

POLITECNICO DI TORINO

Master's Degree in Electronic Engineering



Master's Degree Thesis

Development and characterization of a USB communication between two microcontrollers general purpose STM32 to analyze the digital IP in order to improve its performance

Supervisors

Prof. Danilo DEMARCHI

Ing. Giuseppe GUARNACCIA

Ing. Giovanni PANGALLO

Candidate

Alessio SIPALA

December 2022

*“Credi in te stesso e in tutto ciò che sei.
Sappi che c’è qualcosa dentro di te
che è più grande di qualsiasi ostacolo.”*

Abstract

The objective that is proposed to achieve with this experience is to evaluate the characteristics, the performances and the limits of a USB communication between two evaluation boards belonging to the *STM32G0* family via a USB *type-C* cable, highlighting anomalies and criticisms.

The two evaluation boards have a USB *type-C* connector on board and they support the USB 2.0 version in full speed mode, i.e. they are able to exchange data between them at a speed of 12 Mbps. The activity was divided into several phases. The first one in which the knowledge on controllers and the USB protocol were acquired, the other one in which the skills to use ST's software and boards were acquired. The last phase, the experimental one, is the one in which a criticism of the USB was stimulated in order to improve its digital design or the software usage.

More specifically, at first a code was implemented that would allow to verify the correct communication between the host board and the device board. Subsequently, to put the digital IP under stress, the amount of data traffic exchanged between the two boards was increased and the used bandwidth has been evaluated. In this way it is possible to compare the theoretical limit of the data sent between two start of frames and the real one.

This analysis was possible also thanks to a "*Teledyne Lecroy Mercury T2C*" protocol analyzer, that, located in series between the two boards, is able to spy on the USB data traffic that is exchanged and allows to evaluate, in addition to the used bandwidth, also other important parameters such as the transfer type, the various packets and their sizes and the possible errors that may occur.

Table of Contents

List of Figures	VII
Acronyms	XI
1 Introduction	1
1.1 Microcontrollers and STM32	1
1.2 UART protocol	3
1.3 SPI protocol	4
1.4 I2C protocol	7
1.5 USB protocol	7
1.5.1 USB version history	8
2 USB protocol	11
2.1 Introduction	11
2.2 Architecture	12
2.3 Electrical and line states	13
2.4 Protocol and transfer types	17
3 USB2 IP by STMicroelectronics	24
3.1 Description of USB blocks	26
3.2 Usage and structure of packet buffers	30
3.2.1 Double buffer in HOST/DEVICE mode	32
3.3 USB registers	34
4 Used resources for the experience	37
4.1 STM32 Evaluation Board	37
4.1.1 Clock Recovery System	40
4.2 Protocol Analyzer	42
4.3 STM32CubeIDE	45

5	Software and hardware implementation	48
5.1	Hardware configuration	48
5.2	Software configurations	49
6	Performance test, results and conclusions	58
6.1	Evaluation of the maximum bandwidth	58
6.2	First test	60
6.3	Second test	64
6.4	Third test	65
	Bibliography	77

List of Figures

1.1	STM32 product line [1]	2
1.2	UART protocol [2]	4
1.3	SPI protocol [3]	5
1.4	SPI protocol - example with CPHA=1	6
1.5	SPI protocol - example with CPHA=0	6
1.6	I2C protocol [5]	7
1.7	USB Type-A and Type-B connectors [6]	8
1.8	USB Micro and Mini connectors	9
1.9	USB Type-C connector	10
1.10	USB Type-C pinout [7]	10
2.1	USB different speeds and respective applications [8]	11
2.2	USB star-topology architecture [9]	12
2.3	USB coaxial cable [10]	13
2.4	USB speed detection based on 1.5 k Ω resistance [11]	14
2.5	state diagram configuration [12]	16
2.6	Non-return-to-zero-inverted encoding [13]	17
2.7	NRZI with bit stuffing [14]	17
2.8	Single packet format [15]	18
2.9	different packet identifier types [16]	19
2.10	token packet format [17]	20
2.11	data packet format	20
2.12	handshake packet format	21
2.13	SOF packet format	21
2.14	OUT and IN types transactions	22
2.15	Example of control transfer made up of 3 different stages	23
3.1	USB peripheral block diagram [18]	25
3.2	Upstream and downstream transceivers [19]	27
3.3	Low-/full-speed Signaling Levels [20]	28
3.4	Packet buffer with examples of buffer description table locations [21]	30

3.5	STATRX status table [22]	33
3.6	Double-buffering buffer flag definition [23]	33
3.7	Control register structure [24]	34
3.8	Interrupt status register structure	36
4.1	STM32G0C1E-EV evaluation board [25]	37
4.2	STM32G0C1E-EV motherboard and daughterboards schematic [26]	38
4.3	STM32G0C1E-EV motherboard layout [27]	38
4.4	CRS block diagram [28]	40
4.5	CRS counter behavior [29]	41
4.6	Instrumentation setup	42
4.7	user interface view at packet level	44
4.8	user interface view at transfer level	45
4.9	HOST mode configuration from ST32CubeIDE side	46
4.10	HOST mode configuration from ST32CubeIDE side	46
4.11	Clock configuration from ST32CubeIDE side	47
5.1	JP3 and JP4 configuration [30]	48
5.2	Corrupted data view	49
5.3	Device library structure [31]	50
5.4	Device handler structure	51
5.5	Host library structure [32]	52
5.6	Host handler structure	52
5.7	Log display after turning on the boards	54
5.8	Host state machine [33]	55
5.9	Transmit function that has been implemented	56
5.10	Receive function that has been implemented	57
6.1	Full-speed BULK transaction limits	59
6.2	Maximum number of bits for the forbidden window	59
6.3	Log on host and device display for the first test	60
6.4	Transaction and packet level view for the first test	61
6.5	Data view of the first transaction	61
6.6	Waveforms view for Packet 73 (first DATA1)	62
6.7	Bandwidth evaluation of the first test	63
6.8	Presence of multiple NAK	64
6.9	Results and bandwidth evaluation of second test	65
6.10	Double buffer bit enable	66
6.11	Double buffer	66
6.12	Debug session: EPKIND bit is not enabled	67
6.13	Process Transmission function	68
6.14	Function used to start the transfer using double buffer feature	68

6.15	Debug session: EPKIND bit is enabled	69
6.16	Case statement within process transmission function	70
6.17	Initialize parameters before transfer	71
6.18	Bandwidth utilization percentage using double buffer	71
6.19	Presence of NAK handshake that reduces the bandwidth percentage	72
6.20	Bulk double-buffering memory buffers usage (Device mode)	73
6.23	Achieved results with the various tests	74
6.21	<i>HAL_PCD_EP_DB_Receive</i> function	75
6.22	Log and bandwidth usage with correct double buffer	76

Acronyms

ACK

Acknowledge

ADC

Analog to digital converter

API

Application Program Interface

BCD

Battery Charging Detection

BSP

Board Support Package

CRC

Cyclic Redundancy Code

CRS

Clock Recovery System

DRP

Dual-Role Port

DAC

Digital to analog converter

EOF

End of frame

EOP

End of packet

FS

Full speed

HFS

Host Frame Scheduler

HS

High speed

IP

Intellectual property

LS

Low speed

NAK

Not acknowledge

PD

Power Delivery

PHY

Physical Interface

PID

Packet identifier

SE0

Single-ended zero

SE1

Single-ended one

SIE

Serial Interface Engine

SOF

Start of frame

Chapter 1

Introduction

1.1 Microcontrollers and STM32

The *microcontroller* (MCU) is a small-sized programmable electronic device that contains all the peripherals necessary for its operation. Generally it integrates one or more data/instruction memories to save results, temporary variables and instructions to be executed, a *central processing unit* (CPU), several PINs and I/O ports to communicate with the outside world and some additional optional peripherals such as converters DAC/ADC and timers. All the integrated peripherals ensure that the microcontroller is able to manage in complete autonomy the operation of the device where it is located, such as computers, smartphones, household appliances and in general in all systems where electronic control is required.

STMicroelectronics was the first company to effectively introduce a general-purpose microcontroller based on an *ARM* Cortex processor, giving life to a family of microcontrollers that takes the name of *STM32*. The integration with ARM processors has led to an important expansion of the company's product portfolio, providing different solutions to the customer based on the characteristics of the product and the fields of use. The product lines are divided into:

- *High performance*: microcontrollers suitable for a context where high performance is required for greater integration and connectivity;
- *Mainstream*: ideal for general-purpose systems where the costs of the final product are limited;
- *Ultra-low-power*: microcontrollers with an excellent trade-off between performance, power and costs, where the main goal is energy saving;

- *Wireless*: used for applications requiring wireless connectivity up to a maximum operating frequency of 2.4 GHz.

For each type of application there are different series of STM32 that offer different characteristics. The following figure locates the different microcontrollers according to the four macro categories listed above.

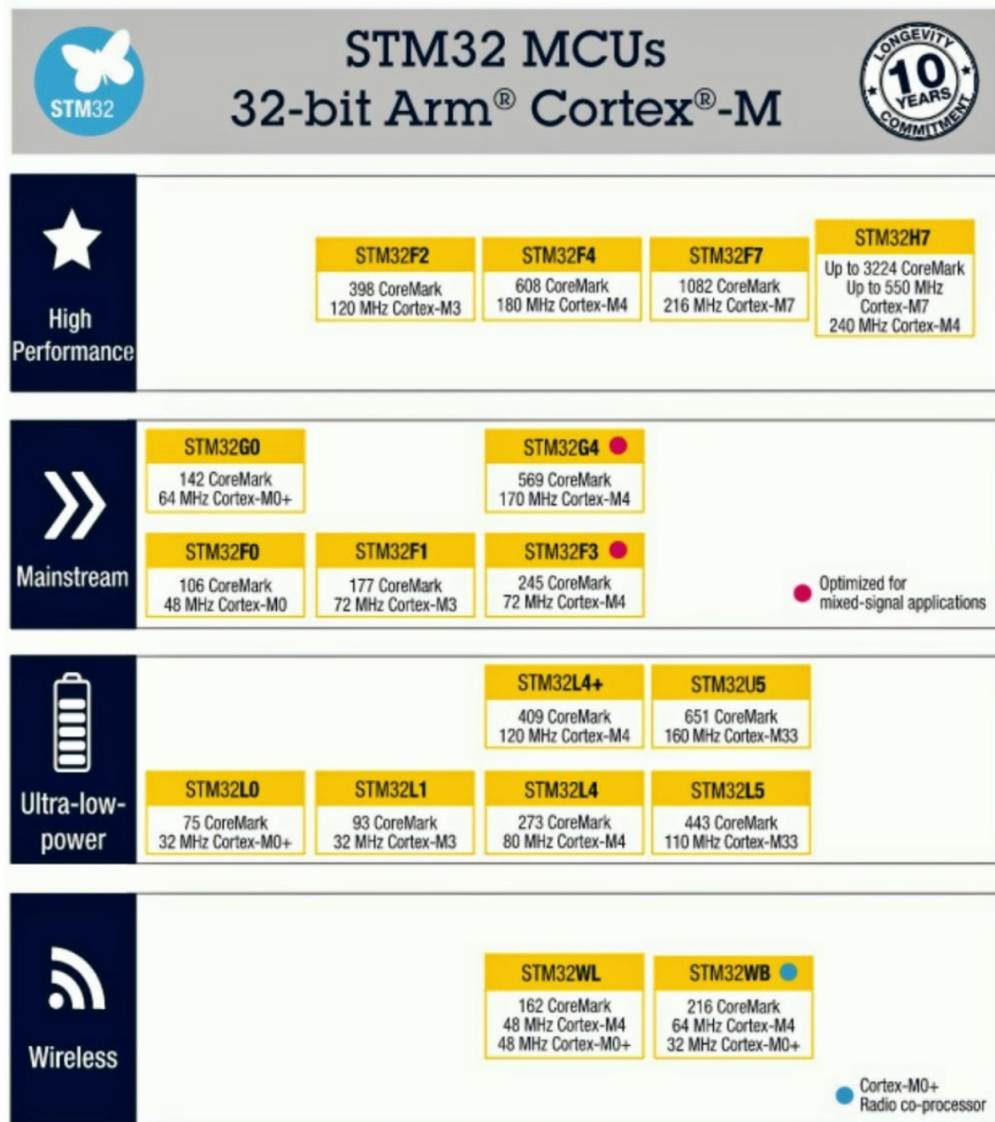


Figure 1.1: STM32 product line [1]

Each microcontroller has one or more peripherals in order to communicate with other microcontrollers or with other external components, in order to exchange data and control or status signals. In general, the communication between two electronic systems can be *parallel* or *serial*. The first one involves the exchange of bits in a parallel manner, therefore it will need a number of data buses equal, at least, to the number of bits to be transmitted; this implies on the one hand higher performance in terms of data transfer speed, on the other a high cost due to the greater resources used. The second communication mode provides, unlike the first, that the bits are sent in serial mode one after the other, making the data transmission speed lower but at the same time substantially reducing production costs, as you will need only one data *BUS*.

Furthermore, at very high frequencies, the cross-talk phenomenon is considerably reduced, increasing the robustness of the system towards disturbances coming from other parts of the circuit. In parallel communication, in fact, due to the greater number of conductors, mutual capacities and inductances are created, which cause interference that modify the voltage levels present on the conductors themselves, making the system much more fragile. This problem is solved by trying to reduce the capacities and parasitic inductances as much as possible, increasing the strength of the drivers that drive the buses, or trying to reduce as much as possible the number of conductors that can disturb each other, as in the case of serial transmission. Within a microcontroller we can find one or more communication protocols, which differ in transmission speed, complexity and timing of the clock signal between the transmitting part and the receiving part. The most widely used serial transmission protocols within an electronic system are briefly described in the following pages.

1.2 UART protocol

The *UART* (Universal Asynchronous Receiver Transmitter) protocol was one of the first protocols to be used due to its simplicity of implementation and low cost; for this reason it has been widely used and is still used in some applications, although in most cases more efficient protocols are preferred.

As suggested by the name itself, this protocol is *asynchronous*: there is no timing signal in common between the transmitter and the receiver. However, to ensure that the data exchange takes place correctly, it is necessary that both parties have the same "*baud rate*", defined as the number of transitions that occur on the data BUS per second. To facilitate transmission and substantially reduce potential errors, the entire string of bits to be transmitted is fragmented into smaller strings

(normally 8-bit strings are used; however the protocol is quite flexible, so different packets size can be used) which is called "*frame*". The first bit that is transmitted takes the name of "*start bit*", that is a transition on the BUS from the high logic level to the low logic level; this is necessary to notify the receiver of the arrival of the data. After transmitting all the frames, the communication is terminated by a "*stop bit*", the opposite transition with respect to the start bit which restores the line to the idle condition. Optionally, a "*parity bit*" can be added, located between the last data bit and the stop bit, and the purpose of this bit is to detect any transmission errors.

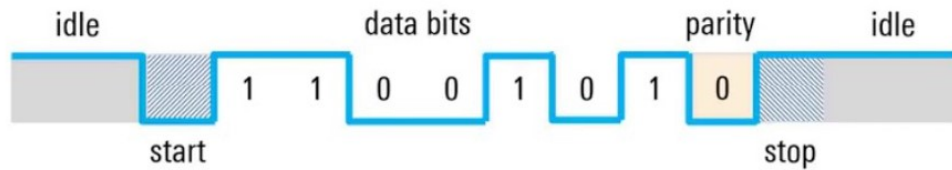


Figure 1.2: UART protocol [2]

The Figure 1.2 shows an example of transmission via the UART protocol. It is possible to notice the presence of the start and stop bits which respectively start and end the transmission, plus an additional bit for error detection.

In conclusion, it can be said that the UART protocol is very simple to use both from the software side and from the hardware side. However, the communication is of the "point-to-point" type, ie reserved only for two devices. Furthermore, the low transmission speed makes the UART protocol less preferable than other more performing protocols.

1.3 SPI protocol

The *SPI* (Serial Peripheral Interface) protocol was introduced by *Motorola* and is widely used in the world of microcontrollers. Unlike the UART protocol, it is a *synchronous* communication standard, thanks to the presence of a line used for the timing signal; this allows data to be transmitted at higher speed with respect to any asynchronous protocol, and this reduce the probability of error.

It is also a master-slave communication, so the master device has complete control: it decides the exact moment in which to start/end a data transfer and it decide the slave device with which to interact.

In general, there are more slaves that are connected to the same master, so we need more lines than the UART protocol:

- *SCLK*: the shared timing signal between master and slave;
- *MOSI* (master output slave input): the signal that indicates the start of a data transmission from the master to the slave;
- *MISO* (master input slave output): the signal that indicates the start of a transmission from the slave to the master;
- *SS* (slave select): the signal that enables the slave with which the master must interact.

The number of lines must be at least 4; however, if there are more than one slave, we will need as many additional selection lines as there are slaves.

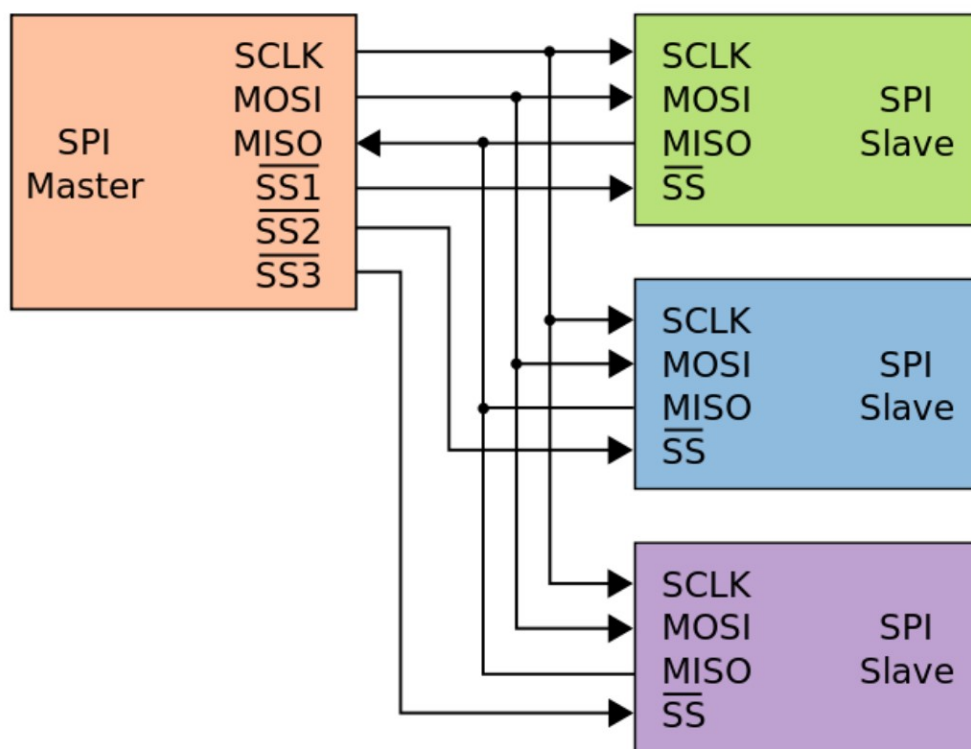


Figure 1.3: SPI protocol [3]

Greater flexibility is guaranteed by the presence of two additional signals, which are normally called "*CPOL*" and "*CPHA*". The first, which stands for "clock polarity", adjusts the polarity of the timing signal: if *CPOL* is at the high logic level, then the clock will be in the idle condition when it gets low logic level; vice versa if *CPOL* is at the low logic level, the clock will be in idle condition when it is at the high logic level. The second signal stands for "clock phase" and discriminates the active sampling edge: if *CPHA* is at the high logic level, then the data is sampled on the rising edge, otherwise on the falling edge. Figure 1.4 and Figure 1.5 show the waveforms of the signals of a transmission via SPI protocol [4] based on the value assumed by the *CPHA* phase bit.

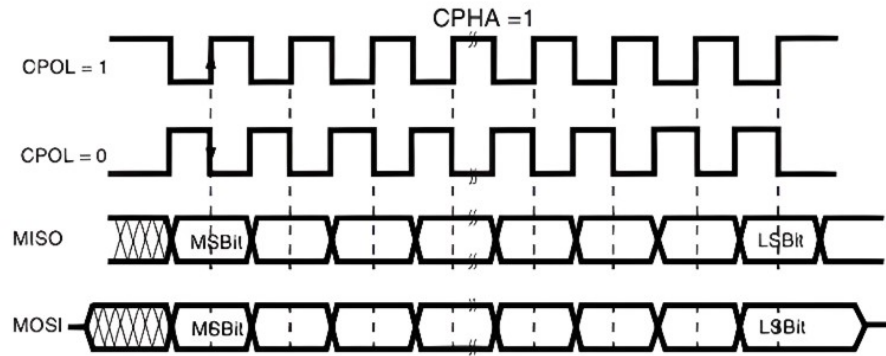


Figure 1.4: SPI protocol - example with $CPHA=1$

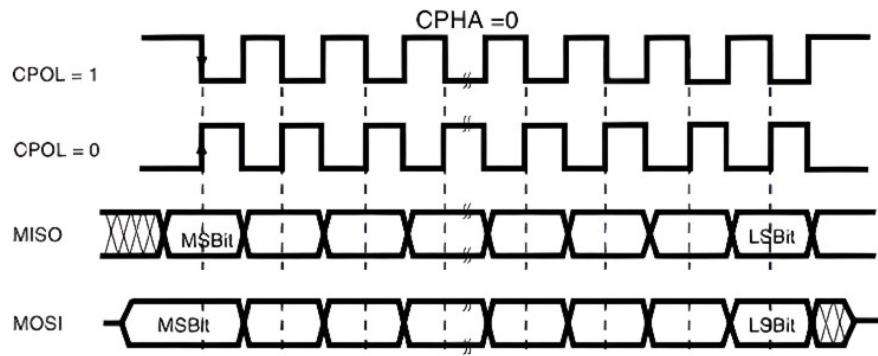


Figure 1.5: SPI protocol - example with $CPHA=0$

1.4 I2C protocol

The *I2C* (Inter Integrated Circuit) protocol is a type of serial communication very similar to the SPI. The substantial difference is that now only two lines are used, one for the timing signal which is called *SCL* (Serial Clock), the other for data exchange *SDA* (Serial Data). To identify the slave with which to interact, a selection line is not used, but a first frame is sent that indicates the unique address of the slave in question. So there will be a start bit that is used to "wake up" the BUS from the idle state. It is recognized by the receiver as it is brought to the low logic value while the timing signal is in idle state. There are then 7 successive address bits, an R/W bit needed by the slave to understand if it is a read or write operation, and finally an ACK bit which is needed by the master to understand if the slave is available for communication. In the following frames there will be the data exchange between the transmitter and the receiver, which ends with a stop bit, i.e. bringing the data line to the high logic level after having brought the timing signal to the high logic level.

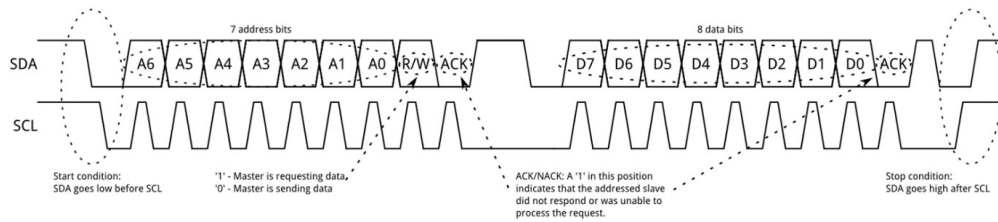


Figure 1.6: I2C protocol [5]

1.5 USB protocol

One of the most used communication standards, especially to connect external peripherals to the computer, is *USB* (Universal Serial Bus).

