Bachelor Thesis
Electrical Engineering
November 2014

USB COMMUNICATION ON STM32F405

DONATAS KATEIVA
ERAY DURAN

Department of Applied Signal Processing
Blekinge Institute of Technology
37179 Karlskrona
Sweden

External Advisors

Gerth Fohlin
Baltic Engineering AB

Anders Bengtsson
Baltic Engineering AB

University examiners:

Sven Johansson

University advisors:

Johan Zackrisson

Department of Applied Signal Processing
Blekinge Institute of Technology
371 79 KARLSKRONA SWEDEN

Internet: www.bth.se/tisb
Phone: +46 455 385000
SWEDEN

Declaration

This thesis work was carried out at Blekinge Institute of Technology (BTH), Sweden in collaboration with Baltic Engineering AB. All the external information used for the completion of this thesis work is given as a reference.

Acknowledgements

As authors we are thankful to our supervisor Sven Johansson who permitted us to start working on this thesis topic and made all the necessary administration processes with Baltic Engineering AB and BTH. We would like to express our gratitude to our supervisor Johan Zackrisson for his support and guidance throughout our thesis work. We are grateful that he has been extremely enthusiastic and that he shared all his knowledge and gave his best whenever we needed some help. Last but not least we would like to express our thanks to Gerth Fohlin and Anders Bengtsson from Baltic Engineering AB for giving us the opportunity to work with them.

Abstract

The goal of this thesis work is to test the High Speed USB 2.0 communication between a custom board and a PC. Baltic Engineering AB has developed a hardware platform based on a microcontroller from ST microelectronics. The microcontroller STM32F405 is equipped with many peripheral functions, one of which is a High Speed USB 2.0 OTG unit. Baltic Engineering AB is planning to use this function in future projects but at the moment they have no experience of implementing software code for this type of communication link.

The work is focused on programming and establishing the connection between the PC and the microcontroller. It is explained how to set up the development environment with CooCox CoIDE and how to write programs in C programming language with the help of the libusb library. The outcome of the project was a successful establishment of the USB FS communication. Furthermore, a bulk transfer was benchmarked and reached the bandwidth of approximately 2 Mb/s.

Glossary

USB	Universal Serial Bus
USB FS	USB Full Speed
USB HS	USB High Speed
USB OTG	USB On-The-Go
RISC	Reduced Instruction Set Computing
UTMI	USB 2.0 Transceiver Macrocell Interface
ULPI	UTMI+ Low Pin Interface
PHY	Physical Layer Device or Protocol
PC	Personal Computer
STM (ST)	STMicroelectronics
DCD	Device Core Driver
ID	Identification
IDE	Integrated development environment
WinUSB	A generic USB driver provided by Microsoft
BSD	Berkeley Software Distribution
OS	Operating System
DLL	Dynamic Link Library
GNU	A free software, mass collaboration project
GitHub	Git repository web-based hosting service
GUI	Graphical User Interface
ARM	A family of instruction set architectures for computer processors
GCC	GNU Compiler Collection
Launchpad	A software collaboration platform
FIFO	First In, First Out
Rx	Receive, receiver or reception
Tx	Transmit, transmitter or transmission
VCP	Virtual COM Port
LED	Light-emitting diode
Mbps	Megabits per second

Contents

1 Introduction

The main requirement of this project was to establish a High Speed USB 2.0 communication between the board provided by Baltic Engineering AB and a PC using C programming skills. The work was carried out by researching how the USB communication works and investigating possible implementations on a PC running Windows. Then followed the verification and troubleshooting. In the end, a successful USB FS (USB Full Speed 1.1) communication was established and was tested by performing read and write operations. The writing was verified by sending data to the board and making one of its LEDs (Light Emitting Diode) blink. The reading was checked by creating a sequence of numbers on the board and transferring all the data correctly on a computer. However, the USB HS (USB High Speed 2.0) was not recognized by the PC.

The thesis is divided into 4 parts. The first part of the report is general overview. It involves the description of the USB protocol, its functionality and necessary elements for the establishment of the communication. It also describes the essential hardware components as well as the libraries used in this project.

