

POLITECNICO DI MILANO

DIPARTIMENTO ELETTRONICA, INFORMAZIONE E
BIOINGEGNERIA

HEAPLAB PROJECT REPORT

Porting Mxgui to STM32F469I

Author:

Alessandro FIORILLO

Supervisor:

Dr. Federico TERRANEO

May 2, 2019



Abstract

The aim of this project is porting the graphical library Mxgui of Miosix to a new development board, which is STM32F469I. The major effort of this task has been writing a working driver for the display, consulting the necessary documentation provided by ST. In this document I will explain what are the main concepts to understand how the driver works and how to use it in a real application.

1 Introduction

The driver is tailored to the display already present on the board. It supports a color depth of 16 bpp and a resolution of 480*800 pixels both in portrait mode and landscape mode. Some simple modifications are needed inside /miosix/config/Makefile.inc to be able to use the library:

- First uncomment the line

```
OPT_BOARD := stm32f469ni_stm32f469i-disco
```

to tell the compiler which is the target board.

- In the section dedicated to board specific options, choose a linker script between

```
LINKER_SCRIPT := $(LINKER_SCRIPT_PATH)stm32_2m+384k_rom.ld
```

and

```
LINKER_SCRIPT := $(LINKER_SCRIPT_PATH)stm32_2m+12m_xram.ld
```

Do not use

```
LINKER_SCRIPT := $(LINKER_SCRIPT_PATH)stm32_2m+16m_xram.ld
```

otherwise the framebuffer for the display will overwrite the data used by the kernel.

- Uncomment the line

```
XRAM := -D__ENABLE_XRAM
```

since the framebuffer is stored in the external memory.

- As clock frequency, select

```
CLOCK_FREQ := -DHSE_VALUE=8000000 -DSYSCLK_FREQ_168MHz=168000000
```

All clock calculations inside the driver assume the board is running on those frequencies.

The driver uses two peripherals on the board, namely the LTDC and the DSI. Its code is written inside `/mxgui/drivers/display_stm32f4discovery.cpp`. It is structured in four parts:

- Clock configuration
- LTDC configuration
- DSI configuration
- Power-up sequence

In the following sections I will explain how those peripherals work at high level and how they are configured inside the library.

2 Design and Implementation

2.1 LTDC

The LTDC is a peripheral which autonomously fetches pixel data from the framebuffer and stores them in an internal FIFO. These pixels are then blended with the background color according to a blending factor, encoded in RGB888 and finally sent to the DSI.

Its configuration is quite straightforward and does not need special attention. What needs more caution is the configuration of its clock. The LTDC fetches a pixel from memory at each falling and raising edge of the LCD clock, which needs to be configured. This clock is generated starting from the HSE oscillator, which runs at 8 MHz on this board and passes through a series of frequency divisors and multipliers. Each of these can have a restricted set of

values and must guarantee an output frequency inside a certain range. More specifically, the HSE oscillator is first divided by the factor M of the main PLL, which is fixed at boot by Miosix and can not be modified (otherwise this would affect the entire system clock). The result of this division then passes through the PLLSAI, which can be freely configured. The following picture represents the flow from the HSE oscillator to the LTDC clock. For each step, the accepted range of output frequency and multiplier value is expressed in black color, while the red color represents the effective value which has been chosen inside the driver.

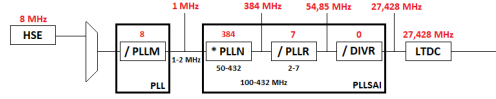


Figure 1: Simplified scheme of LTDC clock

To transport pixel data and synchronization events, many GPIOs are needed. The following table shows which GPIOs are used by the driver and their corresponding alternate function.

