Politenico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

HEAPLab Project Report

Implementation of display driver for MiosixOS

Author:
Davide Salaorni
Matteo Velati

Supervisor:
Ing. Federico Terraneo

March 15, 2021

Abstract

The work consists in the implementation of a display driver for Miosix, an OS kernel designed to run on 32-bit microcontrollers. The purpose of the code is to extend the collection of MXGUI drivers with the 1,8” TFT-LCD ST7735, which is required to perform the operations sent by the attached board, in this case a STM32F4Discovery with STM32F407VG MCU.
The results and well behaviour of the implemented driver are proved by the testing phase with benchmarks, a set of test included inside the MXGUI framework.

1 Introduction

We started our work by studying the architecture of MXGui library, from the structure of already implemented display drivers and applying reverse engineering to understand the underlying code.
Of course, we spent a meaningful amount of time reading the display datasheet and the board reference manual, which were fundamental to fully grasp the device architectures we were using.
Once understood the composition of Miosix kernel, we move on the code, implementing enough functions to turn on the display and draw at least a single pixel. After being able to accomplish this first important step, we continued gradually in the completion of the set of graphics functions provided by MXGui library and the testing phase, to ensure a perfect integration of ST7735 display with MiosixOS environment on the STM32F4Discovery board.

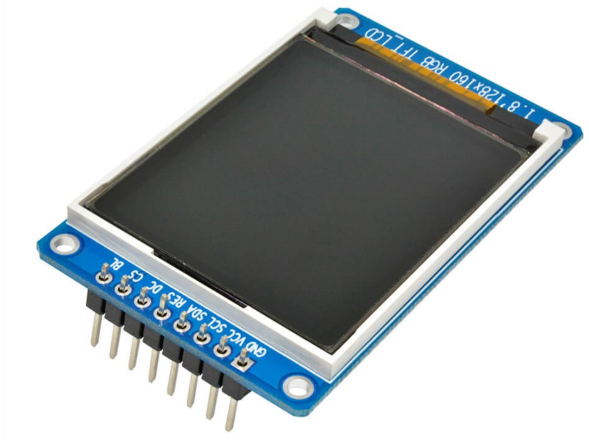


Figure 1: ST7735 display

2 Design and Implementation

In this section we want to describe how we move after studying datasheets and course materials. We want to highlight which was the starting point of our work and how we carried out the implementation of the code.
In the beginning, we spent a meaningful amount of time trying to understand the structure of MiosixOS and MXGUI framework, comparing already existing drivers and unrolling underlying files which link drivers code with the OS.
In the meanwhile, we have studied how the communication between the display and the board works, we have chosen the protocol to use (SPI) and we have studied the initialization sequence.
Then, we have implemented the *DisplayImpl* class merging the notions achieved in the previous steps, reaching the final structure of the driver, which was ready to be tested.