The second part of the report is focused on explaining how to set up the working environment. It contains information about downloading, installing the software and the necessary drivers, configuring the IDEs and linking the libraries. Useful software is briefly introduced that is helpful for troubleshooting.

The third part is about the requirements for establishing the USB FS communication, reading and writing to and from the board. It covers the programming on both the PC and the board. Only the essential parts of the coding is explained.

The fourth part is about the USB HS communication's requirements, troubleshooting and suggestions.

2 Background

Many electronic devices nowadays support Universal Serial Bus (USB). USB is known as an industry standard for short-distance digital data communication, in our case between a laptop and the electronic circuit provided by Baltic Engineering AB.

The key difference between the USB HS and the USB FS mode is the data transfer speed. USB FS is transferring data at 12 Mbps while the USB HS is transferring at 480 Mbps. USB On-The-Go (OTG) is an extension of the USB 2.0 specification. It is a dual-role device that can act both as a host and as a peripheral (for example mouse, memory stick or custom made hardware).

2.1 Host and device roles

In order for the USB communication to be established there must be a host and a peripheral. A host is usually defined as a computer which controls the interface. It initiates a communication session and the peripheral must wait and respond. Moreover, it has to know each of the devices that are attached to it and their capabilities. The host deals with devices that have different requirements and ensures that all the devices attached have the possibility to send and receive data at the same time. To make sure that data is sent and received without errors, the host adds error-checking bits. The host takes care of a process called enumeration and provides the devices with power. During enumeration the host assigns an address to the device and request additional information from the device to establish the communication between each other.

The peripheral acts as a “slave” in the USB communication. Like the host the peripheral adds error-checking bits when it transmits data and makes error-checking calculation when data is received. If it detects that there is some error in the transmission, it informs the host to retransmit the data.

2.2 Communication and transfer types

Typically USB communication is divided into two types: configuration communications and application communications. During configuration communication or enumeration, the peripheral identifies and responds to control requests from the host. After the enumeration process is done, the application phase follows, where the application specific communication can start.

The USB standard describes four basic types of communication: control, bulk, interrupt, and isochronous.

- Control transfers have two main uses. One is carrying the standard requests that are used to learn about and configure devices. The other is transferring the custom requests defined by a vendor or class. All devices must support control transfers over the endpoint zero. There is a portion of the bus bandwidth reserved for control messages: 10% for low- and full-speed and 20% for high-speed and SuperSpeed buses. This ensures that the control transfers are performed as fast as possible.

- Bulk transfers are used to transfer large and bursty types of data. This method provides the highest throughput and is used when the time is not critical. A typical example for a bulk transfer is sending some data to a printer. The Bulk Transfers are used to reach the maximum speed for our communication. Bulk Transfer is very fast when the bus is idle, however, the data can be delayed when there are other transfers with higher priority. Bulk transfer provides error detection and re-transmission mechanisms.

- Interrupt transfers are requests that need immediate action. Typical examples are keyboards and game controllers. Those types of transfers have limited bandwidth at low and full speed, but high speed enables an interrupt endpoint to transfer almost 400 times more data than full speed.

- Isochronous transfers occur continuously or periodically and are used for data that needs a guaranteed on-time delivery rate. This is the only transfer type with no data re-transmission or error detection, but guarantees constant bit rate transfers.

More information about the transfer types can be found in the book USB Complete [1, Ch. 3].

In this work data integrity is crucial and data needs to be transmitted without errors. Therefore, bulk type transfers were used.

Endpoints. In USB devices the endpoint is used for data transfers. *It is a buffer that typically stores multiple bytes and consists of a block of data memory or a register in the device-controller chip* [1, p.34-35]. Each endpoint has an address (a value between 0 and 15) and a direction (IN or OUT). The direction is determined from the host's perspective, i.e., IN - from device to host, OUT - from host to device. Endpoint 0 is set as a control endpoint. More information about the endpoints can be found on the book USB Complete [1, p.34-35].