Introduced for the first time in the market in 1996 thanks to a collaboration of 7 companies (*Compaq*, *Hewlett-Packard*, *IBM*, *Microsoft*, *NEC* and *Nortel*), it was soon adopted in the world and integrated in most of the six electronic devices. The reason why the USB standard has become so popular and is still evolving after more than 25 years, despite the technological evolution, is to be found in several aspects:

- **Versatility:** The same connector is used to connect different peripherals, instead of using different connectors for each peripheral. This greatly reduces costs and complexity, as well as creating an universal bus for each device.

- **Simplicity:** just connect the USB cable on both sides to automatically start the recognition of the inserted device and the installation of the drivers. In this way, all the initial configuration part is hidden from the end user, increasing ease of use.
- **Robustness:** the USB protocol stands out from the others for its robustness and rigidity, which make it one of the safest among those in existence. Plug and Play: allows you to connect/disconnect a device connected via USB cable simply by disconnecting it, without damaging the computer or the device itself.

1.5.1 USB version history

The first version to be introduced was the one that takes the name of "USB 1.0" and was used for *HID* (human interface device) as mouse and keyboards. In fact, its maximum data transfer rate of about 1.5 Mbps, defined as low-speed, makes it impossible to use this version for applications that require higher performance, such as video or audio transfer, but it turned out to be ideal for where performance is not of primary importance. The biggest problem with this version is that the cable length can reach a maximum of 3 meters. The connectors used for this version are called "*Type-A*" and "*Type-B*": Two years later, in 1998, the USB 1.1 version was



Figure 1.7: USB Type-A and Type-B connectors [6]

introduced with the aim of increasing the performance and solving some problems of the previous version. More specifically, the data transfer speed was increased up to 12 Mbps (full speed) and the maximum cable length reached 5 meters.

The USB 2.0 standard was introduced in the first half of 2000. Within just 2 years from the previous version, there were significant improvements in data

transmission speed, which went up to 480 Mbps (high speed). This made it possible to adopt the USB standard for applications that required higher performance. An important novelty was that of battery charging: the device connected to the PC via USB cable is not powered by an external battery, but is powered by the cable itself, which in addition to carrying information, is able to deliver up to a maximum of 100 mA. A winning idea was to make the USB 2.0 standard completely backward compatible with previous versions. However, using a USB 2.0 cable to connect devices that can only support previous standards, the transmission speed must be adapted and limited to the maximum speed of the devices themselves. The “Type-A” and “Type-B” connectors remained, and four more were introduced, which took the name of “*Mini*” and “*Micro*”. Thanks to the small connectors, they were used to connect devices with reduced thickness such as tablets, cameras or satellite navigators.



Figure 1.8: USB Micro and Mini connectors

In 2008, the USB 3.0 version was introduced, which offers the customer even greater performance (a speed of 4.8 Gbps can be achieved) for most applications. In order to achieve this speed, it was necessary to slightly modify the structure of the connector by adding 5 new pins to support high-speed optical connections. However, in order to maintain backward compatibility with versions 1.0 and 2.0, the position and size of the pins must be such that they do not come into contact with the conductors of the previous version.

The version currently most used on the market is the one that takes the name of USB 3.1 (*SuperSpeed+*), whose specifications were announced starting from 2013. The "*Type-C*" connector was introduced for the first time, now much more small and performing compared to its predecessor.



Figure 1.9: USB Type-C connector

In addition to the addition of some pins to support power delivery (in addition to the exchange of information, at the same time the exchange of power is now also allowed to power devices such as monitors and computers), the connector was made reversible, i.e. the ability to be inserted in any direction.

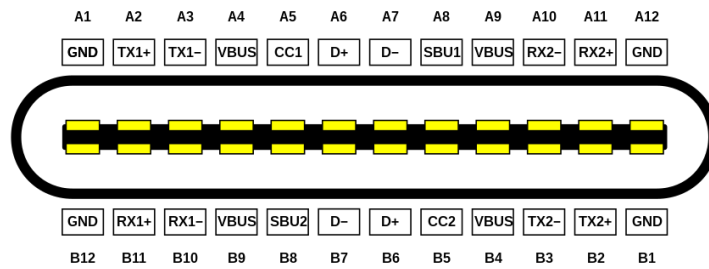


Figure 1.10: USB Type-C pinout [7]

Backward compatibility with previous versions was maintained, which had to be done via adapters or docking stations. The compactness of the connector, versatility and performance have made Type-C the most suitable adapter, at the moment, for latest generation devices where the main goal is to lighten and save space.

Chapter 2

USB protocol

2.1 Introduction

The USB 2.0 protocol provides three different data transfer speeds depending on the application being used. We can identify them in ascending order respectively with: low-speed, full-speed, high-speed.

<u>PERFORMANCE</u>	<u>APPLICATIONS</u>	<u>ATTRIBUTES</u>
LOW-SPEED <ul style="list-style-type: none">• Interactive Devices• 10 – 100 kb/s	Keyboard, Mouse Stylus Game Peripherals Virtual Reality Peripherals	Lowest Cost Ease-of-Use Dynamic Attach-Detach Multiple Peripherals
FULL-SPEED <ul style="list-style-type: none">• Phone, Audio, Compressed Video• 500 kb/s – 10 Mb/s	POTS Broadband Audio Microphone	Lower Cost Ease-of-Use Dynamic Attach-Detach Multiple Peripherals Guaranteed Bandwidth Guaranteed Latency
HIGH-SPEED <ul style="list-style-type: none">• Video, Storage• 25 – 400 Mb/s	Video Storage Imaging Broadband	Low Cost Ease-of-Use Dynamic Attach-Detach Multiple Peripherals Guaranteed Bandwidth Guaranteed Latency High Bandwidth

Figure 2.1: USB different speeds and respective applications [8]

The LS is able to reach a theoretical speed of about 1.5 Mbps, so it is reasonable for applications that do not require particular performance, such as the use of mice, keyboards or interactive devices in general. The FS was initially designed for all other devices that were not HID and reached a data transfer speed of about 12 Mbps. With the technological evolution, the HS was introduced in conjunction with the USB 2.0 protocol, which allowed the use of more complex and performing applications, such as video transfer and storage. It should be noted that all speeds maintain the fundamental characteristics that distinguish the USB protocol from the other existing ones: ease of use, ability to manage multiple peripherals, automatic and secure connection / disconnection between the two devices. However, the price to pay for better performance is to increase production costs due to greater control required.

2.2 Architecture

USB communication provides a *master-slave* architecture, in which the master takes control of the bus and starts the communication, while the slave waits and responds accordingly. The protocol requires the presence of only one master, which takes the name of "*Host*", and one or more slaves, which is called "*Device*". To expand the interconnection between the host and the various devices, it is possible to use devices called "*Hub*"; the latter are conceptually comparable to a multiple socket that is connected to the host or to another hub and allows the output to connect other devices.

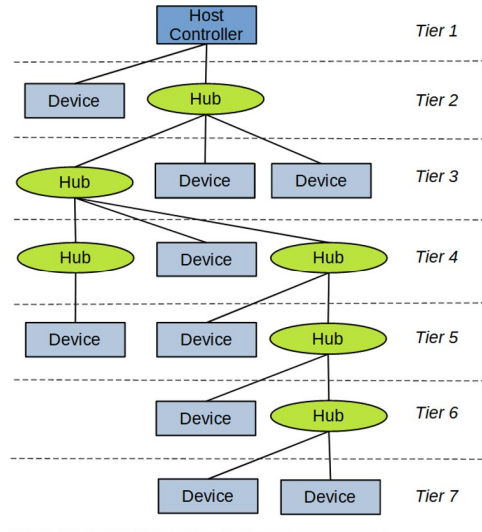


Figure 2.2: USB star-topology architecture [9]

This pyramid structure is called "*star topology*" and provides for a maximum of 127 connected devices, as the address space is 7 bits and the first address is reserved for particular uses. However, such a large number of connections could substantially reduce the maximum bandwidth and weigh on the overall speed. In addition, the maximum length of each cable to ensure reasonable performance is 5 meters, so you can instantiate 7 different levels and a total system length of 30 meters.

2.3 Electrical and line states

From an electrical point of view, the system is composed of a coaxial cable with a certain characteristic impedance Z_{∞} that connects the host with the device. The data travels inside the cable along two differential lines which are called "*D+*" and "*D-*"; in this way the system is made more robust and protects itself from electromagnetic interference that can change the voltage levels present on the bus.

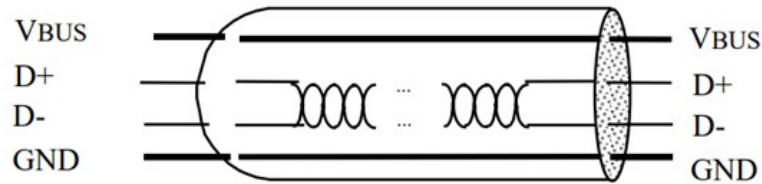


Figure 2.3: USB coaxial cable [10]

In fact, the receiving part is composed of a differential amplifier that receives the two lines and discriminates a "0" from a logic "1" thanks to the voltage difference on the lines themselves. In this way, any disturbance coming from other parts of the circuit will affect both conductors while maintaining the voltage difference constant. In the termination of the coaxial cable, on the device/hub side, there is a pull-up resistor with a nominal value of 1.5 k Ω .

This resistance is essential to make the host understand the maximum data transfer speed of the device: if the "*D+*" line is brought to the power supply voltage, the communication will be of the FS type with a maximum speed of 12 Mbps, otherwise it will be of type LS with a maximum speed of 1.5 Mbps.

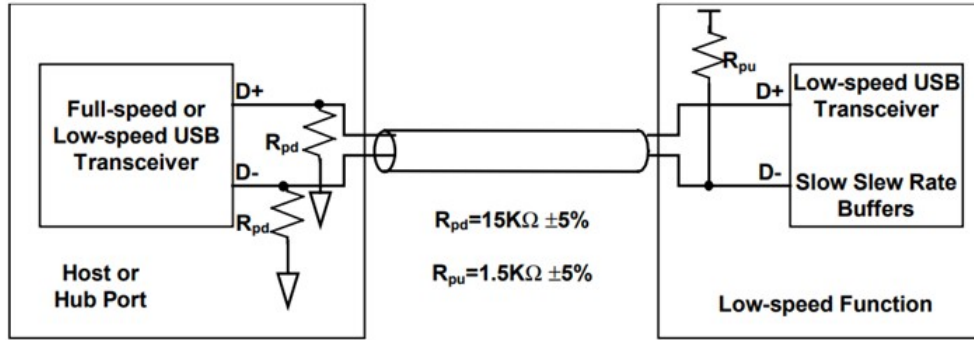


Figure 2.4: USB speed detection based on 1.5 k Ω resistance [11]

The status of the connection between host and device can assume different states based on the voltage level applied on the differential lines. It is possible to distinguish the following states:

- Detached: no device is connected to the host, therefore both lines are kept at logic zero through the two pull-down resistors.
- Attached: at least one device has been connected and one of the two lines is brought to the logical one; through the position of the pull-up resistor of the device, the host distinguishes FS or LS.
- Idle: at least one device is connected but is waiting to receive a bit packet from the host. As for the attach state, the line in which there is the pull up resistor is kept at the high logic state.

These two states are the fundamental ones to discriminate the connection of a new Device. However, once the connection between Host and Device has been established, other configurations of the bus lines are possible:

- Idle: at least one device is connected but is waiting to receive a bit packet from the host. As for the attach state, the line in which there is the pull up resistor is kept at the high logic state.
- J state: coincides with the idle state in which there is a differential "1", that is to say that the line in which the pull-up resistor is connected is brought to the high logic value.
- K state: coincides with the opposite polarity with respect to the previous state.

- Single-ended one (SE1): it is a prohibited configuration that occurs when both differential lines are brought to the high logic value due to a conflict between two drivers.
- Single-ended zero (SE0): both lines are at the low logic level.
- Reset: both lines are brought to logic zero for a time greater than or equal to 10 ms. Normally this condition is used to configure a new device after connecting with the host.
- End of Packet (EOP): as soon as a packet is transmitted, both lines are brought to logic zero (SE0) for 2 bit times followed by a J state for 1 bit time (Idle).
- Suspend: when hosts and devices no longer have to communicate for an indefinite time, switching on the bus is avoided to save energy. This condition is obtained when the idle condition is maintained for a time greater than or equal to 3 ms.
- Resume: when you want to pass from a suspend condition to an activity condition, you have to switch the lines, thus passing from J state to K state for a time greater than or equal to 20 ms.

In order to establish a connection to exchange data, the USB protocol provides some initialization steps to configure the device. The steps can be summarized through a state diagram that takes into account all the possible scenarios that can occur during communication.

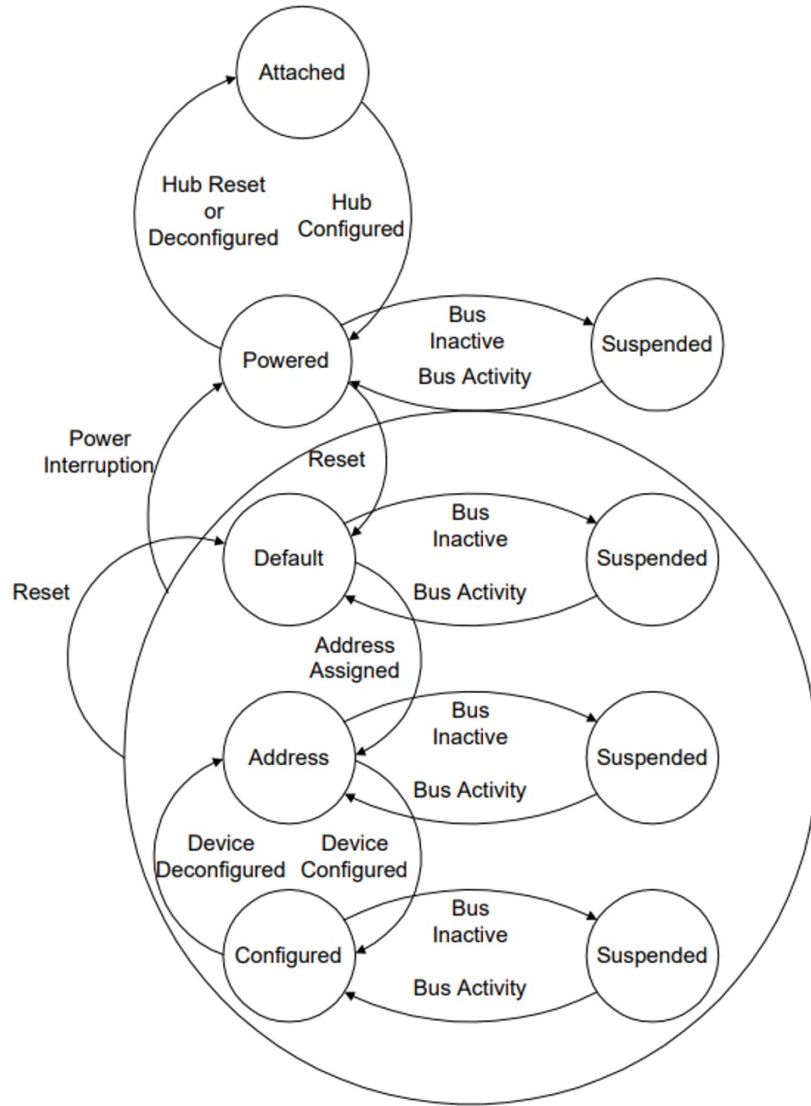


Figure 2.5: state diagram configuration [12]

After detecting a device connection (attached state), the host proceeds with the initial configuration of the hub and begins to provide the power needed to power the device (powered state). A device can be powered via an external power supply or via battery charging, or using the same USB cable which, in addition to carrying information, also carries power. In the first case we speak of a "self-powered device", in the second we speak of a "bus-powered device". It is possible that a device supports both types and switches from one to the other if it is not possible to have the necessary power in one of the two configurations. Subsequently, the host resets the device and assigns it a default address that corresponds to the first address of

the address space, which is then reserved for the initial configuration of the devices. The final step is to assign the first available address uniquely, based on the number of devices already connected (address state) and device configuration (configured state).

2.4 Protocol and transfer types

The host and each device have a well-defined number of "*endpoints*", which can be defined as a two-way virtual communication channel that coexists between the two devices. Each endpoint is associated with a type of transfer depending on its application field. Normally the first endpoint, the one with an address equal to '0', is reserved for particular functions such as the enumeration and general configuration of the device. The exchange of information between two endpoints of the host and the device takes place through data "*packets*", which coincide with the smallest elementary block that can be sent. The transmitted data is encoded via the *NRZI* (non-return to zero inverted). In this case, to transmit a '1', the logic level present on the bus is not changed. On the contrary, a '0' is represented by a change in the logic level. However, this encoding could lead to synchronization

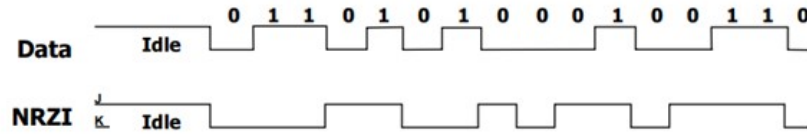


Figure 2.6: Non-return-to-zero-inverted encoding [13]

problems when many consecutive '1' are transmitted. For this reason, what is called bit stuffing is used, i.e. a '0' is transmitted after six consecutive '1'. This technique therefore forces the inversion of the logic state on the bus ensuring a situation of data and clock lock.

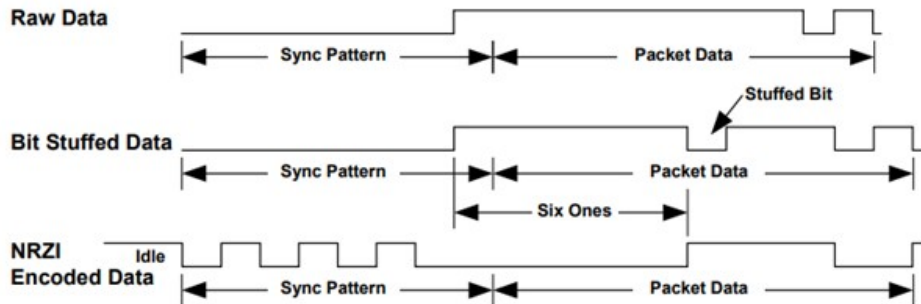


Figure 2.7: NRZI with bit stuffing [14]

At the beginning and at the end of each transmitted packet, the bus is in the idle condition and the packet is made up of some fundamental fields:

- SYNC: is the field reserved for synchronization between the transmitting and receiving part and corresponds to the first 8 MSBs of each packet. When transmitting these bits, the receiving part uses a 48 MHz clock to over-sample the bits from the second device and synchronize with it.
- DATA: this field is variable according to the type of transfer to be used and is made up of some “sub fields”, including the one that contains the bytes of information that must be transmitted on the bus.
- EOP: each packets ends with an end of packet.

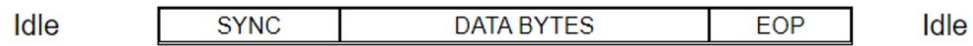


Figure 2.8: Single packet format [15]

The DATA field, as already mentioned, can be divided into several parts. The first 8 bits are always the same regardless of the type of transfer and are called the “*Packet Identifier*” (PID). Based on the combination of the first 4 bits (the 4 MSBs are complementary), the type of packet being transferred is identified between token packet, data packet, handshake packet and special packet. The table (...) lists all the possible combinations of bits, each of which is associated with a PID.

PID Type	PID Name	PID<3:0>*	Description
Token	OUT	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-Frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe (see Section 5.9.2 for more information)
	MDATA	1111B	Data packet PID high-speed for split and high bandwidth isochronous transactions (see Sections 5.9.2, 11.20, and 11.21 for more information)
Handshake	ACK	0010B	Receiver accepts error-free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver (see Sections 8.5.1 and 11.17-11.21)
Special	PRE	1100B	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	(Handshake) Split Transaction Error Handshake (reuses PRE value)
	SPLIT	1000B	(Token) High-speed Split Transaction Token (see Section 8.4.2)
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint (see Section 8.5.1)
	Reserved	0000B	Reserved PID

*Note: PID bits are shown in MSb order. When sent on the USB, the rightmost bit (bit 0) will be sent first.

Figure 2.9: different packet identifier types [16]

Therefore, the format and size of the single packet can change based on the type of PID that is specified within the first byte of data, after sending the SYNC. The various packages can be cataloged in:

- Token packet: is used to specify that the next data packet to be transferred will be in the IN (from the device to the host) or OUT direction (from the host to the device), to transmit a start of frame, i.e. a packet that serves to

keep the device without going into suspend mode, or to transmit a SETUP, that is a packet necessary for the initial configuration of a new device that is connected. In this case, therefore, the data field will be formed, in addition

Sync	PID	ADDR	ENDP	CRC5	EOP
	8 bits	7 bits	4 bits	5 bits	

Figure 2.10: token packet format [17]

to the SYNC and the PID, by 7 address bits corresponding to the device with which you want to speak, 4 address bits relating to a specific endpoint of that specific device in which you want to send the communication, and 5 redundancy bits that take the name of "*Cyclic Redundancy Check*" to check for any errors. The last bits are intended for the EOP.

- Data packet: it can be DATA0 or DATA1 alternately, in order to understand if the communication is continuing correctly or if there has been some problem. In this last case, two consecutive non-alternating DATA0/1 are received and the transmitting device will always send the same packet until it receives the packet of the opposite type. The two PIDs corresponding to *DATA2* and *MDATA* are reserved for high speed communication. Normally this packet is

Sync	PID	DATA	CRC16	EOP
	8 bits	(0-1024) x 8 bits	16 bits	

Figure 2.11: data packet format

sent immediately after the token packet in which the address and endpoint of the device has already been specified. Therefore, in addition to the classic overhead formed by SYNC, PID, CRC and EOP, the remaining bytes are used for the actual data to be transmitted, which can be up to 1024 bytes.

- Handshake packet: this is the last packet within a transaction that is sent. If it is the host that sends data to the device, the handshake packet will be in the IN direction, that is, from the device to the host. Conversely, the host will send this packet to the device.

Sync	PID	EOP
	8 bits	

Figure 2.12: handshake packet format

The handshake can have different meanings depending on the type of PID transmitted: ACK if the receiving device has correctly received all the packets and the communication was successful; NAK if the receiving device is “busy” in other operations and is unable to receive/transmit data; STALL if there is a problem with that particular endpoint or NYET in case of delays. For this type of packet, in addition to the SYNC and the EOP which must necessarily exist for each packet, only the PID that specifies the type of handshake received is required.

- Start of frame packet: it is a special packet that is transmitted on the bus on a regular basis every 1 ms by the host.

Sync	PID	Frame No.	CRC5	EOP
	8 bits	11 bits	5 bits	

Figure 2.13: SOF packet format

The single packets, if sent with a certain order, form more complex structures called “*transactions*”. In this way, a communication that is as simple as effective and secure is used. Each transactions is made up of 3 packets, respectively token packet, data packet and handshake packet. So, for example, an OUT transaction will be formed by an OUT type token packet (the respective PID will give 0001 in binary), a DATA0/1 type data packet that originates from the host to the device and finally a handshake packet, whose content depends on the PID which expresses the status of the communication at that particular moment. Similarly, an IN transaction, will consist of an IN token packet, a data packet from the device to the host and a handshake packet.

In some cases, in OUT and IN transactions, the handshake packet is omitted because it is useless. This is the case of the so-called “*real-time*” communications, for example communications in real time in which the same data cannot be sent again in case of errors. A “correct reception” packet, therefore, would be useless as the data that you wanted to transmit at that moment has now been lost and can’t be sent again.

Finally, it is also possible to distinguish a SETUP transaction whose format is similar to OUT and IN transactions. The only difference is that in this case the data packet, by convention, will always be of the DATA0 type. Furthermore, this transaction is normally used for device configuration, which is why it cannot be a real-time communication and the handshake packet is also required.