2.1 Underlying structure

The design of the driver is focused on the implementation of the functions showed in the following snippet of code, taken from the .h file of the driver. These are the basis functions that a user has to call to work with ST7735.

```
class DisplayImpl : public Display {
public:

    /**
     * Return an instance to this class(singleton)
     */
    static DisplayImpl& instance();

    /*
     * Turn the display On after it has been turned Off
     */
    void doTurnOn() override;

    /**
     * Turn the display Off. It can be later turned back On
     */
    void doTurnOff() override;

    /**
```

```
     * Set display brightness.
     * (ST7735 lacks of this feature)
     */
    void doSetBrightness(int brt) override;

    /**
     * Return a pair with the display height and width
     */
    std::pair<short int, short int> doGetSize() const
   override;

    /**
     * Write text to the display. If text is too long it will
     * be truncated
     */
    void write(Point p, const char *text) override;

    /**
     * Write part of text to the display
     */
    void clippedWrite(Point p, Point a,  Point b, const char
   *text) override;

    /**
     * Clear the Display. The screen will be filled with the
     * desired color
     */
    void clear(Color color) override;

    /**
     * Clear an area of the screen with a color
     */
    void clear(Point p1, Point p2, Color color) override;

    /**
     * Draw a pixel with desired color.
     */
    void setPixel(Point p, Color color) override;

    /**
     * Draw a line between point a and point b, with color c
     */
    void line(Point a, Point b, Color color) override;

    /**
```

```
     * Draw an horizontal line on screen.
     * Instead of line(), this member function takes an array
     * of colors to be able to individually set pixel colors
     * of a line.
     */
    void scanLine(Point p, const Color *colors, unsigned
   short length) override;

    /**
     * Return a buffer of length equal to this->getWidth()
     * that can be used to render a scanline.
     */
    Color *getScanLineBuffer() override;

    /**
     * Draw the content of the last getScanLineBuffer() on an
     * horizontal line on the screen.
     */
    void scanLineBuffer(Point p, unsigned short length)
   override;

    /**
     * Draw an image on the screen
     */
    void drawImage(Point p, const ImageBase& img) override;

    /**
     * Draw part of an image on the screen
     */
    void clippedDrawImage(Point p, Point a, Point b, const
   ImageBase& img) override;

     /**
     * Draw a rectangle (not filled) with the desired color
     */
    void drawRectangle(Point a, Point b, Color c) override;
    ...
}
```

The class *DisplayImpl* presents also a nested class called *pixel_iterator*, which is necessary to correctly draw texts on the display. In facts, methods of this class are called by underlying functions contained inside *font.h* which take care of writing texts. Thus, we didn't care of spacing and letter sizes because they are handled transparently by MXGui framework.

2.2 The implementation

Before starting the implementation phase, we have decided which settings and protocols were better for us.
At first, we chose to work only with the display normal mode (not the partial one), with a resolution of 128x160 pixels, and a color depth set to 16bit/pixel. For the communication protocol, needed to interface the display with the board, we opted for the 4-line Serial Peripheral Interface (SPI), in which data packet contains just the transmission byte and the control bit *DCX* is transferred with its own pin, conversely to what happens in 3-line SPI, where data packet encapsulates both the control bit and the transmission byte.

2.2.1 Pin selection and first feedbacks

At this point, the first question (and the one which lasted longer) was the pin selection problem. In the beginning, we chose to use the SPI1 interface, thus we selected those pins of the board which were used for SCK (Serial Clock) and MOSI (Master Output Slave Input), without consideting the MISO (Master Input Slave Output) since we only needed to write into the display and not to read from it. Instead, for RST (Reset), DCX (Data/-Command) and CSX (Chip Select) we opted for free pins that we could have handled manually.
Unfortunately, we didn't notice that the pins we chose were used at the same time by other internal peripherals of the board, so the signal was not clean for our purpose. Thus, we found other exploitable pins for MOSI and SCK, which weren't employed from other functions. However, neither this time we were able to correctly work since we found out a totally broken pin (PB3), making the SPI1 interface pretty unusable.

After all this struggles, we saw the right path: the SPI2 interface which communicates at a frequency of 42MHz on APB1 peripheral bus, against the 84MHz of SPI1 on APB2. With a clock divider of 4, we reduced the transmission frequency between the board and the display to 10.5 MHz.
After having found correctly the pins we needed, we started to receive some vital signs from the display, which, till that moment, had being always blank. What we had at the time were only some randomly initialized pixels, which was a huge feedback for us.

However, that had being the only feedback we received for a long period, despite of all the different combinations of commands we tried. At a certain point, we got the opportunity to test our environment with a oscilloscope that was helpful to reveal that our timing delays were incompatible with the timing chart for the 4-lines SPI writing protocol (section 9.3.1 of the datasheet) and the initial reset sequence.
Once those timings were fixed, the display started behaving correctly and we could proceed in our work.