Descriptors. Descriptors are data structures that describe the peripheral. There are many descriptor types, but the essential ones for most implementations are (in the hierarchical order): device, configuration, interface, endpoint. String descriptor is optional, but can be useful. It can contain descriptive text to provide more information about the device. Strings are encoded in Unicode and can support different languages. More information about the desriptors can be found in the book [1, Ch. 4] or this website [2].

2.3 Hardware

The microcontroller used in this thesis work is STM32F405 which belongs to STM32xx family manufactured by ST Microelectronics. It is based on the high performance ARM Cortex M4 32-bit RISC core, operating at a frequency up to 168MHz. The STM32F405xx include an USB OTG full-speed device/host/OTG peripheral with integrated transceivers [3, p.12]. For HS communication, an external USB controller is needed connected to ULPI in this case a USB3300 chip. The STM32F405 is used in a custom board that has two physical USB ports. One is connected directly to the microcontroller and supports full-speed communication (12 MB/s). The other port is attached to the USB3300 chip.

According to its datasheet [4], *the USB3300 is an industrial temperature Hi-Speed USB Physical Layer Transceiver (PHY).* It uses ULPI low pin count interface. The USB3300 PHY can work in device, host and OTG modes.

In order to write a program to the STM32F405 microcontroller the ST-Link/V2 is used, which is a debugging and programming interface between the board and the PC [5].

2.4 STM USB Library

STM USB library offers a simplified way to program the USB devices. It takes care of the low-level communication and provides an interface to control the USB transfers. It consists of three layers: USB low-level driver module, USB library module and Application module [6, §6.1]. Most of the coding should be done in the upper layers, since the library provides the configuration files and the callbacks of the functions to the low-lever driver.

Using the library, it is possible to implement classes that conform to the specification of the protocol [6, §6.7]. A custom class (also called vendor-specific class) can be added by using the `USBD_Class_cb_TypeDef` [6, §6.5] structure that provides callbacks for different events such as initialization, de-initialization, setup, data in/out and etc. When these callbacks are called, the transfers can be managed by using Device Core Driver (DCD) layers functions found in `usb_dcd.h` and `usb_dcd.c` files.

2.5 Libusb and libusb-win32

Libusb is a C library that gives applications easy access to USB devices on many different operating systems [7]. Libusb-win32 is an open source libusb project for the Windows operating system [8]. It supports all types of USB transfers: Control, Bulk, Isochronous and Interrupt as well as all standard device requests. The control transfers support sending both standard requests and vendor specific messages.

3 Setting up the working environment

The operating system used in the project is Windows 8.1. The working environment consists of two IDEs: CooCox CoIDE [9] and Code::Blocks [10]. The former has been chosen because it is used by Baltic Engineering AB for the development of their products. It is used for the microcontroller development. The latter has been chosen for coding for the applications to be run on a PC. Code::Blocks uses the MinGW (Minimalist GNU environment for Windows) compiler. It is not compulsory to use Code::Blocks for this application. It is possible to use any other compiler.

3.1 Accessing the USB driver on a PC

Creating applications for a PC to test the USB communication required a way to access the PC's USB driver. On a Windows system two options were considered: WinUSB and libusb-win32. The libusb-win32 was selected because it is based on a cross-platform library that is available on multiple platforms: Linux, Windows, Mac OS X, BSDs and Android. This makes it easier to port to other platforms, if needed. The library (DLL and import lib, examples, installers) is open source and distributed under GNU Lesser General Public License (LGPL) [8].

There are two ways [11] to install the driver: using an INF (.inf) file or Zadig. Zadig is a Windows application that can install generic USB drivers, e.g. WinUSB, lisbusb-win32. It is based on libwdi, a Windows driver installer library for USB devices.

The installation instructions when using INF files are described in the Installation section in libusb-win32 wiki page [8] or libusb wiki. This way is valid for most of the Windows versions before Windows 8. Due to the stricter rules for the installation of unsigned drivers, another method has to be used on Windows 8. An application called Zadig [12] can be used to install the generic USB drivers. To install the driver, download and execute Zadig, select the correct USB device and install libusb-win32 driver. A more thorough usage guide can be found on Zadigs wiki page on GitHub [13].