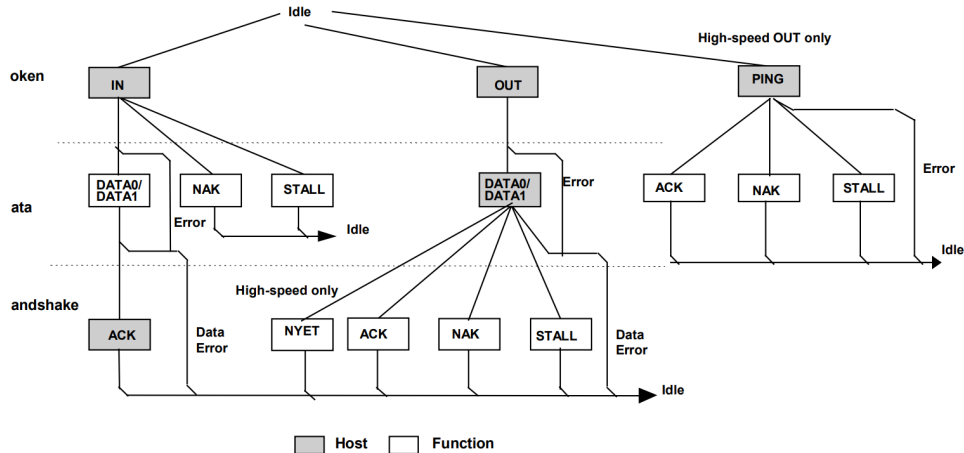


Figure 2.14: OUT and IN types transactions

The sequence of multiple transactions form the highest level structures that are called "*transfers*", which can be of four types depending on the application being used and the endpoint with which you intend to communicate. It is possible to distinguish four different types of transfers:

- **Bulk transfer:** allows the transmission of a large amount of data without errors. This type of transfer can't take place in LS and allows a packet size of up to 64 bytes. After identifying an OUT or IN endpoint, an OUT or IN transaction using that specific endpoint is required to initiate a bulk transfer.
- **Isochronous transfer:** allows the transmission of "real-time" data, i.e. for those applications that need to exchange instant information, such as audio devices like microphones and speakers. As for bulk transfer, the possible types of transactions are OUT or IN, with a maximum size of 1024 bytes. However, unlike bulk transfers, there is the absence of the handshake packet after data transmission.
- **Interrupt transfer:** allows the transmission of data for HID applications such as the use of mouse and keyboards. As soon as the instantaneous position of the mouse is changed, or if a key on the keyboard is pressed, an interrupt bit is activated that alerts the host of a new event. It will then initiate an

interrupt transfer to update the position of the mouse or display the character corresponding to the key pressed. As for the previous transfers, the possible transactions are of type OUT or IN, even if in this case the IN transaction is more used. The maximum packet size in FS is 64 bytes and the handshake packet is present.

- Control transfer: this type of transfer is necessary for the initial configuration of a new device and allows the host to receive all the information necessary for the correct behavior of the device. This transfer is the most "complicated" of all those already listed, as it is made up of 3 stages, each of which corresponds to a different transaction.

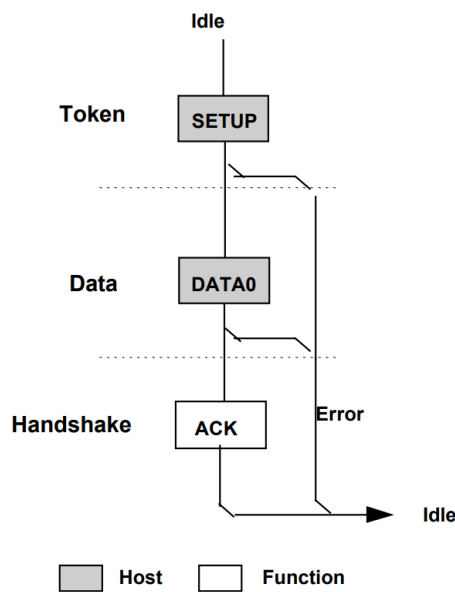


Figure 2.15: Example of control transfer made up of 3 different stages

The first stage corresponds to the “*setup stage*”, where a SETUP transaction is sent, in which the data packet is 8 bytes and gives some information about the size of the bytes transferred to the next stage, which is called the “*data stage*”. This second stage is optional and consists of a variable number of OUT/IN transactions of the DATA0/1 type. The third and last stage is what takes the name of “*status stage*” and is formed by an OUT/IN transaction of zero length opposite to that of the previous stage. For example, if IN transactions are transmitted in the second stage, an OUT transaction with a byte length of zero will be transmitted in the last stage.

Chapter 3

USB2 IP by STMicroelectronics

The USB 2.0 digital IP designed by *STMicroelectronics* supports data transfer in FS, i.e. with a speed of 12 Mbps. Both Host and Device mode are supported, with a programmable number of endpoints reaching up to 8 endpoints: the first is normally reserved for control transfers, in order to allow the correct configuration of a new connected device, while the other endpoints can be used for other transfers, including the isochronous one. Furthermore, the USB communication provides a dedicated memory for the storage of the received/sent data which takes the name of “*packet buffer memory*”, and in this case reaches a maximum size of 2048 bytes. Suspend/resume operations are supported to disable digital IP when a USB communication to the external world is no longer required and to save on power consumption. This is done by writing a specific bit in the control register and allows to disable the clock using the clock gating technique in order to disable the IP. Any new activity on the bus will wake up the peripheral enabling the clock asynchronously. Finally, the Device mode implements the featuring of battery charging, and can therefore be powered via a hub or via the same host using the USB Type-C cable. As per USB specification, there is the possibility to connect/disconnect the USB cable without damaging the IP itself. This occurs thanks to the presence of an integrated pull-up resistor on the D+ line of the host. The block diagram of the entire IP is shown in Figure 3.1.

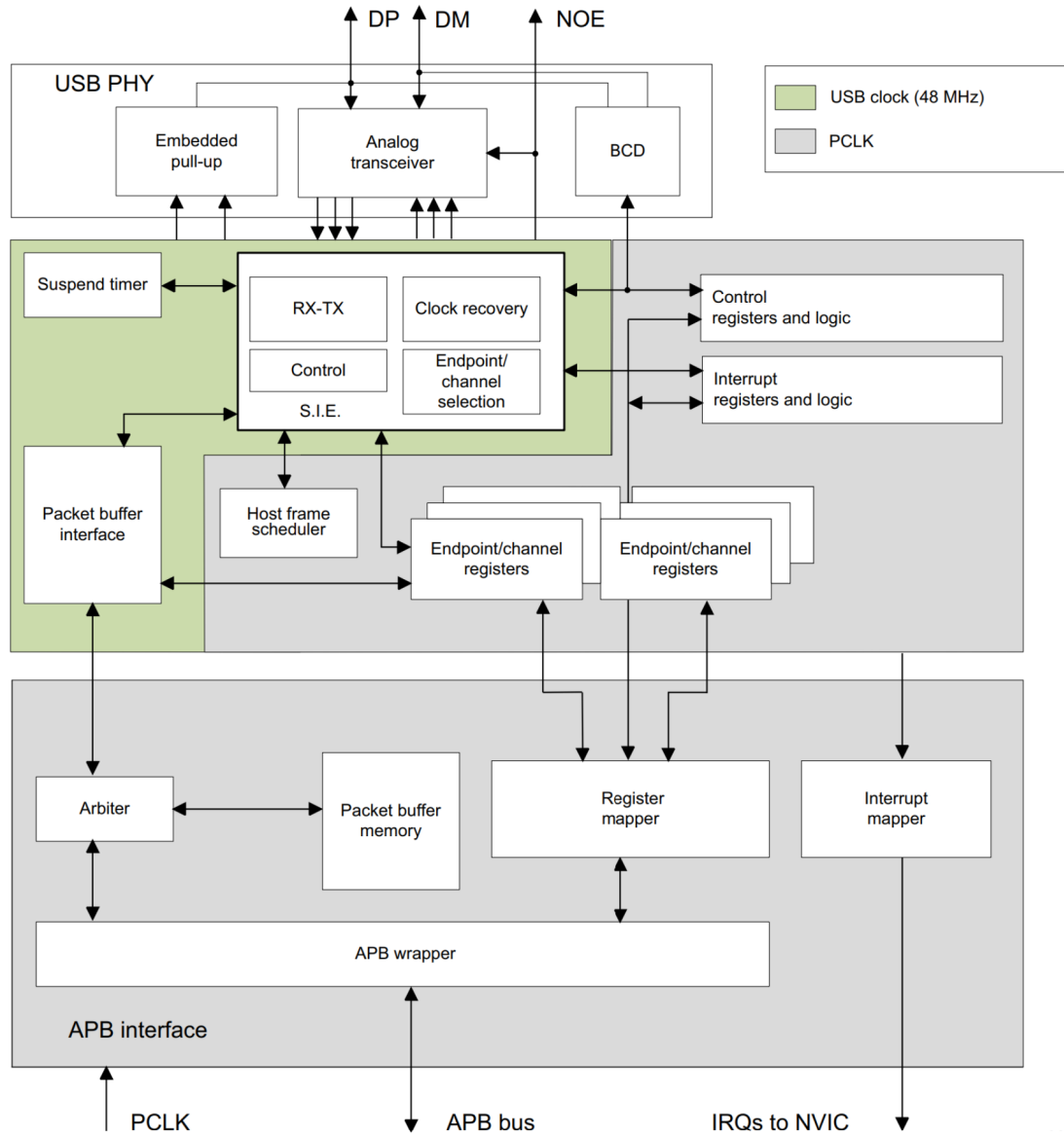


Figure 3.1: USB peripheral block diagram [18]

The data exchanged between an external USB device and the system memory takes place via the packet buffer memory, which is directly accessible from the USB device. Each endpoint is associated with a "*buffer description block*", that is a buffer in which the specific memory location for that particular endpoint, its size and the size of the bytes to be transmitted are indicated.

Once the USB device recognizes the arrival of a token on a specific endpoint, a data transfer takes place and the data received are momentarily saved in a 32-bit register; the memory is accessed on the flight and the contents of the buffer are saved. When all data is received, an handshake packet is generated by the host or device, depending on the direction of the previous packets. When the first transaction is completed, a specific interrupt is generated and the microcontroller determines some parameters, such as what will be the next type of transfer to take place and which endpoint must be served for the next transaction. A very interesting feature is the one called "*double buffer*", in which two buffers are used instead of one, so as to always have a buffer available to receive new data while the microcontroller uses the other to load it inside the memory.

3.1 Description of USB blocks

From the block diagram of Figure 3.1 it is possible to identify various blocks connected to the USB interface, including:

- USB physical interface: this block consists of the pull-up resistor to manage the attach/detach of a device, a support to detect the battery charging feature (battery charging detection) and a differential transceiver. The latter is composed by many drivers which allow to force J state or K state to transmit data from the transmitter side, and to discriminate one of the two states from the receiver side. The left transceiver is called "upstream transceiver" as it is related to the host, while the one on the right is connected to the device and is called "downstream transceiver", in which we can also identify the connected pull-up resistor to line D+ (FS mode). The upstream and downstream transceivers linked respectively to the host and the device, are formed by some receivers for the transmission and reception of the signal. There are both differential and single-ended receivers: in the first receivers, both the D+ and D- lines are linked to the input and the voltage difference between the two lines is evaluated, while the latter have only one of the two lines at the input. Both types of receivers have a minimum threshold V_{IL} equal to 0.8 V and a maximum threshold V_{IH} equal to 2.0 V. It is also necessary that the single-ended receiver is composed by a voltage comparator with hysteresis, in order to have a greater robustness to external disturbances. The USB protocol provides very precise specifications regarding the voltage levels that must be present on the D+ and D- lines in order to discriminate the different states of the bus.

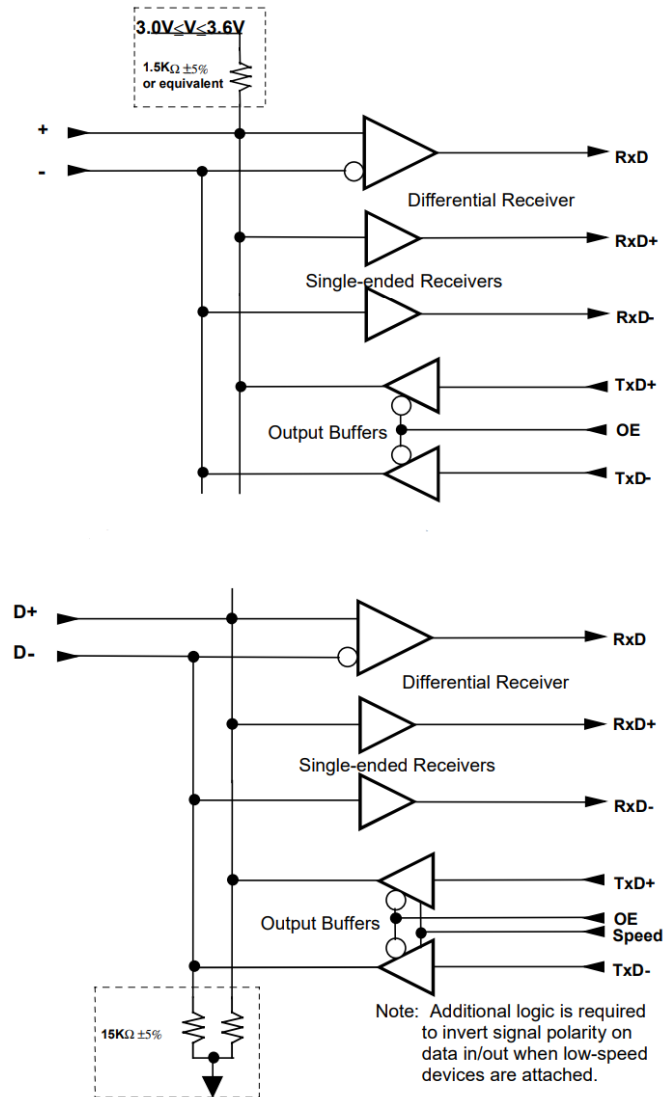


Figure 3.2: Upstream and downstream transceivers [19]

From the table in Figure 3.3, it is possible to analyze some fundamental parameters. For example, to correctly detect a Differential "1", the voltage difference between the D+ line and the D- line must be greater than 200 mV and at the same time, the voltage on D+ must be greater than 2 V . Therefore, the worst case to be able to detect a Differential "1", would be with a voltage on the D+ line equal to 2.01 V and a voltage on D- equal to 1.80 V. According to the USB specification, the SE1 coincides with a prohibited condition in the which the bus should never be in during normal operation.

An SE1 is detected by the receivers when both the D+ and D- lines are above 0.8 V. The conditions of Differential "1" and of SE1 are mutually exclusive, as different voltage comparators are used.

Bus State	Signaling Levels		
	At originating source connector (at end of bit time)	At final target connector	
		Required	Acceptable
Differential "1"	D+ > VOH (min) and D- < VOL (max)	(D+) - (D-) > 200 mV and D+ > VIH (min)	(D+) - (D-) > 200 mV
Differential "0"	D- > VOH (min) and D+ < VOL (max)	(D-) - (D+) > 200 mV and D- > VIH (min)	(D-) - (D+) > 200 mV
Single-ended 0 (SE0)	D+ and D- < VOL (max)	D+ and D- < VIL (max)	D+ and D- < VIH (min)
Single-ended 1 (SE1)	D+ and D- > VOSE1(min)	D+ and D- > VIL (max)	
Data J state: Low-speed Full-speed	Differential "0" Differential "1"	Differential "0" Differential "1"	
Data K state: Low-speed Full-speed	Differential "1" Differential "0"	Differential "1" Differential "0"	
Idle state: Low-speed Full-speed	NA	D- > VIH (min) and D+ < VIL (max) D+ > VIH (min) and D- < VIL (max)	D- > VIH (min) and D+ < VIH (min) D+ > VIH (min) and D- < VIH (min)
Resume state	Data K state	Data K state	
Start-of-Packet (SOP)	Data lines switch from Idle to K state		
End-of-Packet (EOP) ⁴	SE0 for approximately 2 bit times ¹ followed by a J for 1 bit time ³	SE0 for ≥ 1 bit time ² followed by a J state for 1 bit time	SE0 for ≥ 1 bit time ² followed by a J state
Disconnect (at downstream port)	NA	SE0 for ≥2.5 μs	
Connect (at downstream port)	NA	Idle for ≥2 ms	Idle for ≥2.5 μs
Reset	D+ and D- < VOL (max) for ≥10ms	D+ and D- < VIL (max) for ≥10 ms	D+ and D- < VIL (max) for ≥2.5 μs

Figure 3.3: Low-/full-speed Signaling Levels [20]

- Serial Interface Engine: the function of this block is to recognize the synchronization bits, the bit stuffing and verify the packet identifier. It is also able to generate some signals such as SOF and RESET.
- Timer: this block detects a suspend condition when there is no type of activity on the bus for a time greater than or equal to 3 ms.
- Endpoint registers: a register is associated with each endpoint, which contains a register where the type of endpoint (based on the transfers it manages) and the current state in which it is located are saved.
- Host Frame Scheduler: this is a block that allows the management of the transfers to be made between two SOFs based on the available bandwidth. In fact, within a frame, different transfers can be sent until the maximum limit is reached, beyond which nothing more can be sent except before the next frame. This organization takes place according to the priorities of the transfers. Normally, real-time transfers are placed first, such as interrupts and isochronous ones, while non-periodic transfers (bulk and control) are placed at the last.
- Control Registers: this block consists of all the registers that contain the control bits of the entire USB peripheral.
- Interrupt Registers: this block consists of all the registers that contain the interrupt detection bits, clear of some states or clear of pending interrupt.
- Packet Memory: consists of local memory which contains all packet buffers. This memory is directly accessible from the software to pick up the received packets and has a maximum size of 2048 bytes.
- APB Wrapper: this block maps all the USB device in the Advanced Peripheral bus address space.

Therefore, the USB interface acts as a "bridge" between the differential lines D+/D- and the APB bus. The latter is a bus that is optimized to consume as little power as possible and is normally used to drive slower microcontroller peripherals, including the UART, timers, keyboard or USB peripheral. Another type of bus is what is called AHB (Advanced High-performance Bus) and is used for connection to processors, memories and DMAs, which require higher performance than the peripherals connected to the APB. Normally the APB and AHB buses are interfaced through a "bridge" that allows you to switch from one bus to another.

3.2 Usage and structure of packet buffers

Each endpoint is configured as bidirectional, which means it can be used to transmit or receive data over the bus. These data are contained in a specific memory location which is reserved for each endpoint and contains two buffers, one for transmission and the other for reception. They can be placed anywhere within the packet memory, as their position and size are specified by the "buffer description table", that is a table made up of many buffers associated with each endpoint.

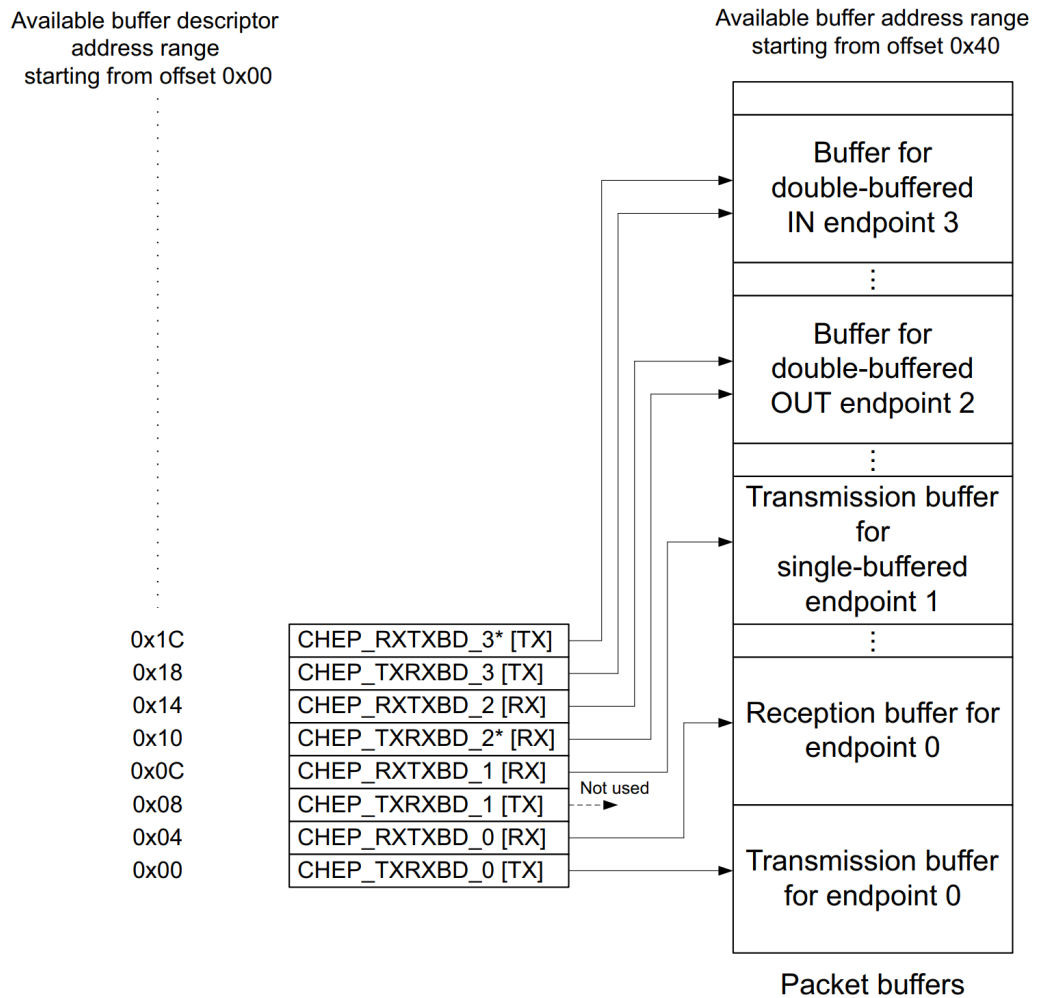


Figure 3.4: Packet buffer with examples of buffer description table locations [21]

When the host sends a data packet to the device, the first thing that is done is to check that the address matches the address of a device endpoint. If there is a coincidence, the USB peripheral of the host accesses the "buffer description table" and selects the "*CHEP_RXTXBD_n*" register for that specific endpoint. Within this register, there are two fields, respectively "*ADDRn_TX*" and "*COUNTn_TX*"; the first contains the address of the first byte inside the packet memory that the host must transmit, while the second contains its size and it is useful to understand how many bytes have been sent and how many are left to be sent. On the other hand, when the device receives a packet from the host, the received address is always checked against the endpoint address. Subsequently, the content of *ADDRn_RX* (which contains the memory address to save the data) of that related endpoint is saved in an internal ADDR register and the COUNT register is reset. The received bytes are transferred to the packet memory starting from the address contained in ADDR. At the same time, an internal counter is initialized with the maximum value of the data size for that specific transfer and is decremented each time a data transfer occurs, while COUNT is incremented. This is essential to detect any overrun situations. When the end of the data transfer is detected, any errors are evaluated through the CRC and proceeds with the handshake. In the event that some error occurs, the data is still copied into memory but an ACK signal is not sent to the host, while an error bit is set in the status register of the USB device. In this case the packet must be resent.

When a large amount of data needs to be transferred, an endpoint configured in bulk mode is normally used. The data processing procedure requires some time to access the various buffers and save the content inside the packet memory. In case a packet is transferred while the previous one has not yet finished processing, a NAK signal is sent to resend the packet again, hoping that the buffer has been managed. The double buffer consists in configuring both buffers contained within the packet memory and addressed by the buffer description table, both modes in TX or RX. In this way, the first data received is saved on the first buffer and processed immediately. The second data is however accepted and saved in the second buffer; while the second buffer has been saved, the first has already been processed by the software and therefore you are available to process the second. This technique allows to increase the working frequency thanks to a higher band. In order to enable the double buffer it is necessary first of all to select the type of endpoint, and then to set a bit inside a control register, which allows you to choose between two unidirectional buffers (both TX or both RX) or a single bidirectional buffer.

