A WS2812 smart LED driver for the Miosix OS

valerio\new

December 4, 2022

Contents

NOTE: IT IS MANDATORY TO FILL IN ALL THE SECTIONS AND SUBSECTIONS IN THIS TEMPLATE

1 Project data

Please refer to the following repositories for the sources:

- https://github.com/valerionew/WS2812-driver
- https://github.com/valerionew/miosix-kernel

2 Project description

The scope of this project is to create a C++ driver for **Miosix** for the `WS2812` **addressable LEDs**. In particular, the `SPI` peripheral will be exploited to generate the data stream for the LEDs. The driver will be independent of the particular platform or framework. The aim of this project is to achieve a **high FPS** refresh update for the LEDs, and the target is set at 30 FPS on an `STM32F411` microcontroller.

2.1 `WS2812` LEDs

The `WS2812` smart LEDs are designed to be connected in a **daisy-chain**, where each smart LED passes to the next smart LED all the packets received, but the first it receives. The packets are **24 bits** long and are structured as 8-bit green, 8-bit red, 8-bit blue (in GRB order).

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Figure 1: The bit ordering for the `WS2812` LEDs

The bits are encoded with a specific encoding, with a **fixed period** for each bit and variable duty to signal '0' and '1'. The data is shown when each led does not receive an input (D = '0') for a fixed amount of time. This is called RESET time.

The aim of this project is to use the `SPI` peripheral of an `STM32F411` micro-controller to generate the data-stream for the LED. This is done by varying the ratio of '0's and '1's in a sent byte to represent each bit. The easiest case is to use the correspondence 1 bit : 8 bits, notice that this approach requires 24 bytes of memory for each LED. However, the number of bits-per-bit, does not influence the refresh rate, as the timing per symbol is fixed.

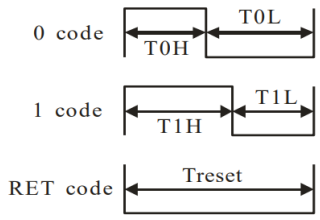

Figure 2: The encoding for the `WS2812` LEDs

2.2 Timing requirements

Timing requirements for this protocol are quite relaxed, and field tests [CPL14] have proven them even more relaxed. Considering the 1 bit : 8 bits correspondence, the total byte time should be at least 650ns and at most 1.85us. This leads to `SPI` speeds between 12.30 Mbit/s and 4.32 Mbit/s, nominally it should be 6.40 Mbit/s. For our implementation we set the `SPI` speed at 9.0 Mbit/s as it is easy to configure the clocks to obtain it, without affecting the system clock. Our symbols of choice are, for '0' `0b11000000`, and for '1' `0b11111100`.

sym	description	time	tolerance
T0H	'0' code, high voltage time	0.4us	±150ns
T1H	'1' code, high voltage time	0.8us	±150ns
T0L	'0' code, low voltage time	0.85us	±150ns
T1L	'1' code, low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	
TH+TL	total period time	1.25us	±600ns

Table 1: Timings for the `WS2812` protocol

2.3 Design and implementation

2.3.1 Class design

- the RGB_t<**typename** T> class, that is a template class that represents the triplet of RGB of an arbitrary type, in our case `uint8_t`;
- the WS2812<**int** LENGTH> class, which is responsible for the driving of the LEDs, the creation of the `SPI` buffer and the call of an user-supplied `SPI` call.
- is an additional header providing already defined RGB_t<uint8_t> colors.

The user has to set the SPI to the correct speed and hand over the SPI function to do the write. The so-composed class is already compatible with the DMA, as the driver is agnostic on how the SPI function works. This also makes the class cross-platform compatible, as long as a spi_transfer(uint8_t* data, uint16_t length) is provided to it.

2.3.2 Thread synchronization

Two threads are implemented on Miosix for this project:

- The main thread, that writes the frame in the frame-buffer;
- The consumer thread, that sets up the DMA controller to read the frame-buffer and output the bytes via the SPI peripheral.