3.2 Useful tools

USBDeview [14] is a free utility which lists the USB devices that are or had been connected to the PC. It also displays information such as Vendor ID, Product ID, device type, driver description etc. The application is useful for checking if the device is recognized by the computer. Also, it is easy to see whether the descriptors are received correctly.

Testlibusb-win is an executable that is included in the downloadable package from the libusb-win32 website. It displays information about the USB driver, such as the device class, number of configurations, interfaces, endpoints, addresses of the endpoints etc. All of the information can also be obtained by using functions in the libusb library. Nevertheless, testlibusb-win.exe displays everything in a GUI and doesnt require additional programming.

3.3 Installing and configuring CooCox CoIDE

In order to set up CooCox CoIDE environment, three things should be installed:

- CooCox CoIDE
- GNU Tools for ARM Embedded Processors
- ST-Link/V2 driver

CooCox CoIDE can be downloaded from the CooCox's main website [9]. The ST-Link/V2 driver can be obtained from the company's website [15]. The GCC compiler can be downloaded from Launchpad's website [16]. After downloading and installing these tools, it is necessary to configure the GCC compiler and the debugger before starting a project in CooCox CoIDE. There is a page on the CooCox website which has an illustrated step-by-step guide that explains the process [17].

3.4 Using Code::Blocks to link the libusb-win32 library

Download libusb-win32 [18] and extract it. Create a project in Code::Blocks. Then copy the `lusb0_usb.h` and `libusb.a` files to the projects directory. Include the `lusb0_usb.h` (`#include "lusb0_usb.h"`) and add the link library `libusb.a` to the project (go to `Project > Build Options > Linker settings`; then add the .a file). The file can be added both as a relative or non-relative path. In this project the file has been added as a non-relative path, since the other option produced linking errors during the compilation.

4 USB FS

To use USB FS mode it is necessary to select the correct core in `usb_conf.h` file (`#define USE_USB_OTG_FS`) and enable it during the library initialization as stated in the STM USB library document [6, §8]. The snippet of code below is an example of how the USB FS core is initialized. It is executed in the `main()` function at the start of the program.

```
USBD_Init(&USB_OTG_dev,
            USB_OTG_FS_CORE_ID,
            &USR_desc,
            &USBD_MY_cb,
            &USR_cb);
```

Furthermore, it is necessary to select the desired mode: device, host or OTG. It is done in `usb_conf.h` file by defining one of these keywords: `USE_HOST_MODE`, `USE_DEVICE_MODE` or `USE_OTG_MODE`. For example, `#define USE_DEVICE_MODE` selects the device mode.

After that it is necessary to configure the USB descriptors. The device and string descriptors (e.g., product and vendor IDs) can be changed in `usbd_desc.c` file.

In this project two files were created (`custom.c` and `custom.h`) for modifying the configuration, interface and endpoint descriptors. They are based on the STM USB library example files `usbd_cdc_core.h` and `usbd_cdc_core.c`. The size of the descriptors is specified in `custom.h` while the descriptors themselves are written inside `custom.c`.

If desired, the Tx and Rx FIFO size can be changed in `usb_conf.h` file [6, §8]. Finally, the endpoints are initialized during the start of the program by calling the `DCD_EP_Open()` function with the correct parameters. In `custom.c` two endpoints are opened, one for sending data (EP IN) and one for receiving (EP OUT). See the example below.

```
/* Open EP IN */
DCD_EP_Open(pdev,
  CDC_IN_EP,
  CDC_DATA_IN_PACKET_SIZE,
  USB_OTG_EP_BULK);

/* Open EP OUT */
DCD_EP_Open(pdev,
  CDC_OUT_EP,
  CDC_DATA_OUT_PACKET_SIZE,
  USB_OTG_EP_BULK);
```

4.1 First test using a Virtual Communications Port (VCP) example

The first test was done by using a VCP example for the STM32F4xx Discovery Board from one of the pages on GitHub [19]. After compiling and running this code the microcontroller was recognised as a virtual COM port. It was then possible to use a serial console (e.g., PuTTY) with a VCP driver (the example's page has information about it) to connect to the microcontroller and send or receive data. In this project, the example was used to verify if the board was functional. It also served as a basis for the development of the custom USB driver for the microcontroller.