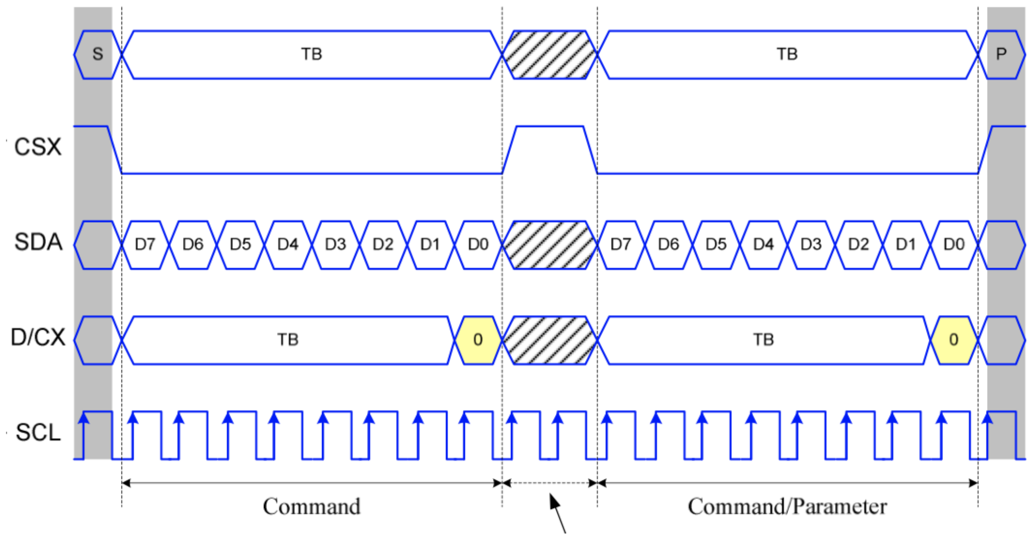


Figure 2: ST7735 4-line serial interface write protocol

2.2.2 Final configuration

The final configuration we used to develop and test the whole system is:

```
// Control interface
typedef Gpio<GPIOB_BASE, 13> scl;   //PB13,  SPI2_SCK  (af5)
typedef Gpio<GPIOB_BASE, 15> sda;   //PB15,  SPI2_MOSI (af5)
typedef Gpio<GPIOB_BASE, 4> csx;    //PB4,   free GPIO
typedef Gpio<GPIOC_BASE, 6> resx;   //PC6,   free GPIO
typedef Gpio<GPIOA_BASE, 8> dcx;    //PA8,   free GPIO
```

We started defining in our code two auxiliary objects, SPITransaction and CommandTransaction, that are responsible for software control of the chip select (CSX) and data command (DCX) pins, respectively.
Whenever we need to communicate to the display controller to run a command, we first instantiate both a SPITransaction and CommandTransaction object, lowering the relative pins. Inside their scope the first 8-bit command is sent. Then the CommandTransaction object is destructed and DCX set to high. A cycle keeps sending the parameters 8-bit per times, if needed, and eventually the SPITransaction object is destructed.
Only the $RAMWR$ command, used to write the GRAM, differs a bit from the previous procedure since it requires 16-bit parameters, which actually are the RGB composition of each single pixel. In this case we split each pixel sending first the most significant bits and then the remaining part, always 8-bit per time.

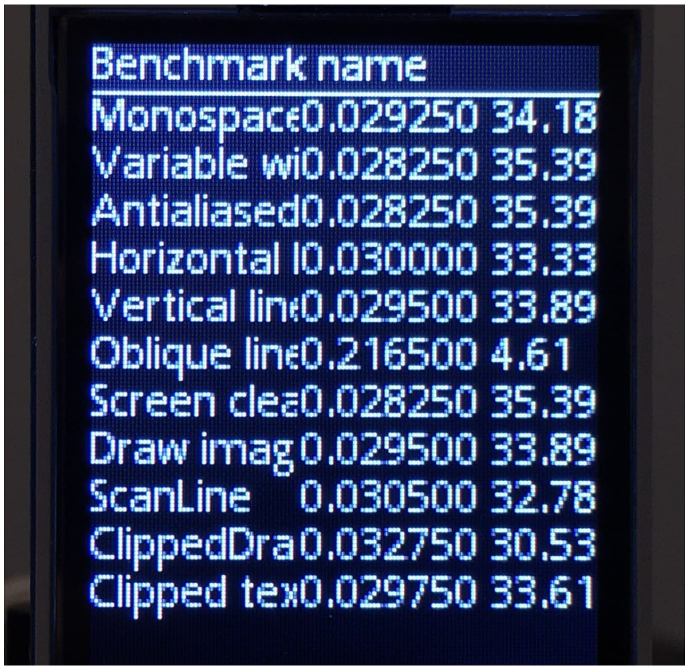
With reference to the Power Control Chart of the datasheet (section 9.13.2), whenever the display is powered, a sequence of hardware reset, software reset and sleep out commands must be sent to put the display in Normal Mode On. The first commands and data that are sent to the display after the power on sequence are the ones for the initialization sequence. Those commands takes care of defining the format of RGB picture data to 16 bit/pixel, setting the frame frequency of the full colors normal mode, setting the output waveform relation of the display, setting the display inversion mode control to line inversion, setting the amount of current drains with power controls commands and finally providing gamma adjustment for both positive and negative polarity which provides accurate colors. After that initial setup, the display is ready to use. We report here the initialization sequence that we used. The first byte of each row identifies the command to send, the second the number of parameters and the other the parameters themselves.

```c
const unsigned char initST7735b[] = {
// ST7735_COLMOD, color mode: 16-bit/pixel
0x3A, 0X01, 0x05,

// ST7735_FRMCTR1, normal mode frame rate
0xB1, 0x03, 0x00, 0x06, 0x03,

// ST7735_DISSET5, display settings
0xB6, 0x02, 0x15, 0x02,

// ST7735_INVCTR, line inversion active
0xB4, 0x01, 0x00,

// ST7735_PWCTR1, default (4.7V, 1.0 uA)
0xC0, 0x02, 0x02, 0x70,

// ST7735_PWCTR2, default (VGH=14.7V, VGL=-7.35V)
0xC1, 0x01, 0x05,

// ST7735_PWCTR4, bclk/2, opamp current small and medium low
0xC3, 0x02, 0x02, 0x07,

// ST7735_VMCTR1, VCOMH=4V VCOML=-1.1
0xC5, 0x02, 0x3C, 0x38,

// ST7735_PWCTR6, power control (partial mode+idle)
0xFC, 0x02, 0x11, 0x15,

// ST7735_GMCTRP1, Gamma adjustments (pos. polarity)
0xE0, 0x10,
    0x09, 0x16, 0x09, 0x20,
    0x21, 0x1B, 0x13, 0x19,
    0x17, 0x15, 0x1E, 0x2B,
    0x04, 0x05, 0x02, 0x0E,

// ST7735_GMCTRN1, Gamma adjustments (neg. polarity)
0xE1, 0x10,
    0x0B, 0x14, 0x08, 0x1E,
    0x22, 0x1D, 0x18, 0x1E,
    0x1B, 0x1A, 0x24, 0x2B,
    0x06, 0x06, 0x02, 0x0F,

// ST7735_NORON, normal display mode on
0x13
};
```

Each function implemented in the driver follow the same pattern: initially, a Memory Data Access Control command is sent, defining the write scanning direction of the frame memory. For text-optimization, the GRAM increment is set to up-to-down first, then left-to-right; conversely for image drawing optimization the increment is left-to-right first, then up-to-down.
Later on, a hardware window on the screen is set, meaning that the actual display size is shrunk (until a rectangle of 1x1 to write a single pixel) or enlarged (up the maximum display size to fill every pixel with the same color as the *clear()* does), with commands Row Address Set and Column Address Set. Then a Memory Write command is sent together with a list of parameters which identifies the pixels to be drawn.
Actually, the drivers supports both vertical and horizontal orientation: this is identified by a define statement in the settings configuration file of MXGui before compiling the kernel.

3 Experimental Results

After having tested each function of the library separately with simple mains, we ran the benchmark from the official MXGui examples directory obtaining the following results:



```
Benchmark name
Monospace0.029250 34.18
Variable wi0.028250 35.39
Antialiased0.028250 35.39
Horizontal l0.030000 33.33
Vertical line0.029500 33.89
Oblique line0.216500 4.61
Screen clea0.028250 35.39
Draw imag0.029500 33.89
ScanLine    0.030500 32.78
ClippedDra0.032750 30.53
Clipped tex0.029750 33.61
```

Figure 3: MXGui benchmark results

In the picture above we can see the performance of each single function performed by the display. Together with the name of the tested function, we can also evaluate the results related to computing time and fps counter for the considered task.

Here we have some pictures of our tests with the board. The first one shows a frame from the video of the plot of a sine curve which grows in width with the time. The second one is simply a loaded image fitting with the resolution we chose to use.

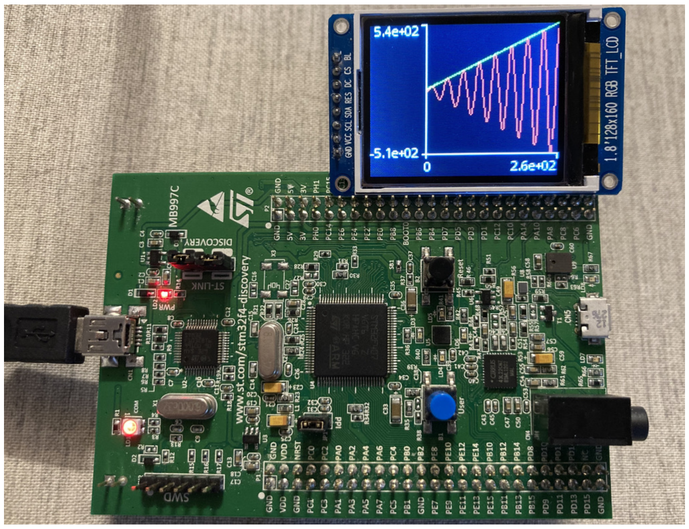


Figure 4: Display test in horizontal orientation mode

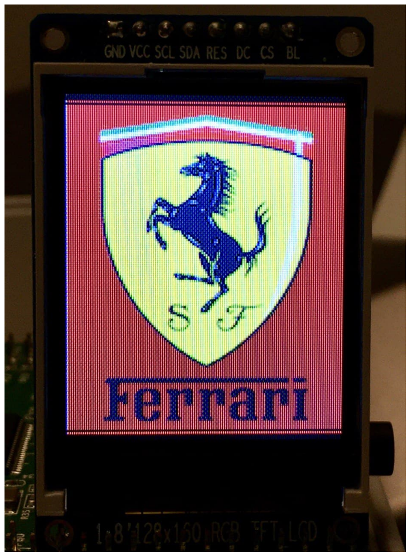


Figure 5: Display test in vertical orientation mode

4 Conclusions

To sum up, from results of benchmarks and testing phase in general, we can say that we have correctly implemented a working driver which can extend the support of MiosixOS also to display ST7735 connected to a STM32F4Discovery board.

5 References

- Miosix Wiki:
  https://miosix.org/wiki/index.php?title=Mxgui
- ST7735 controller datasheet:
  https://www.displayfuture.com/Display/datasheet/controller/ST7735.pdf
- UM1472 (Discovery kit with STM32F407VG MCU):
  https://www.st.com/resource/en/user_manual/dm00039084-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf
- RM0090 (Reference manual):
  https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf
- Adafruit ST7735 library:
  https://github.com/adafruit/Adafruit-ST7735-Library