| Task | GPIO | Alternate function |
|-------------|------|--------------------|
| R0 | PC10 | 14 |
| R1 | PJ2 | 14 |
| R2 | PA11 | 14 |
| R3 | PA12 | 14 |
| R4 | PJ5 | 14 |
| G0 | PA6 | 14 |
| G1 | PG10 | 9 |
| G2 | PJ13 | 9 |
| G3 | PH4 | 9 |
| G4 | PC7 | 14 |
| G5 | PD3 | 14 |
| B0 | PD6 | 14 |
| B1 | PG11 | 14 |
| B2 | PG12 | 9 |
| B3 | PA3 | 14 |
| B4 | PB8 | 14 |
| Data enable | PF10 | 14 |
| Clock | PG7 | 14 |
| VSync | PA4 | 14 |
| HSync | PC6 | 14 |

All of these GPIOs run at high speed. Note that an additional GPIO, which is PH7, is used once to reset the display (as suggested by the documentation) but then it is never reused. Some GPIOs are part of the J port, which alongside the K port was not powered up at boot by Miosix. For this reason the kernel has been updated to power up also those ports at boot, modifying the code in `/miosix/arch/cortexM4_stm32f4/stm32f469ni_stm32f469i-disco/interfaces-impl/bsp.cpp`.

2.2 DSI

The DSI is a sophisticated peripheral to interface to various types of display, either with or without an internal controller and GRAM. This is possible since it can work in two different modes:

- *Video mode*: the DSI continuously sends a stream of pixels and synchronization signals to feed the display. This mode is suitable for displays

without an internal controller and GRAM. However this consumes high bandwidth and requires an accurate configuration of synchronization events. It is also required a second framebuffer to avoid tearing.

- *Command mode*: the DSI sends a set of commands to update the display only when explicitly requested by the programmer. This drastically reduces memory bandwidth, since the display is updated only when the framebuffer changes its content. Synchronization events are managed automatically, and there is no need to use a second framebuffer to avoid tearing. However this mode works only on displays with an integrated controller and GRAM.

Fortunately, the display on board of this controller is compatible with command mode, so this is the modality used by the driver.

The DSI peripheral is located between the LTDC and the display interface. It is composed by many parts which control the flow of pixels to the display.

- *DSI Wrapper*: it is the first component encountered by the pixels generated in the LTDC. Its task is to route those pixels in the correct part of the peripheral depending on the mode used (video or command).
- *DSI Host*: the core of the peripheral, it takes pixel data and generates a set of packets which will be redirected to the next layer, the D-PHY.
- *D-PHY*: it is the last layer of the peripheral, whose task is to serialize the received packets and send them to the display using a physical protocol called PPI.
- *DSI PLL*: it is a dedicated PLL which generates the different clocks needed in the various components of the peripheral.
- *Regulator*: this component provides energy to all parts of the peripheral, both the PLL and the DSI Host.

As anticipated in this list, the DSI does not send bare pixels, but encapsulates them in a series of packets. The format of these packets is specified by the DSI protocol. Both in video and command mode, the packets are created automatically by the peripheral, but it is needed to know how they

are structured, since they can be built and sent by software. This actually happens in the driver when the display is initialized in the power up sequence.

The DSI packets are divided into two categories: *short packets* and *long packets*. Short packets are always 4 bytes long and they are structured in this way:

- *DataID*: the first byte is called DataID, which is splitted in two sub-parts. The most significant two bits are called *Virtual Channel ID*, and they identify the target peripheral. In this driver, the Virtual Channel ID for the display is zero. The other six bits are the *Data Type*. The complete list of data types is specified in the DSI standard. Specific data types allow the peripheral to send, for example, a synchronization event without any payload (since the data type already specifies that this is, for instance, an HSYNC event). Other data types are generic and allow to send any data regardless of their meaning. Those are the ones used in the driver. The code *0x15* identifies the generic short packet (formally it is called *DSI_DCS_SHORT_PKT_WRITE_P1* in the standard).
- *Byte 0*: the first parameter sent.
- *Byte 1*: the second parameter sent.
- *ECC*: an error correction code for the header (automatically generated by the peripheral).