3.2.1 Double buffer in HOST/DEVICE mode

As described above, double buffer is a technique that is used to transfer a large amount of data between two systems via the USB protocol, maximizing the use of the available bandwidth. For this purpose, the two buffers *CHEP_RXTX_BD* and *CHEP_TXRX_BD*, which contain the number of bytes received/transmitted and the addresses related to the endpoint in the packet memory area, are made unidirectional, i.e. both in the OUT direction (for data exchange from the host to the device) or both in the IN direction (for data exchange from the device to the host). In our case, the two buffers were both configured to support OUT transactions and this allowed us to increase the percentage of bandwidth used by a single frame. In the case of single buffer, the USB hardware device used the *CHEP_TXRX_BD* buffer to fill it with the data just received and then waited for the software to write the buffer in memory, as the *CHEP_RXTX_BD* buffer was reserved for IN transactions. By enabling the double buffer instead, both buffers can be written with the received data, so the USB device can fill a buffer while the software takes care of processing the second.

Therefore, the fact that the USB device does not necessarily have to wait for the software to manage the data being received but can switch to fill another buffer, allows you to increase the maximum bandwidth and transfer data faster. It could happen that the host is faster than the device and sends OUT transactions very close to each other on the bus. The device software, being slower, is unable to free the buffer to be processed in time to accept the new transaction from the host. In this case, the device will respond with a NAK handshake to tell the host that it is unable to accept new data as the buffers are both occupied. The host will always send the same transaction until the device is able to accept and manage it. In the case of double buffer, one of the two signals between STATTX and STATRX must be activated according to the type of endpoint that is configured (in the case of OUT transactions STATRX will be set to a value other than 0, while STATTX will be disabled and will assume 'the value 0). This signal contains information about the status of the endpoint involved in the communication.

On the device side, the hardware will configure STATRX to NAK immediately after there has been a correct transaction, so that the software has time to process the data just received before setting it back to VALID, send an ACK and be ready to receive a new transaction. On the host side, however, this signal is configured to VALID when the channel is ready to start a new transaction; in this case it will enter the HFS execution queue and will wait for its confirmation in order to start the transmission.

In the case of double buffer, VALID can be set if an ACK is received from the device: the second channel will remain valid and ready to receive a new transaction, while the first will be managed by the software to save the data in memory. When the host receives a NAK from the device, the channel is suspended and the hardware will write VALID only at the beginning of the next frame, trying to send that data again. However, the software may decide to retry transmission immediately by configuring STATRX to VALID. In order to handle the double buffer function, the

STATRX[1:0]	Meaning
00	DISABLED: all reception requests addressed to this endpoint/channel are ignored.
01	STALL: Device mode: the endpoint is stalled and all reception requests result in a STALL handshake. Host mode: this indicates that the device has STALLED the channel.
10	NAK: Device mode: the endpoint is NAKed and all reception requests result in a NAK handshake. Host mode: this indicates that the device has NAKed the reception request.
11	VALID: this endpoint/channel is enabled for reception.

Figure 3.5: STATRX status table [22]

structure must be slightly changed. To discriminate which buffer is currently in use by the software and which buffer is about to be filled by the hardware, two signals are used, DTOGRX and DTOGTX, which are bit 14 and bit 6 of the register of that particular endpoint, respectively. In single buffer, only one of these two signals was used, depending on the direction, to signal the correct reception of a packet along with the STATRX bit. In particular, this signal is updated by the hardware when an ACK is received and contains the expected value of the data bit that must be alternated (DATA0 = 0, DATA1 = 1). In double buffer, DTOGRX is normally used as in the case of single buffer, while DTOGTX is renamed to *SW_BUF* and indicates which buffer is currently used by the software.

Buffer flag	'Transmission' endpoint	'Reception' endpoint
DTOG	DTOGTX (USB_CHEPnR bit 6)	DTOGRX (USB_CHEPnR bit 14)
SW_BUF	USB_CHEPnR bit 14	USB_CHEPnR bit 6

Figure 3.6: Double-buffering buffer flag definition [23]

In the case of OUT transactions, as specified in Figure 3.6, DTOGRX is used by the hardware as in the case of single buffer, while *SW_BUF* (DTOGTX) is used by the software to save the data already received in memory. The double buffer is enabled by the software by configuring two bits:

- UTYPE: corresponds to bits 9 and 10 of the endpoint register and is set to 00 to specify the type of endpoint (BULK)
- EPKIND: corresponds to bit 8 of the endpoint register and is set to 1 (*DBL_BUF*)

The software initializes the DTOGRX and *SW_BUF* signals based on the buffer that is used by the software and hardware. At the end of each transaction, DTOGRX is removed from the hardware so that the software already knows the next buffer to take. As soon as the software completes the processing of the buffer, it changes the *SW_BUF* signal in such a way as to notify the hardware device that the buffer is available and that it can receive a new transaction.

3.3 USB registers

The USB device has several registers to be able to manage all the features that are implemented. It is possible to list the registers in two large groups, common registers and endpoint registers. The first group contains the control and interrupt management registers, while the second contains the endpoint configuration and status registers. Furthermore, there is a third type of registers which is called "*USB SRAM registers*", which includes the buffer description table, which is used to locate the packet buffers and is contained within a section of the packet memory.

USB control register

This control register (*USB_CNTR*) is part of the common registers and is a 32-bit register. Some of them are reserved and not configurable by the user, as they can be configuration bits or redundancy bits in view of future functionality. Bit 31

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
HOST	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	THR 512M
r/w															r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTRM	PMA OVRM	ERRM	WKUP M	SUSP M	RST_D CONM	SOFM	ESOF M	L1REQ M	Res.	L1RE S	L2RE S	SUS PEN	SUSP RDY	PDWN	USB RST
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w		r/w	r/w	r/w	r	r/w	r/w

Figure 3.7: Control register structure [24]

"HOST" is initialized when the USB peripheral of the evaluation board is switched on. This value is set to "1" if the board is to be used as a host, or to "0" if the board is to be used as a device. The value of the bit can be changed every time the IP is turned on or reset so that its role can be changed. A feature born with USB 2.0 and used above all in the field of mobile telephony, is the one that takes the name of "On-The-Go" (OTG), whose purpose was to alternate the role of a mobile device between Host and Device to be able to interface both with external Hard Disks to save data, and with a host PC to be viewed as mass memory. The substantial difference with respect to the functioning of the evaluation board (which does not support the OTG) is that bit 31 "HOST" can be changed during normal operation of the device, without the need to restart it. Bit 0 "USBRST" allows you to reset the device. In case the board is configured in "device mode", this signal would allow to reset the internal state machine. In the case of "host mode", the reset signal is asserted by the software to drive the reset status on the bus and to initialize the device. To completely disable the USB device and turn off all analog parts, you can use bit 1 "PDWN". In the case of "device mode", receiving a suspend interrupt (when there is no activity for at least 3 ms) enables the "SUSPEN" bit. As soon as the suspend state is propagated inside the peripheral, the activity of the device is stopped and the "SUSPRDY" bit is raised to logic 1. Instead, the host can enable this bit when no activity is expected to be sent on the bus; in this case the SOF generation is stopped. In the event that a SUSPEN signal is received while data is being sent on the bus, it is necessary to first finish sending the data itself and then put the device into suspend mode. All the other bits are used to enable/disable some interrupts such as error interrupt (ERRM), wakeup interrupt (WKUPM) or SOF interrupt (SOFM), so they are used to mask or unmask these kind of interrupt.

USB interrupt status register

This register (*USB_ISTR*) contains the status bits of all interrupts, in order to determine which event caused an interrupt request. These bits are driven by the hardware and generate an interrupt request if and only if the corresponding bit in the control register is enabled. After executing the routine code linked to that type of interrupt, the corresponding bit in this register is brought back to the logical state '0'; if this does not happen, the interrupt is still pending and the bit is kept high. Also, if multiple interrupt requests occur, only one is executed based on the assigned priority. To assign priority to each interrupt, the user can specify within the software the order of the interrupts to be served. Furthermore, to be more robust against external noise that could alter the voltage levels by canceling some interrupts, more bits are used instead of just one bit. Bit 30 "*LS_DCON*", if the logical value is high, indicates the detection of an LS device and is available

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	LS DCON	DCON- STAT	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	THR 512
	r	r													rc_w0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTR	PMA OVR	ERR	WKUP	SUSP	RST- DCON	SOF	ESOF	L1REQ	Res.	Res.	DIR	IDN[3:0]			
r	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0			r	r	r	r	r

Figure 3.8: Interrupt status register structure

only in host mode, while bit 29 "DCON_STAT" indicates the connection status: '0' if no device is connected, '1' if the LS device is connected. To indicate that a transfer was successful, the hardware configures bit 15 "CTR". Then we have other important bits:

- PMAOVR (14): this bit is brought to the high logic level by the microcontroller when it is not able to manage a memory access in time. In this case an interrupt is generated which is used by the host to try to send the packet again.
- WKUP (12): this bit is managed by the hardware to "wake up" the USB device after a suspend period because activity on the bus is detected. Due to this interrupt, the SUSPRDY bit in the control register is reset.
- SUSP (11): this bit is managed by the hardware and activated when no activity is detected for a period of time greater than or equal to 3 ms.
- SOF (9): this bit indicates the start of a new frame and is configured when a new SOF packet is detected.
- IDN[3:0]: these 4 bits indicate the endpoint that generated the interrupt that is being executed.

From Figure 3.8 it is possible to notice that the status bits can only be read by the software as they are marked with the letter "r", while the interrupt request bits can be written.

Chapter 4

Used resources for the experience

4.1 STM32 Evaluation Board

STMicroelectronics offers a wide range of evaluation boards. These boards can be considered as a development and testing platform for the microcontrollers that are located on the boards of the ST family. For this experimental project it was decided to use the *STM32G0C1E-EV*, which is part of the *STM32G0* mainstream product line. This evaluation board offers a wide range of applications and a high level of integration thanks to the *Arm® Cortex®-M0+ 32-bit* microprocessor equipped with a 512 kB flash memory to contain the startup and application files, and a data memory 144 kB RAM, essential for saving data and temporary variables. On

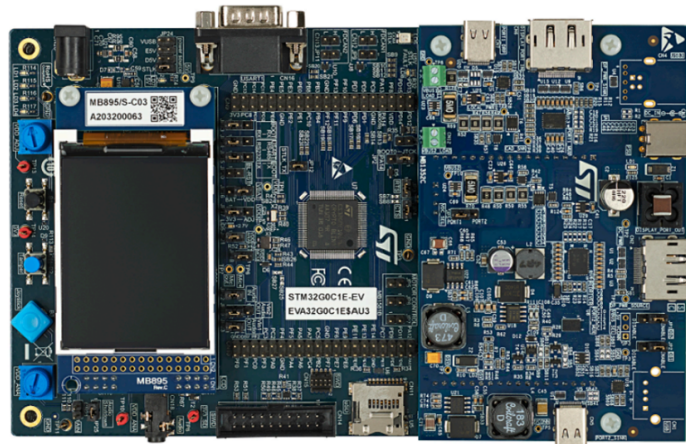


Figure 4.1: STM32G0C1E-EV evaluation board [25]

the front of the board it can be seen a large 2.4" LCD display, which allows you to display the implemented application and the results obtained on the screen. In this project, the presence of the LCD was essential for following the flow of the connection between the two boards and for debugging, showing on the screen all the various configuration steps and the transmitted string once the communication between host and device was established. Two more daughterboards are mounted on top of the motherboard, which increase flexibility and extend the variety of applications through which various peripherals can be tested.

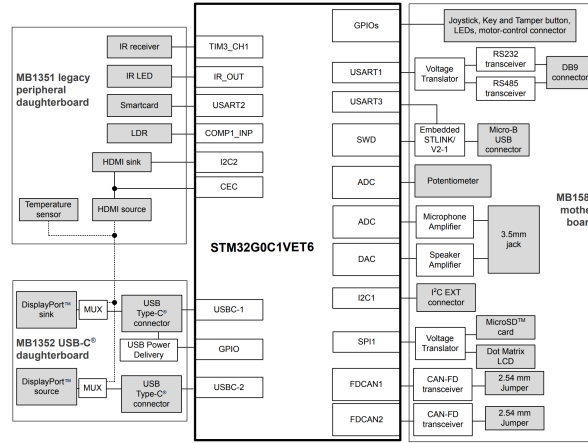


Figure 4.2: STM32G0C1E-EV motherboard and daughterboards schematic [26]

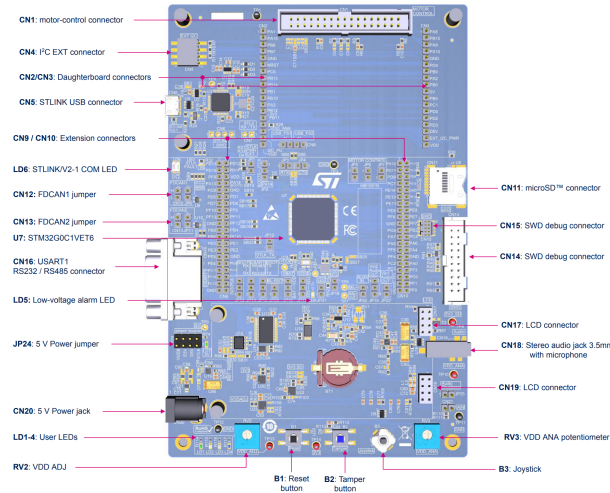


Figure 4.3: STM32G0C1E-EV motherboard layout [27]

The hardware block diagram in Figure 4.2 illustrates the various connections between the microcontroller, the motherboard and the daughter boards, while Figure 4.3 refers to the layout of the motherboard and all the I/O connectors to communicate with the outside world. It should be noted that the microcontroller has two independent USB Type-C peripherals that are connected to the I/O connectors of the daughter board. The first connector is a DRP and can deliver up to 45W of power, while the second only acts as a sink. However, both ports support PD to supply other devices and USB 2.0 FS for data transfer. In addition to the two Type-C connectors, there are other connectors that allow the use of various types of applications, including:

- MicroSD card: useful in mass storage applications. For example, it is possible to load an image into the MicroSD card and show it on the 2.4 "LCD screen;
- SWD debug connector: ARM debug interface that uses 2 pins, which allow data to be released (for example flags, printf, error signals, ...) using specific protocols
- RS232/RS485 connector: output port for communicating the board with another device via RS232 or RS485 serial protocol
- DisplayPort Input: useful in applications where a second device is capable of providing a video via a USB Type-C cable. In this case, the board is connected both to an external monitor via the DisplayPort and to the device that transmits the video via a USB Type-C cable.

In order to work properly, the motherboard must be powered with a voltage of 5V. This voltage can be supplied in different ways according to the position of the jumper J24 present in the lower part of the board. By placing the jumper in "*STLK*" position, the board will be powered using a Micro-B USB cable, by placing it in "*E5V*" it will be powered through the PSU jack, or in "*D5V*" it will be powered by the connectors on the daughterboard. Another possibility is to position the jumper in "*U5V*": in this case the PD feature is exploited and the board will be powered via a USB Type-C cable that will be attached to the connector mounted on the daughterboard. The microcontroller, on the other hand, is powered via the VBAT pin, which can be connected to different power supply voltages based on the positions of jumpers J16 and J17: it is possible to choose between 3V, 3.3V (standard configuration) or *VDD_ADJ*. In the latter case, a trimmer allows you to adjust the voltage supplied to the microcontroller from a minimum of 1.65V to a maximum of 3.5V; this feature can be useful in some applications where the microcontroller voltage can be lowered to save power consumption. Below we will analyze in depth a module present within the evaluation board essential for the correct functioning of the USB device.

4.1.1 Clock Recovery System

From the USB protocol specifications, it can be seen that for good communication to take place, the data-rate must be $12,000 \text{ Mb/s} \pm 0.25\%$. This high accuracy requires the use of a very precise clock signal, which is normally generated by a quartz crystal external to the microcontroller. Through magnetic resonance or mechanical deformation, the crystal begins to oscillate with a very precise frequency. However, an high quality quartz is very expensive, and this would increase the production cost of the board itself. In order to generate a timing signal of a device, an internal or an external component can be used. Another possibility is to use an internal signal whose precision is variable.

The *Clock Recovery System* is a peripheral whose purpose is to obtain a timing signal with very high precision without using an external oscillator, but simply using an input signal as a reference to generate an error capable of controlling the accuracy of the timing signal. It is similar to a digital *PLL*: it requires a very precise external clock signal to generate a replica of the input signal at the output. Normally, the board host is equipped with an external high-speed oscillator (*HSE*), while the board device is equipped with a less precise internal oscillator (*HSI*) and a CRS module. In this case, therefore, the synchronization signal is generated by the SOF packet sent by the host on the bus every 1 ms.

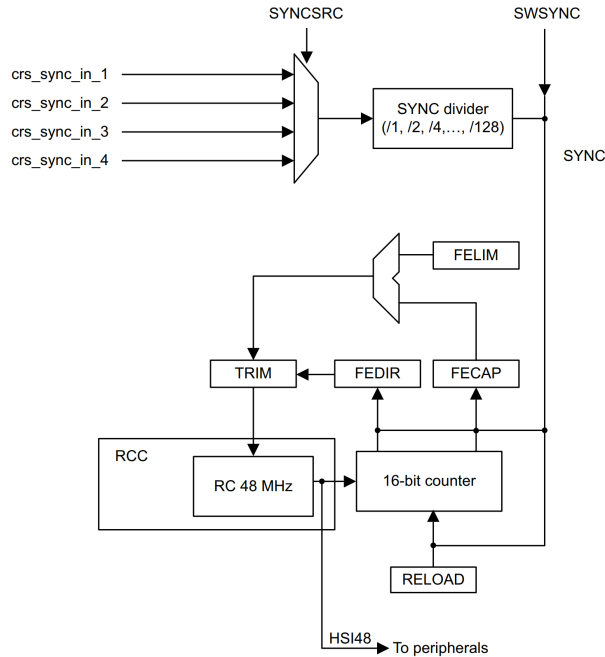


Figure 4.4: CRS block diagram [28]

In the second scenario, if the counter is in that specific area and the direction is of the "DOWN" type, it means that the current frequency is less than the desired one. Then, the new TRIM value is increased by one unit and the counter at the next RELOAD will start from this last updated value. If, on the other hand, the counter is always within that area but the direction is of the "UP" type, it is updated by decreasing TRIM by one unit. In the third scenario, a similar procedure takes place based on the direction of the counter, but it is incremented/decremented by 2 units. If the two frequencies are synchronized, an ESYNC signal is generated and the counter is not updated.

4.2 Protocol Analyzer

The protocol analyzer is a non-intrusive verification tool able to "spy" and "capture" the data traffic present on a communication channel without altering the voltage level. To achieve the purpose of this thesis it was necessary to purchase a hardware protocol analyzer and, after a careful analysis of the products on the market, the "Teledyne Lecroy Mercury T2C" was selected with all the necessary accessories. The correct setup involves placing the analyzer in series between the two devices that are communicating. Therefore it is necessary to connect the second end of the USB Type-C cable of the host board to port 2, while the cable connected to port 1 will be connected to the board device.

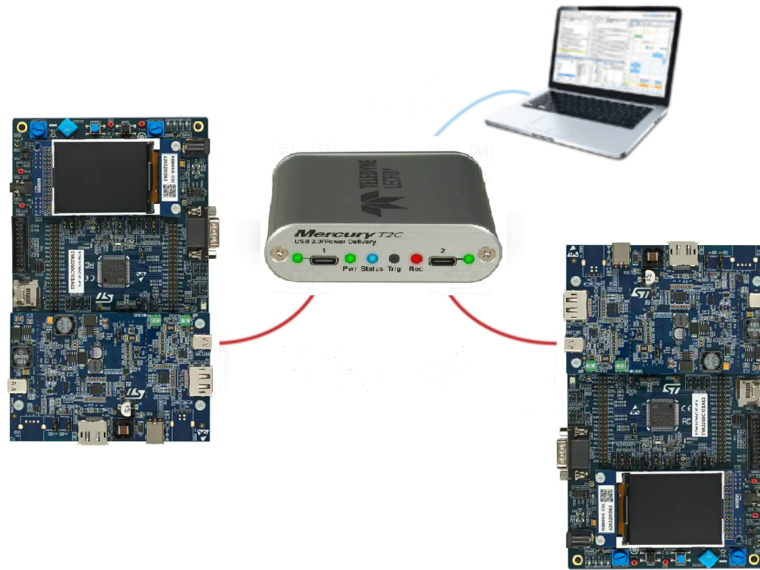


Figure 4.6: Instrumentation setup

On the back of the protocol analyzer there is a third USB Type-C port for powering the device itself, which must be done via the PC in which the Teledyne Lecroy proprietary software is installed, which takes the name of "USB Protocol Suite". Once the protocol analyzer is powered, the program must be launched from the PC to configure the device before capturing data on the bus. After the configuration it is possible to connect the device in series and start recording by clicking on "Start recording".

Through a special option, you can configure some recording parameters such as:

- **Trigger Mode:** it is possible to select the automatic trigger (snapshot), the manual trigger via a button to click, or the event trigger configurable via a specific option. In this last case, the software will start recording the condition of the pre-trigger bus. As soon as the selected event has been verified, the software will automatically trigger and the post-trigger packets will be captured.
- **External Trigger Settings:** useful if you want to use an external signal as a trigger (for example a button or the voltage variation on a certain pin)
- **Spooled Recording:** it is possible to choose the duration of the recording by setting a start and end time or the maximum size (in MB) of the recording file.

A very powerful feature is to choose the trigger based on the occurrence of a certain event or multiple events in sequence. For example, you can start triggering as soon as a first SOF signal occurs or if a SOF - TOKEN sequence occurs. To enable this option, you need to switch to advanced mode and then, through the appropriate section, choose the events or trigger sequences. Once registration has started, the interface visible to the user is the one in Figure 4.7. It is a colorful graphic interface, in which the different fields are highlighted with a different color for easy reading and better viewing of data, errors and other conditions.

