Politenico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

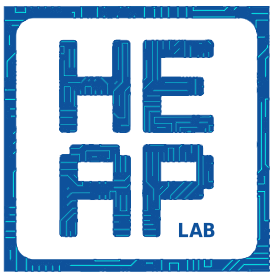
HEAPLab Project Report

mxusb on STM32F4xx

Author:
Luca Morandini
Davide Giacomini

Supervisor:
Dr. Federico Terraneo

September 15, 2021



Abstract

Operating systems for embedded systems are each day more common in our daily life. In this paper we will show our work in order to extend `mxusb`[1], a library used by `Miosix`[2] for handling the USB protocol over embedded systems.

1 Introduction

In this report we are going to illustrate our work. The objective of our project was to use the `Miosix` library module `mxusb` in order to support the USB protocol for the `STM32F407VG`[3] board, which from now on we are going to call `Discovery`.

`Miosix` already supported `Discovery`, but the USB protocol was not supported yet. We took advantage of the `mxusb` library to avoid implementing the support for the board from scratch, and at the same time it sped up the initial process.

As we will see later in this report, `mxusb` was originally thought to support the `STM32F1xx Series`[4], hence the module had not much of a separation between abstract and physical layer. We therefore decided to apply software engineering concepts and split up the logical layers of the implementation.

1.1 Contents

In the first paragraph we will talk about the design and the implementation of our work. Then we will show some results from regression tests and new tests. We used the `STM32F103C8T6`[5] board (which from now on we are going to call `Blue Pill`) in order to check if previous implementations that we modified still work. We will therefore illustrate results, for the same board, before and after the separation of the logical layers, and we will thus provide the proof that our implementation did not change the previous outcomes thought for the other board. We will then show the results of the implementation for the `Discovery` board, explaining what we have finally come down to and what can be done to carry on the project.

[1] https://github.com/fedetft/mxusb
[2] https://github.com/fedetft/miosix-kernel
[3] https://www.st.com/en/evaluation-tools/stm32f4discovery.html
[4] https://www.st.com/en/microcontrollers-microprocessors/stm32f1-series.html
[5] https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html

2 Design and Implementation

In this section we will see how we used two main design patterns common in software engineering to ensure the extendibility of our project.

Then, we will show what implementations we had worked on, concentrating on details when it is necessary for understanding our work.

2.1 Design

Before actually designing the software, we faced the necessity to separate the logical from the physical layer. Indeed, in the previous implementation the code did not have a conceptual separation between the USB protocol and the actual hardware implementation. That issue often brings to code duplication and it is therefore particularly error-prone.

We analyzed the code and identified those sections referring to the actual board implementation and those sections generalizable to a higher-level software interface. In order to build an extendible software, we inserted a layer between the USB protocol and the device implementation. In this way, the code related to the USB protocol, ideally, is portable to every possible new board. At Figure 1 the Hardware Abstraction Layer (`HAL`) is shown. It is basically an interface that separates the USB protocol from the device implementation. This interface, with the addition of some preprocessor defines, allows to select a driver at compile time that is used by the independent high-level code without knowing anything about the underlying hardware version. We created four interfaces and each of them represents a precise task of the USB protocol (more details at Figure 2):

- `USBgpio` is responsible for handling the physical pins used by the peripheral to communicate with the host.
- `USBperipheral` is conceptually an interface which handles all the functions useful to correctly configure the peripheral. It handles the endpoint 0 too, as the ep0 is physically handled differently inside the boards. For example, in the `Discovery` board the registers for the ep0 are completely different from the ones of the other endpoints.
- `EndpointBaseImpl` is the interface dedicated to all the functions related to the endpoints. `EndpointImpl` class, that is device-dependent, inherits from `EndpointBaseImpl`.

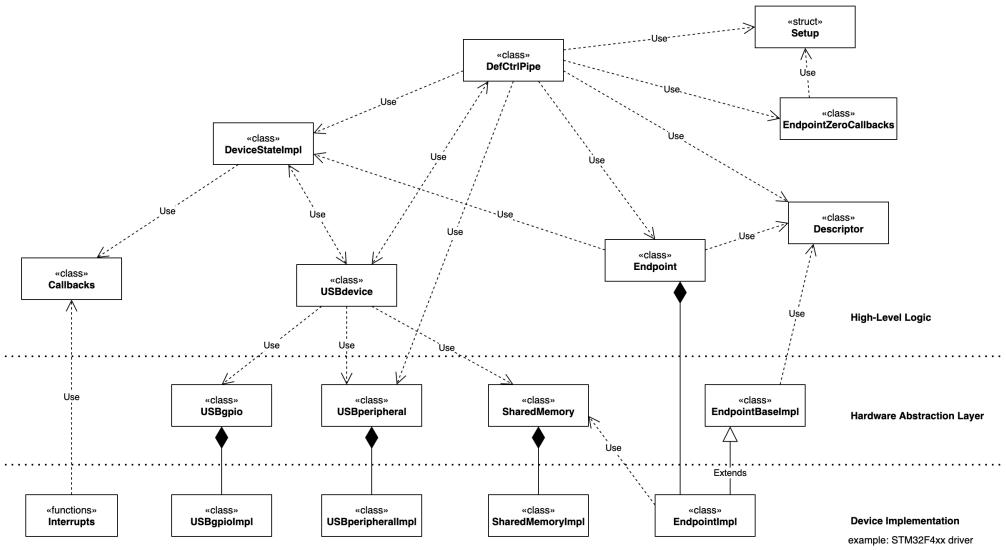


Figure 1: UML of the design with the hardware separated from the software.

- `SharedMemory` is thought to handle the memory shared by the USB peripheral and the application. In our case, it has been tricky generalizing this concept, as the `Blue Pill` and the `Discovery` handle memory in totally different ways.

Moreover, we decided to use two software engineering design pattern which helped us in separating the logical layers.

The `template pattern` has been used for each class and it has been really useful to define the general structure of the algorithm and leave to each subclass the task to implement specific subsets of it.

The `singleton pattern` has been used for the `USBgpio`, `USBperipheral`, `SharedMemory` classes since there is a need to have only one instance inside the entire application but at the same time it is necessary to use inheritance to implement the `template pattern` and for this reason static classes are not a feasible solution.

2.2 Implementation

After the first design, we had to implement code for both the `Blue Pill` and the `Discovery`.

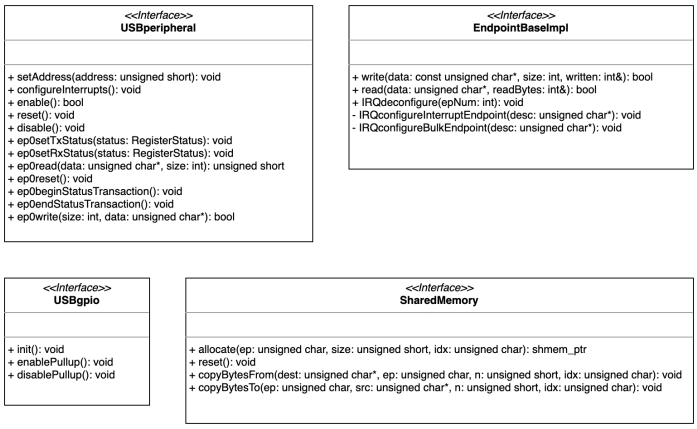


Figure 2: Hardware Abstraction Layer implementation.

The starting point was to identify and isolate the parts of the code related to the low-level device implementation, dependent from the target board, and the high-level code that implements the abstract logic and control flow of the USB protocol independent from the device.

For the `Blue Pill`, we simply moved already existing code to the new device implementation classes, except from `SharedMemory`, which is explained later. For the `Discovery` board, we used the user manual documentation of STM[6][7] and we took inspiration from the old code and other sources we found on internet. The main source has been `libusb_stm32`[8], then we also looked at some code from the official library of STM on GitHub[9].

When we started to develop the implementation of the SharedMemory class for the `Discovery` board driver, we noticed that the previous version of the library `mxusb` was accessing memory directly through pointers to the shared RAM. This was problematic, as the `Discovery` peripheral manages the shared memory through a FIFO queue that doesn't use any pointer at all. For this reason, we have rewritten the `SharedMemory` class to avoid using pointers out of the class, but allowing other classes to access the memory only with the endpoint index. In this way, the endpoint index in the `Discovery` implementation is used to access the corresponding FIFO, meanwhile in the

[6] manual-reference
[7] user-manual
[8] https://github.com/dmitrystu/libusb_stm32
[9] https://github.com/STMicroelectronics/stm32_mw_usb_device

`Blue Pill` driver the index is used to access a dedicated data structure (entirely managed inside the `SharedMemory` class) that maps the endpoint index to the correct pointer in the shared RAM.

Regarding the `USBperipheral`, it is divided into two main sections, implemented separately: the methods referring to the ep0 has been mostly implemented through reverse engineering from `libusb_stm32`, using the reference manual of STM as a starting point and then for double-check. Indeed, the reference manual was not so clear and generally a little vague on the steps to follow. The other methods (referring to configuration and initialization) have been programmed following step by step the reference manual. particularly, the `enable()` method is divided into three methods that represent three specific different stages:

1. `power_on()`: this method handles all the actions to be done regarding clock gates and power on of the device.

2. `core_initialization()`: this method handles the initialization of the device regardless its mode (host or peripheral).

3. `device_initialization()`: it handles the initialization of the device only if in peripheral mode. Note that we did not implement a fourth method as we never used the host mode.

The class `USBgpio` takes care of those pin that let the device communicate with the host and it is a very small class because it just have to configure the low-level GPIOs. In the `Discovery` driver, some methods are empty because on this board the pins are directly handled by the hardware peripheral.

Finally, `EndpointBaseImpl` did not follow exactly the same design patterns of the other classes. We follow the same template concept as explained before, but we did not use the singleton pattern because inside the application there must be one instance of the class for every physical endpoint. Moreover, there are some static methods in the `EndpointImpl` class that, of course, can not be defined in an abstract class and inherited later on. Another reason to motivate this design choice is that in the `EndpointImpl` there is a static array that holds a reference to every instance associated to the physical endpoints. A possible alternative could have been to move the static array and methods to the `Endpoint` class but, since this class is part of the external interface of the library that is used by the users, it exposes a limited number of methods and for this reason it can not be used as an

abstract class for the `EndpointImpl` class. The final solution was to define the `EndpointBaseImpl` class inside the HAL layer that defines a common interface and is inherited by all the `EndpointImpl` classes defined in every low-level device driver. This architecture preserves the same interface for the `Endpoint` class and at the same time allows to reuse most of the code inside the `EndpointImpl` implementation classes defined in each device driver.

3 Experimental Results

Since we modified the previous code to make it extendible, we needed to be sure that no error was introduced. To ensure that, we performed some regression testing, using the test suites defined in the `mxusb` library. The original code was thought to work on the `STM32F1xx Series`, hence we used the `Blue Pill` board to test our modifications.

We ran all the tests and we obtained similar results without any error. We therefore assumed that the new code is semantically equal to the original one. Before showing the results of the test suites, we would like to point out that the second test suite (`testUsbDisconnected()`) did not pass both the original code and our modification. We therefore excluded it as it would be meaningless to our purpose. We think that there is some timing issue, but we were unfortunately unable to fix it.

Following, we show the results of the tests.

```
>>> OLD VERSION <<<
Testing custom requests on endpoint zero... OK
Testing interrupt transfers... OK
 Time required=0.999s
Testing bulk transfers... OK
 Time required=0.158s
Testing bulk out speed... OK
 Time required=1.19s
Average packets per frame=15.9664
Speed=997KB/s
Testing bulk in speed... OK
 Time required=1.312s
Average packets per frame=14.4817
Speed=904KB/s
```

```
Test passed

                >>> NEW VERSION <<<
Testing custom requests on endpoint zero... OK
Testing interrupt transfers... OK
 Time required=0.983s
Testing bulk transfers... OK
 Time required=0.191s
Testing bulk out speed... OK
 Time required=1.214s
 Average packets per frame=15.6507
 Speed=977KB/s
Testing bulk in speed... OK
 Time required=1.454s
 Average packets per frame=13.0674
 Speed=816KB/s
Test passed
```

Regarding the `Discovery` board, to facilitate the debugging and testing phase, we introduced in the Tracer module a new method to allow the logging of arbitrary strings. This logging system uses a queue where all the logs that are pushed will be later processed and executed on a different thread. With this approach, all the log calls are not blocking and can be used also inside interrupts routines. At the beginning we planned to use this logging system only during the debugging phase but at the end we decided to leave it in the final version after some small fixes and improvements.

Unfortunately we could not perform the same test suites as before on the `Discovery`. We are going to explain in detail what happened with our board.

Although we were able to enable the peripheral relatively easy (just following very carefully the indications on the manual documentation), we struggled with the initialization of the ep0. Precisely, the initialization flow crashes after the enumeration interrupt, but it is not so deterministic. Sometimes it begins printing the next instruction, sometimes it doesn't print anything and some other times it goes a little further than the expected.

We initially hypothesized an interrupt issue. In fact, if not correctly cleared, the interrupt flags could create problems and eventually could have the program blocked. Hence we connected with the in-circuit debugger and we printed all the registers we were interested for, in order to see the state of the flags at each step. However, not only we did not see anything strange in

the registers we printed, but we even managed to continue the computation and the flow seemed not blocking anymore.

After some further modifications and tests, we have come back to the original idea that maybe it could be a problem with interrupt flags. In fact, we have been tricked to think that everything worked fine with the in-circuit debugger, because we placed the breakpoint inside the suspicious interrupt and we saw that is was executed correctly. However, after placing the breakpoint at the beginning of the ISR, it began to block also during the debug phase, and in fact we were able to detect an infinite loop in the call of the interrupt routine. Printing the state of the interrupt flag register has been complicated, as in debug mode it results almost non deterministic. More specifically, depending on the speed that we used to step over the instructions, the interrupt flag register was each time in a different state (executing the very same exact sequence). It is our understanding that this behaviour is due to the fact that the host sees the peripheral not responding, and it could, for example, send multiple USB reset signals, getting the register 'dirty'.

Before concluding, we would like to highlight other steps done until now:

- We have updated the `DefCtrlPipe` class because in some cases it did not read every packet received by the USB peripheral. This creates problems in the `Discovery` driver, because every received packet is supposed to be popped from the receiving FIFO, otherwise the application is not able to read any subsequent packet.

- The interrupt that most probably is creating problem is called RXFLVL, and it is triggered every time a packet is correctly pushed by the peripheral to the shared receiving FIFO. Unfortunately the flag of this interrupt is read-only because it is automatically cleared by the hardware peripheral when all the received packets are popped from the FIFO. We deeply analyzed this problem, but it clearly requires a complicated analysis in order to identify why the flag is not cleared from the hardware side.

To conclude the part related to this problem, we want to clarify that this description is just a collection of our hypothesis that we derived from many tests (with and without the in-circuit debugger), from some researches on the Internet and from a lot of discussions and reasoning made together during the last weeks. We therefore cannot exclude any particular problem, even though we hypothesized everything trying to avoid absurd speculations.

4 Conclusions

We illustrated our work and the issues that we encountered, highlighting our line of reasoning and motivating every choice that we made.

It is our believing that designing the code prior implementing it and following a software-engineering approach is fundamental in order to set a baseline that is the least error-prone possible. In this way, we hope that the highest-level structure has almost never to be changed, and that future developers will be able to take advantage of our structure, thus avoiding writing from scratch a new board implementation.

We apologize for delivering a non fully-functioning library. We described everything in details to help whoever is willing to continue this extension and we hope that these descriptions can be useful. In any case, we are happy to be aware that we put our best effort to improve this library.

In conclusion, in this paper we explained in details the design patterns used to implement an extensible library. Furthermore, we described our main issues and reported all the tests conducted.