited from Container

virtual **Rect**	**getContainedArea**() const

	Gets a rectangle describing the total area covered by the children of this container.
virtual void	**moveChildrenRelative**(int16_t deltaX, int16_t deltaY)
	Calls moveRelative on all children.

Protected Attributes inherited from Container

Drawable *	**firstChild**
	Pointer to the first child of this container. Subsequent children can be found through firstChild's nextSibling.

Public Functions inherited from Drawable

virtual void	**childGeometryChanged**()
	This function can be called on parent nodes to signal that the size or position of one or more of its children has changed.
virtual void	**draw**(const **Rect** & invalidatedArea) const =0
	Draw this drawable.
	Drawable()
	Initializes a new instance of the **Drawable** class.
void	**drawToDynamicBitmap**(**BitmapId** id)
	Render the **Drawable** object into a dynamic bitmap.
Rect	**getAbsoluteRect**() const
	Helper function for obtaining the rectangle this **Drawable** covers, expressed in absolute coordinates.
virtual **Drawable** *	**getFirstChild**()
	Function for obtaining the first child of this drawable if any.
int16_t	**getHeight**() const
	Gets the height of this **Drawable**.
virtual void	**getLastChild**(int16_t x, int16_t y, **Drawable** ** last) =0
	Function for obtaining the the last child of this drawable that intersects with the specified point.
Drawable *	**getNextSibling**()

	Gets the next sibling node.
Drawable *	**getParent**() const
	Returns the parent node.
const **Rect** &	**getRect**() const
	Gets the rectangle this **Drawable** covers, in coordinates relative to its parent.
virtual **Rect**	**getSolidRect**() const =0
	Get (the largest possible) rectangle that is guaranteed to be solid (opaque).
virtual **Rect**	**getSolidRectAbsolute**()
	Helper function for obtaining the largest solid rect (as implemented by **getSolidRect()**) expressed in absolute coordinates.
virtual void	**getVisibleRect**(**Rect** & rect) const
	Function for finding the visible part of this drawable.
int16_t	**getWidth**() const
	Gets the width of this **Drawable**.
int16_t	**getX**() const
	Gets the x coordinate of this **Drawable**, relative to its parent.
int16_t	**getY**() const
	Gets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**handleClickEvent**(const **ClickEvent** & evt)
	Defines the event handler interface for ClickEvents.
virtual void	**handleDragEvent**(const **DragEvent** & evt)
	Defines the event handler interface for DragEvents.
virtual void	**handleGestureEvent**(const **GestureEvent** & evt)
	Defines the event handler interface for GestureEvents.
virtual void	**invalidate**() const
	Tell the framework that this entire **Drawable** needs to be redrawn.
virtual void	**invalidateRect**(**Rect** & invalidatedArea) const
	Request that a region of this drawable is redrawn.
bool	**isTouchable**() const

Gets whether this **Drawable** receives touch events or not.

bool **isVisible**() const

Gets whether this **Drawable** is visible.

virtual void **moveRelative**(int16_t x, int16_t y)

Moves the drawable.

virtual void **moveTo**(int16_t x, int16_t y)

Moves the drawable.

void **setPosition**(const **Drawable** & drawable)

Sets the position of the **Drawable** to the same as the given **Drawable**.

void **setPosition**(int16_t x, int16_t y, int16_t width, int16_t height)

Sets the size and position of this **Drawable**, relative to its parent.

void **setTouchable**(bool touch)

Controls whether this **Drawable** receives touch events or not.

void **setVisible**(bool vis)

Controls whether this **Drawable** should be visible.

void **setWidthHeight**(const **Bitmap** & bitmap)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Drawable** & drawable)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(const **Rect** & rect)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

void **setWidthHeight**(int16_t width, int16_t height)

Sets the dimensions (width and height) of the **Drawable** without changing the x and y coordinates).

virtual void **setX**(int16_t x)

Sets the x coordinate of this **Drawable**, relative to its parent.

void **setXY**(const **Drawable** & drawable)

Sets the x and y coordinates of this **Drawable**.

void	**setXY**(int16_t x, int16_t y)
	Sets the x and y coordinates of this **Drawable**, relative to its parent.
virtual void	**setY**(int16_t y)
	Sets the y coordinate of this **Drawable**, relative to its parent.
virtual void	**translateRectToAbsolute**(**Rect** & r) const
	Helper function for converting a region of this **Drawable** to absolute coordinates.
virtual	**~Drawable**()
	Finalizes an instance of the **Drawable** class.

Protected Attributes inherited from Drawable

Drawable *	**nextSibling**
	Pointer to the next **Drawable**.
Drawable *	**parent**
	Pointer to this drawable's parent.
Rect	**rect**
	The coordinates of this **Drawable**, relative to its parent.
bool	**touchable**
	True if this drawable should receive touch events.
bool	**visible**
	True if this drawable should be drawn.

Public Types Documentation

ZoomMode

enum ZoomMode

A ZoomMode describes in which direction the image will grow/shrink when do a zoom animation.

FIXED_CENTER	The small image will grow from the center of the large image.

FIXED_LEFT	The small image will grow from the middle of the left side of the large image.
FIXED_RIGHT	The small image will grow from the middle of the right side of the large image.
FIXED_TOP	The small image will grow from the middle of the top of the large image.
FIXED_BOTTOM	The small image will grow from the middle of the bottom of the large image.
FIXED_LEFT_AND_TOP	The small image will grow from the top left corner of the large image.
FIXED_RIGHT_AND_TOP	The small image will grow from the top right corner of the large image.
FIXED_LEFT_AND_BOTTOM	The small image will grow from the bottom left corner of the large image.
FIXED_RIGHT_AND_BOTTOM	The small image will grow from the bottom right corner of the large image.

Protected Types Documentation

States

enum States

Animation states.

ANIMATE_ZOOM	Zoom animation state.
ANIMATE_ZOOM_AND_MOVE	Zoom and move animation state.
NO_ANIMATION	No animation state.

Public Functions Documentation

cancelZoomAnimation

void cancelZoomAnimation ( )

Cancel zoom animation.

The image is left in the position and size it is currently at.

getAlpha

virtual uint8_t getAlpha ( ) const

Gets the current alpha value of the widget.

