MSP Graphics Library 3.21.00.00 version

USER'S GUIDE

Copyright

Texas Instruments
Post Office Box 655303
Dallas, TX 75265
http://www.ti.com/msp430

Revision Information

This is version 3.21.00.00 of this document, last updated on February 15, 2016.

Table of Contents

February 15, 2016

1   Using Template Driver files

1.1   Modifying the Template Driver File

This template driver is intended to be modified for creating new LCD drivers. It is setup so that only Template_DriverPixelDraw() and DPYCOLORTRANSLATE() and some LCD size configuration settings in the header file `Template_Driver.h` are **REQUIRED** to be written. These functions are marked with the string "TemplateDisplayFix" in the comments so that a search through `Template_Driver.c` and `Template_Driver.h` can quickly identify the necessary areas of change.

Template_DriverPixelDraw() is the base function to write to the LCD display. Functions like Write-Data(), WriteCommand(), and SetAddress() are suggested to be used to help implement the Template_DriverPixelDraw() function, but are not required. SetAddress() should be used by other pixel level functions to help optimize them.

This is not an optimized driver and will significantly impact performance. It is highly recommended to first get the prototypes working with the single pixel writes, and then go back and optimize the driver. Please see application note `SLAA548` for more information on how to fully optimize LCD driver files. In short, driver optimizations should take advantage of the auto-incrementing of the LCD controller. This should be utilized so that a loop of WriteData() can be used instead of a loop of Template_DriverPixelDraw(). The pixel draw loop contains both a SetAddress() + Write-Data() compared to WriteData() alone. This is a big time saver especially for the line draws and Template_DriverPixelDrawMultiple(). More optimization can be done by reducing function calls by writing macros, eliminating unnecessary instructions, and of course taking advantage of other features offered by the LCD controller. With so many pixels on an LCD screen each instruction can have a large impact on total drawing time.

2   Circle API

2.1   Introduction

The Circle API provides simple functions to draw a circle on the display. There are two different functions used to draw a circle; one which draws the outline, and the other which draws a filled-in circle. The clipping of the circle is performed within the routine; the display driver's circle fill routine is used to permore the actual circle fill.

The code for this API is contained in `grlib/circle.c`, with `grlib/circle.h` containing the API definitions for use by applications.

2.2   API Functions

Functions

- void Graphics_drawCircle (const Graphics_Context ∗context, int32_t x, int32_t y, int32_t radius)
- void Graphics_fillCircle (const Graphics_Context ∗context, int32_t x, int32_t y, int32_t radius)

2.2.1   Detailed Description

The Circle API is broken into two separate functions both of which write to the display.

The function which draws a circle is handled by

- GrCircleDraw()

The function which draws a filled-in circle is handled by

- GrCircleFill()

2.2.2   Function Documentation

2.2.2.1   Graphics_drawCircle

Draws a circle.

Prototype:
```
void
Graphics_drawCircle(const Graphics_Context *context,
```

```
                                     int32_t x,
                                     int32_t y,
                                     int32_t radius)
```

Parameters:

> ***context*** is a pointer to the drawing context to use.
>
> ***x*** is the X coordinate of the center of the circle.
>
> ***y*** is the Y coordinate of the center of the circle.
>
> ***radius*** is the radius of the circle.

Description:

> This function draws a circle, utilizing the Bresenham circle drawing algorithm. The extent of the circle is from *x - radius* to *x + radius* and *y - radius* to *y + radius*, inclusive.

Returns:

> None.

2.2.2.2   Graphics_fillCircle

Draws a filled circle.

Prototype:

```
    void
    Graphics_fillCircle(const Graphics_Context *context,
                        int32_t x,
                        int32_t y,
                        int32_t radius)
```

Parameters:

> ***context*** is a pointer to the drawing context to use.
>
> ***x*** is the X coordinate of the center of the circle.
>
> ***y*** is the Y coordinate of the center of the circle.
>
> ***radius*** is the radius of the circle.

Description:

> This function draws a filled circle, utilizing the Bresenham circle drawing algorithm. The extent of the circle is from *x - radius* to *x + radius* and *y - radius* to *y + radius*, inclusive.

Returns:

> None.

2.3   **Programming Example**

```
tContext sContext;

//
// Initialize the graphics context
//
GrContextInit(&sContext, &g_sharp400x240LCD);
GrContextForegroundSet(&sContext, ClrBlack);
GrContextBackgroundSet(&sContext, ClrWhite);
```

```
GrClearDisplay(&sContext);

GrCircleDraw(&sContext, 275, 100, 30);
GrCircleFill(&sContext, 50, 100, 30);

GrFlush(&sContext);
__no_operation();
```

3 Context API

3.1 Introduction

The Context API provides simple functions to initialize a drawing context, preparing it for use on the display. The display driver will be used for all subsequent graphics operations.

The code for this API is contained in `grlib/context.c`, with `grlib/context.h` containing the API definitions for use by applications.

3.2 API Functions

Functions

- void Graphics_clearDisplay (const Graphics_Context ∗context)
- void Graphics_drawPixel (const Graphics_Context ∗context, uint16_t x, uint16_t y)
- void Graphics_flushBuffer (const Graphics_Context ∗context)
- uint16_t Graphics_getDisplayHeight (Graphics_Context ∗context)
- uint16_t Graphics_getDisplayWidth (Graphics_Context ∗context)
- uint8_t Graphics_getFontBaseline (const Graphics_Font ∗font)
- uint8_t Graphics_getFontHeight (const Graphics_Font ∗font)
- uint8_t Graphics_getFontMaxWidth (const Graphics_Font ∗font)
- uint16_t Graphics_getHeightOfDisplay (const Graphics_Display ∗display)
- uint16_t Graphics_getWidthOfDisplay (const Graphics_Display ∗display)
- void Graphics_initContext (Graphics_Context ∗context, const Graphics_Display ∗display)
- void Graphics_setBackgroundColor (Graphics_Context ∗context, int32_t value)
- void Graphics_setBackgroundColorTranslated (Graphics_Context ∗context, int32_t value)
- void Graphics_setClipRegion (Graphics_Context ∗context, Graphics_Rectangle ∗rect)
- void Graphics_setFont (Graphics_Context ∗context, const Graphics_Font ∗font)
- void Graphics_setForegroundColor (Graphics_Context ∗context, int32_t value)
- void Graphics_setForegroundColorTranslated (Graphics_Context ∗context, int32_t value)

3.2.1 Detailed Description

The Context API is broken into two separate functions both of which initialize the context for the display, but differ in the way they set the clipping regions of the screen. The clipping region is not allowed to exceed the extents of the screen, but may be a portion of the screen. The supplied coordinates are inclusive for the clipping region. As a consequence, the clipping region must contain at least one row and one column.