It is clear how there is concurrency between these threads on the frame-buffer. The access to the frame-buffer is regulated with a **thread synchronization** mechanism. The main thread warns the consumer thread that the data is ready using the full **condition variable**. The Thread, Mutex and ConditionVariable classes are native to Miosix and readily available.

This **mutex** and **condition variable** paradigm allows for **thread-safe** access of the shared resources. This approach renders also efficient the execution of the consumer thread, that remains on hold up until when new data is available to write on the peripheral and the full condition variable is signaled. The main thread is put to sleep for $\tau = 1/\text{frame-rate}$.

The **lock** on the mutex is acquired by the thread according to the **RAII** paradigm. This C++ paradigm is very common in this context, where the lock is instantiated in a scope and gets destroyed at the end of it. The advantage of this technique is that it prevents errors such as forgetting to release the lock when exiting the critical section. The lock is also considered released if the execution of the holder is, for any reason, disrupted.

The same technique is used to prevent interrupts from disrupting some operations that shall be **atomic**, such as the setup of the RCC (Reset and Clock Control) unit, that may be accessed by multiple peripherals concurrently. By acquiring a lock on the interrupts with FastInterruptDisableLock Lock, the interrupts are disabled until the object is destroyed.

2.3.3 DMA and interrupt handling

The DMA implementation does also exploit some **interrupts** to warn the software that the hardware DMA has completed the transmission. With this mechanism it is possible to release the lock on the shared buffer only on completion.

The interrupt handling on it's own requires some operating systems mechanisms. After clearing possible previous interrupts and enabling the interrupts for the DMA, the DMA is set, the process is placed in a **wait queue** and the control is yielded back to the **scheduler**. When the interrupt arrives, after reading possible errors and clearing the interrupt flag, the interrupt handler compares the current thread priority with the priority of the consumer. If the waiting thread has higher priority, it is scheduled right away. Otherwise the interrupt handler just returns, and the consumer thread will be resumed when it's time.

Clearing the **interrupt** flag is done by simply writing 1 in the Flag Clear Register of the DMA peripheral, in contrast with the clear by writing 0/1, that is used in registers that can be written both by the hardware and by the software. This is probably done because the FCR register is a special register that triggers an hardware mechanism for clearing the flag in a **safe** manner.

3 Project outcomes

3.1 Concrete outcomes

The main demo for this project is composed by a `NUCLEO-F411` development board with a strip of 60 `WS2812` LEDs. The demo can update the LED animation at more than 300 FPS, far exceeding the required frame-rate. With this amount of spare resources available, it would be fairly easy to stay above the 30 FPS mark for as much as 600 `WS2812` LEDs.

3.2 Learning outcomes

This project was very useful to learn and apply all the techniques and paradigms seen in the course, especially in an embedded context. It also taught the author how to approach a new (large) code base, and how to debug in it. Debugging with GDB and a logic analizer were also skills acquired during this project.

Figure 3: The demo for the project

3.3 Existing knowledge

The Sensor System course was helpful for this project, as it taught the basics of STM32 HAL development, which were very useful in the first stage of the project, where the `WS2812` Class was being tested in a known environment. A spare time read, [Kor21], was also helpful in mastering the `C++` concepts from an embedded point of view.

3.4 Problems encountered

The two biggest problems encountered while developing this project were:

- Short time available: this project was developed due to the need to change from the previous project. The project was therefore developed in an handful of weeks.
- Non-functioning UART on the `STM32F411 Nucleo`: the UART was not functioning on the target board. Some debugging was carried out, the source code for the UART was verified, the hardware was swapped, but the problem was not solved.

References

[CPL14] Blog CPLDCPU. Understanding the ws2812. `https://cpldcpu.wordpress.com/2014/01/14/light_ws2812-library-v2-0-part-i-understanding-the-ws2812/`, January 2014.

[Kor21] C. Kormanyos. *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer Berlin Heidelberg, 2021.