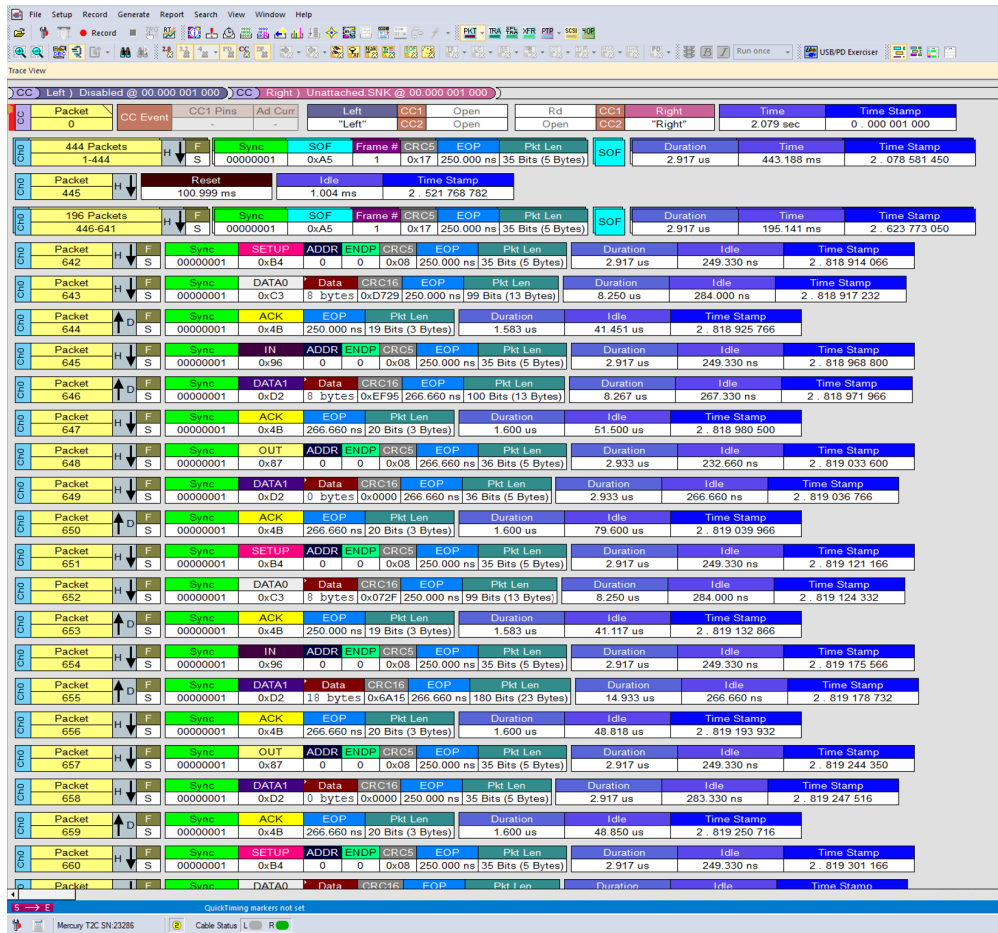


Figure 4.7: user interface view at packet level

Individual packets that are transmitted on the USB bus can be analyzed. Within a single line visible on the screen, numerous information is shown, such as the number of the channel used (in this case channel 0), the number of the packet transmitted, the direction (IN / OUT), the speed transmission, the various fields of the packet (SYNC, PID, ADDR, ENDP, CRC, EOP) and some information about the duration of transmission of the packets and the time that elapses between the end of one packet and the beginning of the next. Using the toolbar at the top you can select the "level view" and you can choose between different options:

- PKT: default configuration in which the single packages are shown;
- TRA: configuration in which transaction types are shown;
- XFR: configuration in which the transfer types are shown;

By clicking on the single transaction or on the single transfer, it is possible to extend the window and see the whole hierarchy. In Figure 4.8 there is an example

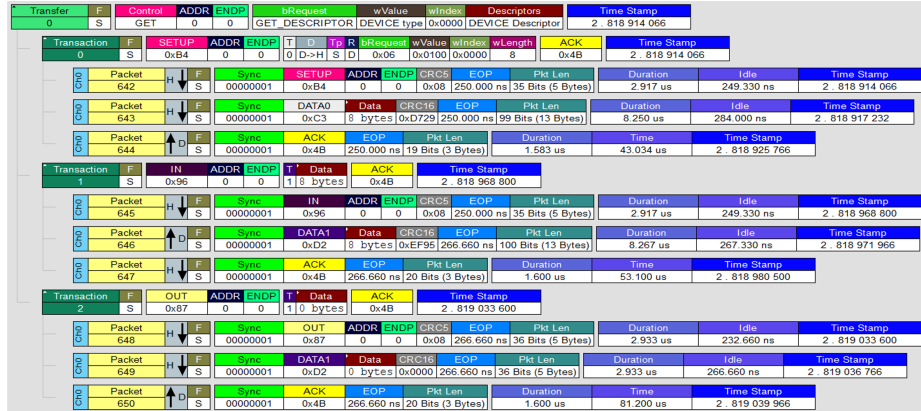


Figure 4.8: user interface view at transfer level

of the first transfer to be transmitted on the bus: it is a control transfer made up of 3 different transactions, as seen in the chapter 2.

Furthermore, the first packet to be sent is a setup packet in the OUT direction to address 0 and endpoint 0, followed by a second data packet and an acknowledge. The other two transactions are respectively IN and OUT, both at address 0 and at endpoint 0. It is, therefore, the first control transfer required to configure a new device after attachment and reset by the host. Other control transfers will follow to receive all the information necessary for the correct configuration of the device and the assignment of a unique address.

4.3 STM32CubeIDE

STM32CubeIDE is a development environment based on the C/C++ programming language designed by STMicroelectronics to implement and test applications on the STM32 priority boards. The initial graphic interface allows you to configure the microcontroller more easily, generating a default code that activates/deactivates the peripherals selected for that specific application. Furthermore, through a search window, it is possible to search quickly and effectively for the board you are using, with the possibility of selecting pre-set test examples that show the operation of the various peripherals. Once the microcontroller or core board has been selected, a new window allows me to easily configure the initial setup, activating/deactivating the peripherals and setting the various parameters. The tool also includes an advanced debugging tool, which allows, step by step, to view temporary variables, core registers, memories and all the peripherals inside the microcontroller.

From the graphic interface of the tool it was possible to configure the board as a host or as a device by selecting the appropriate peripheral from the drop-down menu and choosing one of the two configurations. This configuration is automatically

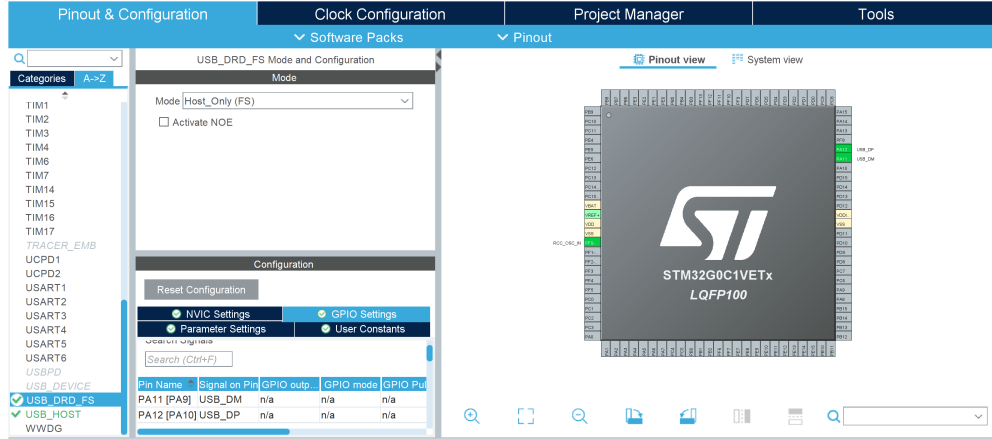


Figure 4.9: HOST mode configuration from ST32CubeIDE side

translated into code by the compiler, which will configure bit 31 of the control register (*USB_CNTR*) according to the desired configuration. After enabling the peripheral by selecting as "HOST", we will see pins PA12 and PA11 of the microcontroller in green: this means that the lines D+ and D- associated with the relative pins have been enabled correctly. Just below, a specific section called "*USB_HOST*" will be enabled, in which you can select the desired class based on the application to be designed. In our case the "Communication Host Class" will be selected.

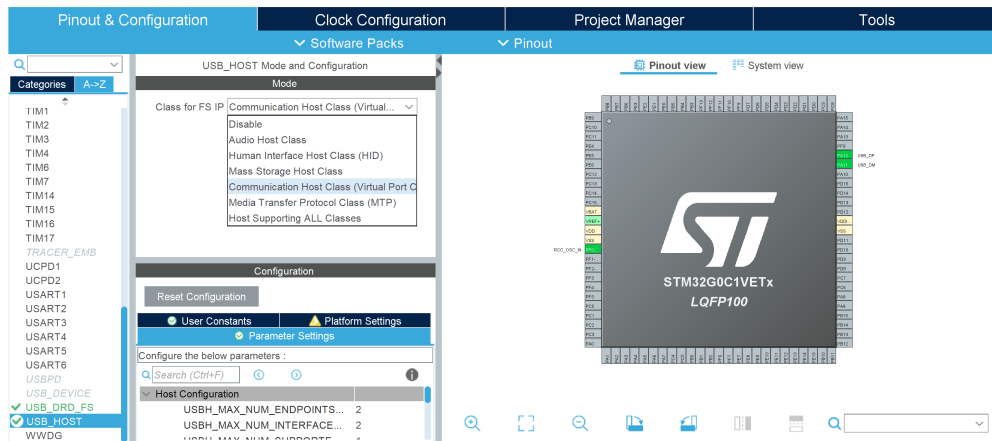


Figure 4.10: HOST mode configuration from ST32CubeIDE side

Another very important section to initially configure our system is what is called "clock configuration". Through the appropriate section, it is possible to select the desired clock source (through an internal oscillator or an external quartz oscillator) and divide/multiply the frequency through various PLLs in such a way as to give the right timing signal to the various peripherals. From Figure 4.11 it is possible to

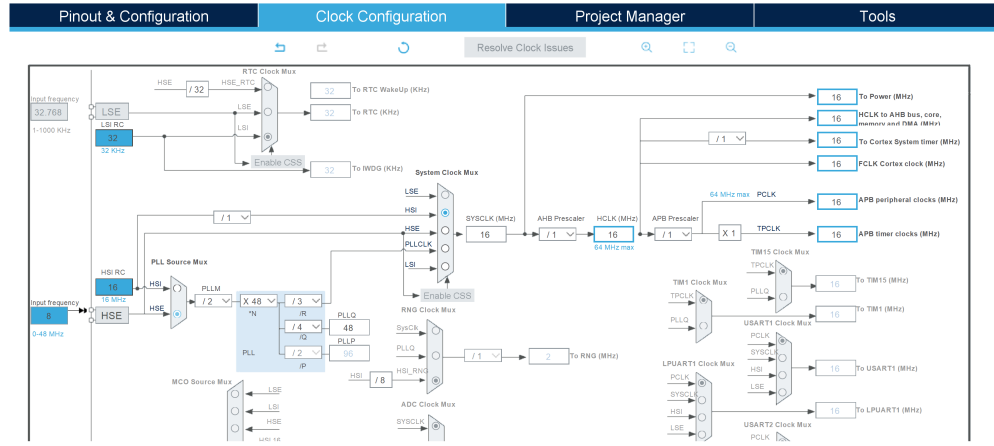


Figure 4.11: Clock configuration from ST32CubeIDE side

notice that the external quartz oscillator was selected as a clock source through the "PLL Source Mux" and subsequently various clock dividers/multipliers were used to have a frequency equal to 48 MHz in input to the PLLQ, whose output will go directly to the USB peripheral. For the other peripherals the internal oscillator is used, as they do not require as high a precision as that required by the USB peripheral. In this case, the internal 16 MHz oscillator was chosen.

Chapter 5

Software and hardware implementation

In this chapter, the hardware and software changes and implementations that were required to properly configure the initial setup will be discussed.

5.1 Hardware configuration

In order to establish a correct communication between the two boards, it was necessary to change the position of some jumpers in both boards. More specifically, from the user manual of the evaluation board, a table is shown which specifies the correct configuration of two jumpers, JP3 and JP4 respectively, in order to use the D + and D- lines correctly.

Pin	Signal	Mother board function	UCPD daughterboard function (MB1352_CN10)	How to disconnect with function block on mother board
1	PA12	USART1_RTS	C1_D+	Keep JP3 OFF
2	PA11	USART1_CTS_3V3	C1_D-	Keep JP4 OFF
3	PA12	USART1_RTS	C2_D+	Keep JP3 OFF
4	PA11	USART1_CTS_3V3	C2_D-	Keep JP4 OFF

Figure 5.1: JP3 and JP4 configuration [30]

The D+ line is connected to pin PA12 of the microcontroller, while the D- line is connected to pin PA11. Both lines are shared with the USART IP, and it is therefore necessary to remove the jumpers JP3 and JP4 in order to isolate and use the lines only with USB communication. The correct configuration of these jumpers is fundamental, as if the lines were shared, the voltage levels would be altered, making it impossible to discriminate between the various states discussed in Chapter 2. In fact, a first problem encountered during the experience was to see

an incorrect communication through the "USB Protocol Suite" protocol analyzer software, as the packets were corrupted.

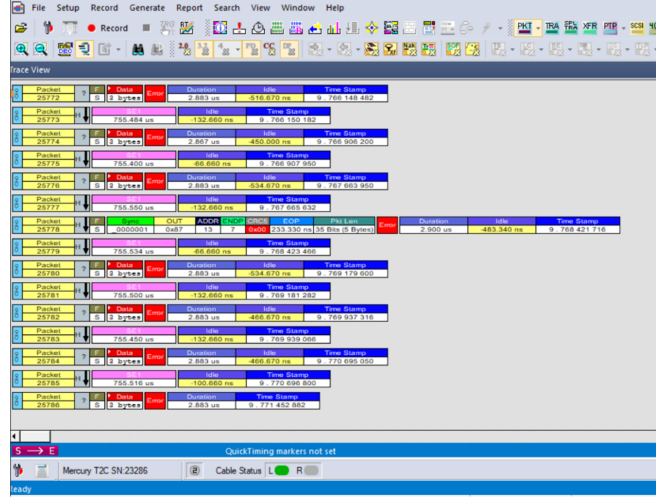


Figure 5.2: Corrupted data view

However, the communication between the two boards was apparently correct, as it was possible to see the transmission of the test string sent from the host to the device thanks to the log error reported on the LCD screen of the two boards. The problem was due to the fact that the sharing of the D+ and D- lines allowed communication between the two boards at the hardware level but altered the voltage levels, making it impossible for the protocol analyzer to decode the various packets. By removing the two jumpers JP3 and JP4 as suggested by the user manual, the graphical interface of the protocol analyzer was similar to that in Figure 4.7.

5.2 Software configurations

Before discussing the implemented implementations, it is useful to describe the structure and hierarchy at the software level of the host and device. In fact, there are many libraries with specific functions that allow to interact with individual blocks depending on the level of abstraction. The libraries implemented for both host and device part, give all the necessary functions to implement an application in which the USB 2.0 FS peripheral of the STM32G0 is involved. In order to include these libraries in the project, a preliminary action is required in which the package is downloaded according to the product line that is used. In our case the downloaded package was “STM32CubeG0 MCU Firmware Package”.

Device library organization

The device library organization is shown in Figure 5.3. At the lowest level we find the USB IP, i.e. the hardware part composed by the various registers and structures of the communication device. An intermediate interface is the one which is called "Device Hardware Abstraction Layer Driver", which includes all the various source files in which there are the functions used to access the registers of the lowest level. The initial configuration of the core takes place through access to these registers and the initialization of the bits inside them. Furthermore, this block acts as a linker between the hardware and the highest level, that is the USB device driver. The latter is made up of 2 sub-blocks: the first, which is called "USB Device Core", includes some files that offer a set of application programming interfaces to manage the main state machine, the various interrupts and modules that allow you to send out error and debug messages intended for the user. The second one includes a file called *"usbd_cdc_if.c"* which allows, through user-implementable functions, to communicate with the application and send/receive packets with a second device. The highest level is the one called "Application". It includes a set of files that wrap all the information related to the device and allow the connection of all parts of the software.

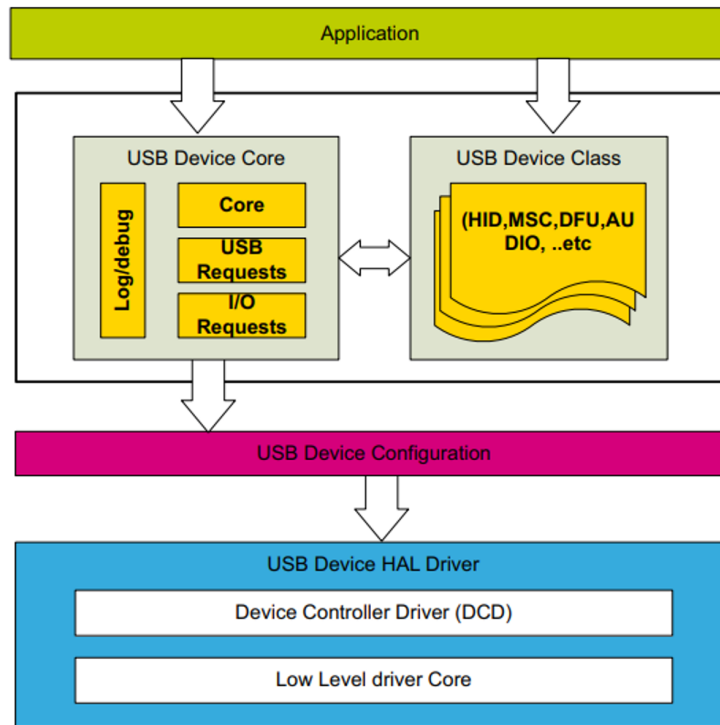


Figure 5.3: Device library structure [31]

The main structure used in the device library is the "device handle", that is a class that has variables and fundamental substructures as attributes, in order to hold together all the information related to the device.

```
typedef struct _USBD_HandleTypeDef
{
    uint8_t          id;
    uint32_t          dev_config;
    uint32_t          dev_default_config;
    uint32_t          dev_config_status;
    USBD_SpeedTypeDef dev_speed;
    USBD_EndpointTypeDef ep_in[16];
    USBD_EndpointTypeDef ep_out[16];
    __IO uint32_t      ep0_state;
    uint32_t           ep0_data_len;
    __IO uint8_t       dev_state;
    __IO uint8_t       dev_old_state;
    uint8_t            dev_address;
    uint8_t            dev_connection_status;
    uint8_t            dev_test_mode;
    uint32_t           dev_remote_wakeup;
    uint8_t            ConfIdx;

    USBD_SetupReqTypeDef request;
    USBD_DescriptorsTypeDef *pDesc;
    USBD_ClassTypeDef *pClass;
    void *pClassData;
    void *pUserData;
    void *pData;
    void *pBosDesc;
    void *pConfDesc;
} USBD_HandleTypeDef;
```

Figure 5.4: Device handler structure

Figure 5.4 shows the structure of the device handler. In the structure, there are some signals that specify the current device configuration type, manage the state machine and specify the number of IN / OUT endpoints present. In addition, there are other substructures, such as "*USBD_SetupReqTypeDef*" which allows you to manage the arrival of a SETUP packet, "*USBD_DescriptorsTypeDef*" which offers callback functions to allow the user to manage the descriptors and "*USBD_ClassTypeDef*" which allows you to select and configure the desired class.

Host library organization

The host library structure (Figure 5.5) is similar to the library of the device. The USB Host Library consists of two main parts: the host core and the host class drivers. The first includes a set of APIs that are called by the user through the application layer and the state machine that manages the connection/disconnection of the device, the enumeration process regardless of the type of class and the transfer of packets during communication with the device. The second includes, in addition to the API, a class handler that is called by the host state machine to

deal with some configurations related to the class (initialization, de-initialization, process).

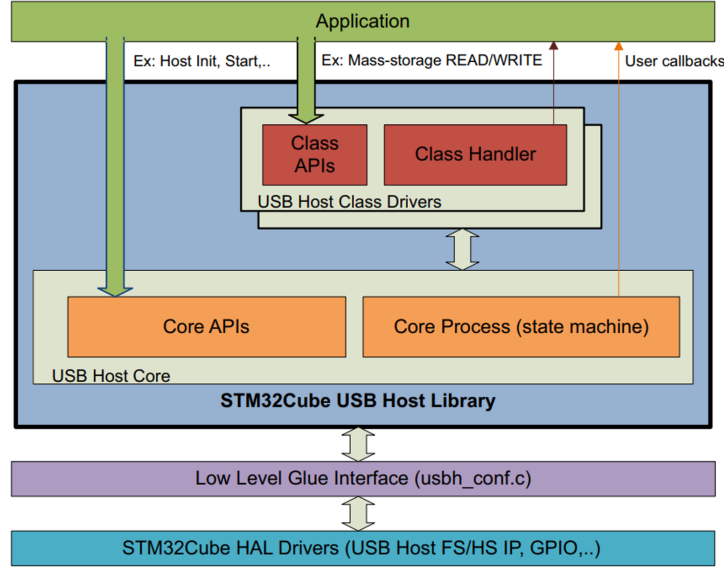


Figure 5.5: Host library structure [32]

As for the device library, also for the host part there is a class called “USBH HandleTypeDef” (Figure 5.6), which is able to wrap all the host configuration parameter inside an unique object.

```
typedef struct _USBH_HandleTypeDef
{
    __IO HOST_StateTypeDef    gState;          /* Host State Machine Value */
    ENUM_StateTypeDef        EnumState;        /* Enumeration state Machine */
    CMD_StateTypeDef         RequestState;
    USBH_CtrlTypeDef         Control;
    USBH_DeviceTypeDef        device;
    USBH_ClassTypeDef        *pClass[USBH_MAX_NUM_SUPPORTED_CLASS];
    USBH_ClassTypeDef        *pActiveClass;
    uint32_t                  ClassNumber;
    uint32_t                  Pipes[16];
    __IO uint32_t             Timer;
    uint32_t                  Timeout;
    uint8_t                   id;
    void                      *pData;
    void (* pUser)(struct _USBH_HandleTypeDef *pHandle, uint8_t id);
} USBH_HandleTypeDef;
```

Figure 5.6: Host handler structure

The "gState" object is part of the "*HOST_StateTypeDef*" class and corresponds to the current value of the host state machine. Therefore, the state machine will always be active in the background and the gState signal will contain the current

state in which the connection between host and device is located. The “EnumState” structure, on the other hand, provides the current state in the state machine destined for the enumeration process. Finally, “RequestState”, provides the current status of a control request between IDLE, SEND or WAIT.

STM32CubeIDE configurations

At the software level, some functions have been implemented, starting from the configuration of the LCD screen. In order to correctly use the display, it was first necessary to include the board support package (BSP) library. Inside there are many functions to initialize and manage the LCD screen, selecting the size of the font to be printed, the row and column and the color of the text. Among the many functions, the ones that were used most for this project were:

- *BSP_LCD_Init()*: enables and initializes the LCD screen;
- *BSP_LCD_SetFont()*: receives as parameter the size of the text to be printed on the screen;
- *BSP_LCD_SetBackColor()*: receives as a parameter the background color to be set (white by default);
- *BSP_LCD_Clear()*: receives the background color as a parameter and deletes all the writings and/or figures that are printed on the screen, while maintaining the last font set;
- *BSP_LCD_SetTextColor()*: receives as a parameter the color of the text you want;
- *BSP_LCD_DisplayStringAt()*: receives 4 parameters, respectively the column index, the row index, the text to be printed and the position (left mode, center mode, right mode).

In the main source code (top layer), that is the "main.c", after enabling the HAL functions and configuring the system clock, the initialization preset function was called first and then a function implemented ad hoc to print to screen the title of the project in red, the type of board implemented (host or device) and the corporate copyright in blue.

After launching and executing the *My_LCD_Setting()* function, the LCD screen of the host and device boards appears as in Figure .

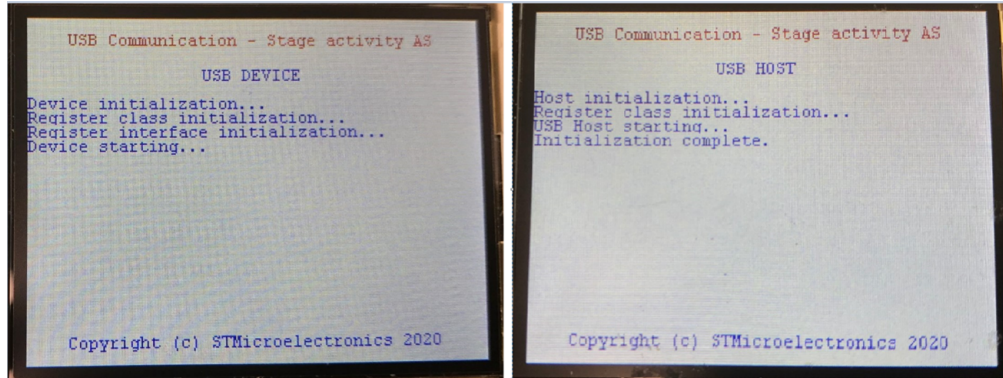


Figure 5.7: Log display after turning on the boards

Within the infinite while loop, there is the *MX_USB_HOST_Process()* function which contains the *USBH_Process()* function. The latter represents the state machine that is called continuously to perform many actions depending on the state and receives the object "hUsbHostFS" by reference as a parameter; it belongs to the *USBH_HandleTypeDef* class, that is the host handler that contains all the parameters and characteristics of the host. The *USBH_Process()* function consists of a “switch” case structure in which the comparison parameter is the gState signal. The entire structure, therefore, can be schematized with the Figure 5.8 . All the states have been analyzed and many print commands have been added to print all the flow on the LCD screen, such as device attachment, choice of class and enumeration.

Another function within the infinite while loop is the one implemented specifically to transmit a string from the host to the device via a BULK transfer. After finishing the flow that leads to the enumeration of the device, the *Transmit()* function is called. If the application is ready to make a transmission, we proceed with the deletion of all the written messages already printed on the screen regarding the correct connection between host and device; the initialization strings of the LCD screen are printed again, regarding the name of the project, the type of device implemented and the corporate copyright. In addition, instructions are printed for the user, who has the option of increasing or decreasing the number to be transmitted to the device by 1 or 10 units. Therefore, through the *BSP_JOY_GetState()* function, it can be acquired the direction of the joystick button that has been pressed.

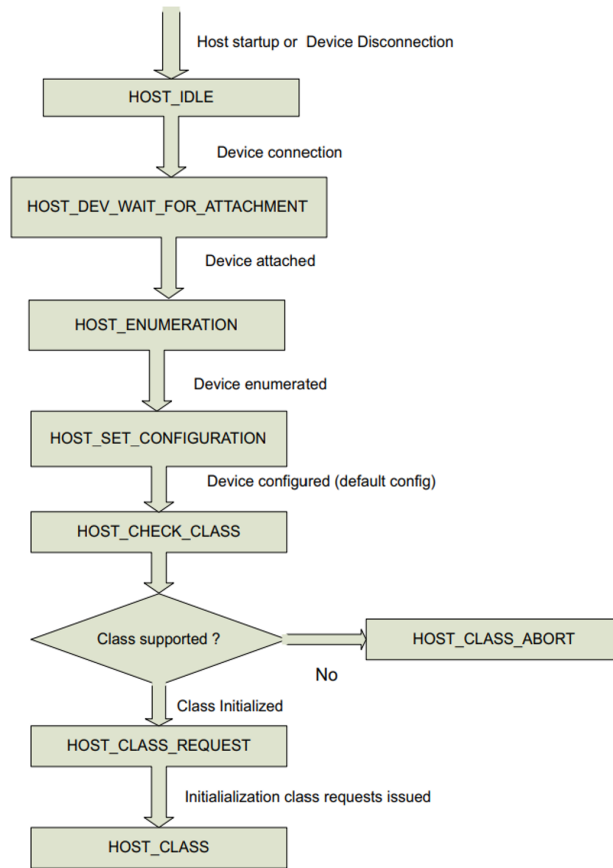


Figure 5.8: Host state machine [33]

At this point 5 alternatives are possible:

- If the UP button (coded with the number 4) is pressed, the number to be transmitted is increased by 1 unit;
- If the DOWN key (coded with the number 1) is pressed, the number to be transmitted is decreased by 1 unit;
- If the RIGHT key (coded with the number 3) is pressed, the number to be transmitted is increased by 10 units;
- If the LEFT key (coded with the number 2) is pressed, the number to be transmitted is decreased by 10 units;
- If no key is pressed, the number to be transmitted always remains the same and is neither increased nor decreased.


```

void Transmit(void)
{
    uint8_t joyState = 0;
    if(Appli_state == APPLICATION_READY)
    {
        if(flag_LCD == 1)
        {
            BSP_LCD_Clear(LCD_COLOR_WHITE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKRED);
            BSP_LCD_DisplayStringAt(0, 10, (uint8_t *)"USB Communication - Stage activity AS", CENTER_MODE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
            BSP_LCD_DisplayStringAt(0, 35, (uint8_t *)"USB HOST", CENTER_MODE);
            BSP_LCD_DisplayStringAt(0, BSP_LCD_GetYSize()- 20, (uint8_t *)"Copyright (c) STMicroelectronics 2020", CENTER_MODE);
            BSP_LCD_DisplayStringAt(0, 55, (uint8_t *)"Push the joystick to increase or decrease the", LEFT_MODE);
            BSP_LCD_DisplayStringAt(0, 65, (uint8_t *)"number to transmit.", LEFT_MODE);
            BSP_LCD_DisplayStringAt(0, 85, (uint8_t *)"Sending to device the number:", LEFT_MODE);
            flag_LCD++;
        }

        joyState = BSP_JOY_GetState();
        if(joyState == 4)
        {
            i++;
            itoa(i,prova,10);
            USBH_CDC_Transmit(&hUsbHostFS,prova,16);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKMAGENTA);
            BSP_LCD_DisplayStringAt(0, 105, (uint8_t *)prova, CENTER_MODE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
        }

        if(joyState == 2)
        {
            i--;
            itoa(i,prova,10);
            USBH_CDC_Transmit(&hUsbHostFS,prova,16);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKMAGENTA);
            BSP_LCD_DisplayStringAt(0, 105, (uint8_t *)prova, CENTER_MODE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
        }

        if(joyState == 3)
        {
            i=i+10;
            itoa(i,prova,10);
            USBH_CDC_Transmit(&hUsbHostFS,prova,16);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKMAGENTA);
            BSP_LCD_DisplayStringAt(0, 105, (uint8_t *)prova, CENTER_MODE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
        }

        if(joyState == 1)
        {
            i=i-10;
            itoa(i,prova,10);
            USBH_CDC_Transmit(&hUsbHostFS,prova,16);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKMAGENTA);
            BSP_LCD_DisplayStringAt(0, 105, (uint8_t *)prova, CENTER_MODE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
        }

        else
        {
            itoa(i,prova,10);
            USBH_CDC_Transmit(&hUsbHostFS,prova,16);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKMAGENTA);
            BSP_LCD_DisplayStringAt(0, 105, (uint8_t *)prova, CENTER_MODE);
            BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
        }
    }
}

```

Figure 5.9: Transmit function that has been implemented

The function used to transmit the number via a BULK transfer is called "*USBH CDC Transmit()*" and accepts the host handler, a string and the length of the string as input parameters. Therefore, before transmitting the number, it was necessary to convert it into a string using the *itoa()* function, which accepts as parameters the number to be transmitted, the buffer in which to insert the converted number and the conversion base. As for the clock configuration, the external HSE oscillator present on the board has been selected for both the host and the device, in order

to increase the performance and stability of the system.

On the device side, a function similar to that of the host has been implemented for the initialization of the LCD screen and the printing of the initial configuration strings. However, in the device code it was necessary to add a function within the code capable of receiving the input string and printing it on the LCD screen. The *USBD_CDC_SetRxBuffer()* function is called whenever a new packet is

```
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
    /* USER CODE BEGIN 6 */
    USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
    USBD_CDC_ReceivePacket(&hUsbDeviceFS);
    BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
    BSP_LCD_DisplayStringAt(0, 95, (uint8_t *)"Receiving...", LEFT_MODE);
    BSP_LCD_SetTextColor(LCD_COLOR_DARKMAGENTA);
    BSP_LCD_DisplayStringAt(0, 135, (uint8_t *)Buf, CENTER_MODE);
    return (USBD_OK);
    /* USER CODE END 6 */
}
```

Figure 5.10: Receive function that has been implemented

sent from the host to the device and cannot be called again before the end of the reception for not having overwriting. Once the data has been processed, the *USBD_CDC_ReceivePacket()* function is used to indicate that the buffer can be used again to receive the next packet.

Chapter 6

Performance test, results and conclusions

6.1 Evaluation of the maximum bandwidth

The main purpose of this thesis project is to evaluate the performance of the digital USB IP designed by STMicroelectronics by evaluating the maximum bandwidth. In the case of USB FS, the data transfer rate is 12 Mbit/s. This means that every second, theoretically, 12.000.000 bits can be transferred from one device to another via USB cable. When you want to do an analysis on the bandwidth used, it is better to refer to a frame, or a time interval equal to 1 ms which corresponds to having two consecutive SOFs. So, in this case, the data transfer rate is 12.000 bits/ms. To evaluate the performance of USB2 IP by STMicroelectronics, we must refer to Figure 6.1, which indicates the maximum number of possible BULK transfers within a single frame, based on the size of the data we want to send (Data Payload). The maximum number of bytes that can be had within a frame is equal to $12.000 \text{ bits} : 8 = 1500$ bytes, which corresponds to the maximum bandwidth if we only had the useful information to transmit. In this case we would therefore have $1500 : 64 = 23$ bulk transfers in a single frame. However, we must consider the presence of the protocol overhead, which in the case of BULK transfers is 13 bytes for each transaction. The maximum transfers are reduced to a value equal to $1500 : (64 + 13) = 19$ transfers. Therefore, $13 \cdot 19 = 247$ bytes will be the bytes due to the overhead and the useful bytes that we can send will be $(64 \cdot 19) + 247 = 1463$ bytes compared to the theoretical 1500.

However, we must also consider the presence of the forbidden window, which is a time window within each frame within which it is not possible to start new transactions. Its purpose is to avoid the start of a new transaction near the SOF which indicates the end of the frame. In the latter case, in fact, the transmission

Protocol Overhead (13 bytes)		(3 SYNC bytes, 3 PID bytes, 2 Endpoint + CRC bytes, 2 CRC bytes, and a 3-byte interpacket delay)			
Data Payload	Max Bandwidth (bytes/second)	Frame Bandwidth per Transfer	Max Transfers	Bytes Remaining	Bytes/Frame Useful Data
1	107000	1%	107	2	107
2	200000	1%	100	0	200
4	352000	1%	88	4	352
8	568000	1%	71	9	568
16	816000	2%	51	21	816
32	1056000	3%	33	15	1056
64	1216000	5%	19	37	1216
Max	1500000				1500

Figure 6.1: Full-speed BULK transaction limits

would be incomplete and could cause errors with subsequent transmissions. Taking into consideration what has been said so far, it is considerable to think that the size of the forbidden window is equal to that of a BULK transfer, i.e. 77 bytes. From the VHDL code written by *STMicroelectronics* digital hardware design engineers, we see that the limit has been configured to 848 bits, which corresponds to 106 bytes. This choice was made considering also other factors that affect the

```

01 architecture rtl of usb2_frame_timer is
02
03   -- Number of bits passed after the forecasted SOF event when an ESOE event
04   -- is notified to the system
05   constant esof_limit : integer := 48;
06   constant eof4_limit : integer := 0;
07   constant eof3_fs_limit : integer := 848;

```

Figure 6.2: Maximum number of bits for the forbidden window

delays that may occur and therefore it is a value with which we can consider ourselves safe. Therefore, also considering the forbidden window we will have a maximum number of transfers equal to 18, which correspond to a maximum achievable band of 92.93%. In fact, $1500 - 106 = 1394$ bytes that can be in a frame. Now dividing 1394 by 77 bytes we get 18.10 maximum possible transfers. It should be noted that the number of transfers is not an integer: the contribution of

0.10 transfers is equivalent to 8 bytes that can be transmitted before the start of forbidden windows. So it could theoretically start a last bulk transfer just before the beginning of the forbidden window and end inside it, before having the end of the frame. In this case, the percentage increase to a maximum of 98

6.2 First test

After configuring the hardware and software of the two evaluation boards, and after setting up the setup with the protocol analyzer (Figure 4.6), the correctness of the communication was verified both through the LCD screen and through the graphical interface of the protocol analyzer. After turning on the two boards, having powered them through a USB Micro-B cable and having connected them through a USB Type-C cable, the LCD screen showed all the strings related to the initialization of the boards. Subsequently, communication began by transferring a data string with a size of 4 bytes inside which the number "1" is contained. In fact, the implemented code described in Chapter 5.2 (STM32CubeIDE configurations), allows you to increase or decrease the number to be transmitted based on the joystick button pressed, starting from number "1".

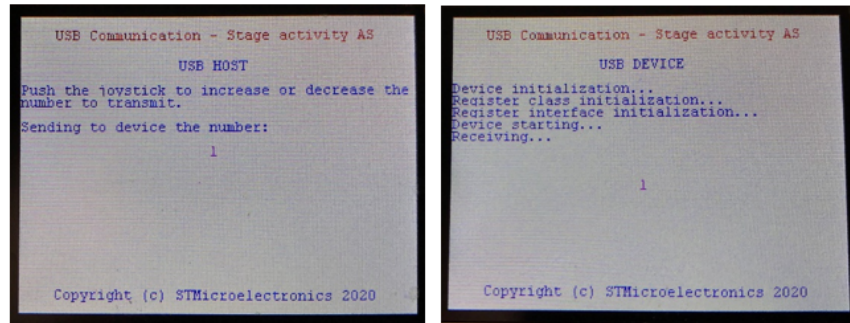


Figure 6.3: Log on host and device display for the first test

Figure 6.3 shows how the number is actually received in the correct way by both boards, showing a correct communication between the host board and the device board. However, it would be of fundamental importance to have practical feedback through the protocol analyzer. By opening the USB Protocol Suite software and starting the recording of USB 2.0 FS data traffic, it is possible to see the transmission of 3 packets repeatedly: the first two packets are in the OUT direction, while the third is in the IN direction. Analyzing it more carefully, it can be seen that the first packet is the one relating to the OUT token packet, the second is the DATA1 data packet, while the third is the ACK handshake packet. This means that many OUT transactions are taking place from the host to the device,

as expected. Also, these transactions are sent repeatedly, as the implemented code relies on a "Transmit()" function within an infinite while loop. This means that every time the software enters the while loop, a number is sent from the host to the device. If no button is pressed, the number sent will always be the same as the previous one.

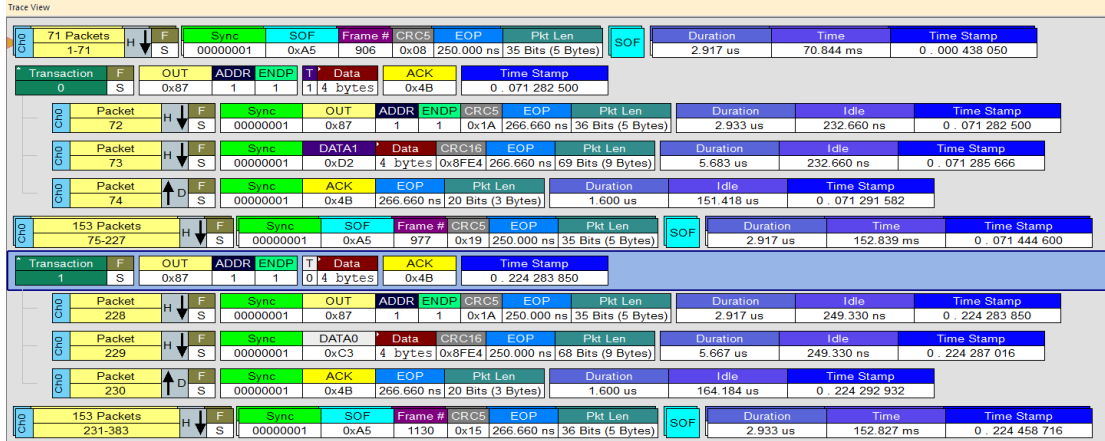


Figure 6.4: Transaction and packet level view for the first test

Raising the level of abstraction of the graphical interface, it is possible to notice that it is a set of BULK transfers, as expected; the chosen class is used to transfer a large amount of data in a non-periodic way and without time constraints, therefore the BULK transfer is the preferred one in these cases. By right-clicking on "Data" label and then on "View data block", you can view the content of the 16-bit data that the host sends to the device. This number is displayed in hexadecimal and is equal to "31", which exactly corresponds to the number "1" according to the ASCII code, which is the number transmitted by the host and displayed on the LCD screen.

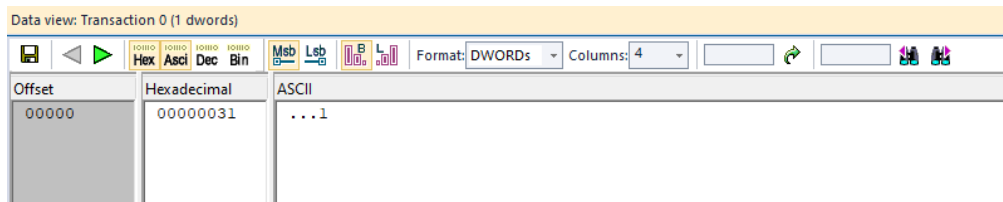


Figure 6.5: Data view of the first transaction

Another very interesting feature is that of being able to view the waveforms of the transmitted packets by right clicking on the packet number and selecting "show raw bits". In this case, therefore, we will have a graphic view of all the bits that

form the various fields of the packet, according to the NRZI encoding used by the USB protocol.

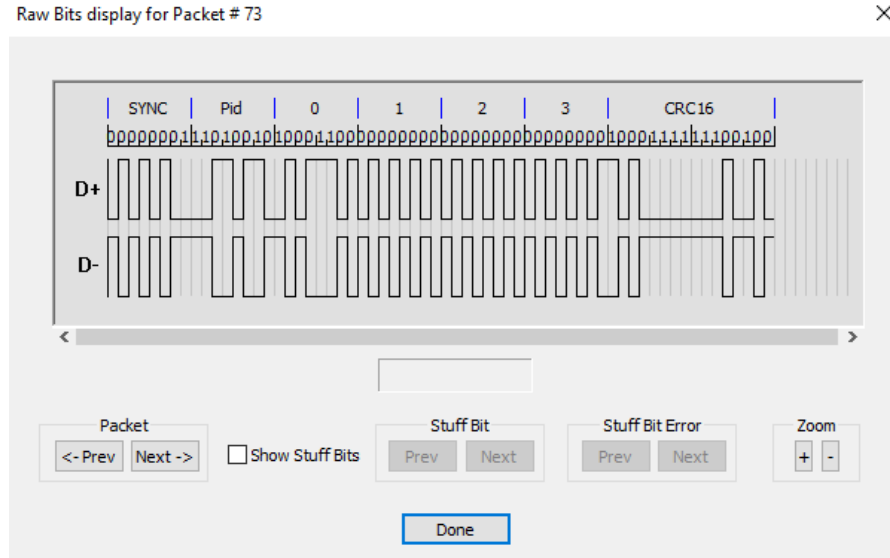


Figure 6.6: Waveforms view for Packet 73 (first DATA1)

Used bandwidth evaluation

As described in Chapter 3, the host frame scheduler is a hardware module that allows you to manage requests on the bus based on the priority assigned to the transfers. The HFS divides a 1 ms full-speed frame into 3 different parts, which are called "windows" and are managed by the same scheduler:

- Periodic service windows: within this window, only periodic type transfers are managed such as isochronous and interrupt;
- Non-periodic service window: non-periodic transfers such as bulk and control are managed within this window.
- Black security window: this is also called "*forbidden window*", as no transfer can be start but it is a necessary window for the host to prepare to send the next SOF. Therefore, in this window no transfer can start, but it is only possible to finish those started just before the start of this window.

Therefore, at the beginning of each frame, the host initially considers all the isochronous and interrupt transfers that are associated with periodic endpoints, then manages the window reserved for non-periodic transfers and subsequently posts all other requests to the frame later, because of the presence of the black

security window. Due to the overhead present within a single packet (SYNC, PID, CRC, ...) and due to the forbidden window, the useful bandwidth used will be less than the theoretical one equal to 12 Mbps, as explained before. There is also a delay, called "inter-packet delay", which would be a time necessary for the hardware to pass from one transaction to the next. This delay represents idles and reduces the maximum bandwidth that the device is able to reach. In order to evaluate the actual bandwidth usage for this test, it can always used the USB Protocol Suite tool by clicking on the "Bus utilization" button. Then the software needs to know and select the interval in which you want to measure the bandwidth, by entering markers or by entering the number of the start packet and the number of the end packet. By inserting two consecutive SOFs as start and end packets, the bandwidth can be calculated by clicking the "Calculate" button.

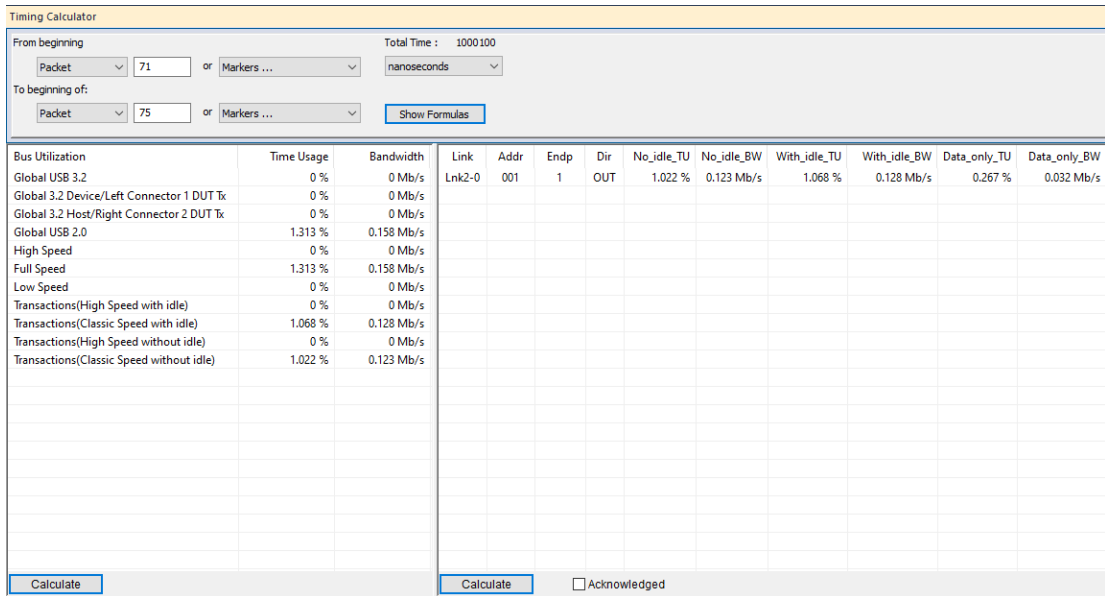


Figure 6.7: Bandwidth evaluation of the first test

The bandwidth usage is very low and equal to 0.158 Mbps. This means that within a frame (between two SOFs), only one bulk transmission is taking place and all the remaining time is "wasted" remaining in the IDLE state, as only SOFs are sent in order to avoid to send the USB device in the suspend state.