The function which initializes the context and who's clipping region is set to the extent of the entire screen is handled by

- GrContextInit()

The function which initializes the context and also sets a clipping region is handled by

- GrContextClipRegionSet()

3.2.2 Function Documentation

3.2.2.1 Graphics_clearDisplay

Forces a clear screen. Contents of Display buffer unmodified

Prototype:
```
void
Graphics_clearDisplay(const Graphics_Context *context)
```

Parameters:
context is a pointer to the drawing context to use.

Description:
This function forces a clear screen.

Returns:
None.

3.2.2.2 Graphics_drawPixel

Draws a pixel.

Prototype:
```
void
Graphics_drawPixel(const Graphics_Context *context,
                   uint16_t x,
                   uint16_t y)
```

Parameters:
context is a pointer to the drawing context to use.
x is the X coordinate of the pixel.
y is the Y coordinate of the pixel.

Description:
This function draws a pixel if it resides within the clipping region.

Returns:
None.

3.2.2.3   Graphics_flushBuffer

Flushes any cached drawing operations.

Prototype:
```
void
Graphics_flushBuffer(const Graphics_Context *context)
```

Parameters:
   context is a pointer to the drawing context to use.

Description:
   This function flushes any cached drawing operations. For display drivers that draw into a local frame buffer before writing to the actual display, calling this function will cause the display to be updated to match the contents of the local frame buffer.

Returns:
   None.

3.2.2.4   Graphics_getDisplayHeight

Gets the height of the display being used by this drawing context.

Prototype:
```
uint16_t
Graphics_getDisplayHeight(Graphics_Context *context)
```

Parameters:
   context is a pointer to the drawing context to query.

Description:
   This function returns the height of the display that is being used by this drawing context.

Returns:
   Returns the height of the display in pixels.

3.2.2.5   Graphics_getDisplayWidth

Gets the width of the display being used by this drawing context.

Prototype:
```
uint16_t
Graphics_getDisplayWidth(Graphics_Context *context)
```

Parameters:
   context is a pointer to the drawing context to query.

Description:
   This function returns the width of the display that is being used by this drawing context.

Returns:
   Returns the width of the display in pixels.

3.2.2.6   Graphics_getFontBaseline

Gets the baseline of a font.

Prototype:
```
uint8_t
Graphics_getFontBaseline(const Graphics_Font *font)
```

Parameters:
>   ***font*** is a pointer to the font to query.

Description:
>   This function determines the baseline position of a font. The baseline is the offset between the top of the font and the bottom of the capital letters. The only font data that exists below the baseline are the descenders on some lower-case letters (such as "y").

Returns:
>   Returns the baseline of the font, in pixels.

3.2.2.7   Graphics_getFontHeight

Gets the height of a font.

Prototype:
```
uint8_t
Graphics_getFontHeight(const Graphics_Font *font)
```

Parameters:
>   ***font*** is a pointer to the font to query.

Description:
>   This function determines the height of a font. The height is the offset between the top of the font and the bottom of the font, including any ascenders and descenders.

Returns:
>   Returns the height of the font, in pixels.

3.2.2.8   Graphics_getFontMaxWidth

Gets the maximum width of a font.

Prototype:
```
uint8_t
Graphics_getFontMaxWidth(const Graphics_Font *font)
```

Parameters:
>   ***font*** is a pointer to the font to query.

Description:
>   This function determines the maximum width of a font. The maximum width is the width of the widest individual character in the font.

Returns:

Returns the maximum width of the font, in pixels.

3.2.2.9   Graphics_getHeightOfDisplay

Gets the height of the display.

Prototype:
```
uint16_t
Graphics_getHeightOfDisplay(const Graphics_Display *display)
```

Parameters:

display is a pointer to the display driver structure for the display to query.

Description:

This function determines the height of the display.

Returns:

Returns the height of the display in pixels.

3.2.2.10   Graphics_getWidthOfDisplay

Gets the width of the display.

Prototype:
```
uint16_t
Graphics_getWidthOfDisplay(const Graphics_Display *display)
```

Parameters:

display is a pointer to the display driver structure for the display to query.

Description:

This function determines the width of the display.

Returns:

Returns the width of the display in pixels.

3.2.2.11   Graphics_initContext

Initializes a drawing context.

Prototype:
```
void
Graphics_initContext(Graphics_Context *context,
                     const Graphics_Display *display)
```

Parameters:

context is a pointer to the drawing context to initialize.

display is a pointer to the Graphics_Display Info structure that describes the display driver to use.

Description:
This function initializes a drawing context, preparing it for use. The provided display driver will be used for all subsequent graphics operations, and the default clipping region will be set to the extent of the screen.

Returns:
None.

3.2.2.12   Graphics_setBackgroundColor

Sets the background color to be used.

Prototype:
```
void
Graphics_setBackgroundColor(Graphics_Context *context,
                            int32_t value)
```

Parameters:
context is a pointer to the drawing context to modify.
value is the 24-bit RGB color to be used.

Description:
This function sets the background color to be used for drawing operations in the specified drawing context.

Returns:
None.

3.2.2.13   Graphics_setBackgroundColorTranslated

Sets the background color to be used.

Prototype:
```
void
Graphics_setBackgroundColorTranslated(Graphics_Context *context,
                                      int32_t value)
```

Parameters:
context is a pointer to the drawing context to modify.
value is the display driver-specific color to be used.

Description:
This function sets the background color to be used for drawing operations in the specified drawing context, using a color that has been previously translated to a driver-specific color (for example, via Graphics_translateColorDisplay()).

Returns:
None.

3.2.2.14 Graphics_setClipRegion

Sets the extents of the clipping region.

Prototype:
```
void
Graphics_setClipRegion(Graphics_Context *context,
                           Graphics_Rectangle *rect)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>
>   ***rect*** is a pointer to the structure containing the extents of the clipping region.

Description:
>   This function sets the extents of the clipping region. The clipping region is not allowed to exceed the extents of the screen, but may be a portion of the screen.
>
>   The supplied coordinate are inclusive; *xMin* of 1 and *xMax* of 1 will define a clipping region that will display only the pixels in the X = 1 column. A consequence of this is that the clipping region must contain at least one row and one column.

Returns:
>   None.

3.2.2.15 Graphics_setFont

Sets the font to be used.

Prototype:
```
void
Graphics_setFont(Graphics_Context *context,
                    const Graphics_Font *font)
```

Parameters:
>   ***context*** is a pointer to the drawing context to modify.
>
>   ***font*** is a pointer to the font to be used.

Description:
>   This function sets the font to be used for string drawing operations in the specified drawing context. If a tFontEx type font is to be used, cast its pointer to a font pointer before passing it as the font parameter.

Returns:
>   None.

3.2.2.16 Graphics_setForegroundColor

Sets the foreground color to be used.

Prototype:
```
void
Graphics_setForegroundColor(Graphics_Context *context,
                               int32_t value)
```

Parameters:
> ***context*** is a pointer to the drawing context to modify.
> ***value*** is the 24-bit RGB color to be used.

Description:
> This function sets the color to be used for drawing operations in the specified drawing context.

Returns:
> None.

3.2.2.17 Graphics_setForegroundColorTranslated

Sets the foreground color to be used.

Prototype:
```
void
Graphics_setForegroundColorTranslated(Graphics_Context *context,
                                         int32_t value)
```

Parameters:
> ***context*** is a pointer to the drawing context to modify.
> ***value*** is the display driver-specific color to be used.

Description:
> This function sets the foreground color to be used for drawing operations in the specified drawing context, using a color that has been previously translated to a driver-specific color (for example, via Graphics_translateColorDisplay()).

Returns:
> None.

3.3   Programming Example