4.2 Writing to the board

For this task a small program is written for a PC based on libusb-win32 library. In the beginning the library is initialized with with `usb_init()`. To find a particular device all buses and devices are enumerated by using `usb_find_busses()` and `usb_find_devices()`. Later the program iterates through each device in each bus. During this process the device descriptors of all connected devices are read. The program checks the values of the Vendor ID and Product ID. If they both match, the `usb_dev_handle` is retrieved, the device is opened with the function `usb_open()`. Later the preferred configuration is selected with `usb_set_configuration()`. Sometimes a peripheral can be programmed to have multiple configurations, e.g., one when it is bus powered and another when it is powered by a different source. Finally, the interface is claimed with `usb_claim_interface()`. After all of these steps the writing operation can be started.

To perform a bulk OUT transfer, the endpoint should be prepared to receive the data. Therefore, `usb_control_msg()` is called. It sends a setup packet with a set of parameters. One of the parameters is `bmRequestType`. It is a number that has a length of 1 byte. The fields of the parameter are specified in the table 1. More information about the setup stage can be found on [1, §5].

Bit	What is specified
7	*Direction*, 0 - host to device, 1 - device to host
6..5	*Type*, 0 - standard, 1 - class, 2 - vendor, 3 - reserved
4..0	*Recipient*, 0 - device, 1 - interface, 2 - endpoint, 3 - other

Table 1: bmRequestType parameter

In this case `bmRequestType` determines these options: type - vendor, direction - device to host. After this packet is sent the USB device prepares the OUT endpoint to receive data. Then `usb_bulk_write()` is called and the host sends 64 bytes of data.

On the board side, the received data is processed in custom.c file `usbd_cdc_DataOut()` function, which is a callback to the devices core layer.

Result: In the test USB device code is modified to process each incoming data packet and check the last byte in the buffer. If the byte equals to `'a'`, the board blinks one of its LEDs. This test was successful and it was possible to send a packet from the PC to the board and blink an LED each time.

4.3 Reading from the board

In order to test the reading from the board, it is necessary to firstly prepare the data that is going to be transferred. This is performed by sending two custom commands to the USB controller. It should be done after initialization and claiming the interface (the same way as described in the beginning of section 4.2). The custom commands are not standard USB requests. Therefore, it is important to implement a way to handle them on both the host and the peripheral. A more detailed explanation of this process will be presented from both host's and device's perspectives.

Host side. In the beginning the host istructs the device to generate a packet of data. This is done by issuing a custom command, which is a control transfer with `bmRequestType` field set to `0xC3` and `bRequest` field set to `0`. Then the host sends another command to tell the device to prepare the data to be sent on the IN endpoint. This time it is a control message which is similar to the first one. The difference is that the `bRequest` field is set to `1`. After these two messages the data is generated and prepared to be read. Therefore, the host can perform the reading operation with the `usb_bulk_read()` function.

Peripheral side. Since vendor-specific requests are used, the device has to be able to interpret them. Hence, an extra rule is added in `USBD_SetupStage()` function (located in `usbd_core.c`). The rule captures the packets with the right `bmRequestType` field and uses a callback to redirect it to `usbd_cdc_Setup()` function (found in `custom.c`). Here the packet's `bRequest` value is checked and the further operations are performed.

There were two configurations used for testing:

- The function was modified to prepare a buffer with limited size. During the test the buffer was filled with a sequence of numbers to be sent. The test was successful and all numbers were received correctly on the PC.
- The function was modified to prepare the buffer to be sent many times (infinitely). This was used for benchmarking. The PC performed the bulk read multiple times. The time of all those operations was calculated as well as the number of bytes read. Finally, the data rate was calculated and the result was around 2 Mbps.