6.3 Second test

This second test was implemented with the aim of increasing the USB data traffic exchanged between the two boards and therefore also increasing the used bandwidth. In a first attempt, the main functions of the previous code have been maintained, while the size of the buffer to be transmitted from the host to the device has been increased. In this case the number "1" has been replaced with a very large string containing the character "A". The "Transmit()" function in the previous code, accept a buffer with a maximum size of 2^{16} bytes as a parameter. However, the new string used for this code is much higher than 4 bytes: this forces the software to break the string into smaller strings with a size of 4 bytes and perform the "Transmit()" function several times to be able to send all the bytes. In this case, on the software analyzer tool, it is possible to notice the presence of many NAKs. The presence of these NAKs is due to the presence of the "DisplayString" function

Ch0	Packet 1	H↓	F	S	Sync	OUT	ADDR	ENDP	CRC5	EOP	Pkt Len	Duration	Idle	Time Stamp
					00000001	0x87	1	1	0x1A	250.000 ns	35 Bits (5 Bytes)	2.917 us	249.330 ns	0.000 329 916
Ch0	Packet 2	H↓	F	S	Sync	DATA1	Data	CRC16	EOP	Pkt Len	Duration	Idle	Time Stamp	
					00000001	0xD2	16 bytes	0x3A4E	250.000 ns	163 Bits (21 Bytes)	13.583 us	266.660 ns	0.000 333 082	
Ch0	Packet 3	↑D	F	S	Sync	NAK	EOP	Pkt Len	Duration	Idle	Time Stamp			
					00000001	0x5A	266.660 ns	20 Bits (3 Bytes)	1.600 us	11.418 us	0.000 346 932			
Ch0	Packet 4	H↓	F	S	Sync	OUT	ADDR	ENDP	CRC5	EOP	Pkt Len	Duration	Idle	Time Stamp
					00000001	0x87	1	1	0x1A	250.000 ns	35 Bits (5 Bytes)	2.917 us	249.330 ns	0.000 359 950
Ch0	Packet 5	H↓	F	S	Sync	DATA1	Data	CRC16	EOP	Pkt Len	Duration	Idle	Time Stamp	
					00000001	0xD2	16 bytes	0x3A4E	266.660 ns	164 Bits (21 Bytes)	13.600 us	266.000 ns	0.000 363 116	
Ch0	Packet 6	↑D	F	S	Sync	NAK	EOP	Pkt Len	Duration	Idle	Time Stamp			
					00000001	0x5A	250.000 ns	19 Bits (3 Bytes)	1.583 us	11.517 us	0.000 376 982			
Ch0	Packet 7	H↓	F	S	Sync	OUT	ADDR	ENDP	CRC5	EOP	Pkt Len	Duration	Idle	Time Stamp
					00000001	0x87	1	1	0x1A	250.000 ns	35 Bits (5 Bytes)	2.917 us	251.330 ns	0.000 390 082
Ch0	Packet 8	H↓	F	S	Sync	DATA1	Data	CRC16	EOP	Pkt Len	Duration	Idle	Time Stamp	
					00000001	0xD2	16 bytes	0x3A4E	250.000 ns	163 Bits (21 Bytes)	13.583 us	266.660 ns	0.000 393 250	
Ch0	Packet 9	↑D	F	S	Sync	NAK	EOP	Pkt Len	Duration	Idle	Time Stamp			
					00000001	0x5A	266.660 ns	20 Bits (3 Bytes)	1.600 us	11.500 us	0.000 407 100			
Ch0	Packet 10	H↓	F	S	Sync	OUT	ADDR	ENDP	CRC5	EOP	Pkt Len	Duration	Idle	Time Stamp
					00000001	0x87	1	1	0x1A	250.000 ns	35 Bits (5 Bytes)	2.917 us	249.330 ns	0.000 420 200
Ch0	Packet 11	H↓	F	S	Sync	DATA1	Data	CRC16	EOP	Pkt Len	Duration	Idle	Time Stamp	
					00000001	0xD2	16 bytes	0x3A4E	250.000 ns	163 Bits (21 Bytes)	13.583 us	282.660 ns	0.000 423 366	
Ch0	Packet 12	↑D	F	S	Sync	NAK	EOP	Pkt Len	Duration	Idle	Time Stamp			
					00000001	0x5A	250.000 ns	19 Bits (3 Bytes)	1.583 us	11.517 us	0.000 437 232			

Figure 6.8: Presence of multiple NAK

inside the "ReceiveFS" function which prints the string received from the device. Since the string is much larger this time, the microcontroller is busy printing it all and wastes a lot of time before printing the next string. Therefore, the device will send a NAK handshake to the board host to inform it that it is busy and to try to send the string again.

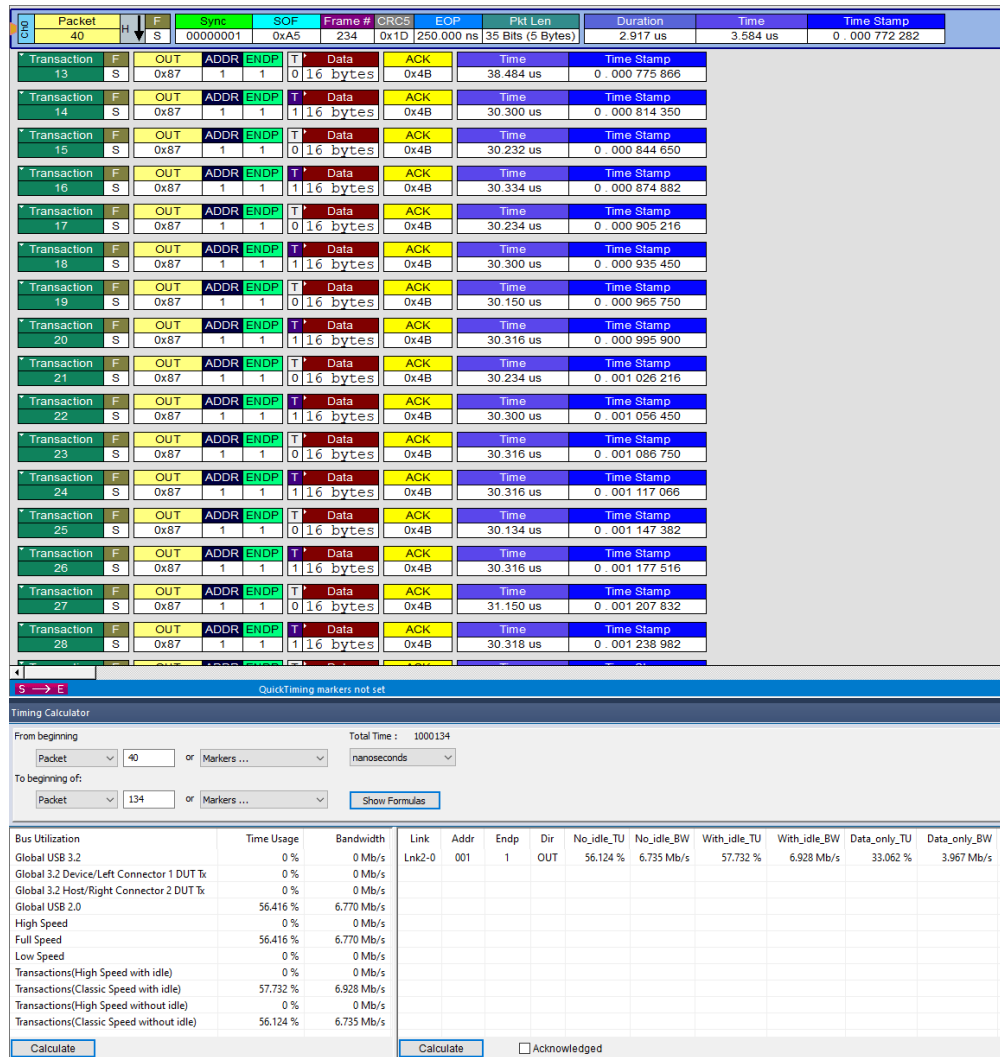


Figure 6.9: Results and bandwidth evaluation of second test

By modifying the code and removing the print string, we get the expected result, that is the reception of many more transactions than before. Furthermore, the bandwidth used is much higher than before, reaching a maximum peak of 6.928 Mbps between packet 40 (first SOF) and packet 134 (subsequent SOF).

6.4 Third test

To further increase the percentage of bandwidth usage, it is necessary to enable the double buffer function already described in the subsection 3.2.1, in order to use both buffers of EP1 only in the OUT direction. For greater debugging capacity, the

string of test 2 has been modified with a size of 512 bytes containing a sequence of 8 alternating characters starting from the letter A and ending at the letter R. In fact, due to how the feature was designed, it is activated only when the size of the data to be sent (512 bytes) is greater than the maximum size of the BULK transfer data (64 bytes in FS). In this way, the string is split into multiple packets and 8 transactions of 64 bytes each are sent. Furthermore, to make the analysis more interesting and understand how the two devices behave according to different situations, various tests were carried out, first enabling the double buffer only for the DEVICE, then enabling it only for the HOST and finally enabling it for both boards.

Double buffer enabled on host/device side

To enable the double buffer on the host side, it was necessary to make some changes to the Test 2 code, starting from the size and content of the string to be sent. In fact, as already mentioned in the introductory part of this chapter, to enable the double buffer it is necessary that the size of the data to be sent exceeds the maximum size of the transfer, so that software and hardware work on two different packets to transmit/receive the data. Subsequently, it was necessary to initialize the double buffer in the file *USB_Host > Target > "usbh_conf.c"* by changing the *"bulk_doublebuffer_enable"* signal present on line 197 from "DISABLE" (0U) to "ENABLE" (1U) To enable the double buffer on the device side, it is necessary

```
197 | hhcd_USB_DRD_FS.Init.bulk_doublebuffer_enable = ENABLE;  
198 | hhcd_USB_DRD_FS.Init.iso_singlebuffer_enable = DISABLE;
```

Figure 6.10: Double buffer bit enable

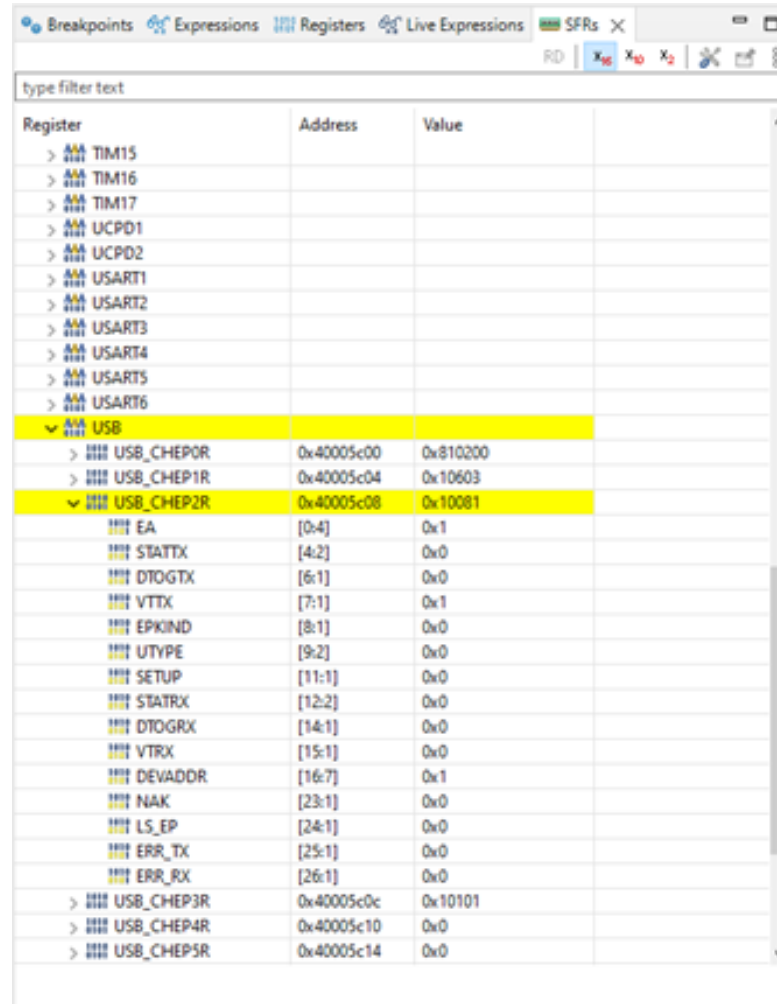
to enable the same bit within the device code. Furthermore, the two buffers of the endpoint involved in the communication inside the packet memory area must be unidirectional (both RX in the case of OUT transaction). So, you have to change *"PCD_SNG_BUF"* to *"PCD_DBL_BUF"* in line 462/463. However,

```
462 | HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , 0x81 , PCD_DBL_BUF, 0xC0);  
463 | HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , 0x01 , PCD_DBL_BUF, 0x110);  
464 | HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , 0x82 , PCD_SNG_BUF, 0x100);
```

Figure 6.11: Double buffer

the percentage of bandwidth used, evaluated by the protocol analyzer, is unchanged. To analyze the problem and try to solve it, a debugging session was carried out. A breakpoint was positioned near the "Transmit()" function and within the register of the endpoint involved in the OUT transaction, it was realized that the UTYPE

signal was correctly "00" (BULK TYPE), but the EPKIND signal turned out to be '0', while from the USB IP specification it should be '1' to enable double buffer.



Register	Address	Value
> TIM15		
> TIM16		
> TIM17		
> UCPD1		
> UCPD2		
> USART1		
> USART2		
> USART3		
> USART4		
> USART5		
> USART6		
▼ USB		
> USB_CHEP0R	0x40005c00	0x810200
> USB_CHEP1R	0x40005c04	0x10603
▼ USB_CHEP2R	0x40005c08	0x10081
EA	[0:4]	0x1
STATTX	[4:2]	0x0
DTOGTX	[6:1]	0x0
VTTX	[7:1]	0x1
EPKIND	[8:1]	0x0
UTYPE	[9:2]	0x0
SETUP	[11:1]	0x0
STATRX	[12:2]	0x0
DTOGRX	[14:1]	0x0
VTRX	[15:1]	0x0
DEVADDR	[16:7]	0x1
NAK	[23:1]	0x0
LS_EP	[24:1]	0x0
ERR_TX	[25:1]	0x0
ERR_RX	[26:1]	0x0
> USB_CHEP3R	0x40005c0c	0x10101
> USB_CHEP4R	0x40005c10	0x0
> USB_CHEP5R	0x40005c14	0x0

Figure 6.12: Debug session: EPKIND bit is not enabled

To identify the problem, it can be started by analyzing the *CDC_Transmit()* function inside the "Transmit()" function in the file *USB_HOST > App > "usb_host.c"*. Internally, this function initializes parameters such as the pointer that scans the vector of letters in the buffer, the size of the buffer and the transmission status. The status "wake up" the *CDC_ProcessTransmission()* function which is responsible for sending the data to the device. Inside we find a switch case which is described in Figure 6.13.

```

658 static void CDC_ProcessTransmission(USBH_HandleTypeDef *phost)
659 {
660     CDC_HandleTypeDef *CDC_Handle = (CDC_HandleTypeDef *) phost->pActiveClass->pData;
661     USBH_URBStateTypeDef URB_Status = USBH_URB_IDLE;
662
663     switch (CDC_Handle->data_tx_state)
664     {
665     case CDC_SEND_DATA:
666         if (CDC_Handle->TxDataLength > CDC_Handle->DataItf.OutEpSize)
667         {
668             (void)USBH_BulkSendData(phost,
669                                     CDC_Handle->pTxData,
670                                     CDC_Handle->DataItf.OutEpSize,
671                                     CDC_Handle->DataItf.OutPipe,
672                                     1U);
673             len_db = CDC_Handle->TxDataLength; //added by AS
674         }
675         else
676         {
677             (void)USBH_BulkSendData(phost,
678                                     CDC_Handle->pTxData,
679                                     CDC_Handle->TxDataLength,
680                                     CDC_Handle->DataItf.OutPipe,
681                                     1U);
682         }
683         CDC_Handle->data_tx_state = CDC_SEND_DATA_WAIT;
684         break;
685     }

```

Figure 6.13: Process Transmission function

Therefore, if we are in "SEND_DATA" state, we must distinguish two cases: the first in which the size of the data to be sent is greater than the maximum size of 64 bytes (in this case the size that is passed to the next called function is equal to 64) and the second case in which the size of the data to be sent is less than 64 bytes (in this case the current size is passed). In any case, the "*USBH_BulkSendData()*" function is called, which is the fundamental one to start the bulk transaction. Going to open all the other functions that are called in "matryoshka" way, we finally arrive at a function called "*HAL_HCD_HC_SubmitRequest()*" in which the index DATA0/DATA1 is toggled. Before terminating this last function, various parameters are assigned to the channel involved in the OUT transmission and then the function to start the transfer is called, which takes the name of "*USB_HC_BULK_DB_StartXfer()*" inside Drivers > *STM32G0xx_HAL_Driver* > "*stm32g0xx_II_usb.c*". The first instruction that is executed within this function is a comparison between the size of the transfer and the maximum size of the BULK, as can be seen in Figure 6.14. So

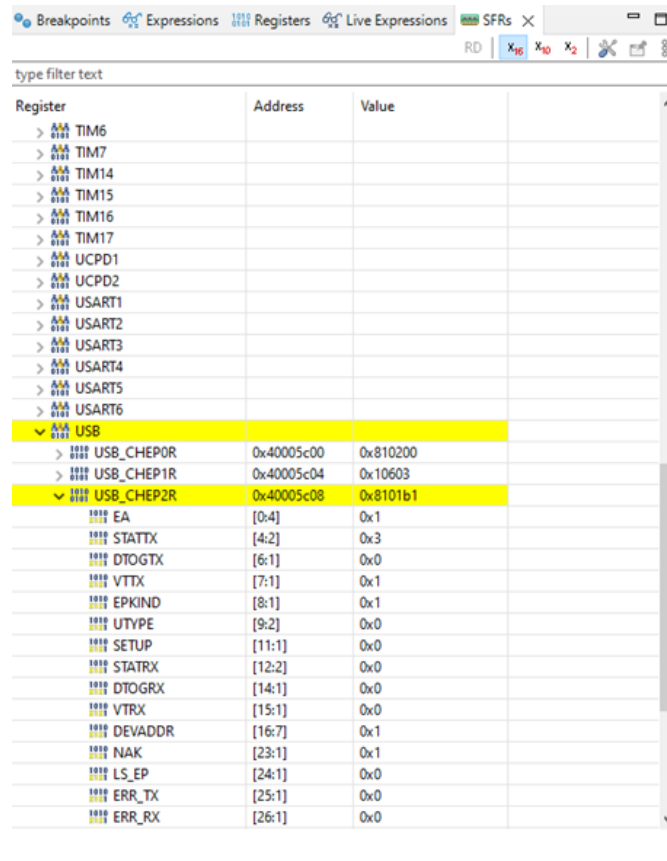
```

1258 static HAL_StatusTypeDef USB_HC_BULK_DB_StartXfer(USB_DRD_TypeDef *USBx,
1259                                                     USB_DRD_HCTypeDef *hc,
1260                                                     uint32_t ch_reg,
1261                                                     uint32_t *len)
1262 {
1263     uint32_t phy_ch_num = (uint32_t)hc->phy_ch_num;
1264
1265     /* -Double Buffer Mangement- */
1266     if (hc->xfer_len_db > hc->max_packet)
1267     {
1268         /* enable double buffer mode */
1269         (void)USB_HC_DoubleBuffer(USBx, (uint8_t)phy_ch_num, USB_DRD_BULK_DBUFF_ENBALE);
1270         *len = hc->max_packet;
1271         hc->xfer_len_db -= *len;

```

Figure 6.14: Function used to start the transfer using double buffer feature

if we want to send a string with a size greater than 64 bytes, the double buffer is enabled and the length of the transfer of 128 bytes is decreased each time (64 bytes for each buffer), so as to know when the transfer is finished and all bytes have been sent. The problem encountered is that the `"xfer_len_db"` parameter is assigned in `HAL_HCD_HC_SubmitRequest()` and in the original code corresponds to the maximum size of the BULK, while it should be the size of the entire transfer. In fact, the EPKIND bit was never enabled because the condition is never verified and the double buffer is never enabled. The solution implemented to try to solve the problem was to store the size of the transfer in a `"len_db"` variable which is used only if the size of the transfer is greater than 64 bytes (line 673 inside the code in Figure 6.13). This variable is declared "external" in the `stm32g0xx_hal_hcd.c` file which contains the `HAL_HCD_HC_SubmitRequest()` function. Then, on line 698, the variable `"len_db"` is used which contains the size of the transfer instead of `"length"` which contains the maximum size of the BULK. As you can see from



The screenshot shows a debugger window with the 'Registers' tab selected. A search filter 'type filter text' is applied. The register list is expanded to show the USB peripheral registers. The register `USB_CHEP2R` at address `0x40005c08` has a value of `0x8101b1`. The bit fields for this register are listed below:

Register	Address	Value
USB		
USB_CHEP0R	0x40005c00	0x810200
USB_CHEP1R	0x40005c04	0x10603
USB_CHEP2R	0x40005c08	0x8101b1
EA	[0:4]	0x1
STATTX	[4:2]	0x3
DTOGTX	[6:1]	0x0
VTTX	[7:1]	0x1
EPKIND	[8:1]	0x1
UTYPE	[9:2]	0x0
SETUP	[11:1]	0x0
STATRX	[12:2]	0x0
DTOGRX	[14:1]	0x0
VTRX	[15:1]	0x0
DEVADDR	[16:7]	0x1
NAK	[23:1]	0x1
LS_EP	[24:1]	0x0
ERR_TX	[25:1]	0x0
ERR_RX	[26:1]	0x0

Figure 6.15: Debug session: EPKIND bit is enabled

Figure 6.15, after this modification the EPKIND bit turns out to be 1 and the

UTYPE signal turns out to be 00 (bulk transfer). Furthermore, thanks to the use of the protocol analyzer, another error was identified: if we wanted to transmit a string with a size greater than 64 bytes, the Transmit() function was called before the end of the current transfer. This means that only the first 64 bytes were sent, while all subsequent bytes were not sent because the Transmit() function was called too early and initialized the buffer again. From Figure 6.13, it can be seen that after the "USBH_BulkSendData()" function, the status is set in "CDC_SEND_DATA_WAIT". Therefore, after sending the first 64 bytes, you will enter the following case:

```

687     case CDC_SEND_DATA_WAIT:
688         URB_Status = USBH_LL_GetURBState(phost, CDC_Handle->DataItf.OutPipe);
689
690         /* Check the status done for transmission */
691         if (URB_Status == USBH_URB_DONE)
692         {
693             if (CDC_Handle->TxDataLength > CDC_Handle->DataItf.OutEpSize)
694             {
695                 CDC_Handle->TxDataLength -= CDC_Handle->DataItf.OutEpSize;
696                 CDC_Handle->pTxData += CDC_Handle->DataItf.OutEpSize;
697             }
698             else
699             {
700                 CDC_Handle->TxDataLength = 0U;
701             }
702
703             if (CDC_Handle->TxDataLength > 0U)
704             {
705                 CDC_Handle->data_tx_state = CDC_SEND_DATA;
706             }
707             else
708             {
709                 CDC_Handle->data_tx_state = CDC_IDLE;
710                 USBH_CDC_TransmitCallback(phost);
711             }
712         }
713
714         #if (USBH_USE_OS == 1U)
715             phost->os_msg = (uint32_t)USBH_CLASS_EVENT;
716             #if (osCMSIS < 0x20000U)
717                 (void)osMessagePut(phost->os_event, phost->os_msg, 0U);
718             #else
719                 (void)osMessageQueuePut(phost->os_event, &phost->os_msg, 0U, 0U);
720             #endif
721         #endif
722     }
723     else
724     {
725         if (URB_Status == USBH_URB_NOTREADY)
726         {
727             CDC_Handle->data_tx_state = CDC_SEND_DATA;
728         }
729         #if (USBH_USE_OS == 1U)
730             phost->os_msg = (uint32_t)USBH_CLASS_EVENT;
731             #if (osCMSIS < 0x20000U)
732                 (void)osMessagePut(phost->os_event, phost->os_msg, 0U);
733             #else
734                 (void)osMessageQueuePut(phost->os_event, &phost->os_msg, 0U, 0U);
735             #endif
736         #endif
737     }
738 }
739 break;
740
741 default:
742     break;
743 }

```

Figure 6.16: Case statement within process transmission function

The buffer and the size of the string are managed within this case. If the size of the data to be sent is greater than 64 bytes, the size is reduced by subtracting 64 (since the first packet has already been sent by the BulkSendData function) and the pointer is updated by adding 64 (in so as to send the next 64 bytes). This procedure is done cyclically until the size of the transfer is reduced to a number less than or equal to 0: in this case the entire string has been sent and you have to put yourself in the IDLE condition in order to possibly start a new one. transmission. The main problem is that the original code never accessed the "*CDC_SEND_DATA_WAIT*" case, as the Transmit() function was automatically called on the next run, which initialized the parameters such as buffer length and pointer to the buffer.

```

589 if ((CDC_Handle->state == CDC_IDLE_STATE) || (CDC_Handle->state == CDC_TRANSFER_DATA))
590 {
591     CDC_Handle->pTxData = pbuff;
592     CDC_Handle->TxDataLength = length;
593     CDC_Handle->state = CDC_TRANSFER_DATA;
594     CDC_Handle->data_tx_state = CDC_SEND_DATA;
595     Status = USBH_OK;

```

Figure 6.17: Initialize parameters before transfer

The code has been modified by adding a flag inside the Transmit() function. Initially this flag is set to a value that initializes the buffer and the pointer. Immediately after carrying out this step, the flag is removed in such a way as to avoid a new initialization on the next lap. Therefore, the buffer is managed correctly and at the end of the transmission (Figure 6.16 line 711), a Callback function is used to return the flag value to the starting one and send a new transaction. After these small changes, the percentage of bandwidth used was calculated. Using the double buffer, we went from 56% to about 70%.

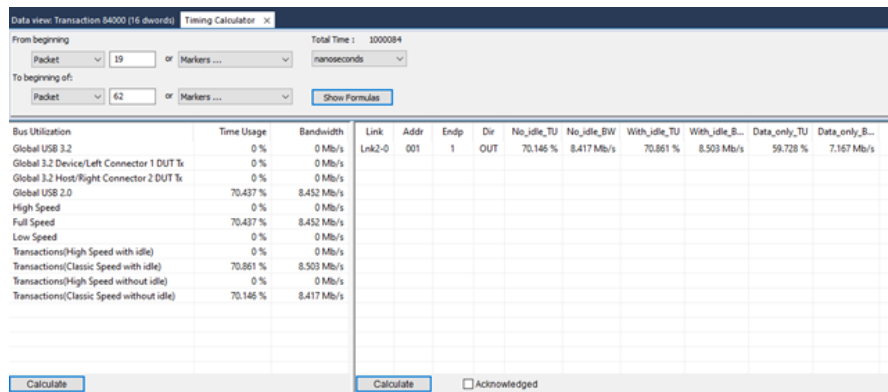


Figure 6.18: Bandwidth utilization percentage using double buffer

However, the result obtained is not the one evaluated theoretically, which is close to 92%. In fact, by observing the packets from the interface of the protocol analyzer, it can be seen that within a frame (i.e. between two consecutive SOFs) only 14 transactions are sent (compared to the maximum 19 possible). This happens due to the presence of the NAKs, which are transmitted from the device to the host to make it understand that it is already busy processing the previous transactions and therefore must try again to send the transaction in the future. In this way the host waits a long time to send the packet again: more precisely, there are about 84 us of delay between receiving the NAK and the start of the next transaction. The

Packet	H	F	Sync	SOF	Frame #	CRCS	EOP	Pkt Len	Duration	Time	Time Stamp
19	S		00000001	0xA5	1320	0x06	250.000 ns	35 Bits (5 Bytes)	2.917 us	3.584 us	0.000.774.182
Transaction 6	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x4B	51.534 us	0.000.777.766		
Transaction 7	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x5A	84.550 us	0.000.829.300		
Transaction 8	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x4B	51.532 us	0.000.913.850		
Transaction 9	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x5A	83.800 us	0.000.965.382		
Transaction 10	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x4B	51.534 us	0.001.049.182		
Transaction 11	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x5A	83.966 us	0.001.100.716		
Transaction 12	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x4B	51.534 us	0.001.184.682		
Transaction 13	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x5A	84.034 us	0.001.236.216		
Transaction 14	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x4B	51.550 us	0.001.320.250		
Transaction 15	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x5A	84.132 us	0.001.371.800		
Transaction 16	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x4B	51.550 us	0.001.455.932		
Transaction 17	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x5A	84.034 us	0.001.507.482		
Transaction 18	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp		
	S	0x87	1	1	0	64 bytes	0x4B	51.550 us	0.001.591.516		
Transaction 19	F	OUT	ADDR	ENDP	T	Data	NAK	Time	Time Stamp		
	S	0x87	1	1	1	64 bytes	0x5A	131.200 us	0.001.643.066		
62	S		00000001	0xA5	1321	0x19	250.000 ns	35 Bits (5 Bytes)	2.917 us	3.584 us	0.001.774.266

Figure 6.19: Presence of NAK handshake that reduces the bandwidth percentage

presence of NAKs indicates that the device software is extremely slow compared to the host, as it is unable to process the packet in time to receive the second before the arrival of a new transaction. Therefore, the hardware must still wait for the software to get rid of the previous buffer before being able to "accept" the arrival of a new transaction. However, the two boards are identical and use the same external quartz oscillator. The doubt is that there is some problem with enabling the double buffer for the board device. Thanks to an analysis using a debug session, it was seen that the DTOGRX and DTOGTX bits did not switch correctly. After a successful transaction, an interrupt is called that brings the VTRX signal to the high logical value. Immediately after, since the data has been correctly written to the buffer, the DTOGRX bit (which indicates the buffer used by the hardware) is removed by the hardware. The software takes care of processing the second buffer and removes the DTOGTX bit, so that the hardware can fill it again. This

behavior is described in an exhaustive way by the table in Figure 6.20. Inside

Endpoint type	DTOG	SW_BUF	Packet buffer used by USB peripheral	Packet buffer used by Application Software
Transmit Host: OUT Device: IN or Receive (OUT)	0	1	USB_CHEP_TXRXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations	USB_CHEP_RXTXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations
	1	0	USB_CHEP_RXTXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations	USB_CHEP_TXRXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations.
	0	0	first transaction: USB_CHEP_TXRXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations After first transaction: None (1)	USB_CHEP_TXRXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations. Also for transmit (first transaction only) prepare second buffer: USB_CHEP_RXTXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations.
	1	1	None (1)	USB_CHEP_RXTXBD_n (ADDR_TX / COUNT_TX) Buffer description table locations

Figure 6.20: Bulk double-buffering memory buffers usage (Device mode)

the code there is "*HAL_PCD_EP_DB_Receive*" which is called every time the VTRX bit is brought to logic '1'. When DTOGRX is worth '1' (first condition), the software reads the first buffer through the *USB_ReadPMA* function and then must toggle DTOGTX (which must be '0' to ensure that the conditions in Figure 6.21 are verified. In fact, the peripheral will autonomously toggle DTOGRX and therefore the final condition will be DTOGRX = 0 and DTOGTX = 1. Similarly, if DTOGRX is '0', the software reads the second buffer and toggles DTOGTX from '1' to '0', so that the final condition after the two toggles will be DTOGRX = 1 and DTOGTX = 0. The two controls on line 2041 and 2074 have been changed as they were wrong and it was the latter that caused problems on the double buffer. After making this last correction, the band calculated by the protocol analyzer turns out to be very close to the maximum calculated and equal to 91.1%, as shown in Figure 6.22. Therefore, it is possible to observe that through the use of the double buffer, which is a fundamental function of the USB peripheral to obtain higher performances, we can obtain an excellent band almost equal to the theoretical one calculated. However, some bugs in the native code created by STM32Cube had to be fixed. These bugs have been communicated to the team and taken into careful analysis.

The results obtained have been summarized in Figure 6.23. Enabling the double buffer on both boards was essential to push digital IP to the limit and achieve maximum performance. In fact, enable the double buffer only for the board of type device, maintains the percentage of bandwidth used equal to 65%, which is equal to the case with double buffer disabled for both cards. In this case, the "bottleneck" regarding the performance is linked to the host, which is unable to quickly send two consecutive packets, while the device is very fast and always has a free buffer waiting for a new packet. Enabling the double buffer only for the host type board, the performance increases slightly and the percentage is around 75% (1 transaction more than before). However, even if the bandwidth is higher, the effective packets exchanged between host and device decrease; the host is faster than the device, which is not always able to accept the new packet as both buffers are already occupied by previous packets. When double buffering is enabled on both boards, the best performance is achieved because the software and digital peripherals on both boards work at the same speed. The implementation of the codes and the enabling of the double buffer led to the resolution of some problems of the native code, and an important feedback was provided to the team located in Tunisia that writes the firmware for the STM32 boards on the market.

TIPO DI TRANSFER	LUNGHEZZA DELLA TRANSFER	NUMERO MASSIMO DI BYTE PER TRANSACTION	DOUBLE BUFFER HOST	DOUBLE BUFFER DEVICE	UTILIZZO TEORICO DELLA BANDA	UTILIZZO REALE DELLA BANDA
BULK	16 BYTE	64 BYTE (1 transaction to complete the transfer)	NO	NO	98% (51 transactions)	49.6% (5.96 Mbit/s) 33 transactions
BULK	512 BYTE	64 BYTE (8 transactions to complete the transfer)	NO	NO	98% (19 transactions)	65.8% (7.89 Mbit/s) 13 transactions
BULK	512 BYTE	64 BYTE	NO	SI	98%	65.8% (7.89 Mbit/s) 13 transactions
BULK	512 BYTE	64 BYTE	SI	NO	98%	70.4% (8.45 Mbit/s) 14 transactions
BULK	512 BYTE	64 BYTE	SI	SI	98%	91.1% (10.9 Mbit/s) 18 transactions

Figure 6.23: Achieved results with the various tests

```

2009 static uint16_t HAL_PCD_EP_DB_Receive(PCD_HandleTypeDef *hpcd,
2010                                         PCD_EPTypeDef *ep, uint16_t wEPVal)
2011 {
2012     uint16_t count;
2013
2014     /* Manage Buffer0 OUT */
2015     if ((wEPVal & USB_EP_DTOG_RX) != 0U)
2016     {
2017         /* Get count of received Data on buffer0 */
2018         count = (uint16_t)PCD_GET_EP_DBUF0_CNT(hpcd->Instance, ep->num);
2019
2020         if (ep->xfer_len >= count)
2021         {
2022             ep->xfer_len -= count;
2023         }
2024         else
2025         {
2026             ep->xfer_len = 0U;
2027         }
2028
2029         if (ep->xfer_len == 0U)
2030         {
2031             /* set NAK to OUT endpoint since double buffer is enabled */
2032             PCD_SET_EP_RX_STATUS(hpcd->Instance, ep->num, USB_EP_RX_NAK);
2033         }
2034
2035         if ((count != 0U))
2036         {
2037             USB_ReadPMA(hpcd->Instance, ep->xfer_buff, ep->pmaaddr0, count);
2038         }
2039
2040         /* Check if Buffer1 is in blocked state which requires to toggle */
2041         if ((wEPVal & USB_EP_DTOG_TX) == 0U) //modified by AS ( != 0 original code )
2042         {
2043             PCD_FREE_USER_BUFFER(hpcd->Instance, ep->num, 0U);
2044         }
2045     }
2046     /* Manage Buffer 1 DTOG_RX=0 */
2047     else
2048     {
2049         /* Get count of received data */
2050         count = (uint16_t)PCD_GET_EP_DBUF1_CNT(hpcd->Instance, ep->num);
2051
2052         if (ep->xfer_len >= count)
2053         {
2054             ep->xfer_len -= count;
2055         }
2056         else
2057         {
2058             ep->xfer_len = 0U;
2059         }
2060
2061         if (ep->xfer_len == 0U)
2062         {
2063             /* set NAK on the current endpoint */
2064             PCD_SET_EP_RX_STATUS(hpcd->Instance, ep->num, USB_EP_RX_NAK);
2065         }
2066
2067         if (count != 0U)
2068         {
2069             USB_ReadPMA(hpcd->Instance, ep->xfer_buff, ep->pmaaddr1, count);
2070         }
2071
2072         /*Need to FreeUser Buffer*/
2073         if ((wEPVal & USB_EP_DTOG_TX) != 0U) //modified by AS ( == 0 original code )
2074         {
2075             PCD_FREE_USER_BUFFER(hpcd->Instance, ep->num, 0U);
2076         }
2077     }
2078 }
2079
2080
2081 return count;
2082 }

```

Figure 6.21: *HAL_PCD_EP_DB_Receive* function

Performance test, results and conclusions

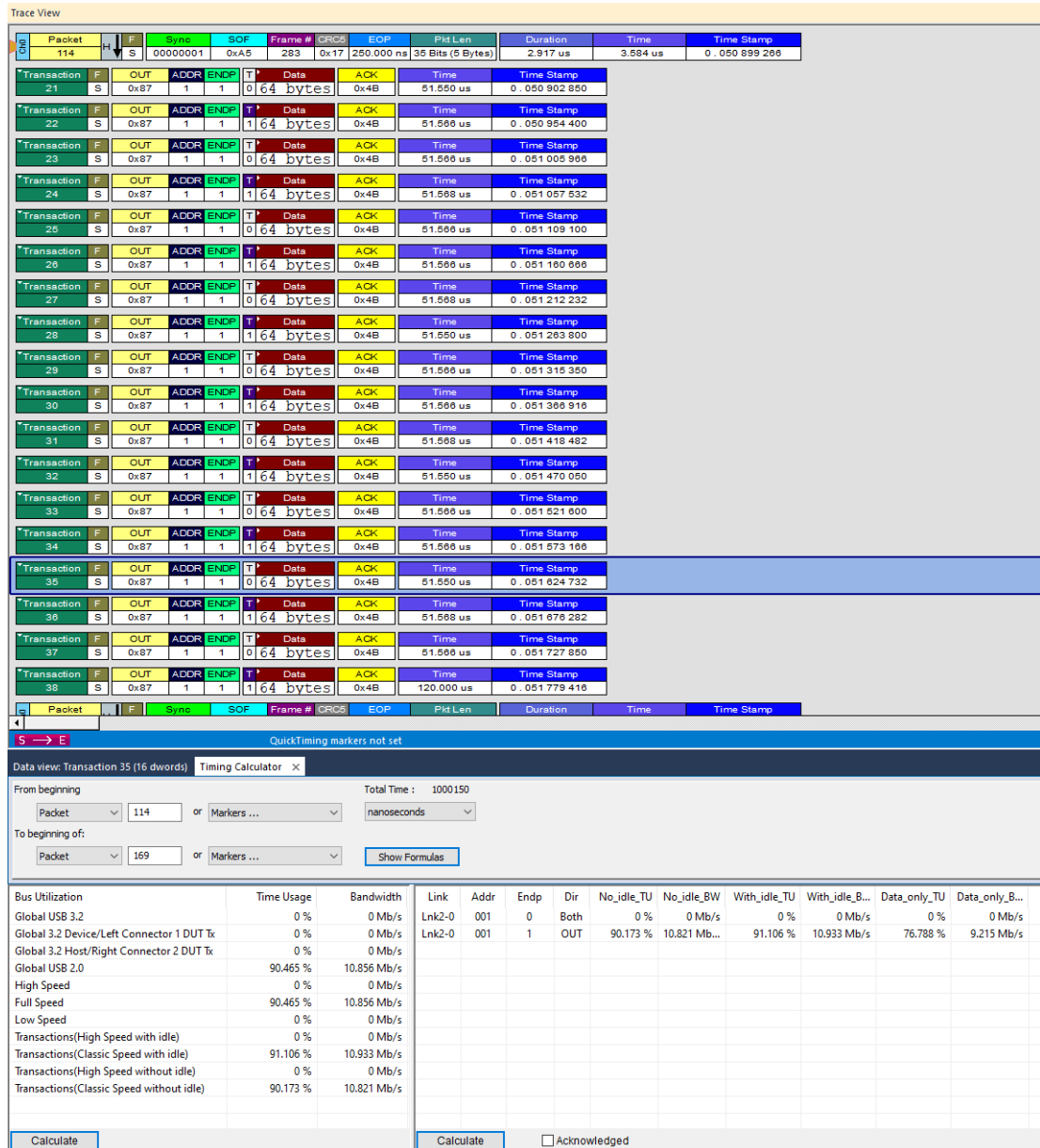


Figure 6.22: Log and bandwidth usage with correct double buffer

Bibliography

- [1] STMicroelectronics. *STM32 32-bit Arm Cortex MCUs*. URL: <https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html> (cit. on p. 2).
- [2] Rohde Schwarz. *Comprensione dell'UART*. URL: https://www.rohde-schwarz.com/it/prodotti/misura-e-collaudo/oscilloscopi/educational-content/comprensione-uart_254524.html (cit. on p. 4).
- [3] Wikipedia. *Serial Pheripheral Interface*. June 7, 2021. URL: https://it.wikipedia.org/wiki/Serial_Peripheral_Interface (cit. on p. 5).
- [4] STMicroelectronics. *Full duplex SPI emulation for STM32 microcontrollers*. August 4, 2015. URL: https://www.st.com/resource/en/application_note/an4678-full-duplex-spi-emulation-for-stm32f4-microcontrollers-stmicroelectronics.pdf (cit. on p. 6).
- [5] Gaspare Santaera. «STM32 Discovery: Le Comunicazioni Seriali». In: *Elettronica Open Source* Decembrer 31 (2015). URL: <https://it.emcelettronica.com/stm32-discovery-le-comunicazioni-seriali> (cit. on p. 7).
- [6] Inventiva. *The USB Connectors: Upbeat journey of USB 1.0 to USB 3.1*. Novembrer 1, 2021. URL: <https://www.inventiva.co.in/stories/the-usb-connectors/> (cit. on p. 8).
- [7] Wikipedia. *USB-C*. September 10, 2022. URL: <https://en.wikipedia.org/wiki/USB-C> (cit. on p. 10).
- [8] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 12 (cit. on p. 11).
- [9] Open4Tech team. *An Introduction to USB Communication*. URL: <https://open4tech.com/an-introduction-to-usb-communication-part-1/> (cit. on p. 12).
- [10] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 17 (cit. on p. 13).
- [11] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 141 (cit. on p. 14).

- [12] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 240 (cit. on p. 16).
- [13] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 157 (cit. on p. 17).
- [14] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 157 (cit. on p. 17).
- [15] USB Made Simple. *Data Flow*. URL: https://www.usbmadesimple.co.uk/ums_3.htm (cit. on p. 18).
- [16] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 196 (cit. on p. 19).
- [17] USB Made Simple. *Data Flow*. URL: https://www.usbmadesimple.co.uk/ums_3.htm (cit. on p. 20).
- [18] STMicroelectronics. «RM0444». In: November 2020, p. 1263 (cit. on p. 25).
- [19] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 146 (cit. on p. 27).
- [20] Compaq-Hewlett-Packard-Intel-Lucent-Microsoft-NEC-Philips. «Universal Serial Bus Specification». In: April 27, 2000, p. 145 (cit. on p. 28).
- [21] STMicroelectronics. «RM0444». In: November 2020, p. 1269 (cit. on p. 30).
- [22] STMicroelectronics. «RM0444». In: November 2020, p. 222 (cit. on p. 33).
- [23] STMicroelectronics. «RM0444». In: November 2020, p. 1275 (cit. on p. 33).
- [24] STMicroelectronics. «RM0444». In: November 2020, p. 1283 (cit. on p. 34).
- [25] STMicroelectronics. *Evaluation board with STM32G0C1VE MCU*. URL: <https://www.st.com/en/evaluation-tools/stm32g0c1e-ev.html> (cit. on p. 37).
- [26] STMicroelectronics. «UM2783». In: December 1, 2020, p. 8 (cit. on p. 38).
- [27] STMicroelectronics. «UM2783». In: December 1, 2020, p. 9 (cit. on p. 38).
- [28] STMicroelectronics. «RM0444». In: November 2020, p. 1300 (cit. on p. 40).
- [29] STMicroelectronics. «RM0444». In: November 2020, p. 223 (cit. on p. 41).
- [30] STMicroelectronics. «UM2783». In: December 1, 2020, p. 30 (cit. on p. 48).
- [31] STMicroelectronics. «STM32Cube USB device library». In: February, 2019, p. 11. URL: https://www.st.com/resource/en/user_manual/um1734-stm32cube-usb-device-library-stmicroelectronics.pdf (cit. on p. 50).
- [32] STMicroelectronics. «STM32Cube USB host library». In: May, 2015, p. 7. URL: https://www.st.com/resource/en/user_manual/dm00105256-stm32cube-usb-host-library-stmicroelectronics.pdf (cit. on p. 52).

- [33] STMicroelectronics. «STM32Cube USB host library». In: May, 2015, p. 18.
URL: https://www.st.com/resource/en/user_manual/dm00105256-stm32cube-usb-host-library-stmicroelectronics.pdf (cit. on p. 55).

Ringraziamenti

Una pagina importante della mia vita si chiude adesso, con l'ultima pagina di questo elaborato finale. Mai avrei immaginato, all'inizio di questo percorso, di possedere la forza e la determinazione che mi hanno portato al raggiungimento di questo fondamentale obiettivo. Mi sembra doveroso dedicare alcune parole a tutte le persone che mi hanno sostenuto ed hanno contribuito alla realizzazione di questo elaborato. Sono certamente consapevole che senza il vostro supporto, questa avventura sarebbe stata molto più difficile, se non impossibile.

Ringrazio il professor Demarchi, per avermi guidato nella fase più importante del mio percorso accademico.

Sono grato all'ingegnere Giuseppe Guarnaccia, tutor aziendale della mia attività, ed a tutto il team Digital IP di Catania per avermi ben accolto ed avermi fatto sentire parte del gruppo sin dal primo giorno. I vostri consigli, dettati dall'esperienza, sono e saranno la base per iniziare il mio percorso professionale. Darò il massimo per ripagare la fiducia che avete riposto in me.

Grazie a tutti i miei colleghi di corso, per avermi sempre incoraggiato fin dall'inizio del percorso universitario ed aver condiviso successi, paure, fatiche e risate. Senza di voi questa avventura sarebbe stata più noiosa e difficile.

Grazie a tutti i miei amici, per avermi regalato momenti di spensieratezza anche quando era lo sconforto a prevalere sul mio stato d'animo. Vi voglio bene.

Un ringraziamento particolare alla mia ragazza, Martina, che è il mio porto sicuro dove rifugiarmi quando il mare è in tempesta: grazie per amarmi incondizionatamente per quello che sono, nonostante gli innumerevoli difetti, e per essere sempre al mio fianco. Grazie per aver sopportato le mie ansie e le mie paure, per aver ridimensionato le mie preoccupazioni. Sono orgoglioso di avere una Donna come te al mio fianco. Auguro a noi di raggiungere altri importanti traguardi, sempre insieme, pronti a sorreggerci l'un l'altro.

Grazie nonna Piera e mastro Giovanni, per avermi donato la vostra infinita dolcezza ed avermi fatto sentire sempre a casa, nonostante la lontananza.

Un pensiero ai nonni Pippo e Concetta, che mi guardano da lassù. Spero che siate orgogliosi dell'uomo che sono diventato.

Grazie ai miei cugini ed ai miei zii per tutte le volte che siete stati presenti quando il mio cuore necessitava di stare insieme a voi. Siete la mia seconda famiglia.

A mia sorella, da sempre la metà che mi completa. Grazie per avermi insegnato ad essere più forte nei momenti difficili, quando tutto sembra andare nel verso sbagliato. Ti proteggerò sempre. Ti amo.

A mio padre, l'uomo che ammiro più di tutti. Grazie per tutti i valori che mi hai trasmesso, per avermi dato la possibilità di studiare e diventare quello che sono oggi. Non sarà mai abbastanza la gratitudine per tutto quello che hai fatto per la nostra famiglia.

Infine, ringrazio mia madre, a cui dedico questo mio traguardo. Sei stata la forza che mi è servita per andare avanti. Con la paura di poter cadere, certo, ma con la consapevolezza di sapersi rialzare più forti di prima. Grazie per aver sempre creduto in me, sin dal primo giorno di università. Questa laurea è anche tua, che insieme a Papà, avete combattuto e stretto i denti al mio fianco, e che spero oggi possiate essere felici. Siete tutta la mia vita.