```
tContext sContext;

  //
  // Initialize the graphics context
  //
  GrContextInit(&sContext, &g_sharp400x240LCD);
  GrContextForegroundSet(&sContext, ClrBlack);
  GrContextBackgroundSet(&sContext, ClrWhite);

  GrClearDisplay(&sContext);

  GrContextFontSet(&sContext, &g_sFontCm26);
  GrStringDraw(&sContext, "Welcome to ", -1, 20, 8, 0);

  GrContextFontSet(&sContext, &g_sFontCm30);
```

```
GrStringDraw(&sContext, "Dallas TX", -1, 20, 180, 0);

GrFlush(&sContext);
__no_operation();
```

4   Image API

4.1   Introduction

The Image API provides simple functions to draw images on the screen. There are two different functions used to draw a image; one which converts the palette of a bitmap image and the other which renders the bitmap image onto the screen.

The code for this API is contained in `grlib/image.c`, with `grlib/image.h` containing the API definitions for use by applications.

4.2   API Functions

Functions

- void Graphics_drawImage (const Graphics_Context ∗context, const Graphics_Image ∗bitmap, int16_t x, int16_t y)
- uint16_t Graphics_getImageColors (const Graphics_Image ∗image)
- uint16_t Graphics_getImageHeight (const Graphics_Image ∗image)
- uint16_t Graphics_getImageWidth (const Graphics_Image ∗image)
- uint32_t Graphics_getOffscreen1BppImageSize (uint16_t width, uint16_t height)
- uint32_t Graphics_getOffscreen4BppImageSize (uint16_t width, uint16_t height)
- uint32_t Graphics_getOffScreen8BPPSize (uint16_t width, uint16_t height)

4.2.1   Detailed Description

The Image API is broken into two separate functions, one to converte the palette and the other to render to the display. Calling the GrImageDraw() function also invokes GrPaletteConversion() as well so the user only needs to be concerned with the GrImageDraw() function.

The image may be either 1-, 4-, or 8-bits per pixel by using a palette supplied in the image data. The image palette is in 24-bit RGB form and by calling GrPaletteConversion(), the palette can then be sent to the LCD usign DpyColorTranslate function. The converted palette is contained in a global buffer while the original image remains the same. The palette can be uncompressed data or it can be compressed using several different compresion types. Compression options are either 4- or 8-bit run length encoding, or a custom run lenth endocing variation written for complex 8-bit per pixel images.

The function which converts the palette of the bitmap is handled by

- GrPaletteConversion()

The function which draws a bitmap image is handled by

- GrImageDraw()

4.2.2 Function Documentation

4.2.2.1 Graphics_drawImage

Draws a bitmap image.

Prototype:
```
void
Graphics_drawImage(const Graphics_Context *context,
                   const Graphics_Image *bitmap,
                   int16_t x,
                   int16_t y)
```

Parameters:
> *context* is a pointer to the drawing context to use.
> *bitmap* is a pointer to the image to draw.
> *x* is the X coordinate of the upper left corner of the image.
> *y* is the Y coordinate of the upper left corner of the image.

Description:
> This function draws a bitmap image. The image may be 1 bit per pixel, 4 bits per pixel or 8 bits per pixel (using a palette supplied in the image data). It can be uncompressed data, or it can be compressed using several different compression types. Compression options are 4-bit run length encoding, 8-bit run length encoding, and a custom run length encoding variation written for complex 8-bit per pixel images.

Returns:
> None.

4.2.2.2 Graphics_getImageColors

Gets the number of colors in an image.

Prototype:
```
uint16_t
Graphics_getImageColors(const Graphics_Image *image)
```

Parameters:
> *image* is a tImage struct

Description:
> This function determines the number of colors in the palette of an image. This is only valid for 4bpp and 8bpp images; 1bpp images do not contain a palette.

Returns:
> Returns the number of colors in the image.

4.2.2.3   Graphics_getImageHeight

Gets the height of an image.

Prototype:
```
uint16_t
Graphics_getImageHeight(const Graphics_Image *image)
```

Parameters:
image is a tImage struct

Description:
This function determines the height of an image in pixels.

Returns:
Returns the height of the image in pixels.

4.2.2.4   Graphics_getImageWidth

Gets the width of an image.

Prototype:
```
uint16_t
Graphics_getImageWidth(const Graphics_Image *image)
```

Parameters:
image is a tImage struct

Description:
This function determines the width of an image in pixels.

Returns:
Returns the width of the image in pixels.

4.2.2.5   Graphics_getOffscreen1BppImageSize

Determines the size of the buffer for a 1 BPP off-screen image.

Prototype:
```
uint32_t
Graphics_getOffscreen1BppImageSize(uint16_t width,
                                   uint16_t height)
```

Parameters:
width is the width of the image in pixels.
height is the height of the image in pixels.

Description:
This function determines the size of the memory buffer required to hold a 1 BPP off-screen image of the specified geometry.

Returns:
Returns the number of bytes required by the image.

4.2.2.6   Graphics_getOffscreen4BppImageSize

Determines the size of the buffer for a 4 BPP off-screen image.

Prototype:
```
uint32_t
Graphics_getOffscreen4BppImageSize(uint16_t width,
                                   uint16_t height)
```

Parameters:
width is the width of the image in pixels.
height is the height of the image in pixels.

Description:
This function determines the size of the memory buffer required to hold a 4 BPP off-screen image of the specified geometry.

Returns:
Returns the number of bytes required by the image.

4.2.2.7   Graphics_getOffScreen8BPPSize

Determines the size of the buffer for an 8 BPP off-screen image.

Prototype:
```
uint32_t
Graphics_getOffScreen8BPPSize(uint16_t width,
                              uint16_t height)
```

Parameters:
width is the width of the image in pixels.
height is the height of the image in pixels.

Description:
This function determines the size of the memory buffer required to hold an 8 BPP off-screen image of the specified geometry.

Returns:
Returns the number of bytes required by the image.

4.3   Programming Example

```
tContext sContext;

//
// Initialize the graphics context
```

```
//
GrContextInit(&sContext, &g_sharp400x240LCD);
GrContextForegroundSet(&sContext, ClrBlack);
GrContextBackgroundSet(&sContext, ClrWhite);

GrClearDisplay(&sContext);

GrImageDraw(&sContext, &infoHugePig, 200, 70);

GrFlush(&sContext);
__no_operation();
```

5 Line API

5.1 Introduction

The Line API provides simple functions to draw lines on the display. There are five different functions used to draw a line; two optimized functions for horizontal and vertical drawing, one generic line drawing function, two functions for clipping. The user needs only to be concerned with the generic line drawing function, GrLineDraw(), as it incorporates the use of all the other functions automatically.

The code for this API is contained in `grlib/line.c`, with `grlib/line.h` containing the API definitions for use by applications.

5.2 API Functions

Functions

- void Graphics_drawLine (const Graphics_Context ∗context, int32_t x1, int32_t y1, int32_t x2, int32_t y2)
- void Graphics_drawLineH (const Graphics_Context ∗context, int32_t x1, int32_t x2, int32_t y)
- void Graphics_drawLineV (const Graphics_Context ∗context, int32_t x, int32_t y1, int32_t y2)

5.2.1 Detailed Description

The Line API is broken into two separate functions; one for drawing and the other for clipping (internal functions).

The functions that draw a line are handled by

- GrLineDrawH()
- GrLineDrawL()
- GrLineDraw()

The user needs only to be concerned with the generic line drawing function, GrLineDraw(), as it incorporates the use of all the other functions automatically.

5.2.2 Function Documentation

5.2.2.1 Graphics_drawLine

Draws a line.

Prototype:
```
void
Graphics_drawLine(const Graphics_Context *context,
                  int32_t x1,
                  int32_t y1,
                  int32_t x2,
                  int32_t y2)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
>
> ***x1*** is the X coordinate of the start of the line.
>
> ***y1*** is the Y coordinate of the start of the line.
>
> ***x2*** is the X coordinate of the end of the line.
>
> ***y2*** is the Y coordinate of the end of the line.

Description:
> This function draws a line, utilizing Graphics_drawLineH() and Graphics_drawLineV() to draw the line as efficiently as possible. The line is clipped to the clippping rectangle using the Cohen-Sutherland clipping algorithm, and then scan converted using Bresenham's line drawing algorithm.

Returns:
> None.

5.2.2.2   Graphics_drawLineH

Draws a horizontal line.

Prototype:
```
void
Graphics_drawLineH(const Graphics_Context *context,
                   int32_t x1,
                   int32_t x2,
                   int32_t y)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
>
> ***x1*** is the X coordinate of one end of the line.
>
> ***x2*** is the X coordinate of the other end of the line.
>
> ***y*** is the Y coordinate of the line.

Description:
> This function draws a horizontal line, taking advantage of the fact that the line is horizontal to draw it more efficiently. The clipping of the horizontal line to the clipping rectangle is performed within this routine; the display driver's horizontal line routine is used to perform the actual line drawing.

Returns:
> None.

5.2.2.3 Graphics_drawLineV

Draws a vertical line.

Prototype:
```
void
Graphics_drawLineV(const Graphics_Context *context,
                   int32_t x,
                   int32_t y1,
                   int32_t y2)
```

Parameters:
>*context* is a pointer to the drawing context to use.
>
>*x* is the X coordinate of the line.
>
>*y1* is the Y coordinate of one end of the line.
>
>*y2* is the Y coordinate of the other end of the line.

Description:
>This function draws a vertical line, taking advantage of the fact that the line is vertical to draw it more efficiently. The clipping of the vertical line to the clipping rectangle is performed within this routine; the display driver's vertical line routine is used to perform the actual line drawing.

Returns:
>None.

5.3   Programming Example

```
tContext sContext;

  //
  // Initialize the graphics context
  //
  GrContextInit(&sContext, &g_sharp400x240LCD);
  GrContextForegroundSet(&sContext, ClrBlack);
  GrContextBackgroundSet(&sContext, ClrWhite);

  GrClearDisplay(&sContext);

  GrLineDraw(&sContext, 130, 30, 275, 200 );
  GrLineDrawH(&sContext, 20, 180, 220);
  GrLineDrawV(&sContext, 30, 50, 160);

  GrFlush(&sContext);
  __no_operation();
```

6 Rectangle API

6.1 Introduction

The Rectangle API provides simple functions to draw a rectangle on the display. There are two different functions used to draw a rectangle; one which draws the outline, and the other which draws a filled-in rectangle. The clipping of the rectangle is performed within the routine; the display driver's rectangle fill routine is used to permore the actual rectangle fill.

The code for this API is contained in `grlib/rectangle.c`, with `grlib/rectangle.h` containing the API definitions for use by applications.

6.2 API Functions

Functions

- void Graphics_drawRectangle (const Graphics_Context ∗context, const Graphics_Rectangle ∗rect)
- void Graphics_fillRectangle (const Graphics_Context ∗context, const Graphics_Rectangle ∗rect)
- int32_t Graphics_getRectangleIntersection (Graphics_Rectangle ∗rect1, Graphics_Rectangle ∗rect2, Graphics_Rectangle ∗intersect)
- bool Graphics_isPointWithinRectangle (const Graphics_Rectangle ∗rect, uint16_t x, uint16_t y)
- int32_t Graphics_isRectangleOverlap (Graphics_Rectangle ∗rect1, Graphics_Rectangle ∗rect2)

6.2.1 Detailed Description

The Rectangle API is broken into two groups; one that draws to the screen and the other which perform checks(internal functions).

The functions which draw rectangles are handled by

- GrRectDraw()
- GrRectFill()

6.2.2   Function Documentation

6.2.2.1   Graphics_drawRectangle

Draws a rectangle.

Prototype:
```
void
Graphics_drawRectangle(const Graphics_Context *context,
                       const Graphics_Rectangle *rect)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
>
> ***rect*** is a pointer to the structure containing the extents of the rectangle.

Description:
> This function draws a rectangle. The rectangle will extend from *xMin* to *xMax* and *yMin* to *yMax*, inclusive.

Returns:
> None.

6.2.2.2   Graphics_fillRectangle

Draws a filled rectangle.