Long packets are used to send more data, and their structure is the following:

- *DataID*: same as before. The data type for a generic long packet is represented by the code *0x39* (formally called *DSI_DCS_LONG_PKT_WRITE* in the standard).
- *Word count*: this field is 2 bytes long and contains the length of the payload.
- *ECC*: an error correction code for the header (automatically generated by the peripheral).
- *Payload*: the actual data.

- *Checksum*: a 2 bytes long checksum for the payload (generated by the peripheral as well).

As in the LTDC, the most critical part to make the DSI work is correctly configuring the clocks. In this case the DSI has a dedicated PLL which can be clocked by the HSE oscillator or by the result of the R division factor in the main PLL. As usual the HSE oscillator is used. The possible ranges for each intermediate clock, multiplier and divisor are indicated in black color in this figure, while the actual assigned values are in red.

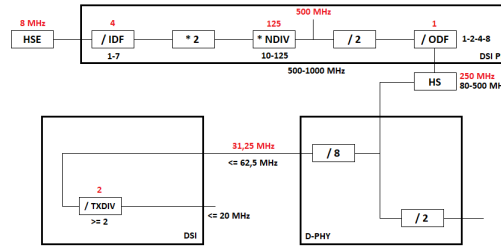


Figure 2: Simplified scheme of DSI clock

Once configured the DSI, the last part to initialize the display is the power-up sequence. This sequence works on all displays based on the OTM8009A and its code can be easily found on the internet. A small change on this code is required to make the display work both in portrait and landscape mode. The commands of the sequence are sent as DSI packets, and this is the reason why it is needed to know how to build them. However it is worth noting another fact: the DSI can send data in *high-speed mode* using the HS clock, or in *low-power mode*. Most displays, when being initialized, can receive data only in low-power mode. For this reason, the DSI is firstly configured to send commands only in low-power mode, then the power-up sequence is sent, and finally the DSI is configured again to send commands in high-speed mode.

As said initially, this driver works in command mode. An update of the display is requested explicitly by setting the bit `DSI_WCR_LTDCEN` inside the `DSI->WCR` register. This operation is executed inside the `update()` method of the library, which is called any time the `DrawingContext` is deallocated, preventing the creation of bad visual effects.

3 Experimental Results

The driver has been tested using the sample benchmark provided inside the library. It works with no problem and these are the performance measured by the benchmark, both in portrait and landscape mode.

| Task | Time spent | Frames per Second |
|---------------------|------------|-------------------|
| Monospace text | 0.046750 | 21.39 |
| Variable width text | 0.038750 | 25.80 |
| Antialiased text | 0.042000 | 23.80 |
| Horizontal lines | 0.012000 | 83.33 |
| Vertical lines | 0.026000 | 38.46 |
| Oblique lines | 0.125500 | 7.96 |
| Screen clear | 0.005500 | 181.81 |
| Draw image | 0.035000 | 28.57 |
| ScanLine | 0.005000 | 200.00 |
| ClippedDraw | 0.066500 | 15.03 |
| Clipped text | 0.042000 | 23.80 |

Table 1: Benchmark in portrait mode

| Task | Time spent | Frames per Second |
|---------------------|------------|-------------------|
| Monospace text | 0.029250 | 34.18 |
| Variable width text | 0.023000 | 43.47 |
| Antialiased text | 0.025000 | 40.00 |
| Horizontal lines | 0.011750 | 85.10 |
| Vertical lines | 0.027250 | 36.69 |
| Oblique lines | 0.125750 | 7.95 |
| Screen clear | 0.005500 | 181.81 |
| Draw image | 0.035250 | 28.36 |
| ScanLine | 0.003000 | 333.33 |
| ClippedDraw | 0.068000 | 14.70 |
| Clipped text | 0.025000 | 40.00 |

Table 2: Benchmark in landscape mode

4 Conclusions

This is the necessary knowledge to understand how the driver works. For more detailed information, you can consult these documents:

- *Application note AN4861* for the LTDC
- *Application note AN4860* for the DSI
- *Reference manual RM0386* for the STM32F469xx and STM32F479xx boards