Important note. One of the common uses of USB transfers is to send and receive data that has a certain size. In that case a setup packet should be sent with `wLength` parameter that indicates the number of bytes to transfer during the data phase. After receiving that kind of control message the peripheral should have a mechanism implemented to keep preparing the buffer to be sent out on IN endpoint until there is no more data to be read.

For example, the maximum size of the data packet in USB FS is 64 bytes. If the host has to read 640 bytes of data, it should firstly send a control message requesting 640 bytes of data. The peripheral should prepare the first 64 bytes to be sent on the IN endpoint. When the host initiates the bulk read, the transfer occurs and the peripheral prepares the next 64 bytes. Both continue doing this until there is no more data to exchange. This way, the host does not need to send a control message before each bulk read. Therefore, the transfer of data is faster. See table 2 as an illustration of this example.

Host	Peripheral
Control (read 640 bytes)	Prepare first 64 bytes
Bulk read (64 bytes)	Prepare next 64 bytes
Bulk read (64 bytes)	Prepare next 64 bytes
...	...
...	Prepare last 64 bytes
Bulk read (64 bytes)	

Table 2: An example of host reading 640 bytes of data

5 USB HS

To enable the HS mode, it is necessary to select the correct core and initialize it similarly to the FS part (see section 4). It is also required to select the mode of operation: device, host or OTG. The physical port of the USB HS is not directly connected to the microcontroller on the board. Instead, it uses a ULPI interface and is connected to USB3000 chip. Before the initialization, it is therefore required to select the correct interface for the HS communication. In this case, ULPI. Furthermore, the desired mode is selected, since USB OTG HS core supports both FS and HS modes. Finally, the pin assignment has to be changed in `usb_bsp.c` to match the microcontrollers connections to the USB3300 chip. To test the setup with these changes, the code was compiled and loaded to the board. The USB connection between the PC and the board was not recognized. There was no response.

5.1 Troubleshooting and future suggestions

Low Power Mode. Some forums about the HS implementation on STM32F4 showed that other people with similar problems were having issues with LPM (low power mode) [20]. To test this suggestion LPM was explicitly disabled in `usb_core.c` by writing to the register GUSBCFG bit 15 (PHYLPCS) [21, p1389-1391]. Still, the device was not detected on host OS.

Checking the clock. One of the troubleshooting ideas was to check the hardware, clocks and supplies. After using the oscilloscope it was possible to verify the functioning of the USB FS clock, but not the USB HS clock.

Suggestions for future troubleshooting. According to one user in a post on the ST forum *the crystal won't oscillate if the chip is held on reset* [20]. One possibility could be to use the oscilloscope to check the status of the reset pin as well as the X3 clock. If the clock is not oscillating and the reset is on, try to find a way to turn it off.

6 Conclusion

This work shows you how to set up a development environment on a Windows PC for programming the STM32F4 microcontroller and using its USB functionality. It explains how to write programs in C language to transfer data between the PC and the microcontroller with the help of libusb library.

In the end, a successful USB FS (1.1) communication was established. It was possible to send and receive the data from the microcontroller. The communication was benchmarked and reached the throughput of approximately 2 Mb/s. The establishment of USB HS (2.0) was not successful. Nevertheless, the requirements to enable it are explained in the document as well as the ideas for further troubleshooting.

7 Appendix

Explanation of the USB HS

The first function in `main()` is `SystemInit()`. Later follows the `ledinit()` function from the `myledcontrol.h` file which has been created to initialize and give an easy control of the LEDs. Then `init()` function initializes SysTick and USB.

`USBD_Init()` calls the function `USB_OTG_BSP_Init()` that initializes the board-specific configurations from `usb_bsp.c`, takes care of the user callbacks and the specified core (in this case HS). One of the callbacks is structure `USBD_Class_cb_TypeDef`. This structure gives access to the class driver and is used inside the `custom.c` to create the functions that are required to control the USB device, e.g., initialization, setup and data in/out stages, sending the configuration descriptor.