Prototype:
```
void
Graphics_fillRectangle(const Graphics_Context *context,
                       const Graphics_Rectangle *rect)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
>
> ***rect*** is a pointer to the structure containing the extents of the rectangle.

Description:
> This function draws a filled rectangle. The rectangle will extend from *xMin* to *xMax* and *yMin* to *yMax*, inclusive. The clipping of the rectangle to the clipping rectangle is performed within this routine; the display driver's rectangle fill routine is used to perform the actual rectangle fill.

Returns:
> None.

6.2.2.3   Graphics_getRectangleIntersection

Determines the intersection of two rectangles.

Prototype:
```
int32_t
Graphics_getRectangleIntersection(Graphics_Rectangle *rect1,
```

```
                              Graphics_Rectangle *rect2,
                              Graphics_Rectangle *intersect)
```

Parameters:
>   ***rect1*** is a pointer to the first rectangle.
>
>   ***rect2*** is a pointer to the second rectangle.
>
>   ***intersect*** is a pointer to a rectangle which will be written with the intersection of *rect1* and *rect2*.

Description:
>   This function determines if two rectangles overlap and, if they do, calculates the rectangle representing their intersection. If the rectangles do not overlap, 0 is returned and *intersect* is not written.

Returns:
>   Returns 1 if there is an overlap or 0 if not.

6.2.2.4   Graphics_isPointWithinRectangle

Determines if a point lies within a given rectangle.

Prototype:
```
bool
Graphics_isPointWithinRectangle(const Graphics_Rectangle *rect,
                                uint16_t x,
                                uint16_t y)
```

Parameters:
>   ***rect*** is a pointer to the rectangle which the point is to be checked against.
>
>   ***x*** is the X coordinate of the point to be checked.
>
>   ***y*** is the Y coordinate of the point to be checked.

Description:
>   This function determines whether point (x, y) lies within the rectangle described by *rect*.

Returns:
>   Returns 1 if the point is within the rectangle or 0 otherwise.

6.2.2.5   Graphics_isRectangleOverlap

Determines if two rectangles overlap.

Prototype:
```
int32_t
Graphics_isRectangleOverlap(Graphics_Rectangle *rect1,
                            Graphics_Rectangle *rect2)
```

Parameters:
>   ***rect1*** is a pointer to the first rectangle.
>
>   ***rect2*** is a pointer to the second rectangle.

Description:

This function determines whether two rectangles overlap. It assumes that rectangles *rect1* and *rect2* are valid with *xMin* < *xMax* and *yMin* < *yMax*.

Returns:

Returns 1 if there is an overlap or 0 if not.

6.3   **Programming Example**

```
tContext sContext;
tRectangle myRectangle1 = { 60, 60, 120, 120};
tRectangle myRectangle2 = { 160, 60, 220, 120};

//
// Initialize the graphics context
//
GrContextInit(&sContext, &g_sharp400x240LCD);
GrContextForegroundSet(&sContext, ClrBlack);
GrContextBackgroundSet(&sContext, ClrWhite);

GrClearDisplay(&sContext);

GrRectDraw(&sContext, &myRectangle1 );
GrRectFill(&sContext, &myRectangle2);

GrFlush(&sContext);
__no_operation();
```

7   String API

7.1   Introduction

The String API provides simple functions to draw strings on the screen. There are several different functions used to draw a string; one which counts the number of leading zeroes, one for obtaining the display width of the string, one for drawing the string to the display, one for setting the location of the current string table, one to set the current language, and the last one for grabbing the string from the current string table. The user should not directly call NumLeadingZeroes() as it is used internally.

The code for this API is contained in `grlib/string.c`, with `grlib/string.h` containing the API definitions for use by applications.

7.2   API Functions

Functions

- void Graphics_drawString (const Graphics_Context ∗context, int8_t ∗string, int32_t length, int32_t x, int32_t y, bool opaque)
- void Graphics_drawStringCentered (const Graphics_Context ∗context, int8_t ∗string, int32_t length, int32_t x, int32_t y, bool opaque)
- uint8_t Graphics_getStringBaseline (const Graphics_Context ∗context)
- uint8_t Graphics_getStringHeight (const Graphics_Context ∗context)
- uint8_t Graphics_getStringMaxWidth (const Graphics_Context ∗context)
- int32_t Graphics_getStringWidth (const Graphics_Context ∗context, const int8_t ∗string, int32_t length)

7.2.1   Detailed Description

The String API available are classified as below.

The functions which calculate and set up parameters are handled by

- GrStringWidthGet()

The function which draws a string to the display is handled by

- GrStringDraw()

7.2.2 Function Documentation

7.2.2.1 Graphics_drawString

Draws a string.

Prototype:
```
void
Graphics_drawString(const Graphics_Context *context,
                    int8_t *string,
                    int32_t length,
                    int32_t x,
                    int32_t y,
                    bool opaque)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>
>   ***string*** is a pointer to the string to be drawn.
>
>   ***length*** is the number of characters from the string that should be drawn on the screen.
>
>   ***x*** is the X coordinate of the upper left corner of the string position on the screen.
>
>   ***y*** is the Y coordinate of the upper left corner of the string position on the screen.
>
>   ***opaque*** is true if the background of each character should be drawn and false if it should not (leaving the background as is).

Description:
>   This function draws a string of test on the screen. The *length* parameter allows a portion of the string to be examined without having to insert a NULL character at the stopping point (which would not be possible if the string was located in flash); specifying a length of -1 will cause the entire string to be rendered (subject to clipping).

Returns:
>   None.

7.2.2.2 Graphics_drawStringCentered

Draws a centered string.

Prototype:
```
void
Graphics_drawStringCentered(const Graphics_Context *context,
                            int8_t *string,
                            int32_t length,
                            int32_t x,
                            int32_t y,
                            bool opaque)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>
>   ***string*** is a pointer to the string to be drawn.
>
>   ***length*** is the number of characters from the string that should be drawn on the screen.

x is the X coordinate of the center of the string position on the screen.

y is the Y coordinate of the center of the string position on the screen.

opaque is **true** if the background of each character should be drawn and **false** if it should not (leaving the background as is).

Description:

This function draws a string of test on the screen centered upon the provided position. The *lLength* parameter allows a portion of the string to be examined without having to insert a NULL character at the stopping point (which would not be possible if the string was located in flash); specifying a length of -1 will cause the entire string to be rendered (subject to clipping).

Returns:

None.

7.2.2.3 Graphics_getStringBaseline

Gets the baseline of a string.

Prototype:

```
uint8_t
Graphics_getStringBaseline(const Graphics_Context *context)
```

Parameters:

context is a pointer to the drawing context to query.

Description:

This function determines the baseline position of a string. The baseline is the offset between the top of the string and the bottom of the capital letters. The only string data that exists below the baseline are the descenders on some lower-case letters (such as "y").

Returns:

Returns the baseline of the string, in pixels.

7.2.2.4 Graphics_getStringHeight

Gets the height of a string.

Prototype:

```
uint8_t
Graphics_getStringHeight(const Graphics_Context *context)
```

Parameters:

context is a pointer to the drawing context to query.

Description:

This function determines the height of a string. The height is the offset between the top of the string and the bottom of the string, including any ascenders and descenders. Note that this will not account for the case where the string in question does not have any characters that use descenders but the font in the drawing context does contain characters with descenders.

Returns:

Returns the height of the string, in pixels.

7.2.2.5   Graphics_getStringMaxWidth

Gets the maximum width of a character in a string.

Prototype:
```
uint8_t
Graphics_getStringMaxWidth(const Graphics_Context *context)
```

Parameters:
>   *context* is a pointer to the drawing context to query.

Description:
>   This function determines the maximum width of a character in a string. The maximum width is the width of the widest individual character in the font used to render the string, which may be wider than the widest character that is used to render a particular string.

Returns:
>   Returns the maximum width of a character in a string, in pixels.

7.2.2.6   Graphics_getStringWidth

Determines the width of a string.

Prototype:
```
int32_t
Graphics_getStringWidth(const Graphics_Context *context,
                        const int8_t *string,
                        int32_t length)
```

Parameters:
>   *context* is a pointer to the drawing context to use.
>   *string* is the string in question.
>   *length* is the length of the string.

Description:
>   This function determines the width of a string (or portion of the string) when drawn with a particular font. The *length* parameter allows a portion of the string to be examined without having to insert a NULL character at the stopping point (would not be possible if the string was located in flash); specifying a length of -1 will cause the width of the entire string to be computed.

Returns:
>   Returns the width of the string in pixels.

7.3   Programming Example

```
tContext sContext;

//
// Initialize the graphics context
```

```
//
GrContextInit(&sContext, &g_sharp400x240LCD);
GrContextForegroundSet(&sContext, ClrBlack);
GrContextBackgroundSet(&sContext, ClrWhite);

GrClearDisplay(&sContext);

GrContextFontSet(&sContext, &g_sFontCm26);
GrStringDraw(&sContext, "Welcome to ", -1, 20, 8, 0);

GrContextFontSet(&sContext, &g_sFontCm30);
GrStringDraw(&sContext, "Dallas TX", -1, 20, 180, 0);

GrFlush(&sContext);
__no_operation();
```

8 Button API

8.1 Introduction

The Button API provides simple functions to draw a button on the display.

8.2 API Functions

Functions

- void Graphics_drawButton (const Graphics_Context ∗context, const Graphics_Button ∗button)
- void Graphics_drawReleasedButton (const Graphics_Context ∗context, const Graphics_Button ∗button)
- void Graphics_drawSelectedButton (const Graphics_Context ∗context, const Graphics_Button ∗button)
- bool Graphics_isButtonSelected (const Graphics_Button ∗button, uint16_t x, uint16_t y)

8.2.1 Detailed Description

The Button API is broken into four separate functions both of which write to the display.

The function which draws a button is handled by

- Graphics_drawButton()

The function which draws a selected button

- Graphics_drawSelectedButton()

The function which draws a released button

- Graphics_drawReleasedButton()

The function which determines if button has been pressed

- Graphics_isButtonSelected()

8.2.2 Function Documentation

8.2.2.1 Graphics_drawButton

Draws a button.

Prototype:
```
void
Graphics_drawButton(const Graphics_Context *context,
                    const Graphics_Button *button)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
> ***button*** is a pointer to the structure containing the extents of the button.

Description:
> This function draws a button. The button will contain a text string and will be created based on the parameters passed in the button struct.

Returns:
> None.

8.2.2.2   Graphics_drawReleasedButton

Draws a released Button.

Prototype:
```
void
Graphics_drawReleasedButton(const Graphics_Context *context,
                            const Graphics_Button *button)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
> ***button*** is a pointer to the structure containing the extents of the button.

Description:
> This function draws a button using the released parameters.

Returns:
> None.

8.2.2.3   Graphics_drawSelectedButton

Draws a selected Button.

Prototype:
```
void
Graphics_drawSelectedButton(const Graphics_Context *context,
                            const Graphics_Button *button)
```

Parameters:
> ***context*** is a pointer to the drawing context to use.
> ***button*** is a pointer to the structure containing the extents of the button.

Description:
> This function draws a button using the selected parameters.

Returns:
> None.

8.2.2.4   Graphics_isButtonSelected

Determines if x and y coordinates are contained in button .

Prototype:
```
bool
Graphics_isButtonSelected(const Graphics_Button *button,
                          uint16_t x,
                          uint16_t y)
```

Parameters:
> ***button*** is a pointer to the structure containing the extents of the button.
>
> ***x*** x-coordinate to be determined if is inside button
>
> ***y*** y-coordinate to be determined if is inside button

Description:
> This function determines if x and y coordinates are contains inside button

Returns:
> true if x and y coordinates are inside button, false if not

8.3   Programming Example

```
Graphics_Button yesButton;

yesButton.xMin = 80;
yesButton.xMax = 150;
yesButton.yMin = 80;
yesButton.yMax = 120;
yesButton.borderWidth = 1;
yesButton.selected = false;
yesButton.fillColor = GRAPHICS_COLOR_RED;
yesButton.borderColor = GRAPHICS_COLOR_RED;
yesButton.selectedColor = GRAPHICS_COLOR_BLACK;
yesButton.textColor = GRAPHICS_COLOR_BLACK;
yesButton.selectedTextColor = GRAPHICS_COLOR_RED;
yesButton.textXPos = 100;
yesButton.textYPos = 90;
yesButton.text = "YES";
yesButton.font = &g_sFontCm18;

Graphics_drawButton(&g_sContext, &yesButton);
```

9 ImageButton API

9.1 Introduction

The ImageButton API provides simple functions to draw a imageButton on the display.

9.2 API Functions

Functions

■ void Graphics_drawImageButton (const Graphics_Context ∗context, const Graphics_ImageButton ∗imageButton)
■ void Graphics_drawReleasedImageButton (const Graphics_Context ∗context, const Graphics_ImageButton ∗imageButton)
■ void Graphics_drawSelectedImageButton (const Graphics_Context ∗context, const Graphics_ImageButton ∗imageButton)
■ bool Graphics_isImageButtonSelected (const Graphics_ImageButton ∗imageButton, uint16_t x, uint16_t y)

9.2.1 Detailed Description

The ImageButton API is broken into four separate functions both of which write to the display.

The function which draws a imageButton is handled by

■ Graphics_drawImageButton()

The function which draws a selected imageButton

■ Graphics_drawSelectedImageButton()

The function which draws a released imageButton

■ Graphics_drawReleasedImageButton()

The function which determines if imageButton has been pressed

■ Graphics_isImageButtonSelected()

9.2.2   Function Documentation

9.2.2.1   Graphics_drawImageButton

Draws a ImageButton .

Prototype:
```
    void
    Graphics_drawImageButton(const Graphics_Context *context,
                             const Graphics_ImageButton *imageButton)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>   ***imageButton*** is a pointer to the structure containing the extents of the ImageButton .

Description:
>   This function draws a ImageButton . The ImageButton will contain the image passed in the ImageButton struct.

Returns:
>   None.

9.2.2.2   Graphics_drawReleasedImageButton

Draws a released ImageButton .

Prototype:
```
    void
    Graphics_drawReleasedImageButton(const Graphics_Context *context,
                                     const Graphics_ImageButton
    *imageButton)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>   ***imageButton*** is a pointer to the structure containing the extents of the ImageButton .

Description:
>   This function draws a ImageButton using the released parameters.

Returns:
>   None.

9.2.2.3   Graphics_drawSelectedImageButton

Draws a selected ImageButton .

Prototype:
```
    void
    Graphics_drawSelectedImageButton(const Graphics_Context *context,
                                     const Graphics_ImageButton
    *imageButton)
```

Parameters:

context is a pointer to the drawing context to use.

imageButton is a pointer to the structure containing the extents of the ImageButton .

Description:

This function draws a ImageButton using the selected parameters.

Returns:

None.

9.2.2.4   Graphics_isImageButtonSelected

Determines if x and y coordinates are contained in ImageButton .

Prototype:
```
bool
Graphics_isImageButtonSelected(const Graphics_ImageButton
*imageButton,
                                uint16_t x,
                                uint16_t y)
```

Parameters:

imageButton is a pointer to the structure containing the extents of the ImageButton .

x x-coordinate to be determined if is inside ImageButton

y y-coordinate to be determined if is inside ImageButton

Description:

This function determines if x and y coordinates are contains inside ImageButton

Returns:

true if x and y coordinates are inside ImageButton, false if not

9.3   **Programming Example**

```
Graphics_ImageButton primitiveButton;

primitiveButton.xPosition=20;
primitiveButton.yPosition=50;
primitiveButton.borderWidth=5;
primitiveButton.selected=false;
primitiveButton.imageWidth=Primitives_Button4BPP_UNCOMP.xSize;
primitiveButton.imageHeight=Primitives_Button4BPP_UNCOMP.ySize;
primitiveButton.borderColor=GRAPHICS_COLOR_WHITE;
primitiveButton.selectedColor=GRAPHICS_COLOR_RED;
primitiveButton.image=&Primitives_Button4BPP_UNCOMP;

Graphics_drawImageButton(&g_sContext, &primitiveButton);
```

10 RadioButton API

10.1 Introduction

The RadioButton API provides simple functions to draw a radioButton on the display.

10.2 API Functions

Functions

- void Graphics_drawRadioButton (const Graphics_Context *context, const Graphics_RadioButton *radioButton)
- void Graphics_drawReleasedRadioButton (const Graphics_Context *context, const Graphics_RadioButton *radioButton)
- void Graphics_drawSelectedRadioButton (const Graphics_Context *context, const Graphics_RadioButton *radioButton)
- bool Graphics_isRadioButtonSelected (const Graphics_RadioButton *radioButton, uint16_t x, uint16_t y)

10.2.1 Detailed Description

The RadioButton API is broken into four separate functions both of which write to the display.

The function which draws a radioButton is handled by

- Graphics_drawRadioButton()

The function which draws a selected radioButton

- Graphics_drawSelectedRadioButton()

The function which draws a released radioButton

- Graphics_drawReleasedRadioButton()

The function which determines if radioButton has been pressed

- Graphics_isRadioButtonSelected()

10.2.2   Function Documentation

10.2.2.1   Graphics_drawRadioButton

Draws a RadioButton.

Prototype:
```
void
Graphics_drawRadioButton(const Graphics_Context *context,
                         const Graphics_RadioButton *radioButton)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>   ***radioButton*** is a pointer to the structure containing the extents of the RadioButton.

Description:
>   This function draws a RadioButton . The RadioButton will contain the image passed in the
>   RadioButton struct.

Returns:
>   None.

10.2.2.2   Graphics_drawReleasedRadioButton

Draws a released RadioButton.

Prototype:
```
void
Graphics_drawReleasedRadioButton(const Graphics_Context *context,
                                 const Graphics_RadioButton
*radioButton)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>   ***radioButton*** is a pointer to the structure containing the extents of the RadioButton.

Description:
>   This function draws a RadioButton using the released parameters.

Returns:
>   None.

10.2.2.3   Graphics_drawSelectedRadioButton

Draws a selected RadioButton.

Prototype:
```
void
Graphics_drawSelectedRadioButton(const Graphics_Context *context,
                                 const Graphics_RadioButton
*radioButton)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>
>   ***radioButton*** is a pointer to the structure containing the extents of the RadioButton.

Description:
>   This function draws a RadioButton using the selected parameters.

Returns:
>   None.

10.2.2.4 Graphics_isRadioButtonSelected

Determines if x and y coordinates are contained in RadioButton.

Prototype:
```
bool
Graphics_isRadioButtonSelected(const Graphics_RadioButton
*radioButton,
                                  uint16_t x,
                                  uint16_t y)
```

Parameters:
>   ***radioButton*** is a pointer to the structure containing the extents of the RadioButton.
>
>   ***x*** x-coordinate to be determined if is inside RadioButton
>
>   ***y*** y-coordinate to be determined if is inside RadioButton

Description:
>   This function determines if x and y coordinates are contains inside RadioButton.

Returns:
>   true if x and y coordinates are inside RadioButton, false if not

10.3 Programming Example

```
Graphics_RadioButton radioButton1 = {
        5,
        15,
        true,
        4,
        GRAPHICS_COLOR_BLACK,
        9,
        GRAPHICS_COLOR_BLACK,
        GRAPHICS_COLOR_WHITE,
        &g_sFontFixed6x8,
        "Option #1"
};

Graphics_drawRadioButton(&g_sContext, &radioButton1);
```

11   CheckBox API

11.1   Introduction

The CheckBox API provides simple functions to draw a checkBox on the display.

11.2   Checkbox_api

Functions

- void Graphics_drawCheckBox (const Graphics_Context ∗context, const Graphics_CheckBox ∗checkBox)
- void Graphics_drawReleasedCheckBox (const Graphics_Context ∗context, const Graphics_CheckBox ∗checkBox)
- void Graphics_drawSelectedCheckBox (const Graphics_Context ∗context, const Graphics_CheckBox ∗checkBox)
- bool Graphics_isCheckBoxSelected (const Graphics_CheckBox ∗checkBox, uint16_t x, uint16_t y)

11.2.1   Function Documentation

11.2.1.1   Graphics_drawCheckBox

Draws a checkbox.

Prototype:
```
void
Graphics_drawCheckBox(const Graphics_Context *context,
                      const Graphics_CheckBox *checkBox)
```

Parameters:
   context is a pointer to the drawing context to use.
   checkBox is a pointer to the structure containing the extents of the checkbox.

Description:
   This function draws a checkbox. The checkbox will be created based on the parameters passed in the checkbox struct.

Returns:
   None.

11.2.1.2 Graphics_drawReleasedCheckBox

Draws a released Checkbox.