The alpha value is in range 255 (solid) to 0 (invisible).

Returns:

The current alpha value.

See also:

setAlpha

getAnimationDelay

virtual uint16_t getAnimationDelay ( ) const

Gets the current animation delay.

Returns:

The current animation delay. Expressed in ticks.

See also:

setAnimationDelay

getLargeBitmap

Bitmap getLargeBitmap ( ) const

Gets the large bitmap.

Returns:

the large bitmap.

See also:

setBitmaps

getScalingMode

virtual ScalableImage::ScalingAlgorithm getScalingMode ( )

Gets the scaling algorithm of the **ScalableImage**.

Returns:

the scaling algorithm used.

See also:

setScalingMode

getSmallBitmap

Bitmap getSmallBitmap ( ) const

Gets the small bitmap.

Returns:

the small bitmap.

See also:

setBitmaps

handleTickEvent

virtual void handleTickEvent ( )

Called periodically by the framework if the **Drawable** instance has subscribed to timer ticks.

See also:

Application::registerTimerWidget

Reimplements: **touchgfx::Drawable::handleTickEvent**

isZoomAnimationRunning

bool isZoomAnimationRunning ( ) const

Is there currently an animation running.

Returns:

true if there is an animation running.

setAlpha

virtual void setAlpha (uint8_t newAlpha)

Sets the opacity (alpha value).

This can be used to fade it away by gradually decreasing the alpha value from 255 (solid) to 0 (invisible).

Parameters:

newAlpha The new alpha value. 255=solid, 0=invisible.

NOTE

The user code must call invalidate() in order to update the display.

See also:

getAlpha

setAnimationDelay

virtual void setAnimationDelay (uint16_t delay)

Sets a delay on animations done by the **ZoomAnimationImage**.

Defaults to 0 which means that the animation starts immediately.

Parameters:

delay The delay in ticks.

See also:

getAnimationDelay

setAnimationEndedCallback

void setAnimationEndedCallback (GenericCallback< const ZoomAnimationImage & > & callback)

Associates an action to be performed when the animation ends.

Parameters:

callback The callback to be executed. The callback will be given a reference to the **ZoomAnimationImage**.

See also:

GenericCallback

setBitmaps

void setBitmaps (const Bitmap & smallBitmap ,
const Bitmap & largeBitmap
)

Initializes the bitmap of the image to be used.

The bitmaps should represent the same image in the two needed static resolutions.

Parameters:

smallBitmap The image in the smallest resolution.
largeBitmap The image in the largest resolution.

NOTE

The size of the bitmaps do not in any way limit the size of the ZoomAnimationImage and it is possible to scale the image beyond the sizes of these bitmaps.

See also:

getSmallBitmap, **getLargeBitmap**

setHeight

virtual void setHeight (int16_t height)

Sets the height of this drawable.

Parameters:

height The new height.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.ZoomAnimationImage diverts from the normal behavior by automatically invalidating which causes a redraw.

Reimplements: touchgfx::Drawable::setHeight

setScalingMode

virtual void setScalingMode (ScalableImage::ScalingAlgorithm mode)

Sets the algorithm to be used.

In short, there is currently a value for fast (nearest neighbor) and a value for slow (bilinear interpolation). Default is **ScalableImage::NEAREST_NEIGHBOR** since moving images do not need to be of the best quality, until they stop moving. If the image moves only a little bit, or moves/resizes slowly, consider using ScaleableImage::BILINEAR_INTERPOLATION.

Parameters:

mode The algorithm to use when rendering.

See also:

ScalableImage::ScalingAlgorithm, **getScalingMode**

setWidth

virtual void setWidth (int16_t width)

Sets the width of this drawable.

Parameters:

width The new width.

NOTE

For most Drawable widgets, changing this does normally not automatically yield a redraw.ZoomAnimationImage diverts from the normal behavior by automatically invalidating which causes a redraw.

Reimplements: touchgfx::Drawable::setWidth

startZoomAndMoveAnimation

void startZoomAndMoveAnimation (int16_t endX ,
int16_t endY ,
int16_t endWidth ,

	int16_t	endHeight ,
	uint16_t	duration ,
	ZoomMode	zoomMode =FIXED_LEFT_AND_TOP,
	EasingEquation	xProgressionEquation =&EasingEquations::linearEaseNone,
	EasingEquation	yProgressionEquation =&EasingEquations::linearEaseNone,
	EasingEquation	widthProgressionEquation =&EasingEquations::linearEaseNone,
	EasingEquation	heightProgressionEquation =&EasingEquations::linearEaseNone
)		

Setup and starts the zoom and move animation.

At end of the animation the image will have been resized to the endWidth and endHeight and have moved from its original position to the endX and endY. Please note that the ZoomMode might influence the actual end position since the zoom transformation might change the X and Y of the image. The ZoomMode **FIXED_LEFT_AND_TOP** ensures that the endX and endY will be the actual end position.

The development of the width, height, X and Y during the animation is described by the supplied **EasingEquations**. The container is registered as a TimerWidget and automatically unregistered when the animation has finished.

Parameters:

endX	The X position of the image at animation end. Relative to the container or view that holds the **ZoomAnimationImage**.
endY	The Y position of the image at animation end. Relative to the container or view that holds the **ZoomAnimationImage**.
endWidth	The width of the image at animation end.
endHeight	The height of the image at animation end.
duration	The duration of the animation measured in ticks.
zoomMode	(Optional) The zoom mode that will be used during the animation. Default is **FIXED_LEFT_AND_TOP**.
xProgressionEquation	(Optional) The equation that describes the development of the X position during the animation. Default is **EasingEquations::linearEaseNone**.
yProgressionEquation	(Optional) The equation that describes the development of the Y position during the animation. Default is **EasingEquations::linearEaseNone**.
widthProgressionEquation	(Optional) The equation that describes the development of the width during the animation. Default is **EasingEquations::linearEaseNone**.
heightProgressionEquation	(Optional) The equation that describes the development of the height during the animation. Default is **EasingEquations::linearEaseNone**.

startZoomAnimation

```
void startZoomAnimation ( int16_t endWidth ,
                          int16_t endHeight ,
                          uint16_t duration ,
                          ZoomMode zoomMode =FIXED_LEFT_AND_TOP,
                          EasingEquation widthProgressionEquation
                          =&EasingEquations::linearEaseNone,
                          EasingEquation heightProgressionEquation
                          =&EasingEquations::linearEaseNone
                          )
```

Setup and starts the zoom animation.

At end of the animation the image will have been resized to the endWidth and endHeight. The development of the width and height during the animation is described by the supplied **EasingEquations**. The container is registered as a TimerWidget and automatically unregistered when the animation has finished.

Parameters:

endWidth	The width of the image at animation end.
endHeight	The height of the image at animation end.
duration	The duration of the animation measured in ticks.
zoomMode	(Optional) The zoom mode that will be used during the animation. Default is **FIXED_LEFT_AND_TOP**.
widthProgressionEquation	(Optional) The equation that describes the development of the width during the animation. Default is **EasingEquations::linearEaseNone**.
heightProgressionEquation	(Optional) The equation that describes the development of the height during the animation. Default is **EasingEquations::linearEaseNone**.

NOTE

The animation follows the specified ZoomMode so the X and Y coordinates of the image might change during animation.

ZoomAnimationImage

```
ZoomAnimationImage ( )
```

Protected Functions Documentation

setCurrentState

virtual void setCurrentState (States state)

Sets the current animation state and reset the animation counter.

Parameters:

state The new state.

startTimerAndSetParameters

```
void startTimerAndSetParameters ( int16_t endWidth ,
                                  int16_t endHeight ,
                                  uint16_t duration ,
                                  ZoomMode zoomMode ,
                                  EasingEquation widthProgressionEquation ,
                                  EasingEquation heightProgressionEquation
                                  )
```

Starts timer and set parameters.

Contains code shared between **startZoomAnimation()** and **startZoomAndMoveAnimation()**. If both delay and duration is zero, the end position and size is applied and the animation is ended immediately.

Parameters:

endWidth	The end width.
endHeight	The end height.
duration	The duration.
zoomMode	The zoom mode.
widthProgressionEquation	The width progression equation.
heightProgressionEquation	The height progression equation.

updateRenderingMethod

virtual void updateRenderingMethod ( )

Chooses the optimal rendering of the image given the current width and height.

If the dimensions match either the small or large bitmap, that will be used, otherwise the large image will be scaled using the defined scaling mode.

See also:

setScalingMode, **setBitmaps**

updateZoomAnimationDeltaXY

virtual void updateZoomAnimationDeltaXY ( )

Calculates the change in X and Y caused by the zoom animation given the current **ZoomMode**.

Protected Attributes Documentation

animationCounter

uint32_t animationCounter

The progress counter for the animation.

animationDuration

uint16_t animationDuration

Duration of the animation.

animationEndedAction

GenericCallback< const ZoomAnimationImage & > * animationEndedAction

The animation ended action.

currentState

States currentState

The current animation state.

currentZoomMode

ZoomMode currentZoomMode

The ZoomMode to use by the animation.

image

Image image

The image for displaying the bitmap when the width/height is equal one of the bitmaps.

largeBmp

Bitmap largeBmp

The bitmap representing the large image.

moveAnimationEndX

int16_t moveAnimationEndX

The move animation end x coordinate.

moveAnimationEndY

int16_t moveAnimationEndY

The move animation end y coordinate.

moveAnimationXEquation

EasingEquation moveAnimationXEquation

The move animation x coordinate equation.

moveAnimationYEquation

EasingEquation moveAnimationYEquation

The move animation y coordinate equation.

scalableImage

ScalableImage scalableImage

The scalable image for displaying the bitmap when the width/height is not equal one of the bitmaps.

smallBmp

Bitmap smallBmp

The bitmap representing the small image.

zoomAnimationDelay

uint16_t zoomAnimationDelay

A delay that is applied before animation start. Expressed in ticks.

zoomAnimationDeltaX

int16_t zoomAnimationDeltaX

The zoom animation delta x.

zoomAnimationDeltaY

int16_t zoomAnimationDeltaY

The zoom animation delta y.

zoomAnimationEndHeight

int16_t zoomAnimationEndHeight

Height of the zoom animation end.

zoomAnimationEndWidth

int16_t zoomAnimationEndWidth

Width of the zoom animation end.

zoomAnimationHeightEquation

EasingEquation zoomAnimationHeightEquation

The zoom animation height equation.

zoomAnimationStartHeight

int16_t zoomAnimationStartHeight

Height of the zoom animation start.

zoomAnimationStartWidth

int16_t zoomAnimationStartWidth

Width of the zoom animation start.

zoomAnimationStartX

int16_t zoomAnimationStartX

The zoom animation start x coordinate.

zoomAnimationStartY

int16_t zoomAnimationStartY

The zoom animation start y coordinate.

zoomAnimationWidthEquation

EasingEquation zoomAnimationWidthEquation

The zoom animation width equation.

Version: 4.16

Globals

The global touchgfx namespace.
All TouchGFX framework enums, Type definitions, global functions and global variables are placed in this namespace.

Enums

BlitOperations

enum BlitOperations

The Blit Operations.

BLIT_OP_COPY	Copy the source to the destination.
BLIT_OP_FILL	Fill the destination with color.
BLIT_OP_COPY_WITH_ALPHA	Copy the source to the destination using the given alpha.
BLIT_OP_FILL_WITH_ALPHA	Fill the destination with color using the given alpha.
BLIT_OP_COPY_WITH_TRANSPARENT_PIXELS	Deprecated, ignored. (Copy the source to the destination, but not the transparent pixels)
BLIT_OP_COPY_ARGB8888	Copy the source to the destination, performing per-pixel alpha blending.
BLIT_OP_COPY_ARGB8888_WITH_ALPHA	Copy the source to the destination, performing per-pixel alpha blending and blending the result with an image-wide alpha.
BLIT_OP_COPY_A4	Copy 4-bit source text to destination, performing per-pixel alpha blending.
BLIT_OP_COPY_A8	Copy 8-bit source text to destination, performing per-pixel alpha blending.

Location: `touchgfx/hal/BlitOp.hpp`

Direction

enum Direction

Values that represent directions.

NORTH	An enum constant representing the north option.
SOUTH	An enum constant representing the south option.
EAST	An enum constant representing the east option.
WEST	An enum constant representing the west option.

Location: `touchgfx/hal/Types.hpp`

DisplayOrientation

enum DisplayOrientation

Values that represent display orientations.

ORIENTATION_LANDSCAPE	The display has more pixels from left to right than from top to bottom.
ORIENTATION_PORTRAIT	The display has more pixels from top to bottom than from right to left.

Location: `touchgfx/hal/Types.hpp`

DisplayRotation

enum DisplayRotation

Values that represent display rotations.

rotate0	The display is oriented like the framebuffer.
rotate90	The display is rotated 90 degrees compared to the framebuffer layout.

Location: `touchgfx/hal/Types.hpp`

DMAType

enum DMAType

Values that represent dma types.

DMA_TYPE_GENERIC	Generic DMA Implementation.

DMA_TYPE_CHROMART	ChromART hardware DMA Implementation.

Location: `touchgfx/hal/Types.hpp`

FrameBuffer

enum FrameBuffer

Values that represent frame buffers.

FB_PRIMARY	First framebuffer.
FB_SECONDARY	Second framebuffer.
FB_TERTIARY	Third framebuffer.

Location: `touchgfx/hal/Types.hpp`

GlyphFlags

enum GlyphFlags

Glyph flag definitions.

GLYPH_DATA_KERNINGTABLEPOS_BIT8_10	The 8th, 9th and 10th bit of the kerningTablePos.
GLYPH_DATA_WIDTH_BIT8	The 9th bit of "width".
GLYPH_DATA_HEIGHT_BIT8	The 9th bit of "height".
GLYPH_DATA_TOP_BIT8	The 9th bit of "top".
GLYPH_DATA_TOP_BIT9	The sign bit of "top".
GLYPH_DATA_ADVANCE_BIT8	The 9th bit of "advance".

Location: `touchgfx/Font.hpp`

Gradient

enum Gradient

Values that represent gradients.

GRADIENT_HORIZONTAL	Horizontal gradient.

GRADIENT_VERTICAL	Vertical gradient.

Location: `touchgfx/hal/Types.hpp`

TextRotation

enum TextRotation

Values that represent text rotations.

TEXT_ROTATE_0	Text is written from left to right.
TEXT_ROTATE_90	Text is written from top to bottom.
TEXT_ROTATE_180	Text is written from right to left (upside down)
TEXT_ROTATE_270	Text is written bottom to top.

Location: `touchgfx/hal/Types.hpp`

WideTextAction

enum WideTextAction

Values that represent wide text actions.

WIDE_TEXT_NONE	Do nothing, simply cut the text in the middle of any character that extends beyond the width of the TextArea.
WIDE_TEXT_WORDWRAP	Wrap between words, ellipsis anywhere "Very long t...".
WIDE_TEXT_WORDWRAP_ELLIPSIS_AFTER_SPACE	Wrap between words, ellipsis anywhere only after space "Very long ...".
WIDE_TEXT_CHARWRAP	Wrap between any two characters, ellipsis anywhere, as used in Chinese.
WIDE_TEXT_CHARWRAP_DOUBLE_ELLIPSIS	Wrap between any two characters, double ellipsis anywhere, as used in Chinese.

Location: `touchgfx/hal/Types.hpp`

Type Definitions

Alignment

typedef uint8_t Alignment

Defines an alignment type.

Location: `touchgfx/hal/Types.hpp`

AnimatedImageClickButton

typedef AnimatedImageButtonStyle< ClickButtonTrigger > AnimatedImageClickButton

Defines an alias representing the animated image click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

AnimatedImageRepeatButton

typedef AnimatedImageButtonStyle< RepeatButtonTrigger > AnimatedImageRepeatButton

Defines an alias representing the animated image repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

AnimatedImageToggleButton

typedef AnimatedImageButtonStyle< ToggleButtonTrigger > AnimatedImageToggleButton

Defines an alias representing the animated image toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

AnimatedImageTouchButton

typedef AnimatedImageButtonStyle< TouchButtonTrigger > AnimatedImageTouchButton

Defines an alias representing the animated image touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

BitmapId

typedef uint16_t BitmapId

This type shall be used by the application to define unique IDs for all bitmaps in the system.

Location: `touchgfx/Bitmap.hpp`

BoxClickButton

typedef BoxWithBorderButtonStyle< ClickButtonTrigger > BoxClickButton

Defines an alias representing the box click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

BoxRepeatButton

typedef BoxWithBorderButtonStyle< RepeatButtonTrigger > BoxRepeatButton

Defines an alias representing the box repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

BoxToggleButton

typedef BoxWithBorderButtonStyle< ToggleButtonTrigger > BoxToggleButton

Defines an alias representing the box toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

BoxTouchButton

typedef BoxWithBorderButtonStyle< TouchButtonTrigger > BoxTouchButton

Defines an alias representing the box touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

EasingEquation

typedef int16_t(* EasingEquation) (uint16_t, int16_t, int16_t, uint16_t)

This function pointer typedef matches the signature for all easing equations.

Location: `touchgfx/EasingEquations.hpp`

fixed16_16

typedef int32_t fixed16_16

A fixed point value using 16 bits for the decimal part and 16 bits for the integral part.

Location: `touchgfx/hal/Types.hpp`

fixed28_4

typedef int32_t fixed28_4

A fixed point value using 4 bits for the decimal part and 28 bits for the integral part.

Location: `touchgfx/hal/Types.hpp`

FontId

typedef uint16_t FontId

Defines an alias representing a Font ID.

Location: `touchgfx/Font.hpp`

GlyphNode

typedef struct touchgfx::GlyphNode GlyphNode

struct providing information about a glyph.

Location: `touchgfx/Font.hpp`

IconClickButton

typedef IconButtonStyle< ClickButtonTrigger > IconClickButton

Defines an alias representing the icon click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconImageClickButton

typedef ImageButtonStyle< IconButtonStyle< ClickButtonTrigger > > IconImageClickButton

Defines an alias representing the icon image click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconImageRepeatButton

typedef ImageButtonStyle< IconButtonStyle< RepeatButtonTrigger > > IconImageRepeatButton

Defines an alias representing the icon image repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconImageToggleButton

typedef ImageButtonStyle< IconButtonStyle< ToggleButtonTrigger > > IconImageToggleButton

Defines an alias representing the icon image toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconImageTouchButton

typedef ImageButtonStyle< IconButtonStyle< TouchButtonTrigger > > IconImageTouchButton

Defines an alias representing the icon image touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconRepeatButton

typedef IconButtonStyle< RepeatButtonTrigger > IconRepeatButton

Defines an alias representing the icon repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconToggleButton

typedef IconButtonStyle< ToggleButtonTrigger > IconToggleButton

Defines an alias representing the icon toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

IconTouchButton

typedef IconButtonStyle< TouchButtonTrigger > IconTouchButton

Defines an alias representing the icon touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

ImageClickButton

typedef ImageButtonStyle< ClickButtonTrigger > ImageClickButton

Defines an alias representing the image click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

ImageRepeatButton

typedef ImageButtonStyle< RepeatButtonTrigger > ImageRepeatButton

Defines an alias representing the image repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

ImageToggleButton

typedef ImageButtonStyle< ToggleButtonTrigger > ImageToggleButton

Defines an alias representing the image toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

ImageTouchButton

typedef ImageButtonStyle< TouchButtonTrigger > ImageTouchButton

Defines an alias representing the image touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

LanguageId

typedef uint16_t LanguageId

Language IDs generated by the text converter are uint16_t typedef'ed.

Location: `touchgfx/Texts.hpp`

RenderingVariant

typedef uint16_t RenderingVariant

Describes a combination of rendering algorithm, image format, and alpha information.

Location: `touchgfx/hal/Types.hpp`

TextClickButton

typedef TextButtonStyle< ClickButtonTrigger > TextClickButton

Defines an alias representing the text click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TextDirection

typedef uint8_t TextDirection

Defines a the direction to write text.

Location: `touchgfx/hal/Types.hpp`

TextRepeatButton

typedef TextButtonStyle< RepeatButtonTrigger > TextRepeatButton

Defines an alias representing the text repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TextToggleButton

typedef TextButtonStyle< ToggleButtonTrigger > TextToggleButton

Defines an alias representing the text toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TextTouchButton

typedef TextButtonStyle< TouchButtonTrigger > TextTouchButton

Defines an alias representing the text touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TiledImageClickButton

typedef TiledImageButtonStyle< ClickButtonTrigger > TiledImageClickButton

Defines an alias representing the tiled image click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TiledImageRepeatButton

typedef TiledImageButtonStyle< RepeatButtonTrigger > TiledImageRepeatButton

Defines an alias representing the tiled image repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TiledImageToggleButton

typedef TiledImageButtonStyle< ToggleButtonTrigger > TiledImageToggleButton

Defines an alias representing the tiled image toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TiledImageTouchButton

typedef TiledImageButtonStyle< TouchButtonTrigger > TiledImageTouchButton

Defines an alias representing the tiled image touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TwoWildcardTextClickButton

typedef TwoWildcardTextButtonStyle< ClickButtonTrigger > TwoWildcardTextClickButton

Defines an alias representing the wildcard text click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TwoWildcardTextRepeatButton

typedef TwoWildcardTextButtonStyle< RepeatButtonTrigger > TwoWildcardTextRepeatButton

Defines an alias representing the wildcard text repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TwoWildcardTextToggleButton

typedef TwoWildcardTextButtonStyle< ToggleButtonTrigger > TwoWildcardTextToggleButton

Defines an alias representing the wildcard text toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TwoWildcardTextTouchButton

typedef TwoWildcardTextButtonStyle< TouchButtonTrigger > TwoWildcardTextTouchButton

Defines an alias representing the wildcard text touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

TypedTextId

typedef uint16_t TypedTextId

Text IDs as generated by the text converter are simple uint16_t typedefs.

Location: `touchgfx/hal/Types.hpp`

WildcardTextClickButton

typedef WildcardTextButtonStyle< ClickButtonTrigger > WildcardTextClickButton

Defines an alias representing the wildcard text click button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

WildcardTextRepeatButton

typedef WildcardTextButtonStyle< RepeatButtonTrigger > WildcardTextRepeatButton

Defines an alias representing the wildcard text repeat button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

WildcardTextToggleButton

typedef WildcardTextButtonStyle< ToggleButtonTrigger > WildcardTextToggleButton

Defines an alias representing the wildcard text toggle button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

WildcardTextTouchButton

typedef WildcardTextButtonStyle< TouchButtonTrigger > WildcardTextTouchButton

Defines an alias representing the wildcard text touch button.

Location: `touchgfx/containers/buttons/Buttons.hpp`

Functions

abs

T abs (T d)

Simple implementation of the standard abs function.

Template Parameters:

T The type on which to perform the abs.

Parameters:

d The entity on which to perform the abs.

Returns:

The absolute (non-negative) value of d.

Location: `touchgfx/Utils.hpp`

ceil28_4

int32_t ceil28_4 (fixed28_4 value)

Round up a fixed28_4 value.

Parameters:

value The fixed28_4 value.

Returns:

The ceil result.

Location: `touchgfx/TextureMapTypes.hpp`

clz

int32_t clz (int32_t x)

Count leading zeros in the binary representation of absolute value of the given int32_t.

Parameters:

x The value to count the number of leading zeros in.

Returns:

The number of leading zeros (from 0 to 31).

See also:

clzu

Location: `touchgfx/Utils.hpp`

clzu

int32_t clzu (uint32_t x)

Count leading zeros in the binary representation of absolute value of the given uint32_t.

Parameters:

x The value to count the number of leading zeros in.

Returns:

The number of leading zeros (from 0 to 32).

See also:

clz

Location: `touchgfx/Utils.hpp`

finalizeTransition

```
static FORCE_INLINE_FUNCTION void finalizeTransition ( Screen *     newScreen ,
                                                       Presenter * newPresenter ,
                                                       Transition * newTransition
                                                       )
```

Finalize screen transition.

Private helper function for makeTransition. Do not use.

Parameters:

newScreen	If non-null, the new screen.
newPresenter	If non-null, the new presenter.
newTransition	If non-null, the new transition.

Location: `mvp/MVPApplication.hpp`

fixed28_4Mul

```
fixed28_4 fixed28_4Mul ( fixed28_4 a ,
                         fixed28_4 b
                         )
```

Multiply two fixed28_4 numbers.

Parameters:

a	The fixed28_4 to process.
b	The fixed28_4 to process.

Returns:

the result.

Location: `touchgfx/TextureMapTypes.hpp`

fixed28_4ToFloat

```
float fixed28_4ToFloat ( fixed28_4 value )
```

Convert fixed28_4 to float.

Parameters:

value The fixed28_4 value.

Returns:

The value as float.

Location: `touchgfx/TextureMapTypes.hpp`

floatToFixed16_16

fixed16_16 floatToFixed16_16 (float value)

Convert float to fixed16_16.

Parameters:

value The float value.

Returns:

The value as fixed16_16.

Location: `touchgfx/TextureMapTypes.hpp`

floatToFixed28_4

fixed28_4 floatToFixed28_4 (float value)

Convert float to fixed28_4.

Parameters:

value The float value.

Returns:

The value as fixed28_4.

Location: `touchgfx/TextureMapTypes.hpp`

floorDivMod

void floorDivMod (int32_t numerator ,

```
                int32_t    denominator ,
                int32_t &  floor ,
                int32_t &  mod
                )
```

Divides two fixed28_4 numbers and returns the result as well as the remainder.

Parameters:

numerator	The numerator.
denominator	The denominator.
floor	numerator/denominator.
mod	numerator%denominator.

Location: `touchgfx/TextureMapTypes.hpp`

FrameBufferAllocatorSignalBlockDrawn

```
void FrameBufferAllocatorSignalBlockDrawn ( )
```

Called by FrameBufferAllocator when a block is drawn and therefore ready for transfer.

The **LCD** driver should use this method to start a transfer.

Location: `touchgfx/hal/FrameBufferAllocator.hpp`

FrameBufferAllocatorWaitOnTransfer

```
void FrameBufferAllocatorWaitOnTransfer ( )
```

Called by FrameBufferAllocator to wait for a LCD Transfer, when the allocator has no free blocks.

The **LCD** driver can use this function to synchronize the UI thread with the transfer logic.

Location: `touchgfx/hal/FrameBufferAllocator.hpp`

gcd

```
T gcd ( T a ,
        T b
        )
```

Find greatest common divisor of two given numbers.

Template Parameters:

T Generic type parameter.

Parameters:

a The first number.
b The second number.

Returns:

The greatest common divisor.

Location: `touchgfx/Utils.hpp`

hw_init

void hw_init ( )

Function to perform generic hardware initialization of the board.

This function prototype is only provided as a convention.

Location: `touchgfx/hal/BoardConfiguration.hpp`

lookupBilinearRenderVariant

RenderingVariant lookupBilinearRenderVariant (const Bitmap & bitmap)

Returns the associated bilinear render variant based on the bitmap format.

This is used for quick determination of the type of bitmap during **TextureMapper** drawing.

Parameters:

bitmap The bitmap.

Returns:

A RenderingVariant based on the bitmap format.

Location: `touchgfx/Utils.hpp`

lookupNearestNeighborRenderVariant

RenderingVariant lookupNearestNeighborRenderVariant (const Bitmap & bitmap)

Returns the associated nearest neighbor render variant based on the bitmap format.

This is used for quick determination of the type of bitmap during **TextureMapper** drawing.

Parameters:

bitmap	The bitmap.

Returns:

A RenderingVariant based on the bitmap format.

Location: `touchgfx/Utils.hpp`

makeTransition

```
PresenterType * makeTransition ( Screen **      currentScreen ,
                                 Presenter **   currentPresenter ,
                                 MVPHeap &      heap ,
                                 Transition **  currentTrans ,
                                 ModelType *    model
                                 )
```

Function for effectuating a screen transition (i.e.

makes the requested new presenter/view pair active). Once this function has returned, the new screen has been transitioned to. Due to the memory allocation strategy of using the same memory area for all screens, the old view/presenter will no longer exist when this function returns.

Will properly clean up old screen (tearDownScreen, **Presenter::deactivate**) and call setupScreen/activate on new view/presenter pair. Will also make sure the view, presenter and model are correctly bound to each other.

Template Parameters:

ScreenType	Class type for the **View**.
PresenterType	Class type for the **Presenter**.
TransType	Class type for the **Transition**.
ModelType	Class type for the Model.

Parameters:

currentScreen	Pointer to pointer to the current view.
currentPresenter	Pointer to pointer to the current presenter.
heap	Reference to the heap containing the memory storage in which to allocate.
currentTrans	Pointer to pointer to the current transition.
model	Pointer to model.

Returns:

Pointer to the new **Presenter** of the requested type. Incidentally it will be the same value as the old presenter due to memory reuse.

Location: `mvp/MVPApplication.hpp`

memset

```
void memset ( void *    data ,
              uint8_t   c ,
              uint32_t size
              )
```

Simple implementation of the standard memset function.

Will write the value of 'c' in 'size' consecutive bytes starting from 'data'.

Parameters:

data	Address of data to set.
c	Value to set.
size	Number of bytes to set.

Location: `touchgfx/Utils.hpp`

muldiv

```
int32_t muldiv ( int32_t factor1 ,
                 int32_t factor2 ,
                 int32_t divisor
                 )
```

Multiply and divide without causing overflow.

Multiplying two large values and subsequently dividing the result with another large value might cause an overflow in the intermediate result. The function **muldiv()** will multiply factor1 and factor2 and divide the result by divisor without causing overflow (unless the final result would overflow). The remainder is used to round the result up or down.

Parameters:

factor1	The first factor.
factor2	The second factor.
divisor	The divisor.

Returns:

(factor1 * factor2) / divisor rounded.

See also:

muldiv(int32_t,int32_t,int32_t,int32_t&)

Location: `touchgfx/Utils.hpp`

muldiv

```
int32_t muldiv ( int32_t    factor1 ,
                 int32_t    factor2 ,
                 int32_t    divisor ,
                 int32_t & remainder
                 )
```

Multiply and divide without causing overflow.

Multiplying two large values and subsequently dividing the result with another large value might cause an overflow in the intermediate result. The function **muldiv()** will multiply factor1 and factor2 and divide the result by divisor without causing overflow (unless the final result would overflow). The remainder from the division is returned.

Parameters:

factor1	The first factor.
factor2	The second factor.
divisor	The divisor.
remainder	The remainder.

Returns:

(factor1 * factor2) / divisor.

NOTE

For large numbers close to the limit of int32_t, the calculation may not be correct.

See also:

muldivu

Location: `touchgfx/Utils.hpp`

muldivu

```
uint32_t muldivu ( uint32_t    factor1 ,
                   uint32_t    factor2 ,
                   uint32_t    divisor ,
                   uint32_t & remainder
                   )
```

Multiply and divide without causing overflow.

Multiplying two large values and subsequently dividing the result with another large value might cause an overflow in the intermediate result. The function **muldiv()** will multiply factor1 and factor2 and divide the result by divisor without causing overflow (unless the final result would overflow). The remainder from the division is returned.

Parameters:

factor1	The first factor.
factor2	The second factor.
divisor	The divisor.
remainder	The remainder.

Returns:

(factor1 * factor2) / divisor.

NOTE

For large numbers close to the limit of uint32_t, the calculation may not be correct.

See also:

muldiv

Location: `touchgfx/Utils.hpp`

operator*

```
Matrix4x4 operator* ( const Matrix4x4 & multiplicand ,
                      const Matrix4x4 & multiplier
                      )
```

Multiplication operator.

Parameters:

multiplicand	The first value to multiply.
multiplier	The second value to multiply.

Returns:

The result of the operation.

Location: `touchgfx/Math3D.hpp`

operator*

```
Point4 operator* ( const Matrix4x4 & multiplicand ,
                   const Point4 &    multiplier
                   )
```

Multiplication operator.

Parameters:

multiplicand	The first value to multiply.
multiplier	The second value to multiply.

Returns:

The result of the operation.

Location: `touchgfx/Math3D.hpp`

prepareTransition

```
static FORCE_INLINE_FUNCTION void prepareTransition ( Screen **     currentScreen ,
                                                      Presenter ** currentPresenter ,
                                                      Transition ** currentTrans
                                                      )
```

Prepare screen transition.

Private helper function for makeTransition. Do not use.

Parameters:

currentScreen	If non-null, the current screen.
currentPresenter	If non-null, the current presenter.
currentTrans	If non-null, the current transaction.

Location: `mvp/MVPApplication.hpp`

shouldTransferBlock

```
int shouldTransferBlock ( uint16_t bottom )
```

Check if a Frame Buffer Block ending at bottom may be sent.

Parameters:

bottom The bottom coordinate of the block to transfer.

Returns:

Non zero if possible.

Location: `touchgfx/hal/PartialFrameBufferManager.hpp`

simulator_enable_stdio

```
void simulator_enable_stdio (  )
```

Simulator enable stdio.

Location: `platform/hal/simulator/sdl2/HALSDL2.hpp`

simulator_printf

```
void simulator_printf ( const char * format ,
                        va_list      pArg
                        )
```

Simulator printf.

Parameters:

format Describes the format to use.
pArg The argument list.

Location: `platform/hal/simulator/sdl2/HALSDL2.hpp`

touchgfx_generic_init

```
HAL & touchgfx_generic_init ( DMA_Interface &  dma ,
                              LCD &            display ,
                              TouchController & tc ,
                              int16_t          width ,
                              int16_t          height ,
                              uint16_t *       bitmapCache ,
                              uint32_t         bitmapCacheSize ,
```

```
                                uint32_t            numberOfDynamicBitmaps =0
                                )
```

TouchGFX generic initialize.

Template Parameters:

HALType The class type of the **HAL** subclass used for this port.

Parameters:

dma	Reference to the DMA implementation object to use. Can be of type **NoDMA** to disable the use of DMA for rendering.
display	Reference to the **LCD** renderer implementation (subclass of **LCD**). Could be either **LCD16bpp** for RGB565 UIs, or **LCD1bpp** for monochrome UIs or **LCD24bpp** for 24bit displays using RGB888 UIs.
tc	Reference to the touch controller driver (or **NoTouchController** to disable touch input).
width	The *native* display width of the actual display, in pixels. This value is irrespective of whether the concrete UI should be portrait or landscape mode. It must match what the display itself is configured as.
height	The *native* display height of the actual display, in pixels. This value is irrespective of whether the concrete UI should be portrait or landscape mode. It must match what the display itself is configured as.
bitmapCache	Optional pointer to starting address of a memory region in which to place the bitmap cache. Usually in external RAM. Pass 0 if bitmap caching is not used.
bitmapCacheSize	Size of bitmap cache in bytes. Pass 0 if bitmap cache is not used.
numberOfDynamicBitmaps	(Optional) Number of dynamic bitmaps.

Returns:

A reference to the allocated (and initialized) **HAL** object.

Location: `common/TouchGFXInit.hpp`

touchgfx_init

```
void touchgfx_init (  )
```

Function to perform touchgfx initialization.

This function prototype is only provided as a convention.

Location: `touchgfx/hal/BoardConfiguration.hpp`

transmitActive

int transmitActive ( )

Check if a Frame Buffer Block is beeing transmitted.

Returns:

Non zero if possible.

Location: `touchgfx/hal/PartialFrameBufferManager.hpp`

transmitBlock

```
void transmitBlock ( const uint8_t * pixels ,
                     uint16_t        x ,
                     uint16_t        y ,
                     uint16_t        w ,
                     uint16_t        h
                     )
```

Transmit a Frame Buffer Block.

Parameters:

pixels	Pointer to the pixel data.
x	X coordinate of the block.
y	Y coordinate of the block.
w	Width of the block.
h	Height of the block.

Location: `touchgfx/hal/PartialFrameBufferManager.hpp`

Variables

BITMAP_ANIMATION_STORAGE

const BitmapId BITMAP_ANIMATION_STORAGE = 0xFFFEU

A virtual id representing animation storage.

Location: `touchgfx/Bitmap.hpp`

BITMAP_INVALID

const BitmapId BITMAP_INVALID = 0xFFFFU

Define the bitmapId of an invalid bitmap.

