Graphic LCD and Touchscreen Functions for Velleman VMA412 with ILI9341 Controller used with a STM32F446 Nucleo Board

Version 0.1rc1

Jesse op den Brouw
The Hague University of Applied Sciences
(THUAS)

July 19, 2020

The Velleman VM412 is an Arduino Uno/Mega compatible 2.8" color Graphic LCD with 320x240 pixels controlled by an ILI9431 graphic controller. The ILI9431 is connected using an 8-bit 8080 interface to the board. More information can be found via `https://www.velleman.eu/products/view/?id=435582`. There are a number second sources for this board such as the Makerfactory 2.8" Touchscreen and the ELEGOO UNO R3 2.8 Inches TFT Touch Screen.

The library is written in C and consists of four driver files and one C test file.

Library functions include plotting a pixel, rectangles, circles, arcs, printing strings, rotating displays and filling an object, and reading X, Y and pressure values. It also has primitive console based printing functions.

The library uses 18-bit colors where each color has 6 bits used.

The library is tested using a STM32F446 Nucleo board and STMCubeIDE version 1.3.1.

The VMA412 has an SD-card socket. This library has **no** support for accessing SD-cards.

Please mail me at: J.E.J.opdenBrouw@hhs.nl

Contents

1 Software License

This software is licensed with the BSD License:

```
Software License Agreement (BSD License)

Copyright (c) 2020 Jesse op den Brouw.  All rights reserved.

Portions based on :
 Copyright (c) 2012 Adafruit Industries.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS␣IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

2 Documentation License

This documentation is licensed with the Latex Project Public License:

```
Copyright 20205 J.E.J. op den BRouw

This work may be distributed and/or modified under the
conditions of the LaTeX Project Public License, either version 1.3
of this license or (at your option) any later version.
The latest version of this license is in
  http://www.latex-project.org/lppl.txt
and version 1.3 or later is part of all distributions of LaTeX
version 2005/12/01 or later.

This work has the LPPL maintenance status ‘maintained’.

The Current Maintainer of this work is J.E.J. op den Brouw.
```

3 Mouting the VMA412

The Velleman VM412 is an Arduino Uno/Mega compatible 2.8" color Touchscreen Graphic LCD with 320x240 pixels controlled by an ILI9431 graphic controller. The VMA412 has an Arduino Uno compatible connection and can be connected to numerous STM32F Nucleo boards. Place the VM412 board as instructed. See Figure 1 for a visual inspection.

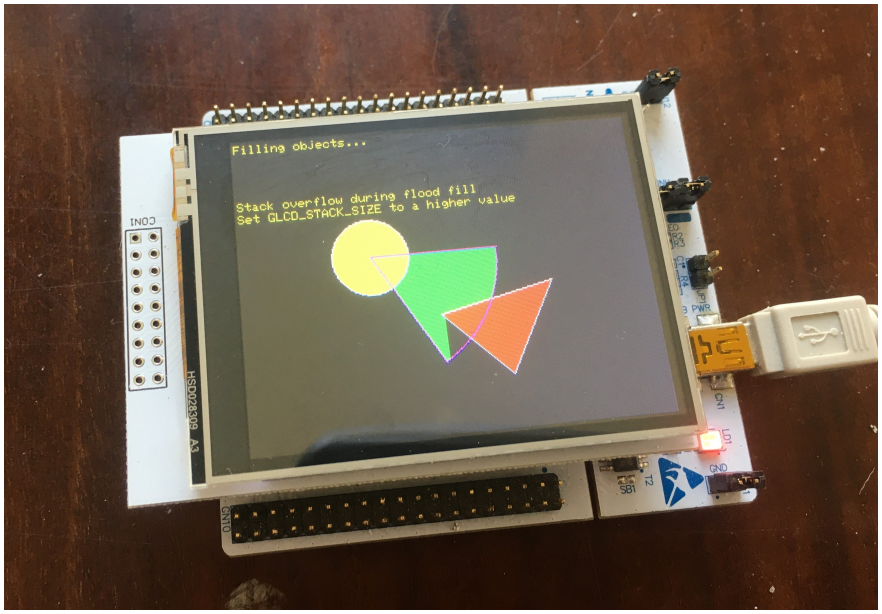

Figure 1: Showing the VMA412 mounted on a STM32F446 Nucleo board.

4 Creating a project

If you have downloaded the code from GitHub, then you already have all files needed. Just start STMCubeIDE and open the project. If you need to create a fresh project, make sure that you have selected the correct device. After creating a project, make sure that the five files are in the appropriate folders. These files are:

```
glcd_ili9341_vma412.h   -- place in Core/Inc, definitions, typedefs etc
glcd_ili9341_vma412.c   -- place in Core/Src, functions to call
touchscreen_vma412.h    -- place in Core/Inc, definitions, typedefs etc
touchscreen_vma412.c    -- place in Core/Src, functions to call
main.c                  -- place in Core/Src, demo using functions
```

5 Running the demo

If you have set up your project just start compilation using `Project→Build Project`. Then start the the demo using `Run→Run`. The demo starts by asking you to touch one of four rectangles to start the GLCD demo, the touchscreen demo, the rotation demo or the character

set demo, see Figure 2. At the end of the GLCD demo, there will be a list of running times for selected graphic functions as can be seen in Figure 3.

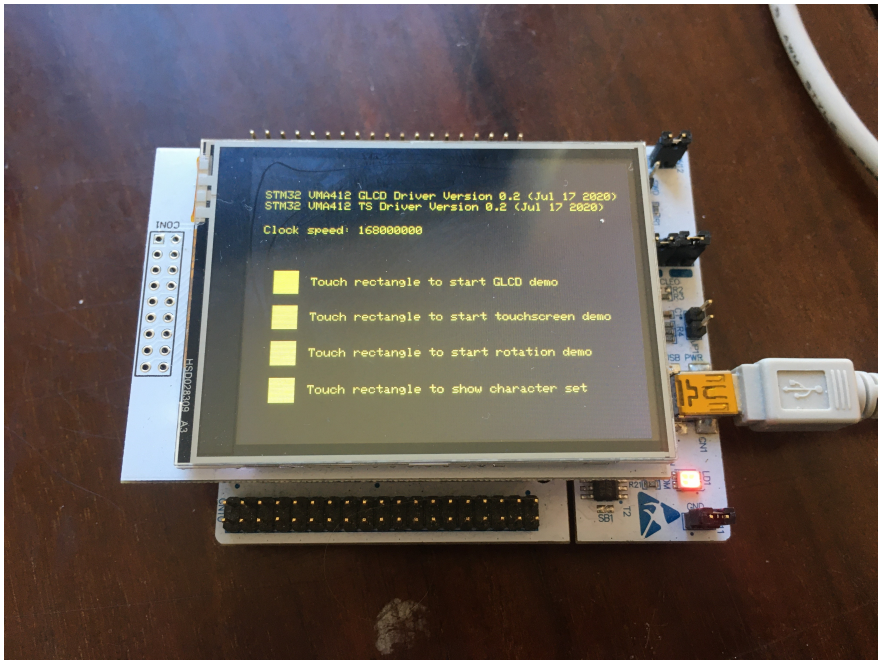

Figure 2: Starting the demo.

Figure 3: Displaying times for selected GLCD functions.

6 Color system

The GLCD is set up to use the 18-bit color specification. This means that colors are specified with unsigned 32 bits. The specification is as follows:

Bits 31-24: should be kept at 0.
Bits 23-16: red – only upper 6 bits are used
Bits 15-8: green – only upper 6 bits are used
Bits 7-0: blue – only upper 6 bits are used

Note: The 16 bit color specification in **NOT** supported, but they can be converted. See § 8.4.32.

Please note that although 8 bits per color can be specified, only the upper 6 bits of each color are used, since the display uses 18 bits of color infomation. The lower 2 bits of each color are ignored. This means that color `0x000000` and `0x000001` are displayed the same, but are different in compares.

The color is specified using the the type `glcd_color_t`.

6.1 Predefined colors

There are some predefines colors:

```
#define GLCD_COLOR_BLACK   (0x000000)
#define GLCD_COLOR_BLUE    (0x0000ff)
#define GLCD_COLOR_GREEN   (0x00ff00)
#define GLCD_COLOR_CYAN    (0x00ffff)
#define GLCD_COLOR_RED     (0xff0000)
#define GLCD_COLOR_MAGENTA (0xff00ff)
#define GLCD_COLOR_YELLOW  (0xffff00)
#define GLCD_COLOR_WHITE   (0xffffff)

#define GLCD_COLOR_GREY50  (0x7f7f7f)

/* THUAS default color */
#define GLCD_COLOR_THUASGREEN ((158<<16)|(167<<8)|0)
```

6.2 Using your own color

To use your own color, please use the color type `glcd_color_t`:

```
glcd_color_t SkyBlue2 = (126<<16)|(192<<8)|238
```

as taken for the LaTeX `xcolor` package (`https://www.ctan.org/pkg/xcolor`).

Note: don't ever change the type `glcd_color_t`.

7 X and Y coordinates, display orientation

The X and Y coordinates use type `uint16_t` for their values. The display is set up in landscape where `x` is between 0 and 319 and `y` is between 0 and 239. Point (0,0) is in the upper left corner, point (239,319) is in the lower right corner. The landscape orientation is with the SD-card socket to the right. The display orientation can be changed, see § 8.4.2.

8 Graphic LCD

8.1 GLCD Initialization

First you have to set up the clock system used by the STM32F microcontroller. If you use the external clock generator, make sure that the value of `HSE_VALUE` is set to the correct frequency. In the case of the Nucleo board it is 8000000 (8 MHz). This value is defined in `stm32f4xx_hal_conf.h` and `system_stm32f4xx.c`.

If you use the internal HSI (`HSIVALUE`), the frequency is 16 MHz by default. Best is to set up the clock speed to the maximum frequency allowed by the microcontroller.

Initialize the display by calling the `glcd_init()` function. After calling that function the display is initialized and ready for use.

8.1.1 Initialize display

Note: call this function **after** the clock system is set up.

This function must be called after the clock system is set up and before using any other GLCD functions. The function prototype is:

```
void glcd_init(void);
```

8.2 Very low GLCD functions

8.2.1 Setting the write delay

Note: use with care.
Sets the delay for write actions. There is no need to call this function as the correct timing is calculated according to the system clock speed after the clock system is set up, but you can tweak this value to get maximum performance. `delay` must be greater than 0. The function prototype is:

```
void glcd_set_write_pulse_delay(uint32_t delay);
```

8.2.2 Setting the read delay

Note: use with care.
Sets the delay for read actions. There is no need to call this function as the correct timing is

calculated according to the system clock speed after the clock system is set up, but you can tweak this value to get maximum performance. `delay` must be greater than 0. The function prototype is:

```
glcd_set_set_read_pulse_delay(uint32_t delay);
```

8.3 GLCD low level functions

There are a number of low level functions. They are normally not needed.

8.3.1 Reading data from the display

Te read data from the display, use the function `glcd_read_terminate`. Reading is explicitly terminated. The function prototype is:

```
void glcd_read_terminate(uint16_t cmd,            // The command
                         uint16_t amount,         // Amount of data
                         glcd_buffer_t data[]);   // The buffer
```

8.3.2 Writing data to the display

Writes data to the display, no explicit terminate. This means that multiple writes can be done in sequence.

```
void glcd_write(uint16_t cmd,                     // The command
                uint16_t amount,                  // Amount of data
                const glcd_buffer_t data[]);      // The buffer
```

8.3.3 Explicit terminating a write

Terminates a write:

```
void gcld_terminate_write(void);
```

8.3.4 Changing the buffer type

The low level functions use an internal buffer. The buffer is of type `glcd_buffer_t`. This normally set to an unsigned 16-bit size. This size can be changed:

Set to `uint8_t` for minimal resources
Set to `uint16_t` for best speed
Set to `uint32_t` for maximum capacity (not recommended)

8.4 GLCD high level commands

8.4.1 Delay

To delay your applications (in milliseconds), use :

```
void glcd_delay_ms(uint32_t delay);
```

8.4.2 Rotating the display

To set the rotation of the display, use:

```
void glcd_setrotation(glcd_rotation_t rot);
```

Where `rot` is one of:

```
GLCD_SCREEN_ROT0          // Standard landscape, default
GLCD_SCREEN_ROT90         // Rotate 90 degrees
GLCD_SCREEN_ROT180        // Rotate 180 degrees
GLCD_SCREEN_ROT270        // Rotate 270 degrees
```

Notes on rotation:

When you rotate the display, the current display contents are not deleted. You have to clear the display explicitly.

The default rotation is 0°. The display is in landscape with the SD-card socket on the right of the display. A rotation of 90° sets the display is portrait mode with the SD-card socket at the bottom of the display. A rotation of 180° sets the display in landscape mode with the SD-card socket to the left of the Display. A rotation of 270° set the display in portrait mode with the SD-card socket at the top of the display.

To be compatible with Arduino-based software, set the rotation to 90°.

Note: the touchscreen is **not** rotated. The touchscreen functions are setup for the standard rotation of 0°. How to handle the touchscreen in case of rotated displays, see § 9.4.

8.4.3 Clear the display

To clear the display with a color, use:

```
void glcd_cls(glcd_color_t color);
```

8.4.4 Plot a pixel

To plot a pixel, use:

```
void glcd_plotpixel(uint16_t x,           // x coordinate
                    uint16_t y,           // y coordinate
                    glcd_color_t color);  // color
```

Note: if `x` is greater that the width of the display minus 1, the pixel is not plotted.
Note: if `y` is greater that the height of the display, minus 1, the pixel is not plotted.

8.4.5 Read a pixel

To read a pixel (getting color information) in 32-bit version, use:

```
glcd_color_t glcd_readpixel(uint16_t x,   // x coordinate
                            uint16_t y);  // y coordinate
```

Note: if `x` is greater that the width of the display minus 1, the returned color is undefined.
Note: if `y` is greater that the height of the display, minus 1, the returned color is undefined.

8.4.6 Plot a horizontal line

To plot a horizontal line (fast), use:

```
void glcd_plothorizontalline(uint16_t x,           // x coordinate
                             uint16_t y,           // y coordinate
                             uint16_t w,           // width
                             glcd_color_t color);  // color
```

Note: if `x+w` is greater than the width of the display minus 1, the pixels in excess are not plotted.

8.4.7 Plot a vertical line

To plot a vertical line (fast), use:

```
void glcd_plotverticalline(uint16_t x             // x coordinate
                           uint16_t y,            // y coordinate
                           uint16_t h,            // height
                           glcd_color_t color);   // color
```

Note: if `y+h` is greater than the height of the display minus 1, the pixels in excess are not plotted.

8.4.8 Plot a line with any angle and length

To plot a line with any angle and length, use:

```
void glcd_plotline(uint16_t x0,           // x start point
                   uint16_t y0,           // y start point
                   uint16_t x1,           // x end point
                   uint16_t y1,           // y end point
                   glcd_color_t color);   // color
```

Note: the part of the line that is exceeding the physical dimensions of the display are not plotted.

8.4.9 Plot a character using buildin font

To plot a character using the buildin font (5x8), use:

```
void glcd_plotchar(uint16_t x,              // x coordinate
                   uint16_t y,              // y coordinate
                   uint8_t c,               // character (0-255)
                   glcd_color_t color,      // color
                   glcd_color_t bg);        // background color
```

Note: character is one of 0 – 255 (no special C treatment)
Note: if `color` is equal to `bg` then pixels having background color are not printed!

8.4.10 Plot a string using buildin font

To plot a string using the buildin font, use:

```
void glcd_plotstring(uint16_t x,                  // x coordinate
                     uint16_t y,                  // y coordinate
                     char str[],                  // the string
                     glcd_color_t color,          // color
                     glcd_color_t bg,             // background color
                     glcd_spacing_t spacing);     // spacing
```

Note: a `\0` terminates a string (as in C). This means that buildin character 0 cannot be printed
Note: if `color` is equal to `bg` then pixels having background color are not printed
Note: `spacing` is one of:

```
GLCD_STRING_CONDENSED  // zero pixels apart
GLCD_STRING_NORMAL     // one pixel apart
GLCD_STRING_WIDE       // two pixels apart
```

8.4.11 Plot a rectangle

To plot a rectangle, use:

```
void glcd_plotrect(uint16_t x,              // x coordinate
                   uint16_t y,              // y coordinate
                   uint16_t w,              // width
                   uint16_t h,              // height
                   glcd_color_t color);     // color
```

Note: `x` and `y` specify the upper left corner.

8.4.12 Plot a filled rectangle

To plot a filled rectangle, use:

```
void glcd_plotrectfill(uint16_t x,            // x coordinate
                       uint16_t y,            // y coordinate
```

```
                          uint16_t w,            // width
                          uint16_t h,            // height
                          glcd_color_t color);   // color
```

Note: `x` and `y` specify the upper left corner.

8.4.13 Plot a rectangle with rounded corners

To plot a rectangle with rounded corners, use:

```
void glcd_plotrectrounded(uint16_t x,           // x coordinate
                          uint16_t y,           // y coordinate
                          uint16_t w,           // width
                          uint16_t h,           // height
                          uint16_t r,           // radius
                          glcd_color_t color);  // color
```

Note: `x` and `y` specify the upper left corner as plotted with `glcd_plotrect`.

8.4.14 Plot a filled rectangle with rounded corners

To plot a filled rectangle with rounded corners, use:

```
void glcd_plotrectroundedfill(uint16_t x,           // x coordinate
                              uint16_t y,           // y coordinate
                              uint16_t w,           // width
                              uint16_t h,           // height
                              uint16_t r,           // radius
                              glcd_color_t color);  // color
```

Note: `x` and `y` specify the upper left corner as plotted with `glcd_plotrect`.

8.4.15 Plot a circle

To plot a circle, use:

```
void glcd_plotcircle(uint16_t x0,            // center x coordinate
                     uint16_t y0,            // center y coordinate
                     uint16_t r,             // radius
                     glcd_color_t color);    // color
```

8.4.16 Plot a filled circle

To plot a filled circle, use:

```
void glcd_plotcircle(uint16_t x0,            // center x coordinate
                     uint16_t y0,            // center y coordinate
                     uint16_t r,             // radius
                     glcd_color_t color);    // color
```

8.4.17 Plot quarters of a circle

To plot quarters of a circle, use:

```
void glcd_plotcirclequarter(uint16_t x0,              -- center x
                            uint16_t y0,              -- center y
                            uint16_t r,               -- radius
                            glcd_corner_t cornername, -- corners
                            glcd_color_t color);      -- color
```

Note: this function doesn't plot the start and end points
Note: `cornername` is one of or a composition of: `GLCD_CORNER_UPPER_LEFT`, `GLCD_CORNER_UPPER_RIGHT`, `GLCD_CORNER_LOWER_RIGHT` and `GLCD_CORNER_LOWER_LEFT`

8.4.18 Plot and fill left and/or right halves of a circle

To plot and fill left and/or right halves of a circle, use:

```
void glcd_plotcirclehalffill(uint16_t x0,                -- center x
                             uint16_t y0,                -- center y
                             uint16_t r,                 -- radius
                             glcd_cornerhalves_t corners, -- see notes
                             int16_t delta,              -- see notes
                             glcd_color_t color);        -- color
```

Notes `delta` is the squeeze factor, positive squeezes left/right, negative squeezes top/bottom, 0 for plain circle.
Note: `corners` is one of `GLCD_CORNER_LEFT_HALF`, `GLCD_CORNER_RIGHT_HALF` and `GLCD_CORNER_BOTH`.
Note: doesn't plot the outer top-to-bottom vertical line.

8.4.19 Plot an arc

To plot an arc, use:

```
void glcd_plotarc(uint16_t xc,              // center x coordinate
                  uint16_t yc,              // center y coordinate
                  uint16_t r,               // radius
                  float start,              // start angle in degrees
                  float stop,               // stop angle in degrees
                  glcd_color_t color);      // color
```

Note: this function is only available if `GLCD_USE_ARC` is defined.
Note: this function uses `sinf` and `cosf` math functions, using the onboard FPU.

8.4.20 Plot a triangle

To plot a triangle, use:

```
void glcd_plottriangle(uint16_t x1,          // start x
                       uint16_t y1,          // start y
                       uint16_t x2,          // second x
                       uint16_t y2,          // second y
                       uint16_t x3,          // last x
                       uint16_t y3,          // last y
                       glcd_color_t color);  // color
```

8.4.21 Plot a 2-color bitmap

To plot a 2-color bitmap, use:

```
void glcd_plotbitmap(uint16_t x,              // x coordinate
                     uint16_t y,              // y coordinate
                     const uint8_t bitmap[],  // the bitmap, see note
                     uint16_t w,              // width
                     uint16_t h,              // height
                     glcd_color_t color,      // color
                     glcd_color_t bg);        // background color
```

Note: `bitmap` consists of bytes (`uint8_t`). A 1 in a byte is converted to `color`, a 0 in a byte is converted to `bg`.

Note: you will get compiler warnings if your bitmap is in RAM, because `bitmap` is qualified as **`const`**.

Note: if you need to convert your image to a 2-bit color bitmap, have a look at `https://lvgl.io/tools/imageconverter`.

8.4.22 Plot a 256-color indexed bitmap

To plot a 256-color indexed bitmap, use:

```
void glcd_plotbitmap8bpp(uint16_t x,              // x start point
                         uint16_t y,              // y start point
                         uint16_t w,              // width
                         uint16_t h,              // height
                         const uint8_t *pic,      // start of image
                         const uint8_t *palette); // start of palette
```

Note: each pixel is specified with a byte (`uint8_t`). This is an index into the palette.
Note: each color is specified with a 32-bit RGB color specification. This means that each color takes up four bytes and the total palette size is 1024 bytes. The byte order for each indexed color is as follows: first byte is red, second byte is green, third byte is blue, fourth byte is not used.
Note: is `palette` is equal to `NULL` then the palette is at the start of the image and the image bytes follow the palette.

Note: you will get compiler warnings if your bitmap is in RAM, because `bitmap` is qualified as **`const`**.

Note: if you need to convert your image to a 256-color indexed image, have a look at `https://lvgl.io/tools/imageconverter`.

8.4.23 Display inversion

Sets the display inversion (or not):

```
void glcd_inversion(glcd_display_inversion_t what);
```

Note: `what` is one of:

```
GLCD_DISPLAY_INVERSION_OFF
GLCD_DISPLAY_INVERSION_ON
```

8.4.24 Display idle

Set the display to idle (or not):

```
void glcd_idle(glcd_display_idle_t what);
```

Note: `what` is one of:

```
GLCD_DISPLAY_IDLE_OFF
GLCD_DISPLAY_IDLE_ON
```

8.4.25 Display on or off

Sets the display on or off:

```
void glcd_display(glcd_display_t what);
```

Note: `what` is one of

```
GLCD_DISPLAY_OFF
GLCD_DISPLAY_ON
```

8.4.26 Flood fill an object

Flood fill an object using a software stack based approach:

```
void glcd_floodfill(uint16_t xs,                // x start point
                    uint16_t ys,                // y start point
                    glcd_color_t fillColor,     // color for pixel == 1
                    glcd_color_t defaultColor); // color for pixel == 0
```

Note: this function is only available if GLCD_USE_FLOOD_FILL is defined.
Note: set GLCD_STACK_SIZE to an appropiate value.
Note: if you have problems filling an object increase the value of GLCD_STACK_SIZE.
Note: the stack uses unsigned 32-bit entries so the complete stack uses GLCD_STACK_SIZE*4 bytes of RAM.

8.4.27 Scroll the display vertical upwards

To scroll the display vertical a number of lines, use:

```
void glcd_scrollvertical(uint16_t lines);   // lines to scroll upwards
```

Note: this is a software based scroll, could be slow. The lines are scrolled upwards off the display (no rotation), and the vacant lines are left untouched.

8.4.28 Console based character printing

To use a simple console based character printing, use:

```
void glcd_putchar(char c);
```

Note: characters are printed in yellow, background is black.
Note: \f (form feed) clears the display.
Note: \n returns and goes to the next line, may cause a vertical shift.
Note: \r return to the beginning of the line.
Note: \b erases last character and goes one character back (ultimate to the beginning of the line).
Note: \t creates tab stops at 8 character intervals
Note: all other characters are printed using the internal font.
Note: if a character "falls off" the display, line wrap will be used, may cause a vertical shift.
Note: this function is aware of the current display rotation.

8.4.29 Console based string printing

To use a simple console based string printing use:

```
void glcd_puts(char str[]);
```

Note: str is terminated with \0 (as in C).
Note: see §8.4.28 for character handling.
Note: if str equals NULL, then (null) is printed.

8.4.30 Get the current display width

To get the current display width, use:

```
uint16_t glcd_getwidth(void);
```

8.4.31 Get the current display height

To get the current display height, use:

```
uint16_t glcd_getheight(void);
```

8.4.32 Converting 16 bit colors

To convert 16 bit standard colors to 24-bit colors, use:

```
glcd_color_t glcd_convertcolor(uint16_t color16)
```

Note: 16 bit colors are composed of 5 bit red (bits 11 – 15), 6 bit green (bits 5 – 10) and 5 bit blue (bits 0 – 4).
Note: although the returned color is specified as a 24-bit value, only 65536 different colors can be composed using 16-bit colors.

8.5 GLCD Tayloring

There are a number of **#define**'s that can be manipulated to taylor the GLCD functions to your needs. They can be found in `glcd_ili9341_vma412.h`:

```
#define GLCD_USE_FLOOD_FILL
#define GLCD_STACK_SIZE (2000)
#define GLCD_USE_FLOOD_FILL_PRINT_IF_STACK_OVERFLOW
#define GLCD_USE_ARC

#define GLCD_WIDTH (320)
#define GLCD_HEIGHT (240)
```

Define `GLCD_USE_FLOOD_FILL` if you need to (flood) fill objects, undefine to save ROM and RAM resources. If you use flood fill, set the stack size `GLCD_STACK_SIZE` to an appropriate size. While testing or debugging, define `GLCD_USE_FLOOD_FILL_PRINT_IF_STACK_OVERFLOW`. This will print warning messages if there is a stack overflow when filling objects. Undefine if you are sure there will be no stack overflow. Define `GLCD_USE_ARC` if you need to plot arcs, undefine to save ROM resources. Plotting arcs use the onboard FPU for sine and cosine calculations.

Leave `GLCD_WIDTH` and `GLCD_HEIGHT` to their respective values.

The GLCD functions use an internal buffer. The size of the buffer elements are set to the type `glcd_buffer_t`. This is normally set to unsigned 16-bit. The buffer length is set to `GLCD_WIDTH`*3+1.

The size of the buffer elements can be changed, see 8.3.4.

```
typedef uint16_t glcd_buffer_t;
```

Set to `uint8_t` for smallest RAM footprint, but causes extra CPU time. Set to `uint16_t` for fastest processing. There is no speed gain when setting to `uint32_t`. It just wastes resources.

9 Touchscreen

9.1 Touchscreen Initialization

Before using any of the touchscreen functions, you have to initialize the touchscreen. Reading the X and Y coordinates of the pressed touchscreen point needs one of the onboard ADC's. The function `touchscreen_init` initializes the selected ADC. Please note that the selected ADC must be dedicated to the touchscreen functions, since the ADC is setup only once. Sharing the ADC with other functions is not recommended.

Please note that the ADC is setup for 10-bit conversions so the values returned are from 0 to 1023 (inclusive).

9.2 Touchscreen low level functions

9.2.1 Read raw X position

To read the current raw X position, use:

```
uint32_t touchscreen_readrawx(void);
```

Note: this value is only of interest if the touchscreen is pressed
Note: returns the raw X position from the touchscreen. This value is an indication of where the touchscreen is pressed. It is not the actual X position as can be used with the GLCD functions. You need to map the raw X position.

9.2.2 Read raw Y position

To read the current raw y position, use:

```
uint32_t touchscreen_readrawy(void);
```

Note: this value is only of interest if the touchscreen is pressed
Note: returns the raw Y position from the touchscreen. This value is an indication of where the touchscreen is pressed. It is not the actual Y position as can be used with the GLCD functions. You need to map the raw Y position.

9.2.3 Read raw pressure

To read the raw pressure, use:

```
uint32_t touchscreen_pressure(void);
```

Note: if the touchscreen is **not** pressed this functions returns a small number, typically 0. Any other values means that the touchscreen is pressed.

9.2.4 Mapping the raw X and Y values to display coordinates

To map raw X and Y values to display coordinates, use:

```
int32_t touchscreen_map(uint32_t value,
                        uint32_t tlow,
                        uint32_t thigh,
                        uint32_t slow,
                        uint32_t shigh);
```

Note: `value` is the raw value from X or Y
Note: `tlow` is lowest raw touchscreen value (X or Y)
Note: `thigh` is highest raw touchscreen value (X or Y)
Note: `slow` is lowest GLCD display value (X or Y)
Note: `shigh` is highest GLCD display value (X or Y)

Note: the returned value can be negative. This is an indication that the touchscreen is not calibrated correctly.
Note: this function internally uses floats to do the calculations. This will be done using the onboard FPU.

The map function corresponds to the linear equation:

$$returned\ value = a \cdot value + b \tag{1}$$

where a is

$$a = \frac{shigh - slow}{thigh - tlow} \qquad (thigh \neq tlow) \tag{2}$$

and b is

$$b = -a \cdot tlow; \tag{3}$$

Note: if $thigh = tlow$ the value `INT_MIN` is returned.

9.3 Touchscreen high level functions

9.3.1 Initializing the touchscreen

To initialize the touchscreen, use:

```
uint32_t touchscreen_init(ADC_TypeDef *used_ADC);
```

Note: this function must be called after the clock system is set up.
Note: `used_ADC` is an ADC handle. Currently only `ADC1` and `ADC2` are supported. `ADC3` is currently **not** supported.
Note: if the initialization succeeds a 1 is return, if failing a 0 is returned.

9.3.2 Testing if the touchscreen is pressed

To test whether the screen is touched, use:

```
uint32_t touchscreen_ispressed(uint32_t p);
```

Note: `p` is the raw pressure value

Note: a 1 is returned is touchscreen is pressed
Note: a 0 is returned is touchscreen is not pressed

9.4 Using the touchscreen on rotated displays

Of course, the touchscreen cannot be physically rotated, so you have to use software to fix the rotation.

When the display is rotated 0°, use:

```
uint16_t raw;
int16_t x, y;

raw = touchscreen_readrawx();
x = touchscreen_map(raw, TOUCH_LEFT, TOUCH_RIGHT, 0, glcd_getwidth());

raw = touchscreen_readrawy();
y = touchscreen_map(raw, TOUCH_BOTTOM, TOUCH_TOP, 0, glcd_getheight());
```

When the display is rotated 90°, use:

```
uint16_t raw;
int16_t x, y;

raw = touchscreen_readrawx();
y = touchscreen_map(raw, TOUCH_LEFT, TOUCH_RIGHT, 0, glcd_getheight());

raw = touchscreen_readrawy();
x = glcd_getwidth() - touchscreen_map(raw, TOUCH_BOTTOM, TOUCH_TOP,
                                      0, glcd_getwidth());
```

Note the swapped `x` and `y`, and the swapped `glcd_getheight` and `glcd_getwidth`.

When the display is rotated 180°, use:

```
uint16_t raw;
int16_t x, y;

raw = touchscreen_readrawx();
x = glcd_getwidth() - touchscreen_map(raw, TOUCH_LEFT, TOUCH_RIGHT,
                                      0, glcd_getwidth());
```

```
raw = touchscreen_readrawy();
y = glcd_getheight() - touchscreen_map(raw, TOUCH_BOTTOM, TOUCH_TOP,
                                       0, glcd_getheight());
```

When the display is rotated 270°, use:

```
uint16_t raw;
int16_t x, y;

raw = touchscreen_readrawx();
y = glcd_getheight() - touchscreen_map(raw, TOUCH_LEFT, TOUCH_RIGHT,
                                       0, glcd_getheight());

raw = touchscreen_readrawy();
x = touchscreen_map(raw, TOUCH_BOTTOM, TOUCH_TOP, 0, glcd_getwidth());
```

Note the swapped `x` and `y`, and the swapped `glcd_getheight` and `glcd_getwidth`.

Note: the pressure is not affected by the rotation.

9.5 Calibrating the touchscreen

The touchscreen of the Velleman VMA412 is constructed of resistive foils. Normally the resistance and hence the voltage measured is linear propotional to the location where the touchscreen is pressed. Small deficienties during fabrication produce differences between samples of the touchscreen. To compensatie for that, calibration is needed. There are a number of **`#define`**'s to calibrate the touchscreen. These can be found in `touchscreen_vma412.h`.

The outer left raw position is set with `TOUCH_LEFT`. The outer right raw position is set with `TOUCH_RIGHT`. The outer top raw position is set with `TOUCH_TOP`. The outer bottom raw position is set with `TOUCH_BOTTOM`.

Whether the touchscreen is pressed or not is calculated according to raw touchscreen values. The lower bound for pressed is set with `TOUCH_PRESSURE_LOW`. The upper bound for pressed is set with `TOUCH_PRESSURE_HIGH`.

Note all values must be within the values 0 to 1023 (inclusive).

Reading raw values is done by oversamping and then the mean of the samples is returned. You can set the number of samples using the define `TOUCH_SAMPLES`. The default value is 16. This value must be 2 or greater. Lower values speed up the reading of the touchscreen.

10 Debugging or production use

If you need to debug the functions, set the compiler optimization to `-O0` (no optimization) or `-Og` (optimize for debug). If you want to exhibit full speed support set the optimization to `-Ofast`. If you need the smallest ROM footprint and have some speed, set the optimization to `-Os`.

11 Nice tricks

To wait until the touchscreen is touched, use one of

```
while (p = touchscreen_pressure(), !touchscreen_ispressed(p)) {}
```

or

```
while (!touchscreen_ispressed(touchscreen_pressure())) {}
```

To wait for the touchscreen to be touched for the first time after some point (like an edge detection circuit), use

```
/* Wait while touchscreen is (still) touched */
while (touchscreen_ispressed(touchscreen_pressure())) {}
/* Wait for the touchscreen is touched after being untouched */
while (!touchscreen_ispressed(touchscreen_pressure())) {}
```

To print variables, the best way is to use a character buffer and the `snprintf` function defined in `stdio.h`. Make sure the buffer is large enough:

```
char buffer[40];

snprintf(buffer, sizeof buffer, "Clock␣speed:␣%lu", SystemCoreClock);
glcd_plotstring(10, 80, buffer, GLCD_COLOR_YELLOW, GLCD_COLOR_BLACK,
                GLCD_STRING_NORMAL);
```

or print via the console print function:

```
glcd_puts(buffer);
```

You could “rewire” the `printf` function by rewriting the `_write` and `_io_putchar` functions. This is not done explicitly because sometimes you need the USART to take control of `printf` and friends.

12 SD-cards are not supported with this library

The SD-card interface needs a complete layer of functions to manipulate the SD-cards (reading, writing, etc.). These functions are not implemented in this library. Best is to use FATFS (`http://elm-chan.org/fsw/ff/00index_e.html`).

13 Using the VMA412 with other hardware

The VMA412 uses 13 pins to do its job. Nearly all of the Arduino compatible pins are used. Only Arduino pin A5 if free, as are digital pins 10 through 13 (for the SD-card functions). But of course you can use the remaining pins on the Morpho connectors.

14 Todo's

Some todo's left:

- Quarter circle with fill plot function, will be faster than flood fill circles;
- Rectangle with rounded corners to support buttons to be pressed;
- Regular polygon plot function, with angle;
- Exchange top/bottom and left/right;
- Fix use of ADC3 for STM32F446;
- Make C++ wrappers;
- Put characters in RAM so new characters can be defined, overwriting older ones;
- High-level touchscreen X and Y read functions (needs rotation information).