Prototype:
```
void
Graphics_drawReleasedCheckBox(const Graphics_Context *context,
                              const Graphics_CheckBox *checkBox)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>
>   ***checkBox*** is a pointer to the structure containing the extents of the checkBox.

Description:
>   This function draws a released checkbox using the selected parameters.

Returns:
>   None.

11.2.1.3 Graphics_drawSelectedCheckBox

Draws a selected Checkbox.

Prototype:
```
void
Graphics_drawSelectedCheckBox(const Graphics_Context *context,
                              const Graphics_CheckBox *checkBox)
```

Parameters:
>   ***context*** is a pointer to the drawing context to use.
>
>   ***checkBox*** is a pointer to the structure containing the extents of the checkBox.

Description:
>   This function draws a selected checkbox using the selected parameters.

Returns:
>   None.

11.2.1.4 Graphics_isCheckBoxSelected

Determines if x and y coordinates are contained in the checkbox.

Prototype:
```
bool
Graphics_isCheckBoxSelected(const Graphics_CheckBox *checkBox,
                            uint16_t x,
                            uint16_t y)
```

Parameters:
>   ***checkBox*** is a pointer to the structure containing the extents of the checkbox.
>
>   ***x*** x-coordinate to be determined if is inside button

y y-coordinate to be determined if is inside button .

Description:
This function determines if x and y coordinates are contains inside checkbox struct.

Returns:
true if x and y coordinates are inside checkbox, false if not

11.3   Programming Example

```
Graphics_CheckBox checkBox1 = {
            5,
            15,
            false,
            4,
            GRAPHICS_COLOR_BLACK,
            GRAPHICS_COLOR_WHITE,
            GRAPHICS_COLOR_BLACK,
            9,
            &g_sFontFixed6x8,
            "Option #1"
};

Graphics_drawCheckBox(&g_sContext, &checkBox1);
```
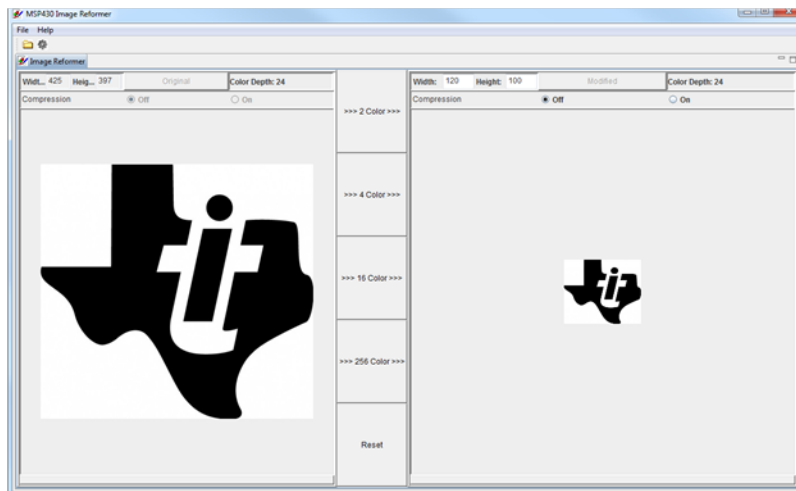
12   Using the MSP Image Reformer Tool

12.1   Introduction

Image Reformer converts images into C code that can be used with the MSP Graphics Library. Import your source image, make your bpp and size settings, generate C code, and then add the resulting file into your project.

12.2   Running MSP Image Reformer Tool

To run the tool go to `{GRLIB_INSTALLATION_PATH}\utils\image-reformer` and run `imagereformer.exe`
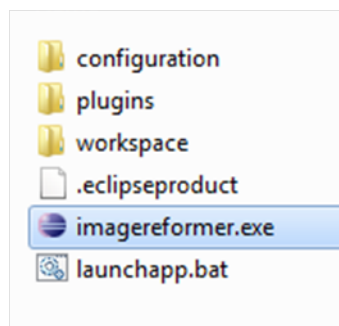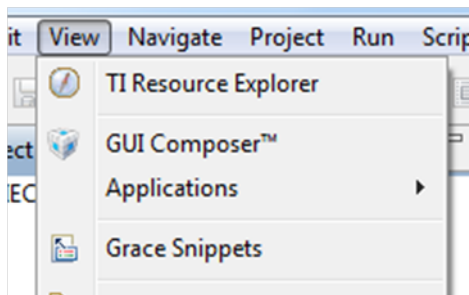

Note:
> In order to keep MSP Graphics Library and Open Source Project the JRE is not shipped with

the Library and it requires that the users have Java 1.5 or later installed in their machines. Currently the tool only has support for Windows OS support.
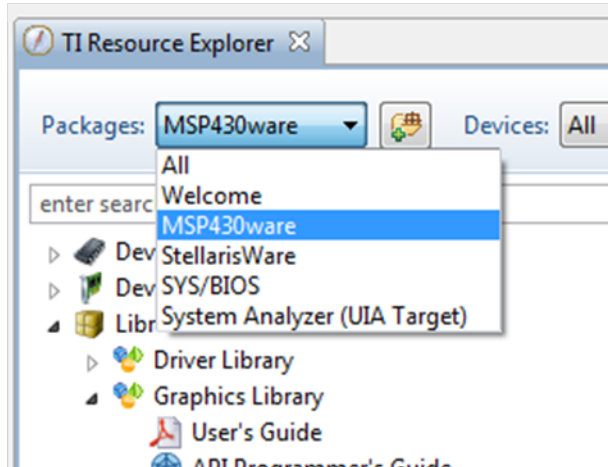
12.3 Launching MSP Image Reformer from TI Resource Explorer

If you download MSP Graphics Library as part of MSPWare, you will have the option to launch the MSP Image Reformer tool from TI Resource Explorer.
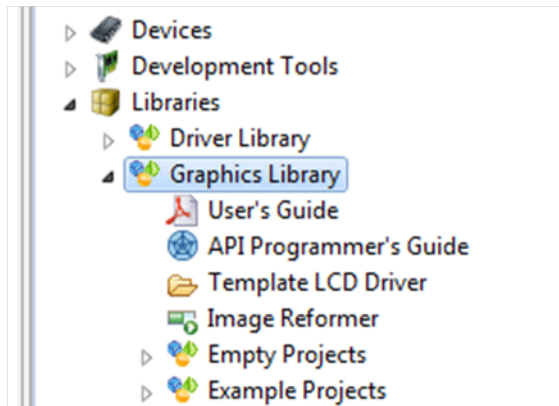
To launch the MSP Image Reformer tool tool, go to TI Resource Explorer windows View -> TI Resource Explorer.
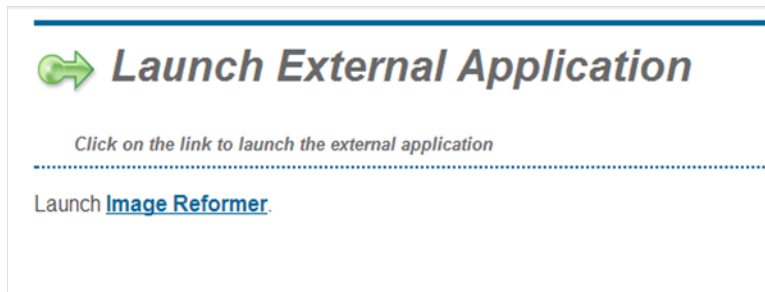

Under Packages select MSPware.

Expand Libraries and Graphics Library and Select Image Reformer.

Finally, click on the "MSP Image Reformer Tool".

Note:

>   To download MSPware go to `MSPWare`.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © , Texas Instruments Incorporated