Location: `touchgfx/Bitmap.hpp`

CENTER

const Alignment CENTER = 1

Text is centered horizontally.

Location: `touchgfx/hal/Types.hpp`

LEFT

const Alignment LEFT = 0

Text is left aligned.

Location: `touchgfx/hal/Types.hpp`

PI

const float PI = 3.14159265358979323846f

PI.

Location: `touchgfx/hal/Types.hpp`

RenderingVariant_Alpha

const uint16_t RenderingVariant_Alpha = 2

The rendering variant alpha bit value.

Location: `touchgfx/hal/Types.hpp`

RenderingVariant_Bilinear

const uint16_t RenderingVariant_Bilinear = 1

The rendering variant bilinear bit value.

Location: `touchgfx/hal/Types.hpp`

RenderingVariant_FormatShift

const uint16_t RenderingVariant_FormatShift = 2

The rendering variant format shift.

Location: `touchgfx/hal/Types.hpp`

RenderingVariant_NearestNeighbor

const uint16_t RenderingVariant_NearestNeighbor = 0

The rendering variant nearest neighbor bit value.

Location: `touchgfx/hal/Types.hpp`

RenderingVariant_NoAlpha

const uint16_t RenderingVariant_NoAlpha = 0

The rendering variant no alpha bit value.

Location: `touchgfx/hal/Types.hpp`

RIGHT

const Alignment RIGHT = 2

Text is right aligned.

Location: `touchgfx/hal/Types.hpp`

TEXT_DIRECTION_LTR

const TextDirection TEXT_DIRECTION_LTR = 0

Text is written Left-To-Right, e.g. English.

Location: `touchgfx/hal/Types.hpp`

TEXT_DIRECTION_RTL

const TextDirection TEXT_DIRECTION_RTL = 1

Text is written Right-To-Left, e.g. Hebrew.

Location: `touchgfx/hal/Types.hpp`

TYPED_TEXT_INVALID

const TypedTextId TYPED_TEXT_INVALID = 0xFFFFU

The ID of an invalid text.

Location: `touchgfx/TypedText.hpp`

Version: 4.16

Presentations

In this section, you will find a wide range of TouchGFX presentations used for demonstrations, seminars, webinars and workshops. The presentations can be used directly or as inspiration for TouchGFX presentations of your own or simply as a source of information on a given topic.

It is the intention that you can piece together a set of presentations to match your needs for topics for a TouchGFX demonstration or seminar. All presentations come with references to relevant articles and to chapters and sections on this documentation site.

Presentations are available for download in slide format PDFs and in "speaker's note" versions where additional presentation information for the speaker can be found.

Not all presentations are available yet, but will be made available as soon as ready.

It is the intention, that in the future most of the presentations will be accompanied by videos that show a presentation of the topic using the slides.

TouchGFX Presentations

TouchGFX Introduction	
Description	A short, general and non-technical introduction to TouchGFX.
Target Audience	Beginner - All
Duration	15 minutes
Links	Not available yet.

TouchGFX Technical Introduction	
Description	A short, technical introduction to doing TouchGFX development. Covers the tools involved and describes the workflow.
Target Audience	Beginner - Any TouchGFX Developers
Duration	14 minutes
Links	**Presentation** **Speaker notes**

	Video available on Youtube

Embedded Graphics - Basic Concepts	
Description	A general introduction to key concepts of graphics on embedded devices. Good background knowledge for doing TouchGFX project development.
Target Audience	Beginner - TouchGFX Application Developers
Duration	55 minutes
Links	**Presentation** **Speaker notes** **Video available on Youtube**

Board Bring Up - Introduction	
Description	Covers key topics you need to address when doing board bring up for a TouchGFX project.
Target Audience	Beginner - Hardware Integrator
Duration	30 minutes
Links	Not available yet.

Abstraction Layer Development - Introduction	
Description	Introduces the TouchGFX Abstraction Layer development process. Describes the key responsibilities of the Abstraction Layer and how to use the TouchGFX Generator.
Target Audience	Beginner - TouchGFX AL Developers
Duration	60 minutes
Links	**Presentation**

Abstraction Layer Development - Advanced Topics	
Description	Dives into advanced topics on TouchGFX Abstration Layer development. Discusses how the TouchGFX AL (generated by TouchGFX Generator) and CubeMX MCU configuration integrate, how to add your own BSP and how TouchGFX is configured manually to support scenarios not configurable in CubeMX.

Target Audience	Medium - TouchGFX AL Developers
Duration	60 minutes
Links	Not available yet.

UI Development - Fundamentals	
Description	Introduction to key TouchGFX topics essential for UI development. Covers topics like the UI Application architecture, how to work with TouchGFX Designer during development, using the PC simulator and writing correct and effient user code.
Target Audience	Medium - TouchGFX Application Developers
Duration	60 minutes
Links	**Presentation** **Speaker notes** **Video available on Youtube**

UI Development - Using Texts and Fonts	
Description	Shows all the details that developers encounters when using texts and fonts in a TouchGFX application.
Target Audience	Medium - TouchGFX Application Developers
Duration	60 minutes
Links	Not available yet.

UI Development - Backend Communication	
Description	Explains the suggested way to interface with the non-GUI part of your system and shows examples hereof.
Target Audience	Medium - TouchGFX Application Developers
Duration	60 minutes
Links	Not available yet.

TouchGFX Hands-on Workshops

UI Development - Getting Started	
Description	Introdution to TouchGFX Application Development. Installing tools, creating and modifying your first application.
Target Audience	Beginner - TouchGFX Application Developers
Duration	60 - 90 minutes
Links	**Presentation** **Speakers note**

UI Development - Fundamentals - Hands on	
Description	Covers UI development fundamentals like reacting to user input, changing screen, persisting and restoring data, doing animations and tips and tricks for efficient development.
Target Audience	Medium - TouchGFX Application Developers
Duration	2 hours
Links	Not available yet.

Abstraction Layer Development - Introduction - Hands on	
Description	Introduction to TouchGFX AL development for an STM32H7B3I-DK board using TouchGFX Generator and STM32CubeIDE.
Target Audience	Beginner - TouchGFX AL Developers
Duration	90 minutes
Links	**Presentation**