The board-specific initialization is handled by `USB_OTG_BSP_Init()` function. In this configuration low power mode is disabled, USB HS core is selected and ULPI mode is chosen. The pin assignment is adjusted to match the custom board connections with the USB3300 chip. In the beginning of the selected configuration AHB1 clock is enabled for the ULPI pins. Then, each pin has a function mapped to it according to the schematic. Later the parameters of each pin are changed through `GPIO_InitStructure` and initialized. Furthermore, `RCC_AHB1PeriphClockCmd()` function enables the AHB1 clocks for HS and ULPI. Finally, the PWR clock is enabled.

References

[1] Jan Axelson, *USB Complete Fourth Edition : The Developer's Guide*. Lakeview Research, 4th edition, 2009.

[2] *USB Descriptors*. [2014-10-03] `http://www.beyondlogic.org/usbnutshell/usb5.shtml`

[3] *STM32F405xx datasheet*. DocID022152 Rev 4, June 2013. [2014-12-04] `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf`

[4] *USB3300: Hi-Speed USB Host, Device or OTG PHY with ULPI Low Pin Interface*. Rev. 1.1 (01-24-13). Microchip. [2014-12-04] `http://ww1.microchip.com/downloads/en/DeviceDoc/3300.pdf`

[5] *ST-LINK/V2 datasheet*. Doc ID 018751 Rev 3, September 2012. [2014-12-04] `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00027105.pdf`

[6] STMicroelectronics, *UM1021 User manual*. Doc ID 18153 Rev 3. [2014-12-04] `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/CD00289278.pdf`

[7] *Libusb.org wiki*. [2014-10-15] `http://www.libusb.org/wiki/WikiStart`

[8] *Libusb-win32 wiki page* [2014-10-15] `http://sourceforge.net/p/libusb-win32/wiki/Home/`

[9] *CooCox CoIDE* [2014-10-15] `http://www.coocox.org/CooCox_CoIDE.htm`

[10] *Code::Blocks* [2014-10-15] `http://www.codeblocks.org/`

[11] *Libusb.org: Driver Installation* [2014-10-15] `http://www.libusb.org/wiki/windows_backend#DriverInstallation`

[12] *Zadig main website* [2014-10-15] `http://zadig.akeo.ie/`

[13] *Zadig wiki on GitHub* [2014-10-15] `https://github.com/pbatard/libwdi/wiki/Zadig`

[14] *Nirsoft USBDeview* [2014-10-15] `http://www.nirsoft.net/utils/usb_devices_view.html`

[15] *ST-LINK/V2* [2014-10-16] `http://www.st.com/web/catalog/tools/FM146/CL1984/SC724/SS1677/PF251168?sc=internet/evalboard/product/251168.jsp`

[16] *GCC Tools for ARM Embedded Processors* [2014-10-16] `https://launchpad.net/gcc-arm-embedded/+download`

[17] *CooCox CoIDE Compiler Settings* [2014-10-16] http://www.coocox.org/CoIDE/Compiler_Settings.html

[18] *Libusb-win32 on SourceForge* [2014-10-16] http://sourceforge.net/projects/libusb-win32/

[19] *STM32 Discovery VCP* [2014-10-16] https://github.com/xenovacivus/STM32DiscoveryVCP

[20] *ST forum: STM32F2/F4 problems with various ULPI USB PHYs* [2014-10-16] https://my.st.com/public/STe2ecommunities/mcu/Lists/cortex_mx_stm32/Flat.aspx?RootFolder=%2Fpublic%2FSTe2ecommunities%2Fmcu%2FLists%2Fcortex_mx_stm32%2FSTM32F2F4%20problems%20with%20various%20ULPI%20USB%20PHYs&currentviews=2865

[21] *RM0090 Reference manual*. STMicroelectronics, Doc ID 018909 Rev 6, February 2014 [2014-12-04] http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf