Master thesis : Electrical Ground Support Equipment (EGSE) of OUFTI2

Auteur : Harika, Soulaimane
Promoteur(s) : Redouté, Jean-Michel
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil électricien, à finalité spécialisée en "electronic systems and devices"
Année académique : 2018-2019
URI/URL : http://hdl.handle.net/2268.2/8089

University of Liege: Faculty of applied science

Electrical Ground Support Equipment (EGSE) of OUFTI2

Graduation work carried out in view of obtaining the master's degree in "Electrical Engineering" by

Harika Soulaimane

Academic year 2018-2019

Abstract

The main objective is to develop and test an Electrical Ground Support Equipment (EGSE). The EGSE is the device that ensures the proper operation of OUFTI-2. Of course, it is essential to fully test any satellite before sending it into space. The choice of only 27 pins or signals was decided in advance for their importance. Their importance comes from the fact that they are essential for the proper operation of OUTFI-2.

The first step was to build the actual device that will perform data acquisition. This will consist of a hardware device that can be interfaced to a PC. Because of their different natures, signals were divided into two categories : the SPI signal and the non-SPI signal. The SPI signals consist of communication signal that allow the On-Board Computer to communicate with some subsystems. For SPI signals, a logic analyzer has been used to acquire them. The logic analyzer had the advantage to be compatible with a software suite that allows easy interface with the PC. For non-SPI signals, the acquisition was done by using a Arduino Uno. To interface the non-SPI signal to the PC, the logic analyser was reused since channels remain available. However, the strategy used for data acquisition turned out to be not possible. Indeed, thanks to numerous tests, and despite concessions, it turned out that the solution found to acquire the data could not work. These early tests avoided implementing a hardware solution that was doomed to fail.

However, another component of the work was to work on data processing. Despite the failure in data acquisition, it was possible to progress in the software part of the EGSE. The EGSE software must allow data processing and display in a graphical interface. To move forward in this part, it was necessary to define the structure of the data acquired by the data acquisition part of the EGSE. Many tests have been done to validate the code as much as possible. Most of the validation was done on real data exchanged by using logic analyser. For the sotware part, the EGSE is up to date with the modules implemented for the moment.

In the end, ideas and observations that I was able to collect to find an alternative for data acquisition as well as an EGSE status report has been provided in order to have another student taking the lead in this project.

Contents

Chapter 1

Introduction and context

1.1 OUFTI-2: brief background

OUFTI-2 is a 100% Belgian educational satellite. It consists of a CubeSat, a cube nanosatellite of 10cm x 10cm x 11.35 cm. Its predecessor is OUFTI-1. It was the first satellite built by students in Belgium! The main objective of OUFTI-1 was to test the use of a digital communication protocol for amateur radio developed by the Japan Amateur Radio League: D-STAR (Digital Smart Technologies for Amateur Radio).

However, after a successful launch on April 25, 2016, an unidentified problem occurred. As a result, it was not possible to use the D-STAR. The only signals transmitted by OUFTI-1 via a system independent of D-STAR, a beacon that uses the Morse code, continued to transmit for 12 days. In the end, all contact with OUFTI-1 was lost on May 7, 2016.

As a successor to OUFTI-1, OUFTI-2 will share the same mission, which is to test and use the D-STAR protocol; this remains the main payload of the nanosatellite. But two other additional payloads (discussed later) will be present in OUFTI-2: **RAD** and **IMU**.

1.2 CubesSat

The CubesSat were born in 1999 from the efforts of 2 professors: Jordi Puig-Suari, from the California Polytechnic State University (Cal Poly), and Bob Twiggs, from Stanford University's Space Systems Development Laboratory (SSDL). The aim was to allow the emergence of educational projects related to space by facilitating access to it by creating the standardised nanosatellite CubeSat.

Indeed, the CubeSat standard defines a reference size, called the CubeSat unit: 1U. A 1U CubeSat corresponds to the dimensions: 10cm x 10cm x 11.35 cm [1]. The weight of a 1U CubeSat is close to 1kg but can vary around 0.8-1.3 kg. OUFTI-2 is an 1U CubeSat like its predecessor. Larger CubeSat can be defined as 3U ($\sim$ 5 kg) which will be 3 x 11.35cm high or even 27U ($\sim$ 35 kg) which will be 27 x 11.35cm high.

In considering these standards, the term *nano*satellite may seem inappropriate for an object of this size, but it should be remembered that a *"small"* satellite in the space domain generally refers to objects below 300-500 kg.

1.3 Presentation of the subsytems present in OUFTI-2

In the scope of this project, it is necessary to discuss the boards and subsystems present in OUFTI-2. As one can obverse in Figure 1.1, different boards are present. One of the particularities is that each board contains a PC104 connector, which facilitates communication and interconnection between the boards. As one can see in Figure 1.1, the PC104 has been designed to allow to stack boards one above

the other. Indeed, thanks to its size (9 x 9.6 cm) which allows to respect the size constraints imposed by the CubeSat standard, PC104 boards are often used in CubeSats.

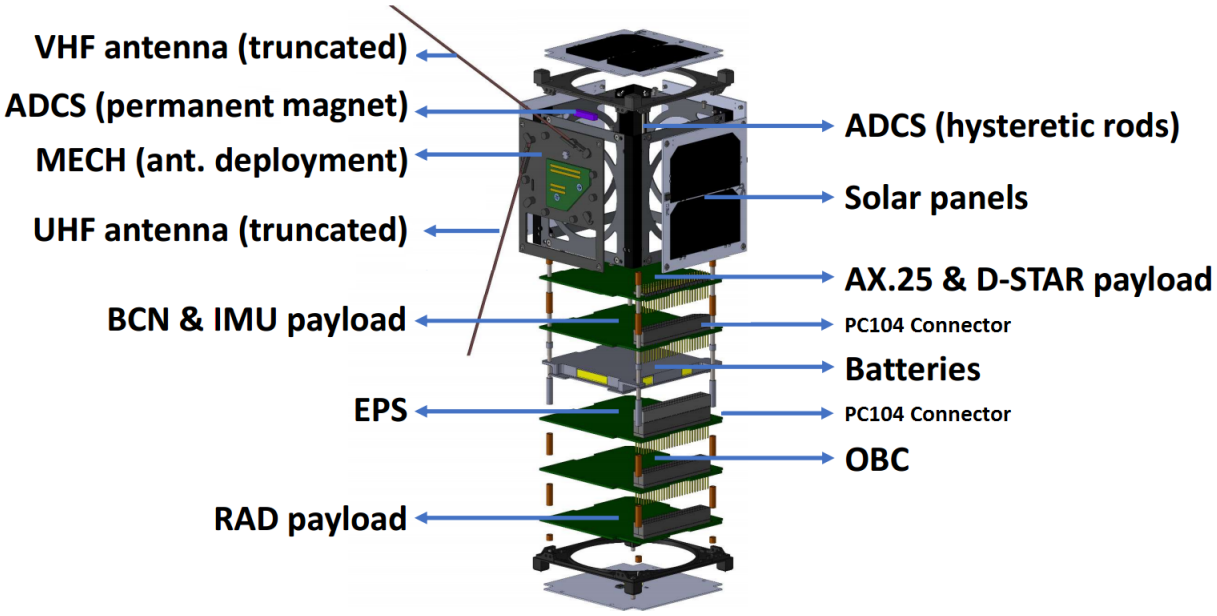

Figure 1.1: Exploded view of OUFTI-2

First, the goal will be to briefly present each board, from top to bottom. Later on, when necessary, more precision about specific module present in these boards will be provided.

The first board is **COMM**, for **comm**unnication, which contains component related to the two communication protocols: AX.25 and the D-STAR payload. The AX.25 is used to communicate with the nanosatellite. As for the D-STAR, as explained above, the main goal of OUFTI-2 will be to act as a relay for D-STAR communications.

The board below contains the beacon that uses the Morse code, named **BCN**, and the **IMU** (**I**nertial **M**easurement **U**nit) payload. The **IMU** is one of the new secondary payload. **IMU** has been developed by secondary school students and uses various sensors to determine the altitude of OUFTI-2. **BCN** is the subsystem that was also present on OUFTI-1 and that had continued to transmit for 12 days. **BCN** is used to send measurement information of other subsystems.

The next board contains the battery and, right below the battery, is the **EPS** (**E**lectrical **P**ower **S**upply) boards. The **EPS** is responsible for powering all other boards. **EPS** is obviously connected to the solar panels and the battery that will be useful for the periods when the solar energy will not be available.

The next board is the board containing the **OBC** (**O**n **B**oard **C**omputer). The **OBC** board supervises the proper operation of OUFTI-2. The microcontroller on this board is the DPC (**D**igital **P**rogrammable **C**ontroller) from ThalesAlenia Space. An important note that will be useful for the future is that, in this board, is present the FRAM (Ferroelectric RAM) which serves as an additional memory for the DPC. For instance, various measurements are communicated to the FRAM to be stored before sending them to Earth. Back to the DPC, it contains 3 cores and has the significant advantage of having been designed to go into space. Indeed, it can withstand radiation much better than a conventional microcontroller.

Speaking of radiation, the last payload is related to this theme. Indeed, the last payload is the other new secondary payload: **RAD**. The purpose of this payload is to determine the degree of protection of different shielding against radiation. To do this, 3 identical circuits are present with different shielding: a reference circuit without shielding, one with aluminium shielding and the last one is a multilayer shielding. Measurements will be sent to the ground to study the effect of these shields.

For the other important elements, on top of Figure 1.1 are the **VHF** (**V**ery **H**igh **F**requency) antenna for transmission and **UHF** (**U**ltra **H**igh **F**requency) antenna reception. Then, the **ADCS** (**A**ttitude **D**etermination and **C**ontrol **S**ystem) is a passive system for controlling the orientation of the nanosatellite. Then, there are, of course, the solar panels and the last element is **MECH**. This is the mechanical system that allows the deployment of Antennas.

1.4 Objectives of the project

The main objective is to develop and test an Electrical Ground Support Equipment (EGSE). The EGSE is the device that ensures the proper operation of OUFTI-2. Of course, it is essential to fully test any satellite before sending it into space.

As discussed above, each board follows the PC104 standard. Each connector contains the same signals i.e. the pin allocation of each connector is the same. This is a prerequisite for stacking 2 boards. **The goal of the EGSE will be to acquire and analyse** only a part of them. Indeed, of the 104 pins, only 27 will be read.

1.4.1 Signals that must be monitored

The choice of only 27 pins or signals was decided in advance for their importance. Their importance comes from the fact that they are essential for the proper operation of OUTFI-2. Three categories of signals can be distinguished:

1. Voltages that supply the different subsystems.
2. Signals related to the activation of certain subsystems and a battery charge indicator.
3. Communication signals that allow the DPC to exchange data with subsystems.

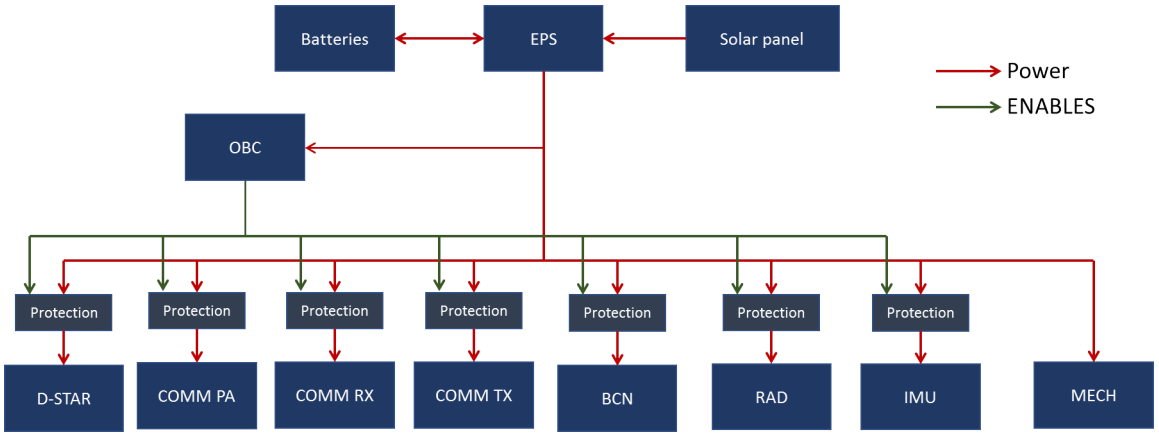

Figure 1.2: Simplified view of the interconnection between subsystems.

In Figure 1.2, a simplified overview of the interactions between subsystems is presented. The voltages to be monitored come from the batteries and the EPS subsystem. A total of 6 voltages will be read by the EGSE. All the others signals that must be retrieved by the EGSE are digital signals. One of them is located in the batteries and indicate that the battery is 100% charged. Still in Figure 1.2, one can observe protection components before some subsystems. Those protection component allows the OBC to activate or deactivate a subsystem by means of digital signals; the ENABLES.

The last digital signals to be retrieved by the EGSE are the communications signals that consist of SPI buses clocked at maximum 3.75 Mhz. The SPI buses are used by the DPC to transmit and receive data with subsystems as shown in Figure 1.3. Indeed, in Figure 1.3, all the elements that can interact with the DPC are listed.

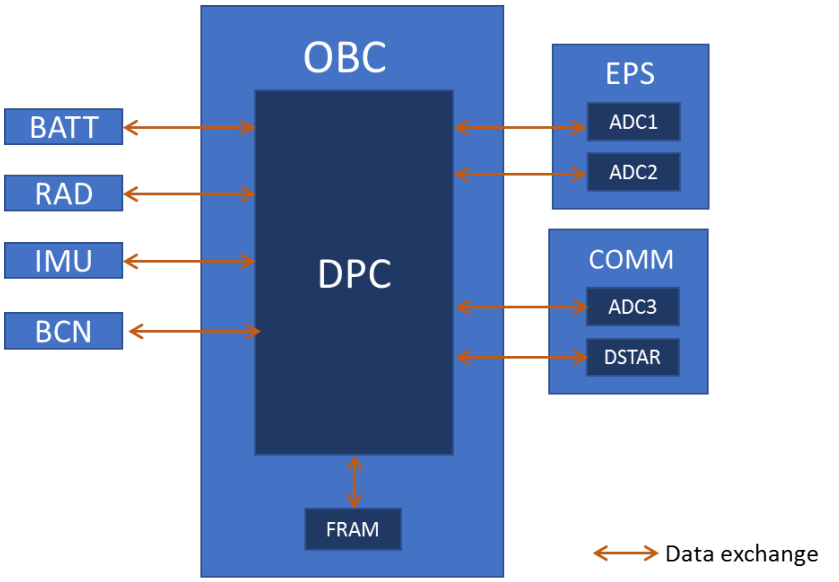

Figure 1.3: Simplified view of subsystems that communicate and exchange data with the DPC through SPI

Actually, we have already discussed about a component present in Figure 1.3: the FRAM. In fact, the FRAM is the memory in which the DPC records measurements and events. The only last element not yet discussed are the ADCs (Analog to Digital Converter) present in the EPS and COMM boards.

1.4.2 Objectives of the EGSE

The first goal of the EGSE will be to acquire the signals discussed above. After acquiring the contents of these pins, some data processing will be performed. Then, the output of the data processing will be displayed on a PC. As summarized in Figure 1.4, during the test phase of OUFTI-2, a PC104 connector will be available and will be connected to the EGSE which will send the contents of the pins to the PC for display.

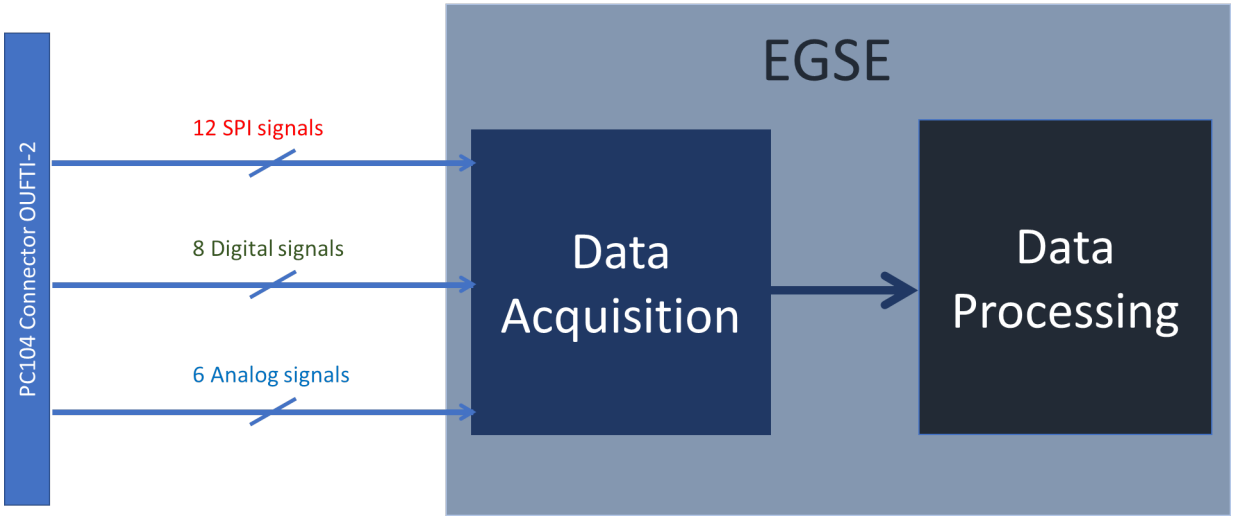

Figure 1.4: From the PC104 connector, three types of signal are available: SPI signals, Digital signals and analog signals. Their content are read by an EGSE and sent to the PC for display.

Figure 1.5 details the actual names of each signal.

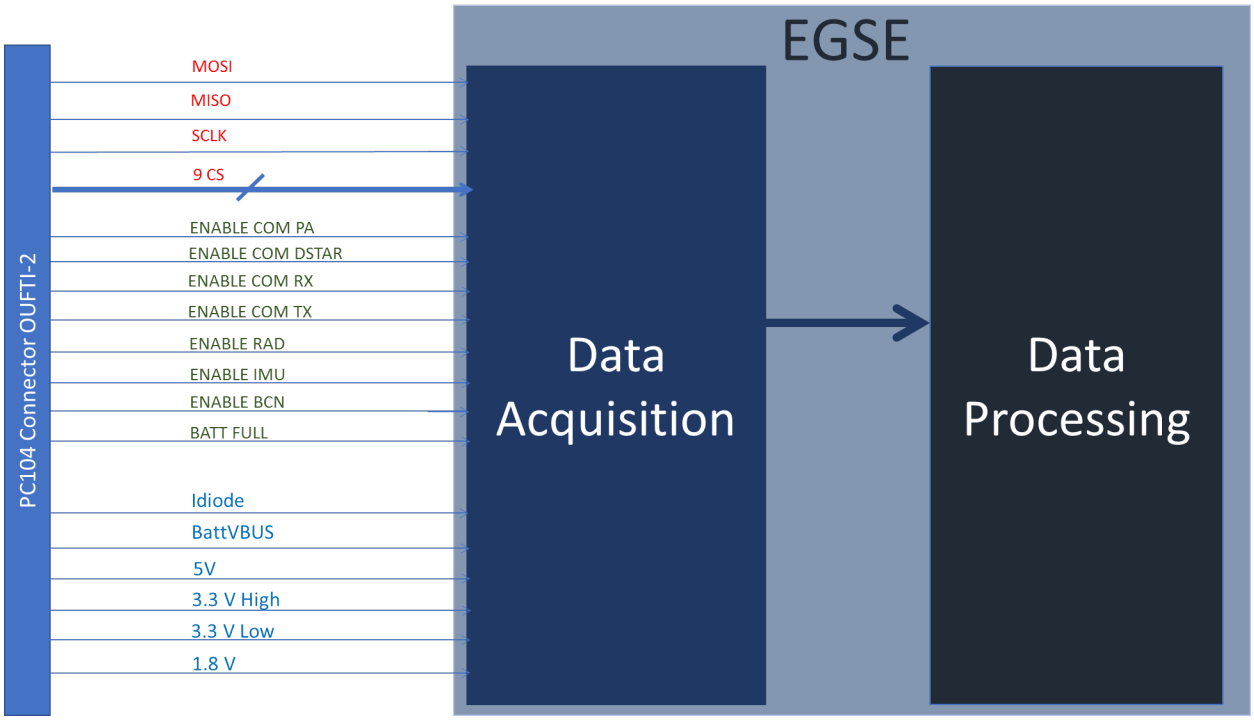

Figure 1.5: From the PC104 connector, three types of signal are available: SPI signals (in red), Digital signals (in green) and analog signals (in blue). Their content are read by an EGSE and sent to the PC for display.

As shown in Figure 1.5 and 1.4, the EGSE consists of two main elements: data acquisition and data processing. Figure 1.6 illustrates the content of the two elements.

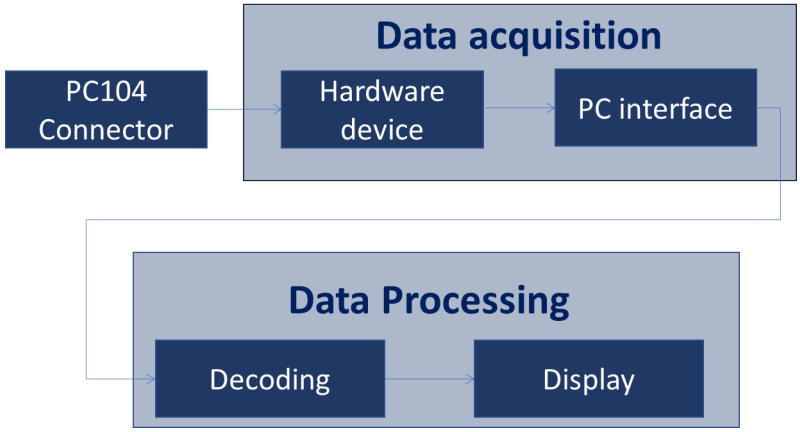

Figure 1.6: Simplified block diagram showing the main elements constituting the EGSE: the data acquisition part (hardware) and the data processing part (software)

Therefore, as shown in Figure 1.6, two main tasks will be required to build the EGSE:

1. To build the actual device that will perform data acquisition. This will consist of a hardware device that can be interfaced to a PC.

2. To process the signal acquired. Concretely, it means creating a software that will take as input the data acquired and decode the data acquired. Then, the result of the data processing will be display in a graphical interface.

Therefore, the first chapters will naturally be focused on the data acquisition. First, the signals of interest will be characterised. Then, depending on signals, ways to acquire them will be discussed. The main solution for data acquisition investigated in this work will be explained. The data acquisition is based on a hardware device called a logic analyzer.

The second part of this document will focus on the way the data are decoded. I was able to develop the JavaFX-based software that allows the acquired data to be processed. Many tests have been done to validate the code as much as possible.

In the end, a status report will summarize the state of the EGSE.

Chapter 2

Signals to be acquired

2.1 List of the signals

To identify the requirement the final equipment will have to meet, the first step is to list the signals that must be acquired by the EGSE. As said in the introduction, 27 signals are directly accessible via a PC104 connector. Figure 2.1 shows a PC104 connector.

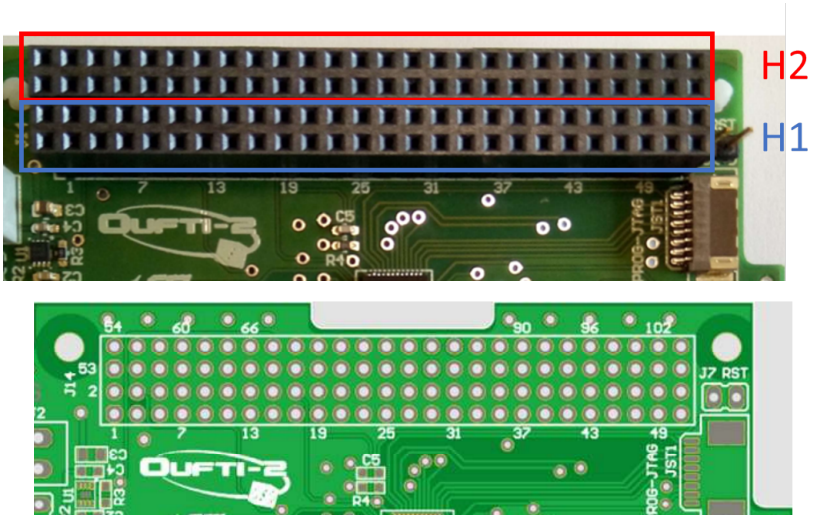

Figure 2.1: PC104 connector: 104 pins divided into 2 headers: H1 is the inner part and H2 is the outer part.

To identify a pin, 2 equivalent methods are possible. The first would be to number them from 1 to 104. The second one, as shown in Figure 2.1, is to define two headers: H1 and H2. Each contain 52 pins, i.e. a total of 104 pins. Depending on the datasheets, this may vary. In OUFTI-2, header 1 is the inner part of the PC104 connector and header 2 is the outer part.

In the following table, one can find a summary of all the signals, chosen in advance because of their importance, that must be read. This is based on a document file, an excel file, last modified in February 2019 (v3.2 version), showing the pin attribution of the PC104 connector.

As can be seen in the table, several types of signals can be distinguished:

- The ENABLES and BAT_FULL
- The SPI_ types
- The Analog data or Voltage

Bus Pin on PC104			Bus Description
H1	1	1	ENABLE_RAD
H1	2	2	ENABLE_IMU
H1	3	3	ENABLE_BCN
H1	4	4	ENABLE_COM_RX
H1	5	5	ENABLE_COM_DSTAR
H1	6	6	ENABLE_COM_PA
H1	7	7	ENABLE_COM_TX
H1	17	8	SPI_CLK
H1	18	9	SPI_MISO
H1	19	10	SPI_MOSI
H1	20	11	SPI_CS_FRAM
H1	21	12	SPI_CS_IMU
H1	22	13	SPI_CS_RAD
H1	23	14	SPI_CS_ADC1
H1	24	15	SPI_CS_ADC2
H1	25	16	SPI_CS_ADC3
H1	26	17	SPI_CS_BCN
H1	27	18	SPI_CS_BATT
H1	28	19	SPI_CS_COM
H2	77-78	20	5V
H2	75-76	21	3V3 GENERAL
H2	81-82	22	DGND
H2	91	23	BAT_FULL
H2	93-96	24	IDIODE
H2	97-98	25	BATVBUS
Not found	/	26	1.8V
Not found	/	27	3.3V

Table 2.1: Pins number and description. (H1 stands for header 1, H2 stands for header 2)

– The Ground

Each signal will be detailed separately below.

2.2 Description of the signals

2.2.1 ENABLES signal and BAT_FULL

ENABLES signal

First of all, as discussed before, the DPC is linked to a number of subsystems. A danger that could arise is that one a malfunction of a subsystem could risk endangering the DPC. Indeed, one of the subsystems may suffer from short circuits. This would mean that the DPC could suffer from this short circuit since the DPC is in direct contact with these subsystems. Thus, to ensure safety and full control of the DPC, it is necessary to allow the DPC to disable a subsystem if it has a short circuit. To do this, a current limit switch is placed directly between the DPC and the subsystem.

The current limit switch used is MAX14575. Its operation can be understood using Figure 2.2.

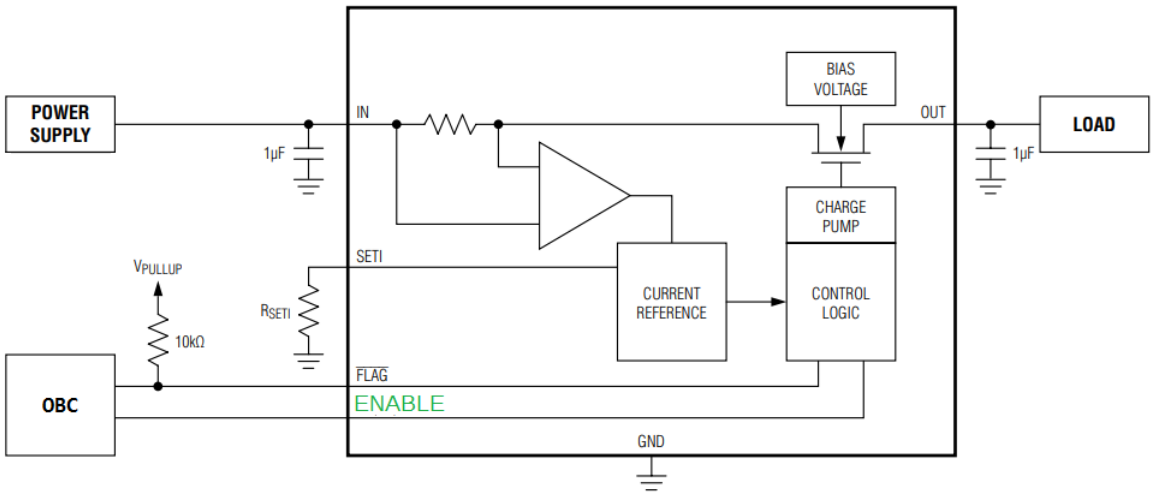

Figure 2.2: Schematics of the current limit switch MAX14575 [2]

The supply voltage of the subsystem first passes through the current limit switch. The different pins are described in Figure 2.3.

PIN MAX14575_	NAME	FUNCTION
1	$\overline{\text{FLAG}}$	Open-Drain Overcurrent Indicator Output. $\overline{\text{FLAG}}$ goes low when the overload fault duration exceeds the blanking time, reverse current is detected, thermal shutdown mode is active, or SETI is connected to ground.
2	SETI	Forward Current-Limit Adjust Input. Connect a resistor from SETI to GND to program the overcurrent limit. If SETI is connected to GND, the switch turns off and $\overline{\text{FLAG}}$ is asserted. Do not leave SETI unconnected or connect any capacitor larger than 20pF to SETI.
3, 4	OUT	Switch Output. Connect OUT pins 3 and 4 together and bypass OUT to GND with a 1µF ceramic capacitor as close to the device as possible.
5, 6	IN	Power Input. Connect IN pins 5 and 6 together and bypass IN to GND with a 1µF ceramic capacitor as close to the device as possible. If necessary, use higher capacitance to prevent large load transients from pulling down the supply voltage.
7	EN	Active-High Enable Input. Drive EN high to turn on the switch. Drive EN low to turn off the switch.
8	GND	Ground
—	EP	Exposed Pad. Internally connected to GND. Connect to a large ground plane to maximize thermal performance. Not intended as an electrical connection point.

Figure 2.3: Description of the MAX14575 pins [2]

To fully understand how it works, it is necessary to consider what happens when there is a short circuit. Once a short has appeared, a flag is raised to inform the DPC. Following this, the DPC may choose to stop powering a subsystem. To do so, the DPC can modify the value of the *EN* pin (in Figure 2.3) to disable the faulty subsystem. Therefore, the purpose of ENABLES is to allow the DPC to enable or disable a subsystem.

In conclusion, the value of the ENABLES can be either Low, to disable a subsystem, or High, to activate it as shown in Figure 2.4.

ELECTRICAL CHARACTERISTICS

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS
EN, $\overline{EN}$ INPUT					
EN, $\overline{EN}$ Input Logic-High Voltage	V_{IH}	1.6			V
EN, $\overline{EN}$ Input Logic-Low Voltage	V_{IL}			0.4	V

Figure 2.4: Voltage threshold to set a LOW or a HIGH [2]

In the case of OUFTI-2, the voltage value of the ENABLES is 0 V in Low (subsystem disabled) and 3.3 V in High (subsystem enabled).

The following table summarises the important information for the ENABLES signals.

Signal	Type of signal	Minimum Value	Maximum Value	Description
ENABLES	Low or High	0 V	3.3 V	Allows the DPC to enable or disable a subsystem.

Table 2.2: Important information about the ENABLES signals

BAT_FULL

The common point with the ENABLES signals is that it consists of a digital signal. Indeed, it can only take 2 possible values: High or Low. The purpose of BATT_FULL is to indicate that the battery is 100% charged.

2.2.2 The SPI_ types

As discussed before, the DPC communicates with various payloads present in OUFTI-2. The communication protocol used by the DPC to communicate is the SPI (**S**erial **P**eripheral **I**nterface).

The SPI communication always requires at least one *master* and one *slave*. In a context of serial communication, a *master* is the module that will have the control of the communication and the *slave* is the module that will execute the instructions ordered by the master.

In the case of OUFTI-2, the master is the DPC and the slaves are different subsystems/modules that interact with the DPC. For example, to have a clear example in mind, one module is an ADC responsible of measuring some parameters like voltage, temperature,... The DPC may request measurement values from the ADCs. In this case, it is always the DPC that initiates the communication. Indeed, it is always the DPC that will ask for the measurement values and the ADCs must respond by sending the measurements. Therefore, in no event may ADCs send measurements to the DPC without a first request from the DPC. That's why the DPC is called the master of communication and the ADC modules are the slaves.

More details about how the SPI communication actually works will be given in section 3.1 as well as the different modules with which it communicates. For the time being, the important point to remember is that SPI signals can only take two values, High and Low, and are used for data communication between the DPC and modules/subsystem/payload.

The following table summarises the important information for an SPI signal.

Signal	Type of signal	Minimum Value	Maximum Value	Description
SPI	Low or High	0 V	3.3V	Data communication between the DPC and payloads

Table 2.3: Important information about the SPI signals

2.2.3 The Analog data

The last signals to be discussed have in common that they are not digital signals, i.e. only High or Low possible, as it was the case with the signals previously discussed (ENABLES, BATT_FULL, or SPI signals) but analog signals.

Two categories can be distinguished:

– Fixed voltages: 5V,3.3V High, 3.3V Low,1.8V

– BATVBUS and Idiode

Fixed voltages

Fixed voltages are quite different from the previous signals that carried only binary information, either High or Low. Indeed, in this case, it will be necessary to measure the actual value of those signals. .

The purpose of each voltage is shown in the following Table

Fixed voltage	Description
5V	To power the beacon (Morse code)
3.3V High (High stands for high current)	To power the transmission communication
3.3V Low (low stands for low current)	To power most of the subsystems
1.8V	To power the DPC

Table 2.4: Purpose of each fixed voltage

Some ADCs will be involved to get the value of those voltages. It will be necessary to define *how accurate* must the measurements of those voltages be. Indeed, the first thing to define is the desired resolution. The required resolution must be at least 8 bits and does not need to be more than 12 bits. Indeed, if one considers the example of 5V, with a resolution of 8bits, variations of 20 mV ($5/2^8$=0.0195V) can be measured which is sufficient for this projet. More detailed will be provided in a section dedicated to the choice of an ADC. The following table summarises the important information for the fixed voltage.

Signal	Type of signal	Expected Value	Resolution	Description
5V, 3.3V (high), 3.3V (low), 1.8V	Analog	Fixed value	8-12 bits	Power voltage

Table 2.5: Important information for the fixed voltage.

BATVBUS and Idiode signal

The Idiode and the BATVBUS signals are part of the BATT subsystem which means it is related to batteries. Both are voltages that will also require an ADC to read this signal. The same resolution as with the fixed voltages will be chosen for this signal. The Idiode corresponds to the battery voltage. The EPS subsystem contains some DC/DC converters and BATVBUS is actually the voltage right in front of the DC/DC converters. The Table 2.6 summarises the important information for both the Idiode and the BATVBUS signals.

Signal	Type of signal	Minimum Value	Maximum Value	Resolution	Description
Idiode	Analog	0V	8.4V	8-12 bits	Battery voltage
BATVBUS	Analog	0V	8.4V	8-12 bits	Voltage before DC/DC converters

Table 2.6: Important information for BATVBUS and Idiode signals

2.3 Summary of the signals

The Table 2.7 summarises the main information for all signals.

Signal	Type of signal	Total number	Description
ENABLES	Digital	7	Allows the DPC to enable or disable a subsystem
BATT_FULL	Digital	1	Satellite powered off or on
SPI	Digital	12	SPI lines to communicate data
Fixed voltage	Analog	4	Power voltage
Idiode	Analog	1	Battery voltage
BATVBUS	Analog	1	Voltage before DC/DC converters

Table 2.7: Summary of all signals

It was essential to clearly identify the nature of the signals to be acquired. Indeed, without this information, it is impossible to establish which requirements the acquisition device to be built will have to meet.

2.4 Overview of the data acquisition solution

Now, to better understand the next sections, it is interesting to first give an overview of the hardware solution for data acquisition. The goal is to give a general view. Obviously, all the justifications related to the choices that led to this hardware solution will be well explained later in the relevant sections.

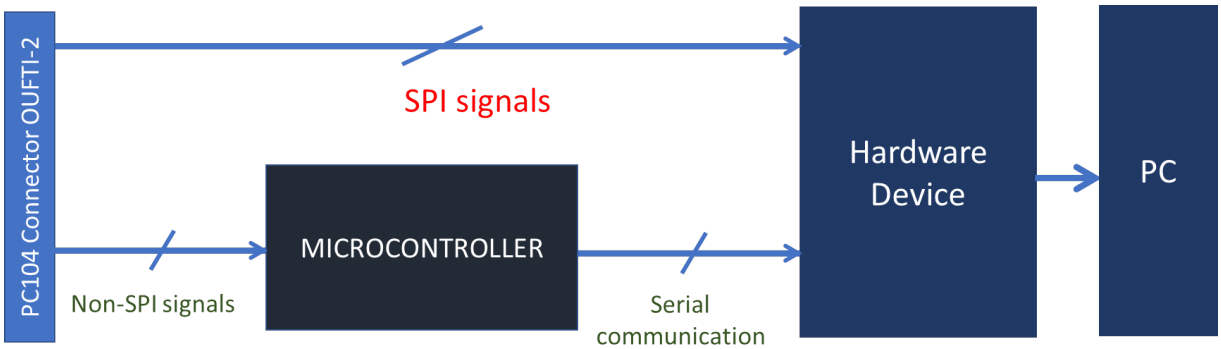

Figure 2.5: Illustration of the hardware solution to acquire data and transmit it through the PC using a specific device

First of all, among different types of signals, the less obvious ones to acquire are the SPI signals due too a high clock rate (3.75 MHz). So, the first step was to find a hardware device that could easily read the SPI data and transmit them to a PC. Then, this device will also be useful to send the rest of the signals, i.e. the non-SPI signals to the PC. Indeed, a microcontroller will send the value of the non-SPI signals to this hardware device. And similarly to the SPI signals, the non-SPI values will then be transmitted to the PC.

The next sections, the acquisition of SPI data will, naturally, be discussed first.

Chapter 3

Acquisition of the SPI signals

The starting point of the reflection is that the most complicated signals to read will be the SPI communication. It is thus important to explain how the SPI protocol works. Indeed, one main objective of the project is to be able to read SPI communications between the DPC and the different modules in OUFTI-2.

3.1 Explanation of the SPI protocol

The SPI requires 4 wires to establish communication between two modules as illustrated in Figure 3.1.:

1. **MOSI**: **M**aster **O**ut **S**lave **I**n, the data sends by the master to the slave
2. **MISO**: **M**aster **I**n **S**lave **O**ut, the data received by the master from the slave
3. **SCLK**: the serial clock controlled by the master.
4. **CS**: **C**hip **S**elect, the master chooses with which slaves the data is exchanged.

Figure 3.1: Illustration of the wires needed for SPI communication. To communicate with one single module, 4 wires are required: MISO,MOSI,SCLK,CS

For **MISO** and **MOSI**, the format of data sent is not standard. Indeed, it is always necessary to check in the datasheet of the slave modules if it is the MSB (**M**ost **S**ignificant **B**it) or the LSB (**L**ess **S**ignificant **B**it) that must be sent first.

For the **CS**, since a master can have several slaves, one can thus have several **CS** lines for each slave module. This is illustrated in Figure 3.2

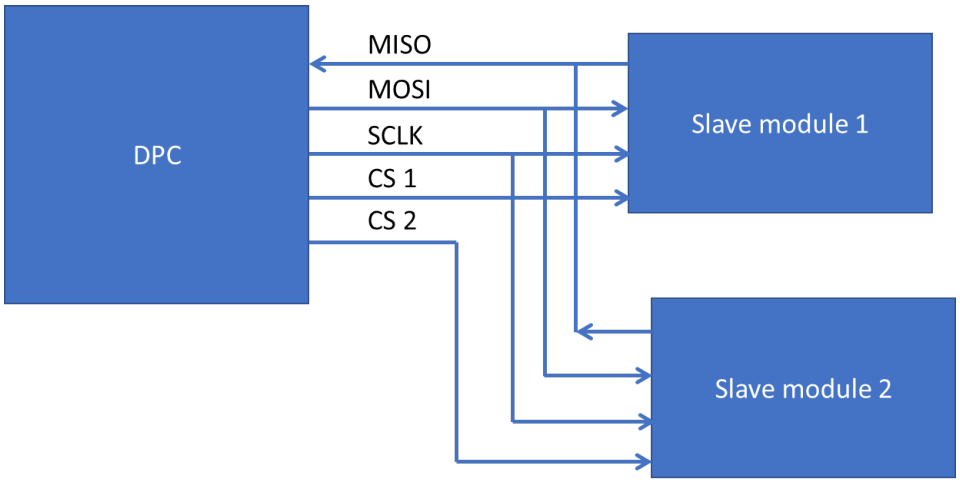

Figure 3.2: To communicate with a specific slave module unambiguously, a **CS** is assigned to each slave module

When the master wants to communicate with a slave module, it simply set the corresponding **CS**.

The line that really needs more details is the **SCLK** line. First of all, **SCKL** corresponds to a clock, i.e. a high and low state oscillation at a certain frequency. A change of state is called a single edge. For a transition from high to low state, the appellation is a falling edge and conversely from low to high state, the appellation is rising edge. So, using single edge, this line allows to give an indication to the master when to send the data to the module and an indication to the module when the data should be sampled.

Since the clock can be idle at low or high states and two single edge, falling and rising edge, can be possible, for a given frequency, different clocks could be defined. Indeed, the **SCLK** can have 4 different modes. Those four modes are characterised by two variables CPOL and CPHA:

1. CPOL characterises the idle state of the clock: logic high and logic low.
2. CPHA specifies when to sample the data on the MISO and/or MOSI line: on the leading edge or on the trailing edge of the clock.

In the next table, one can find the four different modes.

CPOL	**CPHA**	**Mode**
0	0	0
0	1	1
1	0	2
1	1	3

Table 3.1: 4 modes possible in SPI

Practically, the difference between the modes is show in Figure 3.3.

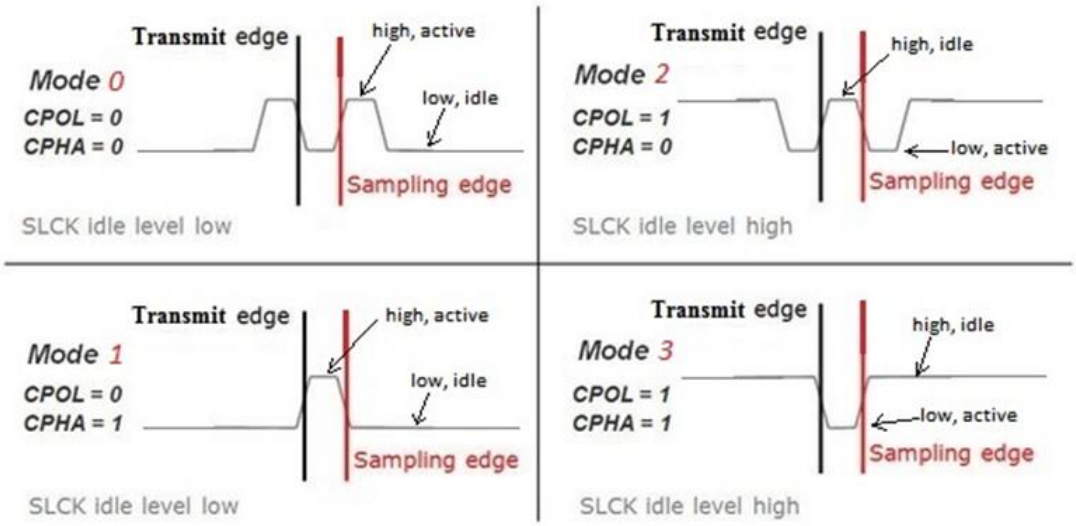

Figure 3.3: Difference between the 4 SPI modes [3]

Why take the time to explain the different possible clocks? For two main reasons. First, if the mode is not known in advance it is not possible to decode correctly because depending on the chosen mode, since the data are not sampled at the same time, the data will be interpreted differently.

A second reason, which will be covered during the decoding part of the modules, is that modules can handle different modes. Moreover, some modules have specificities that depend on the clock mode. This is the case for some modules, the ADCs, which, depending on the mode chosen, allows different functionalities. At the opposite, some modules do not support certain modes. So, the mode of every module must be clear in advance.

After this explanation, it becomes obvious why, surely, the SPI signal were the more complicated signal to acquire properly. Indeed, it is first necessary to sample at a sufficiently high frequency, to be attentive to the change of each chip select, once a select chip is activated, it is necessary to be attentive to the clock and to read the data according to the clock mode.

Only at the right moment, which must match the SPI mode, one bit must be sampled. Once the word size, typically 8bits, is reached, the byte can be recorded. Then, whenever a byte is available, it can be sent for processing. And at the end, depending on the selected chips select, it will be possible to give the specific meaning each module after decoding the byte sent properly.

In our case, it is DPC that will communicate with the different modules. So,after having described precisely how the SPI works, it will be necessary to describe the different modules and the possible interactions with the DPC. The detailed interactions between the DPC and the modules will be largely discussed in the decoding part but for now, the different modules can be already briefly described.

3.2 Modules and payloads interacting with the DPC

From the Table 3.2, the different chips select names were already teasing the 9 modules with which the OBC interacts.

The different modules, briefly discussed during the introduction, that communicate with the DPC are shown in Figure 3.4.

Bus Pin on PC104			Bus Description
H1	20	11	SPI_CS_FRAM
H1	21	12	SPI_CS_IMU
H1	22	13	SPI_CS_RAD
H1	23	14	SPI_CS_ADC1
H1	24	15	SPI_CS_ADC2
H1	25	16	SPI_CS_ADC3
H1	26	17	SPI_CS_BCN
H1	27	18	SPI_CS_BATT
H1	28	19	SPI_CS_COM

Table 3.2: Part of Table 3.2 taking only the CS signals

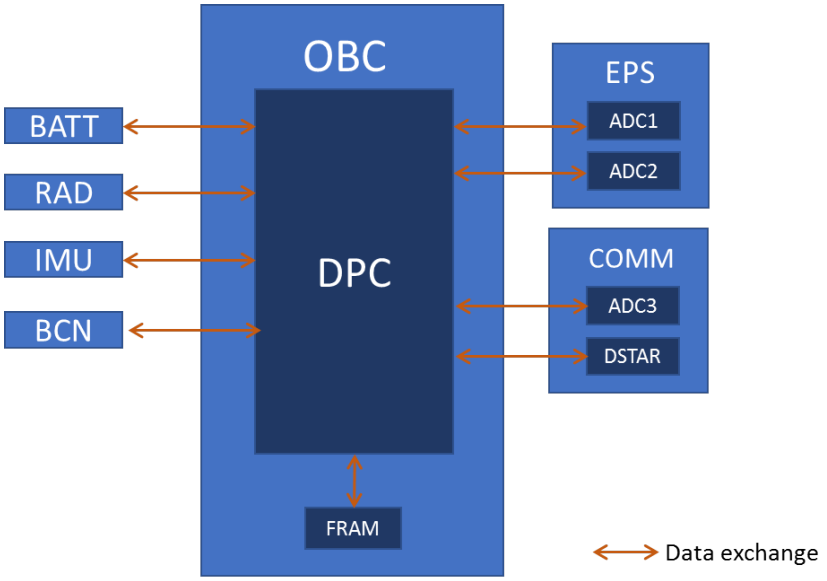

Figure 3.4: All subsystems that communicate and exchange data with the DPC through SPI

From Figure 3.4, the different modules are:

1. **FRAM**: Memory to save measurement and events before sending them to Earth.
2. **IMU**: IMU payloads.
3. **RAD**: RAD payloads.
4. **BCN**: BCN subsystem.
5. **ADC1**, **ADC2**, **ADC3**: some Analog to Digital Converter. More details below.
6. **BATT**: Detailed below.
7. **COM**: Detailed below.

ADC1 and ADC2 are located on the EPS board. The reference of the ADCs is the MAX1231 from **Maxim Integrated**. ADC3 is located in the COMM board. It the same reference. Many signals are digitized thanks to these ADCs: temperatures, voltages,... A temporary scheme of some of the signals digitized by the ADCs is shown in Figure 3.5.

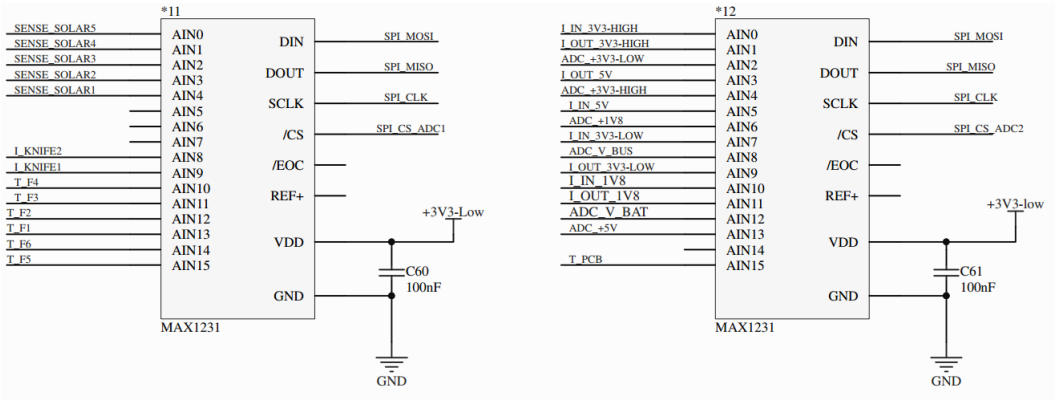

Figure 3.5: Schematic prints of the ADC. ADC1 is on the left. ADC2 is on the right. Various signals are digitized and are different for the two ADCs

The BATT stands for the batteries. One remark about the batteries is that it cannot communicate in SPI but uses another protocol communication: the I2C. In the scope of this project, it is not necessary to explain how the I2C works in details but it is like the SPI. It uses a common clock but only one data line to communicate. So, in order to communicate with it, an I2C-SPI converter is used. For clarity's sake, this converter is not shown in Figure 3.4.

The last module is the **COM**. It refers to the **COMM** subsystem and more precisely the D-STAR payload. The content of the communication is more detailed on the decoding part.

One important remark is that the clock frequency depends on the module as shown in Figure 3.6.

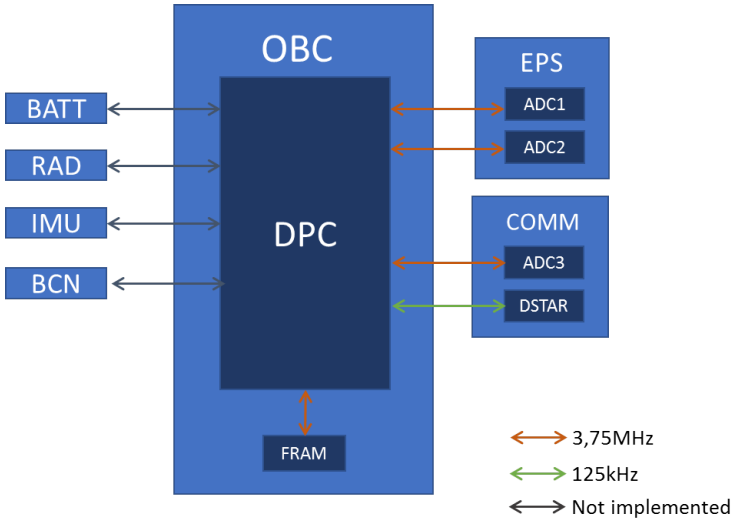

Figure 3.6: Different clock frequency used by the DPC to communicate with the module

Indeed, each module has its own maximum clock frequency that cannot be exceeded in order to communicate with it. Therefore several clock frequencies can coexist. For example, the D-STAR communicates at the maximum frequency of 125 kHz. So, in order for the DPC to communicate with the D-STAR, it will set the clock frequency at 125 kHz. However, the DPC has a maximum clock frequency which is around 3.75 MHz. Therefore, the DPC will communicate with the others modules like the ADCs and the FRAM with a clock frequency of 3.75 MHz because they can withstand this clock frequency.

3.3 Requirements to be met

To design correctly the hardware device, it is necessary to specify the requirements to be met.

The first one is the number of channels i.e. the number SPI lines. There are 3 lines for the **MISO**, **MOSI**, **SCLK** and 9 lines for the **CS**, one for each module. So, in total, the minimum number of channels must be 12.

The number of channels is mainly due to the information of which module or subsystem is selected. So, if it is not possible to find a device that can support 12 channels, one will have to find a way to reduce the number of channels. The only way to reduce the number of channels is to compress the information of which slave module is selected. But, on the other hand, it will induce more complexity on the hardware and the software side. Indeed, for the hardware part, a component that takes all the 9 **CS** lines and outputs the information of which slaves has been selected will have to be found. And for the software part, one has to be coherent with the hardware choice and decode the information consistently. So, if it is possible to avoid those complications, it is simpler to choose the number of channels that fits the applications which is 12 in this case.

The second characteristic to consider is the sampling frequency of the device. Indeed, if the communication is done at the given frequency f, according to Nyquist criteria, one must samples **at least** at twice the frequency f of the communication. Since the clock frequency is at maximum 3.75MHz, the sampling frequency must be **at least** 8MHz, which is a bit more than twice the frequency. Of course, the higher the sampling, the better.

The last constraint is of course the price. A reasonable budget was allocated to the project.

In summary, the hardware device will have to meet those requirements:

1. 12 channels: 3 for MISO, MOSI and CLK and 9 for the slave selection
2. Sampling frequency of **at least** 8MHz
3. Price less than a 300 euros

3.4 Existing device

For OUFTI-1, there was an equivalent project in which it was also necessary to read, not an SPI, but an I2C communication. The solution found was to buy a hardware device that made it possible to complete the task straight away.

Therefore, the first step was to see if an existing device could complete this task. Indeed, it is useless to reinvent the wheel. In terms of acquisition devices in general, different types of devices exist: oscilloscope, picoscope, datalogger, logic analyzer,... The purpose of this section is to understand why most of them are not relevant for this application by explaining the purpose and limitations of each of these devices.

3.4.1 Datalogger

The datalogger is typically a device that can be used for our type of application. dataloggers design devices that measure, record and analyse data. They are often provided with software that facilitates the analysis of these data.

Thus, several known suppliers sell dataloggers that are in line with the needs of this application. They allow data to be acquired and analysed in real time using the software provided with the device. It is interesting to review them because finding a device that meets the requirements directly will solve the problem of data acquisition, which is a significant part of this work.

At **NationaI Instrument**, there is a range: C Series modules. These modules *"can connect to any sensor or bus and allow for high-accuracy measurements that meet the demands of advanced data acquisition and control applications"*.

It proposes to use LabVIew to process the data as shown in the datasheet extract in Figure 3.8.

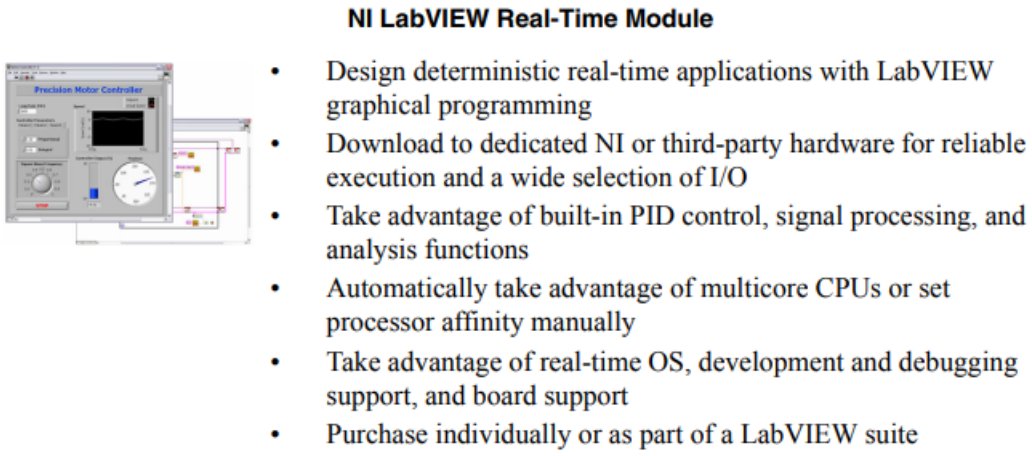

Figure 3.7: Datasheet extract of a C Series module: the **NI-9205 model.**. **LabVIEW** can be used for real time application. [4]

Among the models that allow a channel number greater than 12 is the **NI-9205 model.** However, the sampling frequency of 250 kHz is not high enough for this application. Moreover, the price close to 900 €is also quite high.

Another company, **DATAQ Instruments**, that also offers a range of dataloggers like **NationaI Instrument** shares also the same issue as **NationaI Instrument**: high prices and more importantly a sampling frequency that is not high enough. Indeed, the maximum sample rate is 200kHz...

At **Pico technology**, which is a company specializing in PC Oscilloscopes and dataloggers, there is a range of dataloggers. Each device is supplied with a software, *PicoLog*, thanks to which *"you can measure, record and analyse your data"*, according to their words. It also offers another software: the **PicoSDK**. This one could be extremely interesting because it allows to control the datalogger using the language of our choice like **LabVIEW** and **MATLAB**, or with programming languages including **C**, **C++** or **Python**. This could possibly allow to use a program wrote especially for this application that process and display the data directly.

Here, the model that would meet all the requirements is the **PicoLog 1000 Series** [5]. Indeed, it has 16 channels, a reasonable price of 139 €. However, it should be remembered that in our case, the goal is to read the data in real time, to process and display them in real time. Thus, with the **PicoLog 1000** the mode that corresponds to a real-time playback is the "streaming mode". However, as shown in Figure 3.8, according to the datasheet, the sampling frequency of 100 000 samples per second is not sufficient.

Specifications

Input		
Model	**PicoLog 1012**	**PicoLog 1216**
Analog inputs	12	16
Resolution	10 bits	12 bits
Accuracy	1% of full scale	0.5% of full scale
Maximum sampling rates: PicoScope PicoLog PicoSDK (block mode) PicoSDK (streaming)	 1 MS/s[1] 1 kS/s[2] 1 MS/s[1] 100 kS/s[1]	
Capture memory PicoScope (and PicoSDK block mode) Sample rates over 100 kS/s: Lower sample rates: PicoLog (and PicoSDK streaming mode):	 8000 samples [1] 1 million samples [1] Up to available PC storage	

Figure 3.8: Datasheet extract of the **PicoLog 1000**: streaming mode sample frequency is only 100k samples per second [5]

3.4.2 Logic analyzer

The last type of acquisition devices is the logic analyzer. The best way to understand the utility of a logic analyzer is to compare it to an oscilloscope and understand their major difference. Indeed, the main purpose of an oscilloscope is to provide as much details as possible on a signal. However, in the case of a logic analyzer, the purpose is to detect logic levels. Thus, for each signal, the logic analyzer compares this value to a threshold voltage and determines whether the value is a High or a Low as shown in Figure 3.9. Often, this threshold voltage can be modified to be able to operate to fit any situations. Another feature of logic analyzers is advanced triggering; it allows to initiate the capture of data only when specific conditions are encountered.

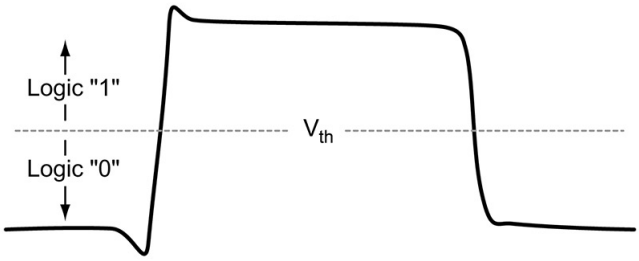

Figure 3.9: A threshold voltage is used to determine whether it is a High or a Low level [6]

Therefore, this type of device is typically suitable for debugging systems that use digital data or checking that a data communication is going on as planned. In the case of SPI signals, this is exactly the goal: to read digital data with no need for analog details.

3.5 Operation of logic analyzers

A wide variety of logic analyzers exist on the market. Each of them has a maximum number of channels available, a maximum sampling frequency,.. and some of them have a software that goes the logic analyzer. This software allows seeing the different waveform. An example can be seen in Figure 3.10.

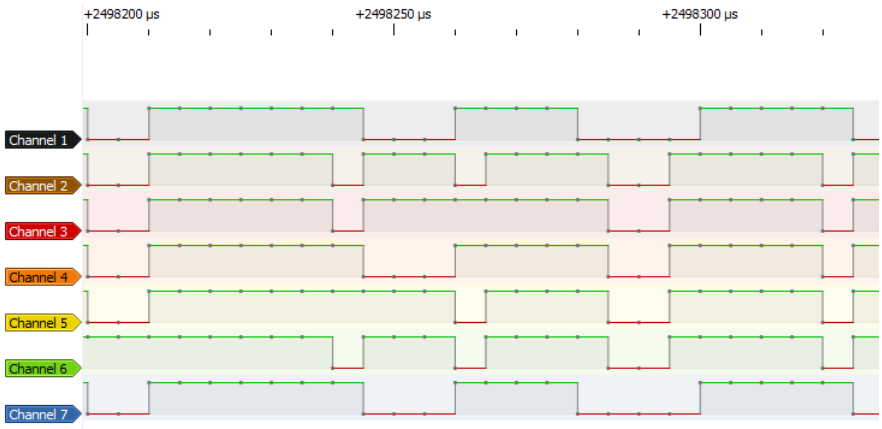

Figure 3.10: Example of the software used by a logic analyzer

Logic analyzers are usually used for debugging but can also allow extracting the sampled data and saving it in a file. The most known and popular is without contest the logic analyzer from the **Saleae** company. They are popular for their software. Indeed, the interface is clean and easy to use. For the hardware constituting a logic analyzer, as we'll see in the next section, it is not complicated. It is interesting to go through the hardware because the majority of the logic analyzer existing on the market are using the same components.

3.5.1 Hardware involved

The basic goal of a logic analyzer is to be an interface between the signal to be measured and the computer. In this section, an overview of a typical logic analyzer will be given, without going into the details of every components.

The core of a logic analyzer is a chip called **Cypress FX2LP microcontroller**, which will be shorted to **FX2LP** in the following sections. In Figure 3.11, one can find a bloc diagram constituting the FX2LP microcontroller.

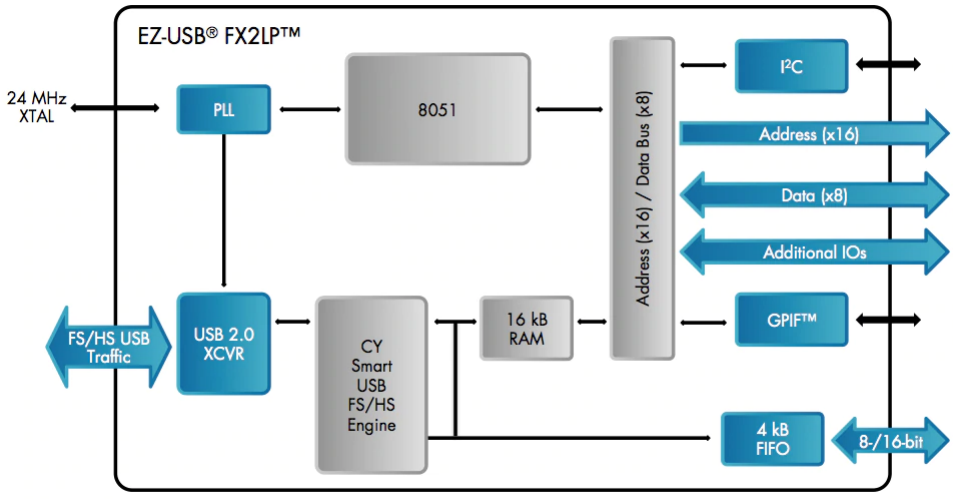

Figure 3.11: Block diagram of the Cypress FX2LP microcontroller [7]

Usually, this is used to send data from a computer to external components like FPGAs, microcontrollers,... In the case of a logic analyzer, it is the contrary, the data is send to the computer. The important part of the diagram is shown in Figure 3.12.

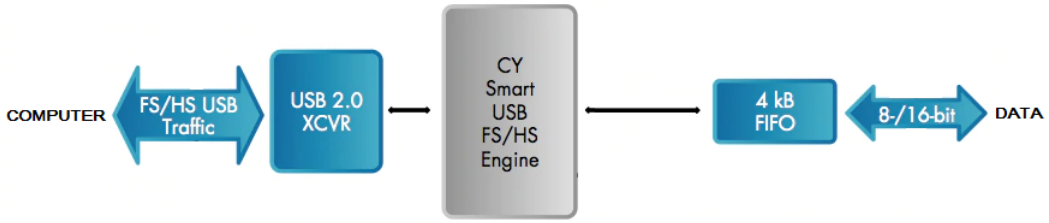

Figure 3.12: Data exchange between a logic analyzer and a computer [7]

It exists a fast path between the USB and the parallel bus of 8/16 bits (can be 8 bits or 16 bits) in width. So, the data are read in the parallel bus, packed into USB packets and send into the computer.

Therefore, the logic analyzer will simply sample the bus asynchronously. It means the sampling frequency is independent of the recorded signal. So, it is up to the user to choose a correct sampling frequency i.e. at least twice the highest frequency contained in the signal (the higher, the better). The maximum sampling frequency depends on the number of the channels used. For example, if only 8 channels are used, the maximum sampling frequency is 24 MHz and 12MHz if 16 channels are used. All the operations are supervised by the 8051 core (an 8-bits microprocessor) that can be seen in Figure 3.11).

In summary, the **FX2LP** can be used thanks to its 8/16 parallel bits bus to read the signals and send it to the computer. However, the **FX2LP** is not sufficient to have a proper logic analyzer. Indeed, some input protection should be added to protect the **FX2LP**. Also, since there are no memory to store the data inside the **FX2LP**, the only mode possible is a streaming mode.

The advantage of the streaming mode is that the storage is not an issue since the computer storage will be the upper limit. However, if the computer is solicited by other USB hub than the **FX2LP**, congestion on the USB bus can happen. This could lead to an interruption of the streaming capture. Indeed, since the **FX2LP** is using USB 2.0, the maximum bandwidth is 200MHz. So, if the signal to be sampled was 24MHz, a small interruption could lead to an interruption of the capture. There are several ways to avoid capture interruption. For example, some logic analyzers are not using USB 2.0 but USB 3.0 or PCI express. Indeed, both have a higher bandwidth than the USB 2.0.

Another way to improve this is to add some extra RAM chips or SDRAM. That way, the data are stored in the RAM inside the logic analyzer before sending to the computer. Therefore, in this case, it is not a streaming mode but is not subjected to interruption anymore. This solution is the most encountered in the market.

In conclusion, the **FX2LP** is the key to easily interface a logic analyzer to a PC. It was important to spend some time on it because whatever the logic analyzer, this element will be present. This will make it easier to explain the final components of the logic analyzer.

3.6 Hardware involved in SPI signal acquisition

Finally, an illustration of the logic analyzer that allows to acquire and send the SPI signal to a PC is shown in Figure 3.13.

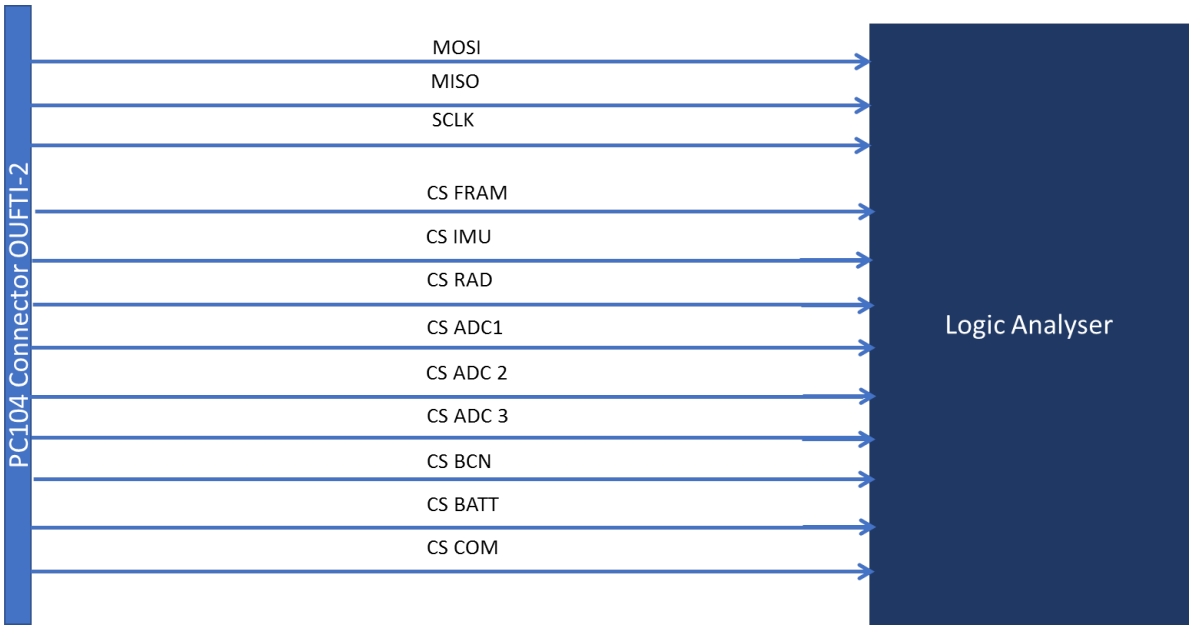

Figure 3.13: Illustration of the SPI acquisition. 12 channels are used: SCLK, MISO, MOSI and the 9 CS

Once the hardware solution has been found, it must, above all, allow an easy interface with the PC. This will drive the choice of the logic analyzer as discussed in the very next section.

3.7 Sigrok: easy interfacing to a computer

Once the hardware allowing an easy interfacing with the PC has been found, it is necessary to find a way to retrieve the data sent in the PC. It turns out that a software specialised in signal analysis exists and is called Sigrok. Sigrok is defined as *"a portable, cross-platform, Free/Libre/Open-Source signal analysis software suite"* [8]. The main work of Sigrok was to write reusable libraries that can be used to communicate with the hardware used in many logic analyzers. That way, once the hardware is found, an easy way to communicate with the PC is to use the softwares developed by Sigrok.

3.7.1 Libraries

This section is about the library upon which the software are based. So, it is necessary to understand it because in the scope of the project, it was necessary to perform some modifications on it. The main library, called **libsigrok**, is used to be the interface with the logic analyzer. It **supports many devices and especially the FX2-based logic analyzer**. It is written in C. With this library, many front-end can be built upon this library. The most well-known program that is based on this library is the graphical interface: PulseView. [21]

PulseView makes it very easy to read and analyse digital signals by displaying each of their tracks in parallel. For example, to illustrate this, consider an SPI communication between the DPC and a module. Thanks to PulseView, it is easy to observe in Figure 3.14 the different signals involved.

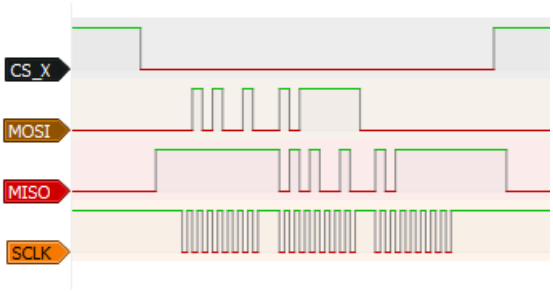

Figure 3.14: SPI communication displayed in PulseView

Thanks to PulseView, it becomes simple to read and debug a microcontroller that would use this protocol to communicate with a module.

Another important and very useful library is called **libsigrokdecode**. Thanks to this library, it is possible to create all types of decoders. A decoder is used to decode certain communication protocols. For example, the list of supported protocols includes UART, I2C, walrus,... And one that is particularly interesting in our case: the SPI.

The library was written in C. It provides an API to allow you to develop a whole bunch of decoders. Indeed, since the project is open-source and to allow as many people as possible to contribute, all decoders are written in Python based on the API provided by the **libsigrokdecode** library. Indeed, Python is easier and more popular than C, and thus facilitates the creation of decoders. Indeed, thanks to the API provided by sigrokdecode, it is not necessary to understand all the implementation details but only to focus on writing the decoder.

Thus, if a person wants to debug a communication protocol that would not already be available in the library, he has two choices: either he saves a communication session using PulseView by exporting the read data and post process in another program (Matlab, Java, Python,...) or he creates a decoder based on **libsigrokdecode** and can read the decoded communication directly on PulseView. If we continue with the example of the SPI communication seen in Figure 3.14, since an SPI decoder already exists, it is possible to have the communication decoded directly! The result can be seen in Figure 7.1.

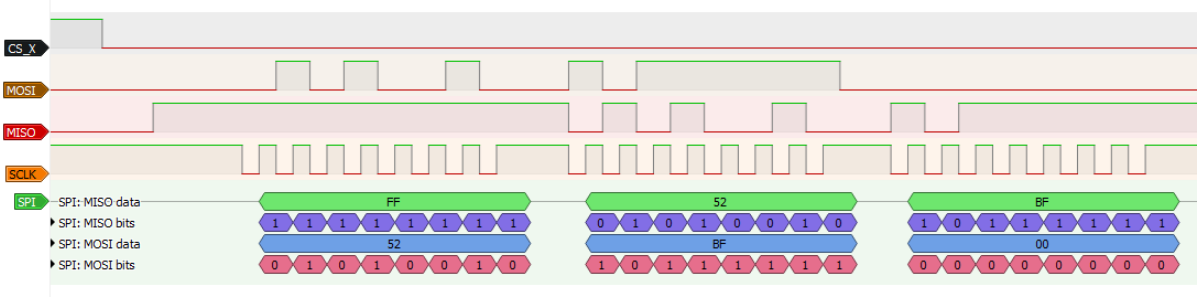

Figure 3.15: Same SPI communication as in Figure 3.14 but the bottom lines show the data sent decoded in terms of bits and bytes. In this case, the module (MISO) sends successively 0xFF, 0x52, 0xBF and the master sends 0x52, 0xBF, 0x00

At this stage, we have the hardware solution that will acquire the communication. But also the software that can communicate with the hardware to get those data into the PC and **decode** them!

3.8 Choice of the logic analyzer: comparison between logic analyzers

It is necessary to find a logic analyzer that is compatible with the Sigrok. In the official Sigrok website, a list of logic analyzers that are compatible with the Sigrok suite is available [9]. Recall that certain constraints must be respected:

1. 12 channels: 3 for MISO, MOSI and CLK and 9 for the slave selection
2. Sampling frequency of **at least** 8MHz
3. Price less than a 300 euros

In the market, the number of channels is standard: either 8 or 16 in 95% of cases. Since we have 12 channels to read, we have to take a 16-channel logic analyzer. At the end, in Table 3.3, one can compare several logics analyser that meets the constraints.

Device	**Saleae Logic Pro 16**	**Logic16 clone**	**DSLogic Plus**	**LAP-C 16064**
Max Sample Rate.	125Mhz when use all 16 channels.	16MHz when use all 16 channels	20MHz when use all 16 channels.	20MHz when use all 16 channels.
Price	$999	$70	$149.00	$225

Table 3.3: Comparison of existing logic analyzers

All of them have in common that they use an **FX2LP**. So, at the hardware level, all the solutions are relatively similar.

The Saleae Logic Pro 16, at $999, does not use Sigrok's programs but has its own dedicated software. The reason for the high price is not the hardware but the software it comes with. It is the most professional logic analyzer. This is typically the logic analyzer that a company would choose to go very fast. However, the price is excessive compared to the possible alternatives. One of them is precisely a clone of Saleae Logic Pro 16 but of much lower quality. The DSLogic Plus is another logic analyzer. The quality is superior to the simple copy of Saleae and it is compatible with Sigrok programs. For the example, another compatible logic analyzer, the LAP-C 16064, has been added but the price difference is not justified, it does not offer anything more compared to the DSLogic Plus.

3.9 Dslogic Plus

In this section, more details specific to the DSLogic will be provided.

3.9.1 Capabilities

First, the characteristics of the DSLogic will be presented. An illustration of the DSLogic is shown in Figure 3.16.

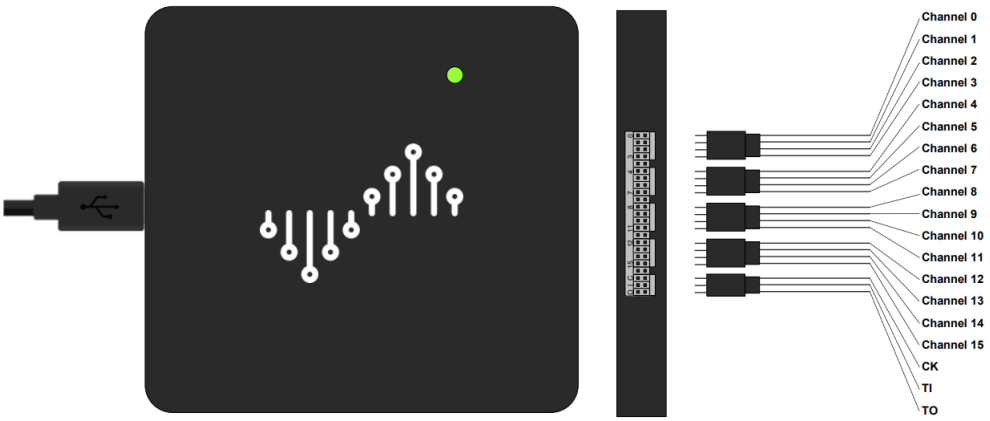

Figure 3.16: Image of the Dslogic: 16 channels and CK (clock input), TI (trigger input), TO (trigger output). [10]

The number of channels is 16 and it can reach a sampling frequency up to 400 MHz using only 4 channels, if accompanied by the appropriate probes.

However, 400 MHz is the maximum sampling frequency. In practice, two modes can be used: stream mode or buffer mode. In stream mode, data can be transferred to PC in real-time using directly the PC storage. In this mode, the sampling frequency, if all the 16 channels are used, is limited to 20MHz. While in buffer mode, data are stored in the on-board memory and transferred to PC after the capture is finished. In this mode, the sampling frequency is limited to 100MHz if all the 16 channels.

The DSLogic allows various threshold voltage as shown in Figure 3.17. This is useful in order to be compatible with to most of voltage standard.

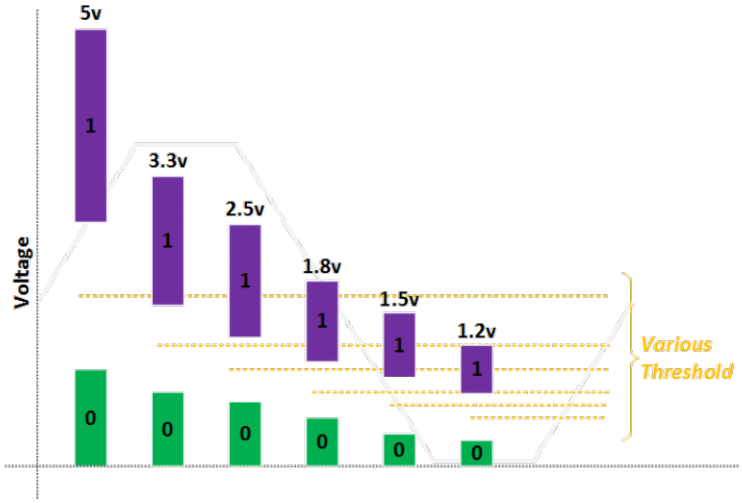

Figure 3.17: Adjustable threshold voltage: 5V, 3.3V, 2.5V, 1.8V, 1.5V, 1.2V. [10]

3.9.2 Hardware involved

In this section, the hardware involved will be presented. First, an annotated view of its PCB is shown in Figure 3.18.

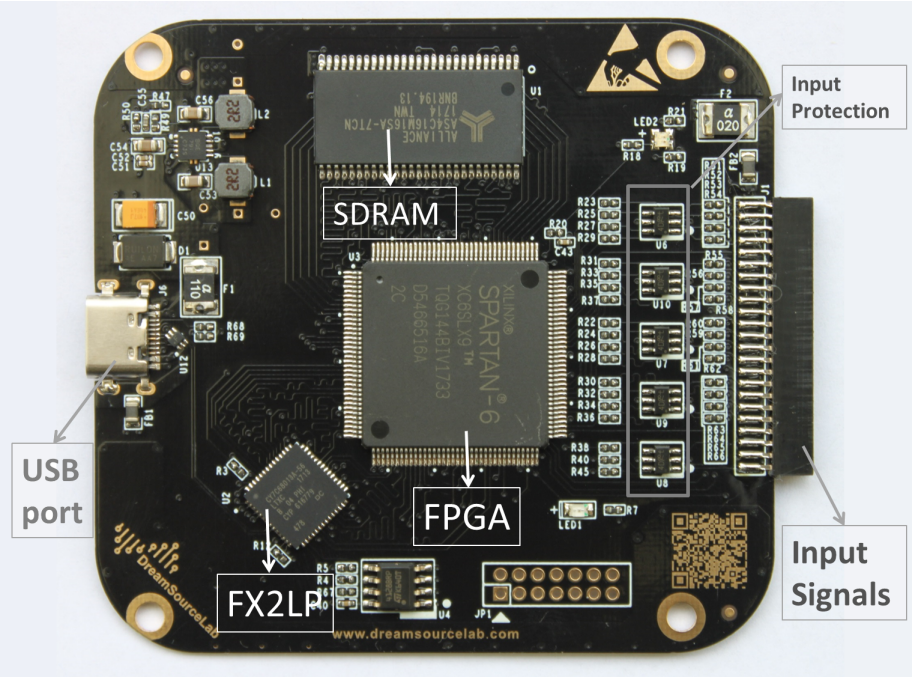

Figure 3.18: an annotated view of the Dslogic PCB. An SDRAM, protection input components, the FPGA and the **FX2LP** chip are highlighted. [11]

First of all, one can notice the **FX2LP** in the bottom left which is used to send the data to the PC. We already discuss its operation before.

And between the **FX2LP** and the inputs signals is an FPGA, the XC6SLX9 from Xilinx. It is the FPGA that sample the input data, packs the data and store them into an internal buffer. It then transfers the data to the **FX2LP** which packs them into USB packets before sending them into the PC. This is typically what is done when the logic analyzer is used in streaming mode.

But, in buffer mode, the data is not sent directly to the PC but to a SDRAM. This is useful when the USB data transfer (200Mbits/s) is too low regarding the frequency of the input signals. Once the SDRAM is full, the FPGA send the data stored into the PC using the **FX2LP** like before.

To protect the FPGA, some protection is present between the FPGA pins and the inputs signals. It consist of an network of diodes to protect against high voltage or voltage spikes.

Chapter 4

Acquisition of the non-SPI signals

Apart from SPI signals, as explained in Chapter 2, there are other types of signals that needs to be acquired. As a reminder, here is the table that summarized the essential information concerning these signals

Signal	Type of signal	Total number	Description
Fixed voltage	Analog	5	Power voltage
Idiode	Analog	1	Battery voltage
ENABLES	Digital	7	Allows the DPC to enable or disable a subsystem
BATT_FULL	Digital	1	Satellite powered off or on

Table 4.1: Main information concerning the non-SPI signals

4.1 General idea

For analog signals, the goal will be to find a hardware solution to digitize analog data and transmit them into a PC. One common way is to use a microcontroller and find a way to communicate the information to the PC via USB. The concern will be that the final hardware product will be impractical if it uses two USB ports. Indeed, recall that the Dslogic already uses an USB port. But, one the other hand, one must also recall that the Dslogic still contains 4 free channels.

So, one could keep the idea of using a microcontroller that allows to digitize all analog data. But, instead of connecting this microcontroller to the PC, which would require to investigate an efficient way to interface it to the PC, one can simply transmit the digitalized data to the 4 still free channels of the DSLogic. This solution is illustrated in Figure 4.1

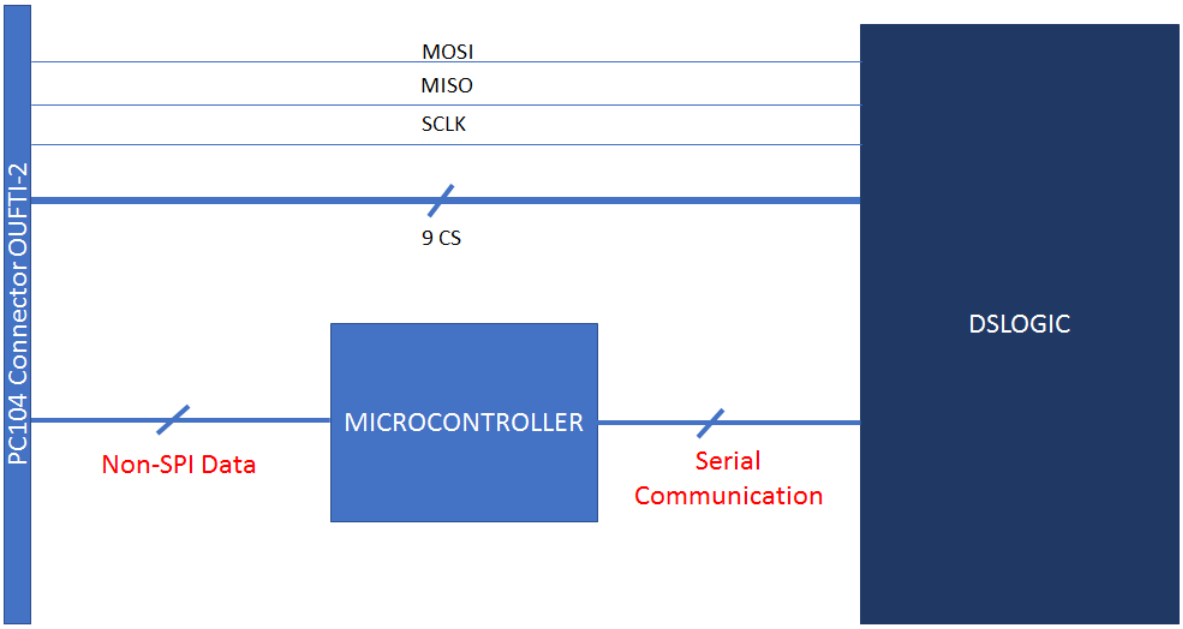

Figure 4.1: Illustration of how to read non-SPI data and transmit it through the PC using the DSLogic

To determine which microcontroller to use, it is essential to specify the needs that it will have to meet. To do this, it is necessary to review the non-SPI signals that will need to be read. Once this is done, the information and how to send the data to the DSLogic will be discussed right after.

A key point to understand about these signals and what makes them intrinsically different is that there is no real urgency to read these data. Indeed, for the SPI signal, it is essential to be as reactive as possible in order not to miss any data exchange. While for non-IPS signals, the stakes are not the same, it is quite the opposite. It is simply a matter of checking whether the voltages are at their expected value for fixed voltages and checking the evolution of the battery charging.

Therefore, for this application, timing is not the main constraint. Indeed, a measurement of these voltages every second for example will be more than sufficient.

4.2 Non-SPI signals: Analog signals

As a reminder, the analog non-SPI signals are recalled in the following Table.

Signal	Type of signal	Total number	Description
Fixed voltage	Analog	4	Power voltage
Idiode	Analog	1	Battery voltage
BATVBUS	Analog	1	Voltage before DC/DC converters

Table 4.2: Analog non-SPI signals description

It is also worth remembering the details of each of these signals as shown in the following Tables.
What is common for both analog signals, i.e. fixed voltages and battery level, is the need for 8 to 12 bits resolution. So, to read this data it will **be necessary to use an 8 to 12 bits ADC of 6 channels.** This is the first requirement. After reading those signals, it will be necessary to send them to the DSLogic.

Signal	Maximum Value	Resolution	Description
Idiode	8.4V	8-12 bits	Battery voltage
BATVBUS	8.4V	8-12 bits	Voltage before DC/DC converters
Signal	**Expected Value**	**Resolution**	**Description**
Fixed voltage: 5V, 3.3V (high), 3.3V (low) , 1.8V	Fixed value	8-12 bits	Power voltage

Table 4.3: Details of analog non-SPI signals

4.3 Non-SPI signal: Digital signals

For non-SPI digital signals, to read the status of each of these signals, 8 pins are needed in total. Therefore, the microcontroller must have 8 pins capable of recognising a high state (3.3V) and a low state (0V). So, **the second requirement is to have 8 digital pins.**

Signal	Type of signal	Total number	Description
ENABLES	Digital	7	Allows the DPC to enable or disable a subsystem
BATT_FULL	Digital	1	Satellite powered off or on

Table 4.4: The 8 digital signals

To send this information, one single byte can be sent, each bit corresponding to one digital signal. As shown in Figure 4.2, the structure of the byte will be as follows:

ENABLE_RAD	ENABLE_IMU	ENABLE_BCN	ENABLE_COM_RX	ENABLE_COM_DSTAR	ENABLE_COM_PA	ENABLE_COM_TX	BAT_FULL
BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0

Figure 4.2: Illustration of 1 byte contain all the digital information

4.4 Choice of the serial communication

In summary, once the microcontroller has read all the non-SPI signals, it has to send those information. Therefore, one must choose a serial communication. The three most common are the UART, I2C and SPI. **The serial communication chosen is the SPI** and it becomes one more requirement for the microcontroller. This choice will be clearly explained later on.

4.5 Choice of the microcontroller

For the choice of the microcontroller, as discussed before, the needs are not critical, the majority of them can perform this kind of task.

It was therefore decided that an Arduino Uno card would be sufficient. As a reminder, the Arduino Uno is a very popular microcontroller board based on the ATmega328P [12] and is widely used for prototyping purposes. One of its main advantages is the simplicity to code and debug, thanks to dedicated IDE software.

It is then necessary to check if the Arduino Uno has all the requirements:

1. 10-bit ADC of 6 channels,
2. 8 digital pins capable of recognising a high state (3.3V) and a low state (0V),
3. Presence of serial communication chosen is the SPI

To verify that this is the case, it should be remembered that the Arduino is based on the ATmega328P. So, it will be necessary to look in its datasheet.

4.5.1 10-bit ADC of 6 channels

As shown in figure 4.3, there is indeed an ADC with 6 channels with a resolution of 10 bits.

24. Analog-to-Digital Converter

24.1 Features

- 10-bit Resolution
- 0.5 LSB Integral Non-linearity
- ± 2 LSB Absolute Accuracy
- 13 - 260µs Conversion Time
- Up to 76.9kSPS (Up to 15kSPS at Maximum Resolution)
- 6 Multiplexed Single Ended Input Channels
- 2 Additional Multiplexed Single Ended Input Channels (TQFP and QFN/MLF Package only)
- Temperature Sensor Input Channel
- Optional Left Adjustment for ADC Result Readout
- 0 - V_{CC} ADC Input Voltage Range
- Selectable 1.1V ADC Reference Voltage
- Free Running or Single Conversion Mode
- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

Figure 4.3: ADC characteristic in the Arduino Uno. Presence of six 10-bit channels [12]

An important detail is that analog inputs cannot support too high input voltages. Indeed, as shown in Figure 4.3, the input voltage is limited to V_{CC}, the supply voltage of the Arduino. In our case, $V_{CC} = 5$ V.

Thus for Idiode, it will be necessary to reduce this tension before reading it with an Arduino. Therefore, a resistive voltage divider with two resistor of 10kΩ that divides the voltage in half will be sufficient since $8.4\text{V}/2 = 4.2\text{ V} < V_{CC}$.

4.5.2 8 digital pins able to recognise a 3.3V high state

As a reminder, the Arduino must be capable of distinguishing between a Low at 0V and High at 3.3V digital signal. In Figure 4.4, the digital pin characteristics are shown.

26.2 DC Characteristics

T_A = -40°C to 85°C, V_{CC} = 1.8V to 5.5V (unless otherwise noted)

Symbol	Parameter	Condition	Min.	Typ.	Max.	Units
V_{IL}	Input Low Voltage, except XTAL1 and RESET pin	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	-0.5 -0.5		$0.2V_{CC}$ (1) $0.3V_{CC}$ (1)	V
V_{IH}	Input High Voltage, except XTAL1 and RESET pins	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	$0.7V_{CC}$ (2) $0.6V_{CC}$ (2)		V_{CC} + 0.5 V_{CC} + 0.5	V

Figure 4.4: Digital pin characteristics. The **IL** in V_{IL} stands for **I**nput **L**ow and the **IH** in V_{IH} stands for **I**nput **H**igh. [12]

A Low will be unambiguously recognised if it is lower than: 0.3 x V_{CC}. This will be the case since a common ground will be shared between the Arduino and the digital signal. Therefore, there is no problem to recognise a 0 V digital signal.

However, it will not be the case for a 3.3V high state. Indeed, the minimum value to be recognised as a High is: V_{IH} = 0.6 x V_{CC}. If you are superior to V_{IH}, it is recognised as high. In our case, we want $V_{IH} \leq 3.3V$ so that the 3.3V will be higher than V_{IH} and will be recognised as High

In the worst case where, the value of V_{IH} is maximum when V_{CC} = 5.5V. So, V_{IH}= 5.5 x 0.6 = 3.3 V. It should be noted here that we are exactly at the limit.

If a small power supply disturbance that induces a V_{CC} = 5.6V, then the value of V_{IH} will become 0.6*5.6V = 3.36V. This means that $V_{IH} \leq 3.3V$ is not satisfied and digital signals, in high state, will not be recognised as high. And even if the power supply is stable, if instead of 3.3V, the high signals are lower than this value, for example at 3.28V, in theory, they will not be recognised as high.

Therefore, it is obvious that being exactly in the limit case is not acceptable, from time to time the supply voltage varies or that the high signals are not exactly at 3.3V.

The solution is to use an intermediate component between the Arduino's inputs and the digital signal to increase the voltage of the signals to be far from the limit $V_{IH} = 3.3V$.

The component used is called a *voltage-level Translator*. One example from **Texas Instruments** is the **TXB0104**. It takes as input any supply voltage from 1.2 V to 3.6 V and outputs any supply voltage from 1.65 V to 5.5 V, provided that the output is greater than the input. **TXB0104** can take 4 inputs so, for 8 signals, two of them will be needed. [13]

With this, the input digital signals will be increased to be far from the limit $V_{IH} = 3.3V$ by increasing them to 5V. In conclusion, by using 2 **TXB0104**, the 8 digital inputs can be recognised as High by increasing the 3.3V to 5V.

4.5.3 SPI communication available

The Arduino Uno can easily use SPI communication thanks to its dedicated library. Four pins, pin 10 to 13, are dedicated to the use of the SPI.

4.6 Illustration of the acquisition of the non-SPI signals

As shown in Figure 4.5, by combining level translator and Arduino, all non-SPI signals can be read and sent via SPI to the Dslogic.

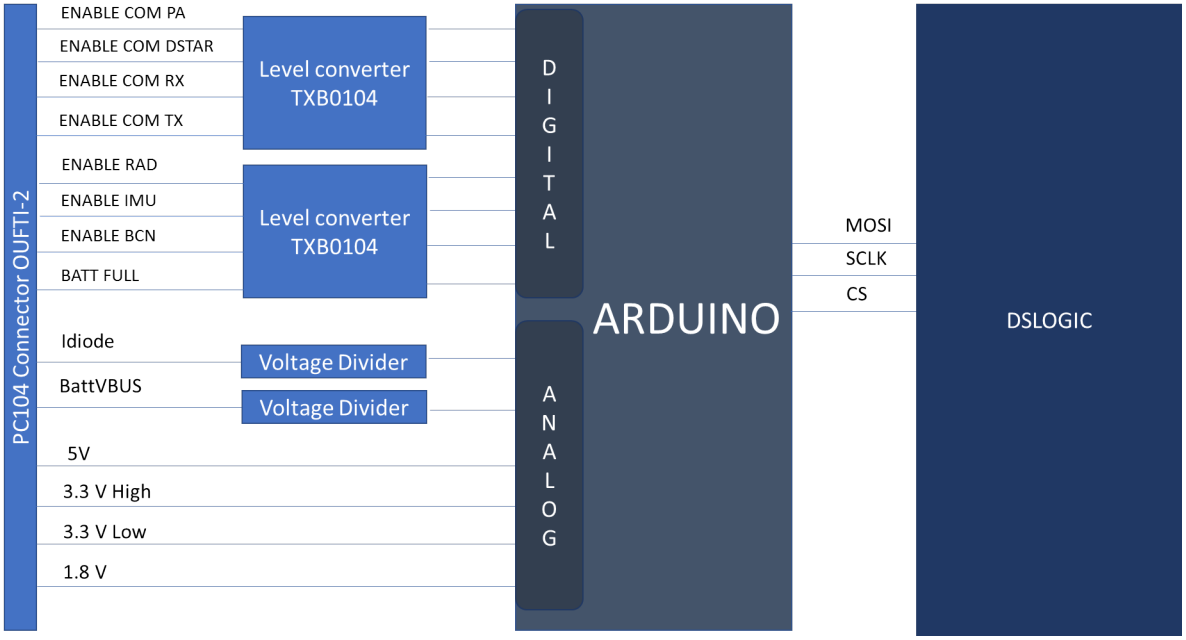

Figure 4.5: All the non-SPI signals are read by the Arduino and sends the data in a specific format through SPI to the DSlogic

It should be noted that since communication is only from the Arduino to the Dslogic, it is not necessary to connect the MISO.

4.7 Format of the data sent to the DSLogic

Most of the time in serial communication, the data is sent in 8-bit packets or 1 byte. The fact that there are 10-bit data for non-SPI data, it is necessary to define a format to send these data correctly.

To send the 10bits data, it will be necessary to send 2 bytes: the first one will correspond to the MSB (most significant Byte), the second will correspond to the LSB (least signifying bytes) as shown in Figure 4.6.

MSB								LSB							
X	X	X	X	X	X	BIT 9	BIT 8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0

Figure 4.6: Illustration of how 10bits data can be split into 2 bytes

For MSB, 2 bits are used to complete the 8 LSB bits. Thus, 6 bits of the MSB are not used for now.

However, if the Arduino only sends the 10-bit measurements as shown in Figure 4.6, it is not possible to deduce the origin of measurements. One way would be to define a send order as shown in the following Table.

Analog Data	Send order
Idiode	1
5V	2
3.3V (high)	3
3.3V (low)	4
BATVBUS	5
1.8 V	6

Table 4.5: Send order of the analog data into the SPI bus

A more robust way is to use the bits not used by the MSB to uniquely define each voltage. Indeed, before each voltage, an ID specific to each voltage can be sent. The list of IDs is presented in the following table.

Data	ID Number
Digital	0
Idiode	1
5V	2
3.3V (high)	3
3.3V (low)	4
BATVBUS	5
1.8 V	6

Table 4.6: ID number of the data into the SPI bus

The same goes for non-SPI digital signals, it will be preceded by a byte containing its ID to be recognized.

Since it is required to define 7 IDs, it is necessary to allocate 3 bits (log2(6) = 2.81) on the MSB. The modified data format is shown in Figure 7.13 for analog data and in Figure 7.14 for digital data.

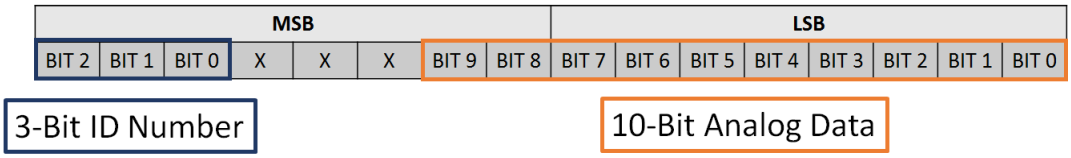

Figure 4.7: Illustration of the analog data structure: 10 bits analog data are preceded by 3 bits ID Number

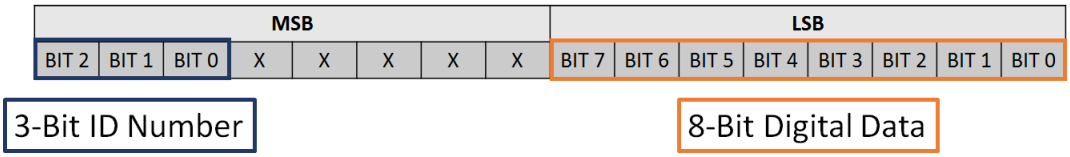

Figure 4.8: Illustration of the digital data structure: 8 bits digital data are preceded by 3 bits ID Number

Chapter 5

Summary of the data acquisition strategy

To summarize data acquisition strategy of the EGSE, illustrated in Figure 5.1, it should be remembered that 23 signals must be read from a PC104 connector. Two types of signals have been distinguished:

1. SPI signals
2. Non-SPI signals

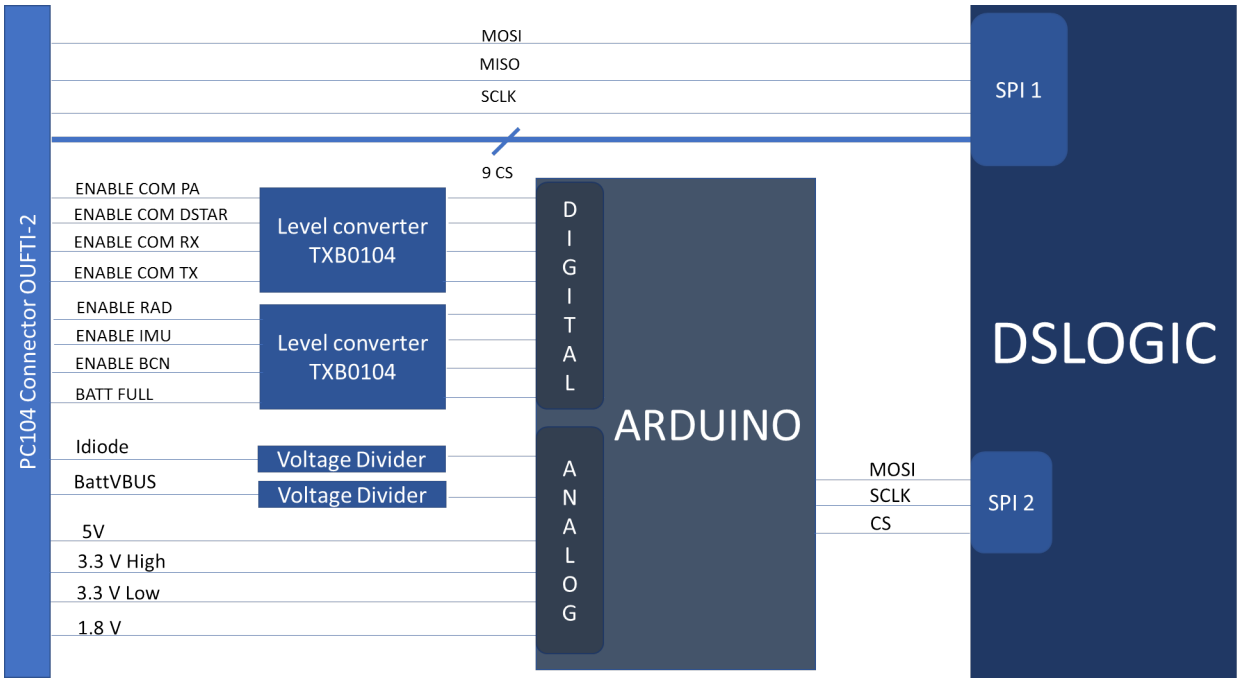

Figure 5.1: Illustration of the data acquisition strategy of the EGSE. The final product will only require a single USB connection and all signals eventually passes through the Dslogic.

For SPI signals, the DSLogic has been used to acquire them. Indeed, the DSlogic has the advantage of being compatible with the sigrok suite that allows easy interface with the PC.

For non-SPI signals, the acquisition was done by using a simple Arduino Uno. To interface with the PC, the DSlogic is reused. Indeed, channels remain available and can be used to receive Arduino data in SPI.

Thus, the Dslogic must acquire the data of two separate SPI communications:

1. SPI1: The basic SPI signals present in the PC104 connector
2. SPI2: The data sent by the Arduino.

The following table gives a clear description of the situation

SPI Communication	**Module**	**Description**
SPI-1 Master: DPC	ADC 1	SPI communication between the DPC and different modules
	ADC 2	
	ADC 3	
	FRAM	
	COMM	
	BCN	
	IMU	
	RAD	
	BATT	
SPI-2 Master: Arduino	Dslogic	SPI communication between the Arduino and the Dslogic to send the digitized analog Data

Table 5.1: Distinction between the two separate SPI communications read by the Dslogic

Chapter 6

Test and failure of the data acquisition part

The purpose of this chapter focused on the difficulties encountered during the various tests. The purpose of these tests was to validate that the data acquisition strategy could work.

6.1 Sigrok-cli

For the moment, the only program in the Sigrok suite that has been discussed is their GUI (PulseView) which allows displaying the signals from each channel. As a reminder, Figure 6.1 shows the results that can be obtained with PulseView

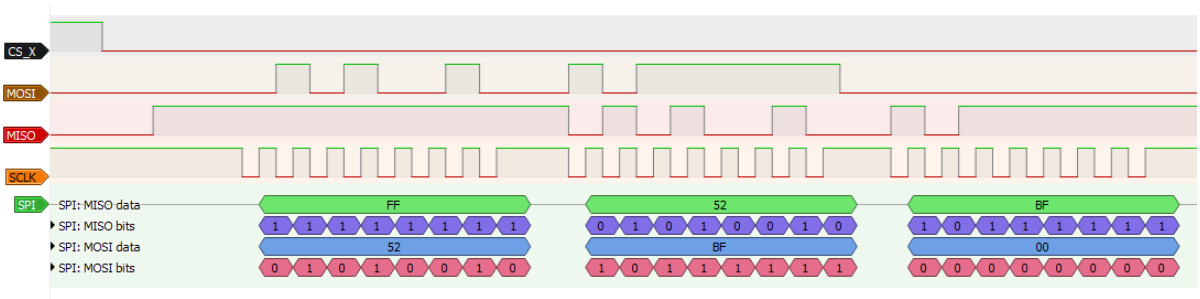

Figure 6.1: Illustration of PulseView. Ideal to obseve digital data

However, in the context of this project, it is not necessary to show the communications. The ideal would be to acquire the data that is read on each channel and, instead of displaying it, it should be sent to a program that will make sense of the data that is being communicated. In concrete terms, the data that pass through the MISO/MOSI lines will be decoded according to the module with which the DPC communicates.

It turns out that the Sigrok suite contains software named Sigrok-cli. It is defined as " *is a cross-platform command line utility for the Sigrok software. It cannot display graphical output, but is still sufficient to run through the whole process of hardware initialization,* ***acquisition*** [...]"

6.1.1 Useful options

To work properly, it is necessary to specify a number of options to connect properly to the logic analyzer. There is a wiki page to have an explanation of all the options that are available [14]. However, the purpose of this section will be to only explain the options I have used and that are interesting in the

context of the project.

The easiest way is to first see the command used and to analyse it step by step. The command used is the following:

sigrok-cli -c "samplerate=10MHz:continuous=on" –driver=dreamsourcelab-dslogic -P SPI:miso=3:clk=0:cs=1

This expression can be easily understood by explaining one option at the time:

- **--driver=dreamsourcelab-dslogic**.
  It is necessary to specify which logic analyzer is connected. For each type of logic analyzer, a firmware is available and it is necessary to specify which firmware should be used by indicating the logic analyzer that is used. Here, the logic analyzer is a Dslogic so it is necessary to indicate the dreamsourcelab-dslogic driver. To know which exact name to indicate, refer to the Supported hardware page (on the official website: https://sigrok.org/wiki/Supported_hardware)

- **-c "samplerate=10MHz:continuous=on"**
  -c stands for config. It is used to configure the most important option in this type of device: the sampling frequency. Indeed, most often, the options chosen are a number of samples (10M samples for example) or an acquisition time (during 1s, for example). Also, it allows you to read the data continuously.

- **-P SPI:miso=3:mosi=2clk=0:cs=1**
  The logic analyzer reads the data, and it has an option - P SPI **to decode the data on the fly**. To do this, you must specify the channels needed to decode.

- **-B SPI**
  This option allows the data to be communicated to another program. This aspect is discussed below.

The last option (**-B SPI**) allows the data to be communicated to another program. This aspect is discussed in section 6.3.

6.2 SPI Decoder

In this application, it is obviously the SPI decoder that is interesting because the communication is done in SPI. The main advantage, with the easy interfacing with the PC, is the presence of an SPI decoder present in the libsigrokdecode library. However, in the basic implementation, the decoder only considers the case of a communication between only one master and one slave. Here, several slaves must be considered. It is therefore necessary to modify the SPI decoder to decode the communication taking into account several slaves. In the decoding part, we will see in detail how this decoder has been modified to work with several slaves.

To go into the details of the basic implementation, as discussed during the presentation of the SPI protocol, the clock mode must be known so that the data can be sampled at the right time, as shown in Figure 6.2.

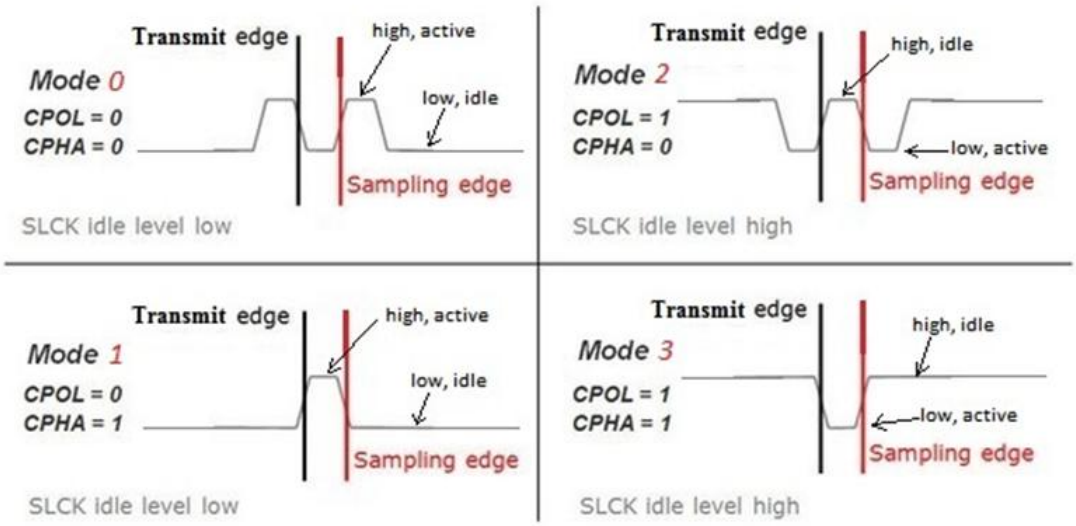

Figure 6.2: Difference between the 4 SPI modes [3]

Once the mode is defined, the MISO and MOSI lines will be sampled at the appropriate time. In concrete terms, it is necessary to be attentive to all clocks changes. For example, in mode 3, to sample correctly, you have to wait for a transition from Low to High to sample. Then, each sampled bit is saved in a variable to form an 8-bit word. Thus, two 8-bit words must be formed, one for MISO and the other for MOSI. Once 8 bits are sampled, both bytes are sent to another program for processing.

6.3 Note on Inter-process communication (IPC)

One of the tasks was to connect Sigrok-cli with the GUI. This type of operation is called an Inter-process communication (IPC). Indeed, Sigrok-cli is responsible to acquire the data while the GUI needs those data for processing.

One of the best known IPCs is to use a program to write data to a file (a simple.txt file for example) and another program reads this file to extract the written data. The file was just an example, it exists many more. In the IPCs terminology, the program that provides the data is called the server and the program that reads the data is called the client.

Usually, in the application similar to what is required in this work, the way two programs are connected is to use a dedicated socket. Indeed, this is typically what is used by the Saleae logic analyzer (the extremely expensive product discussed above, dedicated to professionals). Indeed, it is one of the solutions suggested in their official website to export the data read in real time: *"use the socket API to automate the capture and export processes"*[16] .

The way the communication via socket works is as follows. The server socket is characterized by an IP address and a port number. The client socket is also characterized by an IP address and a port number. The client reads what is sent to the server by connecting to the server-specific port. This way, everything that is sent to the server can be communicated to the client. Of course, all this can be done on the same machine by specifying the localhost as the port number.

Unfortunately, Sigrok-cli does not have an option to send data to a socket and allow another program to use the data sent to the same socket.

It allows many things like recording a data read session by exporting it in a wide variety of formats. This will not be very useful because the purpose of the application is to be able to display all the information in real time.

However, Sigrok-cli has an option to send the data to what is called a pipe. A pipe is a one-way communication.

Figure 6.3: IPC using a pipe

This type of communication can only be used on the same machine. Data written from one side of a pipe can be read from the other side of the pipe as shown in Figure 6.3. In practice, the output of a program is buffered in a virtual file that is linked to a second program. Thus, Sigrok-cli allows to output the data acquired on a pipe which can then be recovered by the GUI.

On Windows, it is very common to use sockets as IPCs. However, the use of pipes as IPCs is very uncommon because they are very complicated to set up. So, to use the application on Windows, you have to find a way to bypass the pipe use.

The first attempt was to investigate how to modify Sigrok-cli to add a socket option. More precisely, Sigrok-cli is based on the two libraries discussed above. The option to send the data to a pipe is available in the library **libsigrokdecode**. Two choices are then possible.

The first option is to modify the library **libsigrokdecode** (written in C). The code was created in 2011 and continues to undergo changes. So, trying to understand the logic of the code and modify it properly would have been ambitious and it would have been difficult to predict whether it would work. Indeed, it must be taken into consideration that my experience in C is limited and that it was my first experience with IPCs, I never configured a socket before for example. This option is not possible.

As discussed before, the SPI decoder (written in Python 3) is based on this library **libsigrokdecode**. So, instead of modifying the library, one can modify the SPI decoder. Indeed, modifying the SPI decoder so that it does not send the decoded bytes to a pipe but to a socket. Thus, it would have been necessary to create the server part of the socket and send all decoded bytes to this socket. On the GUI side, it would behave like a client that reads the socket. So, to summarize, the modifications would have involved the creation of a Python socket which would be the server and the creation of a Java socket which would be the client. This solution may have been possible but considering my experience in IPCs, it involved risks in terms of implementation time and especially debugging to ensure that the communication is correctly configured.

The simplest and safest solution is not to try to modify the SPI decoder nor the **libsigrokdecode** library but to use the pipe which is the only way to export the data already available.

However, this solution is not possible in Windows but is more suitable for Linux where pipes are widely used than for Windows. Even the few examples of project real-time application were implemented in Linux.

It is easy to understand why. Indeed, the use of pipes is much more natural on Linux where its use is very widespread often without realizing it. For example, when, in the terminal, the following command is used:

process1 | process2 | process3

Behind this command are hidden pipes that are represented by the vertical bar (|). Indeed, the output of the process1 is sent in input of the process2 which itself sends its output as input of the process3. In the end, what will be displayed will then be the output of process 3. This is illustrated in Figure 6.4

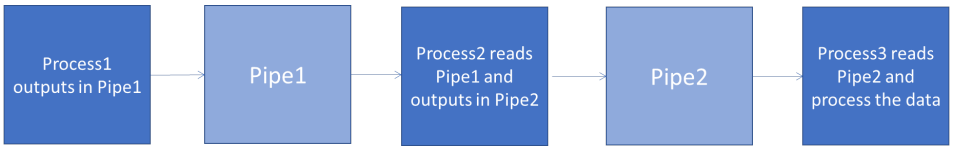

Figure 6.4: Pipe utilisation hidden behind process1 | process2 | process3 command

Thus, it was decided that GUI should be used on Linux.
In Figure 6.5, the current situation is summarized.

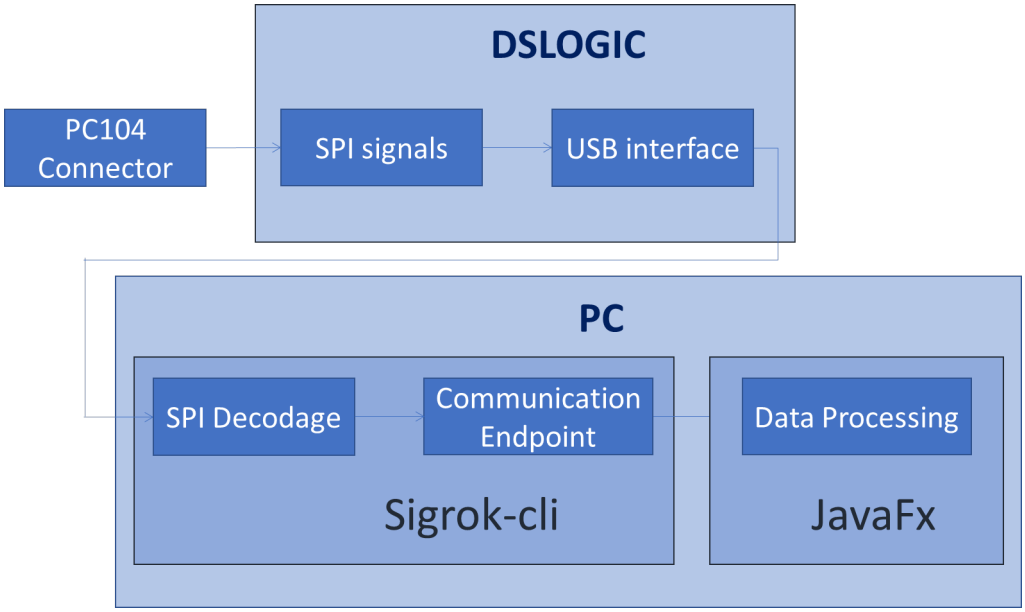

Figure 6.5: Summary of the componenents currently constituing the EGSE.

1. SPI signals: provided by the PC104 and the Arduino (not shown for clarity purpose)
2. USB interface: via the FX2 chip
3. SPI decoder: done by sigrok-cli (thanks to libsigrokdecode library)
4. Endpoint communication: pipe created by sigrok-cli
5. Data processing: detailed below

For now, the Dslogic takes care of reading the signals provided by the PC104 (arduino not shown for clarity purpose) and provides an USB interface. On the PC side, sigrok-cli provides an easy interface that allow to decode the SPI communications. Then, the output is made accessible through a pipe so that any program that can link to it can have access to it.

In conclusion, the combination of sigrok-cli and DSlogic allows data acquisition and data transfer to the PC. In the PC, the data is sent to a part of the memory called a pipe. These data are then available for the processing part of the data. As shown in Figure 6.5, it will be a GUI based on JavaFx.

6.4 Problem due to Sigrok-cli

In this section, the aim will be to discuss all the tests performed and the problems encountered when trying to use the DSlogic.

6.4.1 IPC

This problem has just been discussed but after many tests to try to link sigrok-cli and the GUI on Windows, it was decided to switch to Linux. I had never used Linux before so it took some time to get used to it.

6.4.2 Limitation

To test the DSlogic, an arduino Uno was used to check the device's capabilities.

The arduino allows simulating the DPC's behavior and will act as a master of SPI communication. The SPI lines of the arduino are then directly connected the DSLogic. Thanks to this setup, it is easy to test the limits, if any, of the Dslogic since the Arduino Uno has specific pins to communicate in SPI. In fact, there are many ready-to-use codes in the Arduino official site that allow you to use an Arduino as the master of an SPI communication.

The first thing to test was the Dslogic's ability to send data in real time. So, the first test of the arduino was to send **one byte every millisecond** to verify that the DSlogic is able to read continuously. The result of this test was conclusive. The reading was done without interruption after several tens of minutes of capture. Dslogic could read the data and send it to the PC and could be displayed via the JAVA application.

Why **one byte every millisecond**? Because I first used a generic script illustrating the use of SPI that is present in the Arduino site. However, in practice, more than one bytes every millisecond will be sent to or from the DPC continuously. Therefore, to ensure that the DSLogic worked as expected, it was necessary to check that it would worked in the worst case for the DSLogic. The most unfavourable case is the case where the DPC continuously sends and/or receives byte without a break.

To test this point, with the arduino, data was sent without any break to the DSlogic. The purpose of the test was to see if the reading could be done **without interruption**. Unfortunately, when testing with the Arduino, the DSLogic was systematically interrupted by displaying **"Device only sent X samples"**. X would vary around 10^6 samples

A problem had just been identified. The question is whether this situation can occur in the operation of OUFTI-2. So, the next logical step is to identify if this situation can occur in OUFTI-2. It turns out this situation can occur with the FRAM.

Indeed, the critical point is the communication with the FRAM. So, I tested the DSLogic directly to the development board where the FRAM is located instead of simulating the behaviour of the DPC with the arduino. I was helped by Guillaume who made sure that a writing in the FRAM was done. On my side I made sure that the DSLogic was well installed and was ready to start an acquisition when Guillaume was ready. By testing directly on the development board, sigrok-cli systematically stops and ends with **"Device only sent X samples"**. By looking for the cause of this problem, this problem had already been pointed out before me. This is apparently a known bug when the signal frequencies are too high.[15]

When Guillaume was writing in the FRAM, he was just collecting measurement done by the ADC, so it wasn't even the worse scenario for the FRAM. To understand to what extent the DSLogic was not powerful enough, for the reading to be done correctly, it was necessary to slow down the SPI communication by adding a delay of 300ms more or less every 100 bytes. The only case where sigrok-cli was able to track the data sent by FRAM was when a 300ms delay was added between each frame of 100 bytes. Which is not tolerable. Indeed, changing the SPI communication to allow data acquisition is obviously not an option. The purpose of the project is precisely to test the functionality of OUFTI-2 as if it were in flight. It is not possible to modify the DPC code to add a 300ms delay just to allow the DSLogic to work properly. So, this is a major problem that can compromise the data acquisition part.

Once this major problem had been identified, the reason for the problem had to be found in order to find a solution. To understand where the problem comes from, it is necessary to remember the two

programs developed by Sigrok: PulseView and sigrok-cli. Pulseview is the graphical interface. During the acquisitions, no reading interruption was ever observed. Sigrok-cli on its side is the equivalent command line of Pulseview, ie anything that can be done with PulseView can be replicated with Sigrok-cli. However, with sigrok-cli, as discussed, systematic interruptions are observed.

Since PulseView is not interrupted, the problem cannot be hardware otherwise the interruption problem would also be present with PulseView. So, I contacted the sigrok developers to look for ways to improve. First, they confirmed the problem. The problem was in fact confirmed by two main developers of Sigrok: abraxa and Uwe. As observed in Figure 6.6, **abraxa**, one of the developers confirms to me that the problem comes from a difference in the implementation of sigrok-cli compared to PulseView.

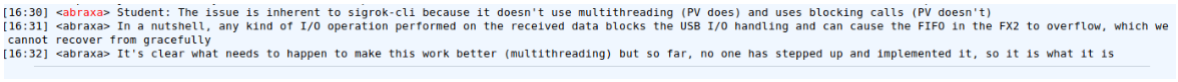

Figure 6.6: Confirmation of the issue by a main developper of Sigrok

Indeed, to quote abraxa: "*The issue is inherent to sigrok-cli because it doesn't use multithreading (PulseView does) and uses blocking calls (PV doesn't)*" Concretely to fix this problem, multithreading must be implemented for sigrok-cli to work as well as PulseView. So, the solution to solve this problem would be to reimplement sigrok-cli.

Sigrok-cli is written in C, has years of development behind it and if the problem was trivial it would already have been fixed by one of the developers. So, it is clear that reimplementing sigrok-cli was not possible for me. By discussing with Mr Dedijcker who immersed himself in the code and in agreement with Mr Broun, another solution had to be found.

My first intuition was to abandon the idea of sigrok but with Mr Broun's advice, it was decided to make compromises on the reading of the FRAM. Indeed, one of the critical cases with FRAM was the repatriation of all measurements. Indeed, by telemetry it can be decided to send back all the measurements recorded to the ground station. However, 3/4 of the memory is occupied by measurements. It corresponds to a slightly less than 200k bytes. This amount of data would inevitably lead to an interruption of reading. As a reminder, a delay of 300ms every 100 bytes was the limit to avoid any interruption...

Therefore, it was decided not to have access to the FRAM reading by the DPC to avoid this situation. However, this is compensated by the fact that in writing, the value of the measurements is accessible.

In summary, FRAM data cannot be read with the DSlogic. However, when writing to the FRAM, the data that is being transmitted may be intercepted. The main idea being that the read data should be the same as the data we collect in writing.

To summarize, the solutions were either to implement sigrok-cli again, to change the solution or to limit the reading of the FRAM. Presented in this way, it is obvious that the last solution is the best but it was not so obvious to find. As Mr Broun advised me, *just because we don't have everything doesn't mean we should abandon the idea directly.* It is a piece of advice that will definitely help me in the future.

6.4.3 Sigrok-cli can't be used.

Despite the compromises made for FRAM, I will explain the methodology used to conclude that sigrok-cli cannot work in this project for data acquisition.

To understand this, it should be remembered that the tests with the Arduino are used to simulate the behaviour of the DPC.

The DSlogic only read 4 channels provided by the Arduino: MISO, MOSI, SCLK and CS. Remember that in the end the DSlogic will actually have to read 12 channels. A number of tests have been carried out, the most important of which are

1. Reading simulation of the 3 ADCs
2. Simulation of writing measurements in the FRAM.

The purpose of these tests is to see if the bug no longer appears. Each test individually, the reading of the ADCs and writing in FRAM, was fine. However, when the reading of the ADCs is directly followed by the FRAM writing, the bug discussed previously reappeared: **"Device only sent X samples"**. This is a first problem.

The second problem is that, for the moment, in all tests, the Dslogic only reads 4 channels instead of the 12 final ones. The last test was to go back to the situation where the Dslogic did not crash, i.e. by individually testing the reading of the ADCs and writing in FRAM. However, instead of only reading 4 channels, it reads 12. In this case, the reading is systematically interrupted. It corresponds to the same bug previously. Compared to the individual test, the Arduino's behaviour is the same. The only difference is that there are additional channels to read. So,adding channels causes the DSlogic to read more channels, which overloads it and as a consequence stops the reading.

To conclude the result of the tests are as follows:

1. With 4 channels, the Dslogic cannot read an SPI communication that includes a reading of the ADCs followed by writing in FRAM.
2. With more channels, the Dslogic cannot read an SPI communication that includes only a reading of the ADCs or a FRAM writing.

In addition to these conclusions, additional details are important. During the tests, the behaviour of the arduino was a favourable case because it is not even complete. Indeed, modules that, for the moment, are not yet present (IMU, RAD,...) have not been simulated so in addition to the reading of the ADCs followed by a FRAM writing, additional bytes will be sent to the OBC.

And it should not be forgotten that the DSlogic will have to read 3 additional channels that will correspond to the MOSI, SCLK and chip select for the digitized analog values.

Taking into account the conclusion of the tests and the additional details, the data acquisition with the Dslogic is clearly compromised.

Chapter 7

Data Processing

Despite the fact that data acquisition is compromised, it was possible to work on the software part of the EGSE, which consists of processing the data and displaying them. To do this, it is necessary to impose a format that should have been given them acquired. To do this many connections with what should have been sent by the DSLogic will be made.

7.1 Format of the data received

In order to decode properly, one must be aware of the format of the data. We saw how SPI communications, i.e. the data that pass through the MISO and MOSI lines, are decoded. The result of the decoding are the bytes exchanged, in hexadecimal, as recalled in Figure 7.1.

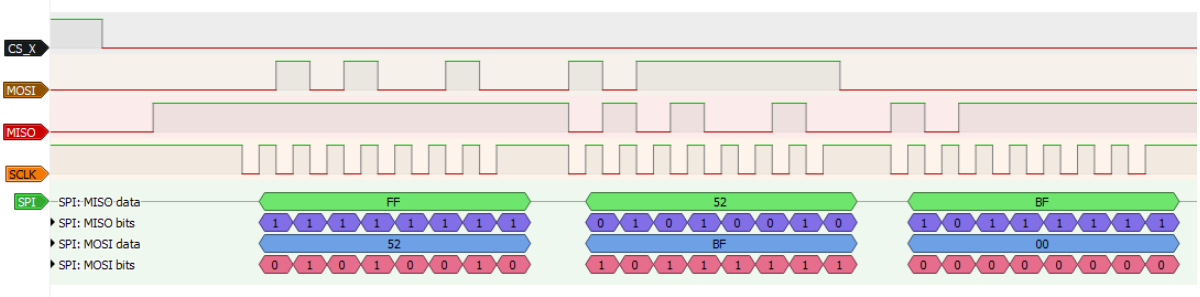

Figure 7.1: The bottom lines show the data sent decoded in terms of bits and bytes. In this case, the module (MISO) sends successively 0xFF, 0x52, 0xBF and the master sends 0x52, 0xBF, 0x00

This feature is also available on sigrok-cli, but modifications had to be made on the SPI decoder. Indeed, the result sent by sigrok-cli is the following:

```
1 byte1  #MISO
2 byte2  #MOSI
3 byte3  #MISO
4 byte4  #MOSI
5 byte5  #MISO
6 byte6  #MOSI
7 byte7  #MISO
8 byte8  #MOSI
9 byte9  #MISO
```

The first change to do concerns the fact that the SPI decoder considers a communication with a single slave module possible. Since we have more than 1 CS, it was necessary to recognize several slave modules. In addition, it is necessary to find **a way to identify each CS to know with which module bytes are exchanged**.

The second modification was to be explicit about the data sent on MISO and MOSI. The decoder implicitly specifies that the odd byte (1st, 3rd,...) correspond to MISO and the even one (2nd, 4th,...) correspond to mosi. It is necessary **to clarify to which line, miso or mosi, the data belong**. For the simple reason that if a problem occurs and some packets of bytes are lost, it is possible to continue decoding on a good basis by knowing exactly to which line the bytes belong.

For the sake of the illustration, we can imagine a case where on a communication of 15 bytes, an interruption occurs at the 7th byte and resume 11st byte. The 7th byte is considered miso, since 7 is an odd number. From the decoder point of view, the next byte it will receive will correspond to mosi because it will be the 8th byte received. So, we end uo in a situation where the 11th byte which belongs to miso is confused as belonging to mosi.

The third is that the dslogic reads 2 SPI communications. Indeed, the first one, SPI1, corresponds to the SPI signals of PC104. The second, SPI2, corresponds to the analog values that pass first through the Arduino and then DSlogic to SPI. Therefore, it is necessary to extend the decoder so that it can track and decode 2 communication. This will be quite easy to set up since the same methodology is used to decode SPI.

The changes made lead to

```
1i: byte1  #MISO
1o: byte2  #MOSI
1i: byte3
1o: byte4
1i: byte5
1o: byte6
1i: byte7
1o: byte8
1i: byte9
```

First, the data exchanged is seen from the master point of view. The *i* stands for what comes **i**nto the master, while the *o* stands for what come **o**ut. So, naturally the *i* will stand for the MISO line (Master In), and the *o* will stand for the MOSI (Master Out).
It solves the problems discussed earlier:

1. **It clarifies to which line, miso or mosi, the data belong** thanks to the use of *i* or *o* right before each byte exchanged.
2. **It is easy to identify each CS** thanks to the number at the beginning of each line. Indeed, each module will have a corresponding number to identify it in a unique way.

SPI Communication	Module	ID number	Description
SPI-1 Master: DPC	ADC 1	1	SPI communication between the DPC and different modules
	ADC 2	2	
	ADC 3	3	
	FRAM	4	
	COMM	5	
	BCN	6	
	IMU	7	
	RAD	8	
	BATT	9	
SPI-2 Master: Arduino	Dslogic	0	SPI communication between the Arduino and the Dslogic to send the digitized analog Data

Table 7.1: ID to identify it in a unique way each module

The choice to inform the module first before the information on the miso or mosi line has a meaning for decoding. Indeed, to decode, it is first necessary to know with which module is concerned and then

decode according to it.

An important point to understand is that at each clock time, data can be transmitted on the MISO or MOSI line. So, at each clock time, two byte are decoded: one from MISO and one from MOSI.

However, once the data was decoded, the notion of clock disappeared. Indeed, just from the data, knowing what happened on the same clock time is impossible. An example of the ambiguity of the situation is given:

```
1o: byte3
1i: byte4
1o: byte5
1i: byte6
1o: byte7
```

From these data, the question that arises is which data is sent first, MISO or MOSI ? If it is MISO, then, on the same clock time, the data are sent in the order: miso then mosi. So, in the example, the couple of decoded byte sent on the same clock time, are: byte4 - byte5 ; byte6-byte7

```
1o: byte3
1i: byte4    | MISO
1o: byte5    | MOSI
1i: byte6         | MISO
1o: byte7         | MOSI
```

But, if it is MOSI first, the couple of decoded byte sent on the same clock time, are: byte3 - byte4 ; byte5-byte6

```
1o: byte3    | MOSI
1i: byte4    | MISO
1o: byte5         | MOSI
1i: byte6         | MISO
1o: byte7
```

That's why it is necessary to decide on a convention to define what happens on the same clock time. The choice was made to first send MISO first and then MOSI. The reason is mainly to simplify decoding.

To understand this, it is useful to consider a simple example: A request is sent by the master on the MOSI line. At the next clock time, a response will be sent by the slave modules. In other words, a request (MOSI) has been sent at time T-1, and the byte of the slave module (MISO), at time T, will correspond to the response.

If considering the MOSI first, the situation is illustrated below.

```
1o: byte3    | MOSI: REQUEST
1i: byte4    | MISO (must be ignored)
1o: byte5         | MOSI: (potential request)
1i: byte6         | MISO: RESPONSE
1o: byte7
```

At time T, the request is made by on the MOSI line. The next byte corresponds to a MISO but is not the response because it is not at time T+1. So, it must be ignored.

At time T+1, it is possible that another request by the master will be made so it will not be possible to simply ignore this MOSI byte. And finally, the response to the request is at the MISO byte.

So,in the end, between the request and the response, 2 byte interpose themselves and turns out to be more difficult to decode than the situation where we consider the MISO first.

Indeed, if a request (MOSI) has been sent by the master at time T-1, the next byte will correspond to the response of the slave module (MISO), at time T, will be the next byte as illustrated below.

```
1o: byte3
1i: byte4    | MISO
1o: byte5    | MOSI: REQUEST
1i: byte6          | MISO: RESPONSE
1o: byte7          | MOSI: (possible new request)
```

7.2 Modules

To decode each module, it is first necessary to briefly describe the role of each module. Indeed, to choose the information to be displayed, it is first necessary to know what information passes through the modules.

Obviously, the goal will not be to provide a summary of the datasheet of each module. The primary goal is to identify the data that can flow between the DPC and the modules. Once all types of communications have been identified, it will be necessary to consider how to decode them.

Among the 9 modules whit which the OBC interacts, only the FRAM, COMM (=D-STAR), 3 ADCs were implemented for the moment. The SPI communication of the IMU, RAD, BCN and BATT subsystem were not implemented. One remark for the BATT is that a library was created to manage SPI communication but it was not fully finalised.

7.3 ADCs

As discussed before, 3 ADCq are present on OUFTI-2. Their purpose is to read analog signals.

7.3.1 Quick overview of the MAX1231 operation

The reference of the ADCs is the **MAX1231** from **Maxim Integrated**. The following table contains the main information about it.

Resolution	12 Bits
Channels	16
Modes possible	Single ended or differential inputs

Table 7.2: Characterists of the ADC

Resolution

About the output of the ADC, the value digitised are encoded on 12 bits. The number of output values is $2^{12} = 4096$ possible values from 0, the minimum value, to 4095, the maximum value.

The data is thus encoded on 12 bits. After the measurements are taken, they are sent to the DPC.

Mode Possible

At no time, differential measurement is required. Indeed, differential measurement means measuring the difference between two inputs which is not used in our case.

So, these are Single-ended measurements since all measurements are referenced to the GND.

In single-ended mode, the only possible mode is unipolar mode. This clarification is not trivial as with differential measurements, the measurement could be unipolar or differential.

Register

In the following Table 7.3, the main registers useful to properly use the ADC.

Register Name	Description
RESET	To reset all register to their default state
SETUP	To setup the clock mode, the reference, the mode unipolar or bipolar
CONVERSION	To specify which channel to read

Table 7.3: To initiate the data, 3 registers must be set

The *Setup* register is the most interesting to to send the measurements. Indeed, as you can see in Figure 7.2 , it is used to define the clock, the reference mode and the choice in unipolar or bipolar mode.

Table 3. Setup Register

BIT NAME	BIT	FUNCTION
—	7 (MSB)	Set to zero to select setup register.
—	6	Set to 1 to select setup register.
CKSEL1	5	Clock mode and $\overline{\text{CNVST}}$ configuration. Resets to 1 at power-up.
CKSEL0	4	Clock mode and $\overline{\text{CNVST}}$ configuration.
REFSEL1	3	Reference mode configuration.
REFSEL0	2	Reference mode configuration.
DIFFSEL1	1	Unipolar/bipolar mode register configuration for differential mode.
DIFFSEL0	0 (LSB)	Unipolar/bipolar mode register configuration for differential mode.

Figure 7.2: Setup Register. [17]

As discussed earlier in single-ended measurement, the only possible mode is unipolar mode.

The selected clock is the external one. Indeed, it will be the one provided by the DPC. The clock frequency, for this module, is 3.8MHz. This frequency is acceptable because the maximum frequency allowed for ADC is 4.8 Mhz.

Therefore, in single-ended mode, the measurement range is between 0 and V_{ref}.

V_{ref} is the reference voltage. It corresponds to the maximum voltage that can be read. The reference voltage must always be greater than the voltages to be measured on the channels. Otherwise, any values greater than V_{ref} will be translated as the maximum value that can be output by the ADC, i. e. V_{ref}.

The **MAX1231** allows you to define V_{ref} in 2 ways: either to provide an external reference voltage or to use the internal V_{ref}. In our case, the V_{ref} chosen is the internal reference which is 2.5V. Thus, the maximum voltage for the channels which must not exceed 2.5V.

The refencence voltage mode is internal (2.5V) and is set to be all the time so as to avoid wasting time with a wake-up delay.

The LSB, i. e. the smallest variation, for example to go from 0 to 1, is given by $V_{ref}/4096 = 2.5/4096 = 0.6mV$. The transfer function to switch from the value digitised by the ADC (on 12 bits), V_{read}, to the voltage value, V, is: $V = V_{read}$ x $\frac{2.5}{4096}$

However, some of the voltages to be read can be higher than 2.5V. For example, 3.3V voltages from the EPS are part of the measurements that are digitised. Since 3.3V is higher than the reference voltage, it must have been necessary to reduce these tensions below 2.5 V to be correctly read. This is often done by using a resistive voltage divider that has a constant transfer function H. So,the real voltage value, before the voltage divider, is then obtained by computing $V = V_{read}$ x $\frac{2.5}{4096}$ x $\frac{1}{H}$.

Now, the last important register that needs explanation is the *Conversion* register. Its goal is to specify which channel to read. Indeed, to recover the value of a channel, the DPC sends to the ADC (via the MOSI line) what is called a byte conversion. Thanks to this byte, the channel to be read is specified. The way this register is defined is shown in Figure 7.3.

Table 2. Conversion Register

BIT NAME	BIT	FUNCTION
—	7 (MSB)	Set to 1 to select conversion register.
CHSEL3	6	Analog input channel select.
CHSEL2	5	Analog input channel select.
CHSEL1	4	Analog input channel select.
CHSEL0	3	Analog input channel select.
SCAN1	2	Scan mode select.
SCAN0	1	Scan mode select.
TEMP	0 (LSB)	Set to 1 to take a single temperature measurement. The first conversion result of a scan contains temperature information.

Figure 7.3: Conversion Register. [17]

Thus, bits 3 to 6 are used to specify which of the 16 ($=2^4$) channels to read. In the following Figure7.4, examples of actual conversion bytes sent to ADC are shown. The channels requested can be deduced, 9 to 12, in this case.

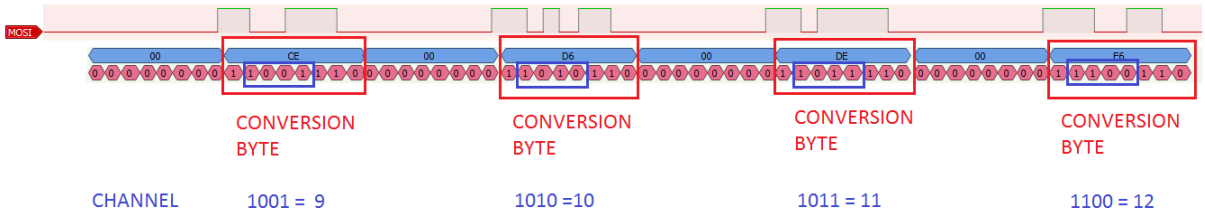

Figure 7.4: Example of conversion bytes sent by the DPC to the ADC.[17]

7.3.2 SPI communication

Now, that the conversion register is understood, one can detail how the measurement value are sent to the DPC. To better explain it, it is easier to use an example of capture of the SPI communication that is shown in Figure 7.5
Once a conversion byte is sent to the MOSI line, the next two bytes of the MISO line will correspond to the measurement of the selected channel as shown in Figure 7.5.

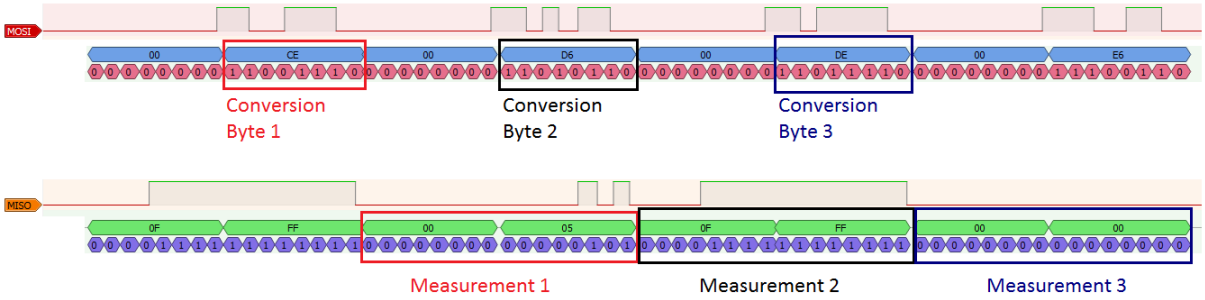

Figure 7.5: Example of communication between ADC and DPC. Right after a conversion byte is sent, the ADC sends 2 bytes corresponding to the measurement requested via the conversion byte.

At each conversion byte corresponds two measurement bytes. The reason for the two bytes is because the first one will correspond to the MSB (most significant Byte), the second will correspond to the LSB (least signifying bytes). Indeed, since the resolution of the ADC is 12 bits, the format of the measurements is illustrated in Figure 7.6

Measure MSB								Measure LSB							
X	X	X	X	BIT 11	BIT 10	BIT 9	BIT 8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0

Figure 7.6: Format of the 12bits-measurements split into 2 bytes

A remark can be underlined about Figure 7.5. To speed up the measurement, the ADC takes advantage of the fact that the SPI is a full duplex protocol communication. When receiving the bytes LSB, you can already configure the byte conversion so that it can read the next channel.

-

7.3.3 Decoding

In continuity with what has been explained in the section 7.1: *Format of the data received*, the decoded data will have the following structure.

```
...
1i: byte1  #MISO
1o: byte2  #MOSI
1i: byte3
1o: byte4
...
```

With all the explanation given in the previous sections, the information that will need to be decoded are:

1. MOSI: Deduce the channel from a conversion byte, recognise the setup byte and the reset byte
2. MISO: Measurement bytes. After a conversion byte, recover the 2 bytes constituting a measurement.

With these information, to each channel will correspond a measurement.

7.4 FRAM

One of the modules with which the DPC communicates is the FRAM. It is the **FM25V20A** from **CYPRESS**.

The main role of FRAM is to record a number of data such as: measurements made by ADCs, satellite status, various events....

However, the goal is not to cover everything that is present in the FRAM but only to focus on what is communicated to the DPC via SPI. For the FRAM, bits are transmitted with Most Significant bit (MSB) first. The FM25V20A can operate in SPI Mode 0 and 3. In this project, the mode 3 is used like the ADCs before.

7.4.1 SPI communication

To communicate with the FRAM, it is necessary to use opcodes. All these opcodes, extracted from the datasheet, are defined in Figure 7.7.

Table 1. Opcode Commands

Name	Description	Opcode
WREN	Set write enable latch	0000 0110b
WRDI	Reset write enable latch	0000 0100b
RDSR	Read Status Register	0000 0101b
WRSR	Write Status Register	0000 0001b
READ	Read memory data	0000 0011b
FSTRD	Fast read memory data	0000 1011b
WRITE	Write memory data	0000 0010b
SLEEP	Enter sleep mode	1011 1001b
RDID	Read device ID	1001 1111b

Figure 7.7: Opcodes. [18]

Among all these opcodes, the only ones that deserve attention are the write and read opcodes. For the others, it is necessary to know that they exist and to recognize them when decoding, nothing more.

The write and read opcodes are important because, as their names suggest, it allows you to initiate writing and reading in the FRAM. Therefore, since the aim will be to decode what is written and read in FRAM, it is necessary to understand how writing and reading in FRAM works.

The operation of writing and reading is quite similar and is illustrated in Figure 7.8. After the opcode write/read, it is necessary to specify which place in the memory should be written/read by indicating the address on 18bits.

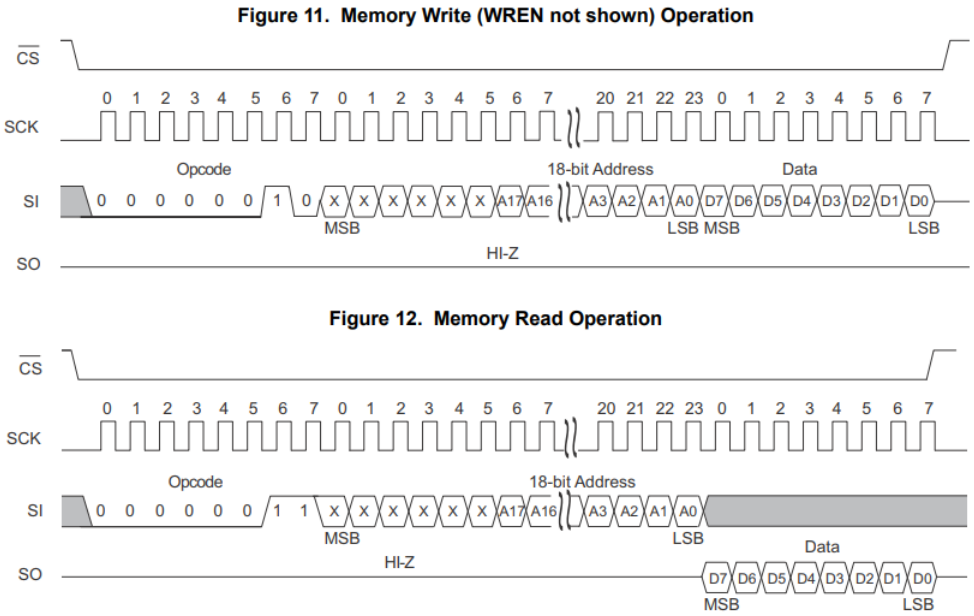

Figure 7.8: Write / Read operation. [18]

For writing, it is the DPC that sends data to be placed in FRAM. The line used will, therefore, only be MOSI and the data to be written will be just after the opcode WRITE and the address where to write.

For reading, the DPC receives the data previously placed in the FRAM. The lines used will be MISO for the data to be read and MOSI line for the request with the READ opcode and the address where to read.

A remark about the addresses length. The reason for the 18 bits is due to the organisation of the FRAM. Indeed, to give an image of the Fram's organisation, it is necessary to imagine a table of 256k lines. Each of these lines is 8 bits long. Therefore, when writing to memory, it is mandatory to specify the line, called the address, on which to write. To specify 256k addresses uniquely it is necessary: $\log2(256000) = 17.97 = 18$ bits. Thus to specify an address, it is necessary to use 18 bits.

The management of the FRAM and its organisation on OUFTI-2 has already been thought by several students. During that year, I had the opportunity to collaborate with a student, Guillaume, who was in charge of coding telemetry. Thanks to his experience, he was able to explain to me how the data was organised within the FRAM by means of a diagram, shown in Figure 7.9, he made.

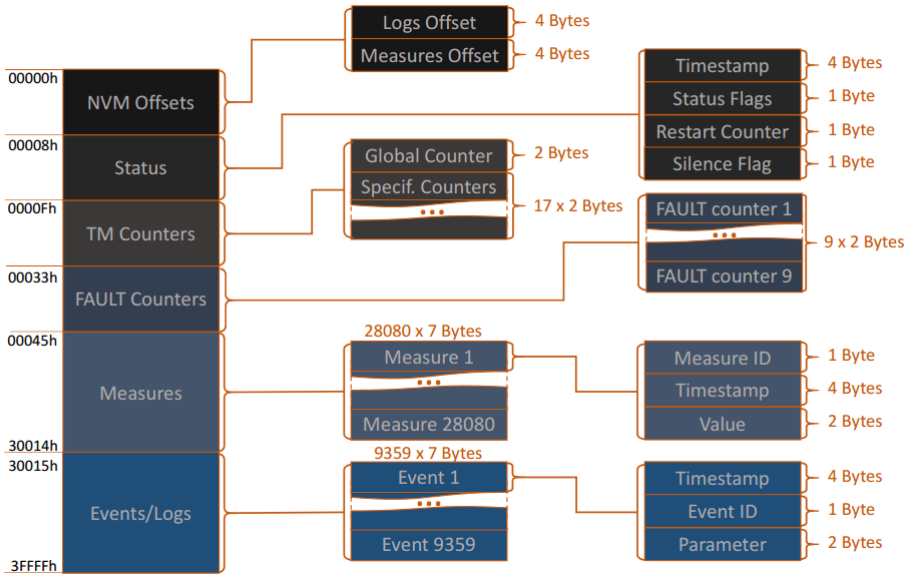

Figure 7.9: FRAM organisation. Structure and type of data stored in the FRAM. **Made by Guillaume ??**

To have all the precise details, it will be necessary to refer to his thesis. For now, it is only necessary to list what constitutes the FRAM to know what needs to be decoded. The data stored in the memory are divided into 6 types:

1. Measures: All measurements made on OUFTI-2. The 3 ADCs module are part of it.
2. Events: Records all types of notable events: subsystem turned off or not, antenna status changes, number of restarts....
3. Status: Timestamp which is the time elapsed since the start, the state of the antennas, the number of restarts of the OBC.
4. Offset measurements and event: To avoid overwriting the data and to know from which address measurements and events can be written
5. TM counter: Counts the number of telemetry requested.
6. Fault counter: Counts the number of time the subsystems encounters faults.

Once the 6 types of data in the FRAM are presented it is necessary to know their structure to correctly decode them. The structures for each of them are explicit in Figure 7.9 and are recalled in the Table 7.4.

Type of Data	Total Bytes	0	1	2	3	4	5	6	7
Offset	8	Event Offset				Measure Offset			
Status	7	Timestamp				Status	Restart	Silence	
Measures	7	Measure ID	Timestamp				Value		
Events	7	Timestamp				Event ID	Parameter		
TM counter	2	Counter							
Fault counter	2	Counter							

Table 7.4: Summary of the structure of type of data

For more details on what is in the FRAM, it is more interesting to see Guillaume's thesis directly. [18]

For now, all the necessary elements to decode have been explained. The next section explains how data transiting between the FRAM and the DPC can be decoded.

7.4.2 Decoding

First of all, to decode it is necessary to remember that the DPC communicates with FRAM using opcodes as discussed before. One of the obvious tasks will be to identify these opcodes. Only one point of attention should be given to opcodes related to writing and reading. Indeed, as seen above, an address directly follows these opcodes followed by the data to be read or written.

As for the data, their structure has also been explained. However, just based on the structure of the data, it is quite difficult to decode. Indeed, even if we have the structures of each type of data in the FRAM, it is complicated to distinguish them from each other.

Indeed, two natural strategies would be:

1. Each type of data has a defined number of byte as can be seen in the table 7.4. For example, if 8 bytes are transmitted, it is necessarily an offset that has been read or written. However, and this is obvious from the table, using length as a basis does not allow the types of data to be distinguished. Indeed, 3 types of data are of size 7 bytes and the counter are on 2 bytes

2. Keep the idea of lengths and for data types of the same length, differentiate them using the fact that each of them has a different structure. For example, both events and measures are 7 bytes in size but have a different structure.

However these strategies are not effective. Indeed, solution 1) is not possible and solution 2) is not ideal.

Indeed, for solution 2), keeping the example of events and measures, the challenge is to determine, from 7 bytes, if it is an event or a measure. Two problems arise:

1. It is necessary to wait for the 7 bytes which induces a delay because the decoding is delayed compared to the communication.

2. There are cases where it is impossible to deduce if it is an event or a measure. Indeed, it is possible to find a certain number of examples where 7 bytes can be both measurement data and event data

In the end, the best solution is hidden in the organisation of memory. Indeed, the memory is divided into 6 distinct parts, each of which contains a type of data. This is clearly visible in Figure 7.9. The following table lists the specific locations where the different types of data in the memory are separated.

Type of Data	Adress start in memory
Offset	0x00000
Status	0x00008
TM Counter	0x0000F
Fault Counter	0x000037
Measures	0x00049
Events	0x30019 to 0x3FFFF

Table 7.5: Memory organisation: Adresses range for each type of data

To read or write, it is mandatory to specify the address where to read or write. Thus, by knowing the organization of the memory in advance, thanks to the address, it is easy to deduce what type of data will be read or written.

To decode, all you have to do is decode the address. Thanks to the address, it is easy to deduce the type of data that will be read or written. Depending on the type of data, it will be necessary to decode it according to its own structure.

To summarise, the information that will need to be decoded are:

1. MOSI: recognise the opcode. If read/write opcode, deduce its adress. If write, decode the data being written using the methodology explained before
2. MISO: If read opcode, decode the data being read using the methodology explained before

7.5 DSTAR

The DPC communicates with the D-STAR microcontroller. This is a part of the these made by Francois Piron, a student who was working on a thesis for OUFTI-2 at the same time. One of its tasks was to establish the way data between the DPC and the D-STAR will be transmitted via SPI.

Once his work was finished, I had the information on SPI communication at my disposal. In order not to repeat what he has already written in his thesis, I will not go into details but if necessary more information can be found in his thesis.

In this work, it will be a question of explaining only what is strictly necessary to correctly decode the SPI communication.

7.5.1 SPI communication

The communication is done in mode 3 and particularity of this subsystem, unlike FRAM or the ADCs which communicated with a clock at a frequency of 3.8 Mhz, the clock frequency is only at 125kHz.

For the D-STAR, communication is based on the use of commands, which reminds us of the opcodes used for FRAM. Thus, all commands are represented by an ASCII byte.[19] Figure 7.10, made by Francois Piron, summarizes the different commands.

Command	Byte	CRC	Parameters	Description
OFF	`'F'`	`0xA4`	*None*	Disables all D-STAR operations (default mode)
REPEATER	`'R'`	`0xBF`	*None*	Repeater mode
PARROT	`'P'`	Yes[5]		Transmits a parrot message
			Message ID:	Pre-recorded frame ID,
				or `0xFF` for re-writable frame
			Repetitions:	Number of additional transmissions[6]
			Interval:	Time in seconds between each transmission[7]
CAPTURE	`'C'`	`0x32`	*None*	Records next frame into re-writable parrot slot
SET_CONFIG	`'S'`	Yes		Changes ADF7021 configuration registers
			Nb config TX:	Number of TX configuration registers
			Config TX:	Variable length, see 3.8.2
			Config RX	Variable length
CLEAR_LOGS	`'Z'`	`0xB1`	*None*	Resets and unlocks the logs
LOGS	`'L'`	No		Flushes the logs on the SPI bus, see 3.8.3
			Offset:	Index of the first returned byte, on 16 bits
(DUMP)	`'D'`	No	*None*	Dumps the re-writable parrot frame
				Should only be used on ground, see 3.8.4

Figure 7.10: Commands available in D-STAR. **Made by Francois Piron** [19]

For communication, it is mainly the OBC that sends the orders and data to the D-STAR microcontroller. It is therefore the MOSI line that is being used. For the MISO line, it is used to check that the data has reached the D-STAR module. Indeed, it sends back to the DPC each byte received to verify that it has been correctly sent. This is shown in Figure 7.12.

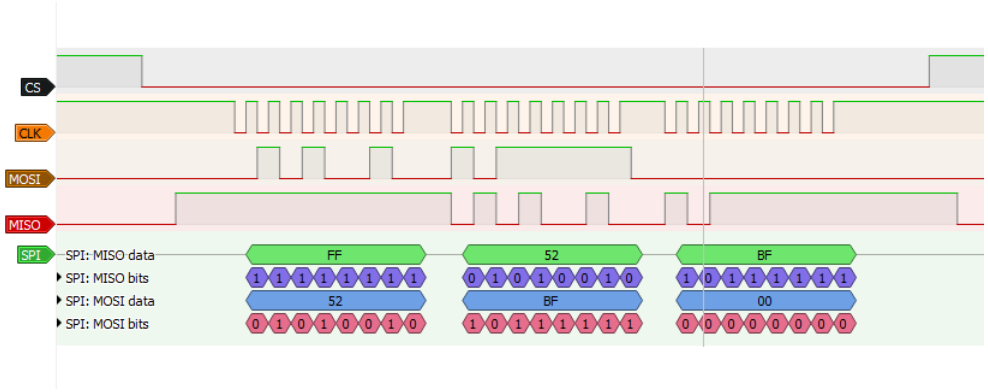

Figure 7.11: Example of DSTAR SPI communication

There is only one command that receives data from the microcontroller and it is the'LOG' command. In this case, MOSI contains the command and an offset (see Figure 7.10), and on the MISO line, after repeating the 'LOG' command and the offset, the logs are sent.

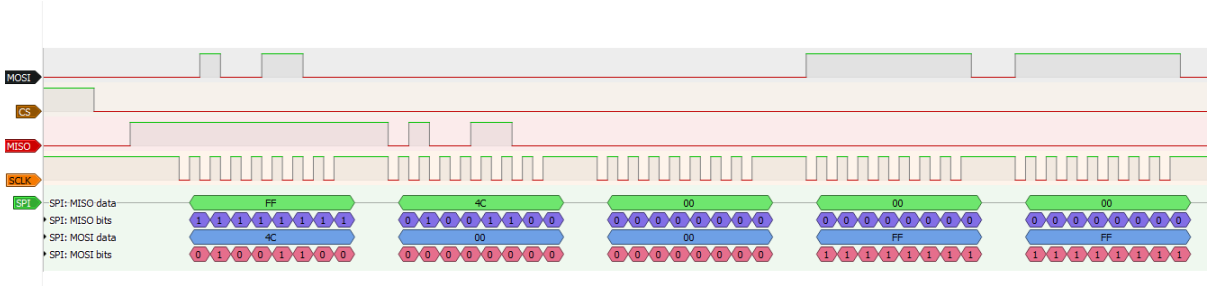

Figure 7.12: Example of DSTAR SPI communication: Log command. (In hexadecimal, the equivalent of ASCII character **L** is 4C.)

7.5.2 Decoding

Since the data sent to the D-STAR is not known, it is not possible to decode what passes through the MOSI line. Indeed, the data that pass through this line contains the frames received by the D-STAR module.

For MISO, it would have been interesting to validate and if it correctly repeats what MOSI sends. However, this part has been validated by Francois Piron so it is not necessary to implement it on our side.

However, it is interesting to decode which of the 8 modes the D-STAR is in.

For the D-STAR the only thing that can be decoded is the mode in which the D-STAR is.

So, the information that will need to be decoded are:

1. MOSI: Recognise the command and deduce the mode.
2. MISO: Nothing.

7.6 Arduino Data

As stated when presenting these signals, for each signal, it will be necessary to send 2 bytes, MSB first. The structure of the data sent by the Arduino has been discussed before and is recalled in Figure 7.13 and 7.14."

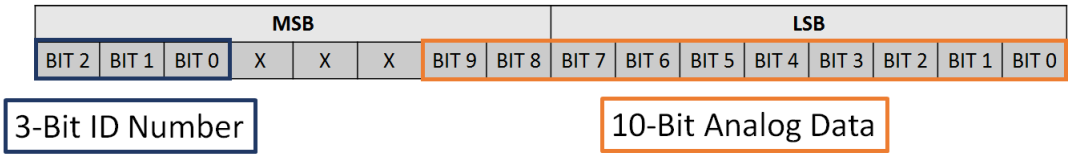

Figure 7.13: Illustration of the analog data structure: 10 bits analog data are preceded by 3 bits ID Number

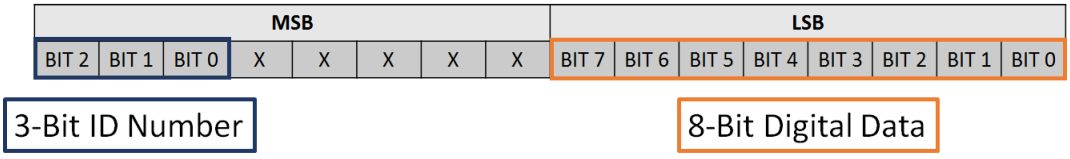

Figure 7.14: Illustration of the digital data structure: 8 bits digital data are preceded by 3 bits ID Number

7.6.1 Decoding

From the explanation above, the information that will need to be decoded are:

1. MOSI: Recognise the ID and decode the 10-bits analog data sent in 2 bytes and the digital coded in 8 bits.
2. MISO: Nothing.

7.7 Software used for data processing and display

From a software point of view, the ultimate goal of the project is to display the important information obtained via the PC104 connector. Therefore, it will be necessary to build a graphical interface.

To build graphical interfaces, several solutions exist. Indeed, most languages allow to build a graphical interface like Python, Java or even Matlab. In my case, the choice of language naturally turned to Java. First of all because it was advised by Mr Dedijcker, one of the supervisors of the OUFTI-2 project. And this choice is quite logical, many GUIs are created in Java so that a large documentation is present. Moreover, I had some basics in Java and for my first experience in the GUI field, having some basics makes it a little easier.

Once the language has been chosen, it is not over. It is necessary to choose the framework. Indeed, two main choices are possible: Swing or JavaFX. Both allow to make GUIs. JavaFX is supposed to be the replacement for Swing. But the transition from Swing to JavaFX is still not done and both are still in use. For my part, I chose to work with JavaFX.

First of all, in an application, 2 parts are present: the graphical interface and the back-end. The graphical interface corresponds to all the visual elements visible in an application. The back-end is in fact the logic that is responsible for correctly modifying the visual elements. For example, imagine that the application needs to display a led that can take two colors: red and green. The visual element is the led, its shape, size, width,... and belongs to the graphical interface. The logic of the led, ie when it changes color, under what conditions,... is supported by the back-end.

What is interesting in javaFX application is that it allows to properly dissociate the graphical interface and the back-end.

7.7.1 Graphical interface

In our case, what should appear in the graphical interface has already been discussed previously. Indeed, everything that needs to be decoded must be displayed. For example, for the ADCs, it is the value of the measurements of the 16 channels, for the D-STAR, it is the current mode, for the FRAM, it is the data sent to the FRAM. The graphical user interface must include and display these elements.

To build the graphical interface, one must create an FXML file, specific to JavaFX, which uses XML syntax. However, to simplify the creation of GUIs, existing programs and to generate FXML files automatically. This is the case with SceneBuilder, the program I used to create the graphical aspect of the JavaFX application as shown in Figure 7.15. Thanks to this program, to add an element to display, it is enough to select the desired element and drag it to the place where it should be placed. An element can be a frame, a button, a text field and many other things. The size of an element can be modified with sliders and all graphic elements such as shape, color, font,... can be defined with SceneBuilder. Once an element has been chosen, to uniquely identify it, an id can be associated with it. Thus, if 2 elements appear similar in appearance, with the id specific to each element, it is then possible, in back-end, to manipulate each element separately.

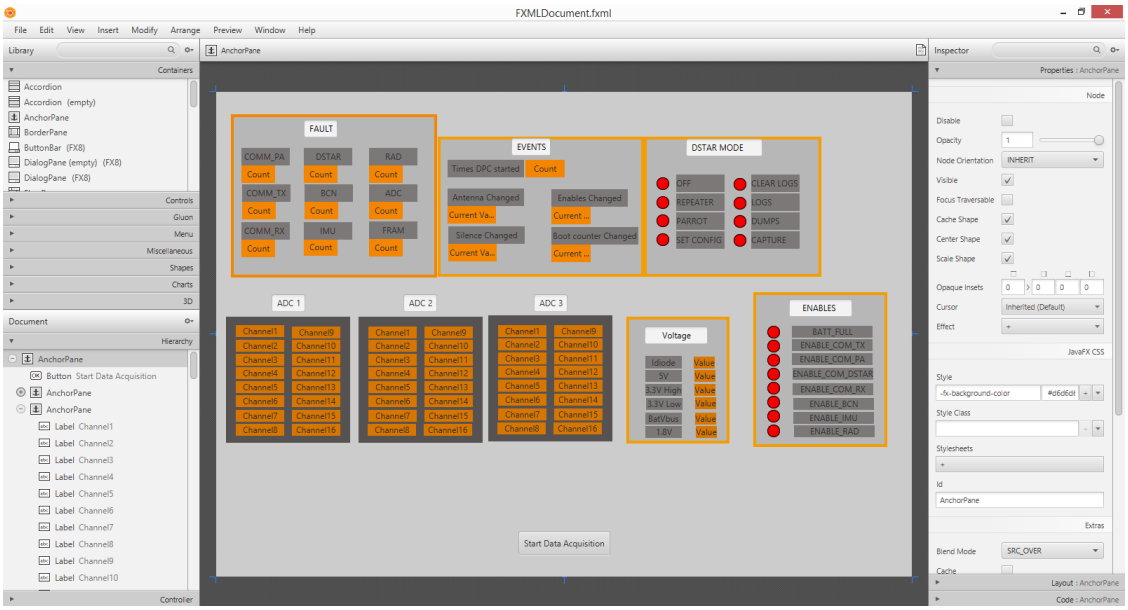

Figure 7.15: Example of the GUI manipulation in SceneBuilder

7.7.2 Back-end

For the back-end, this is really the logical and implementation part of the decoding discussed above. The main back-end tasks are:

1. Generate the GUI
2. Read the decoded data (specific format)
3. Recognize the slave module
4. Decode MISO and MOSI line, depending on the slave module.
5. Log and display on the GUI

To code these parts, it was important to make a clear code. Indeed, as previously discussed, there are still 4 modules left to be implemented. So, in the future, it will be necessary to supplement the code with additional decoders for each remaining module. So, for reasons of clarity, and in order to be completed by future students, I have made sure to create a class for each of these tasks.

As a reminder, the language chosen is Java and corresponds to an object-oriented language. Each class is associated with a task listed above. The rest of this section will focus on a short explanation of each class so that future students can continue with the same logic.

DataReading

A class is in charge of initiating the data acquisition via the sigrok-cli program. As discussed above, sigrok-cli can send the data to a pipe. So, the first thing to do is to have access to this pipe. In short, the data is available in an endpoint that a pipe created by sigrok-cli. This endpoint is accessible via Java and and manipulate as a simple buffer. In concrete terms, in terms of code, we can run the sigrok-cli program from Java and get the bytes decoded by sigrok-cli via a buffer.

As explained before, the data available in the buffer will have the following structure.

```
...
1i: byte1   #MISO
1o: byte2   #MOSI
1i: byte3
1o: byte4
...
```

DataManagement

Once the access to the data is done, the next step is to manipulate these bytes. The first step is to identify the slave module. This is done by identifying the ID of the slave module by referring to the Table that had been shown previously.

SPI Communication	Module	ID number	Description
SPI-1 Master: DPC	ADC 1	1	SPI communication between the DPC and different modules
	ADC 2	2	
	ADC 3	3	
	FRAM	4	
	COMM	5	
	BCN	6	
	IMU	7	
	RAD	8	
	BATT	9	
SPI-2 Master: Arduino	Dslogic	0	SPI communication between the Arduino and the Dslogic to send the digitized analog Data

Table 7.6: ID to identify it in a unique way each module

Once the slave module is identified, it is up to the decoders to give meaning to the bytes received by following what has been explained in the decoding part of each module. This class also displays the information once it has been decoded.

Decoder

Since each module has its own specificities, a class for each module has been created. Indeed, the way of decoding MISO and MOSI line is specific to each. The only thing they actually have in common is that they have to implement a method to decode the MISO and MOSI line. Therefore, an abstract class has been created. This class has only two methods: DecodeMiso and DecodeMosi. All classes decode inherited from this abstract class.

Each decoder class implement methods to execute what has been explained in the decoding part of each module.

LogData

Before displaying the information, it is necessary to log them. This class takes care of the methods to allow the information to be recorded correctly by specifying when it was recorded.

Chapter 8

Testing and validation of the data processing part.

In this project two distinct parts had to be realized, the data acquisition and the data processing. As seen above, the real-time data acquisition part is compromised. But this does not prevent us from working on the decoding part of the data. The practical implementation has already been discussed previously however it was necessary to test if the Java code was working as expected.

8.1 Input data collection

Therefore, the question arises as to how to test the implementation of the code if we do not have access to SPI communication. Indeed, as an input, the Java application expects:

```
...
1i: byte1  #MISO
1o: byte2  #MOSI
1i: byte3
1o: byte4
...
```

As a reminder, the Java application must be linked to an endpoint to read the data. This endpoint can be of several types: socket, pipe,... More information had been provided in the explanation of IPCs (Inter Process Communication).

So,for the operation of the Java application, regardless of the endpoint, provided it is properly linked to the application, its operation will be the same. Thus to simulate the reading of the Java application at one of these endpoints, a simple text file can be used. Indeed, by placing the elements that are expected in the endpoint in a text file, it will have a totally equivalent behavior from the application point of view.

The question that arises is how to produce the element that is expected in the endpoint in a text file.

The first idea would be to produce them "by hands". This means starting from the explanations of the SPI communications of each module and producing the expected input. Typically for ADCs, the expected pattern is as follows:

```
1i: 00          #MISO
1o: Conversion Byte 1  #MOSI

1i: Measurement MSB 1  #MISO
1o: 00          #MOSI

1i: Measurement LSB 1  #MISO
1o: Conversion Byte 2  #MOSI
```

```
1i: Measurement MSB 2   #MISO
1o: 00          #MOSIO

1i: Measurement LSB 2   #MISO
1o: Conversion Byte 3   #MOSI
...
```

However, this method is first of all very long. Indeed, writing each line is tedious and it is not guaranteed that the final result is without error.

Instead of trying to generate the expected input, the ideal would be to have direct access to the data. The solution found uses Dslogic, which only works with Pulseview in the end. Indeed, thanks to Pulseview which does not interrupt its data reading, it is easy to acquire the data actually exchanged by the DPC and the modules. To test the Java application, this is exactly what we need except one detail: it must respect the expected input structure.

8.1.1 Methodology to acquire the real data exhanged with the DPC

The methodology is as follows. First of all, connect the pins correctly to the Dslogic and launch Pulseview.

In Pulseview, configure the sampling rate and reading time correctly. After reading, use the SPI decoder by defining which pins are used as MISO, MOSI, SCLK and CS. This is explained in Figure 8.1

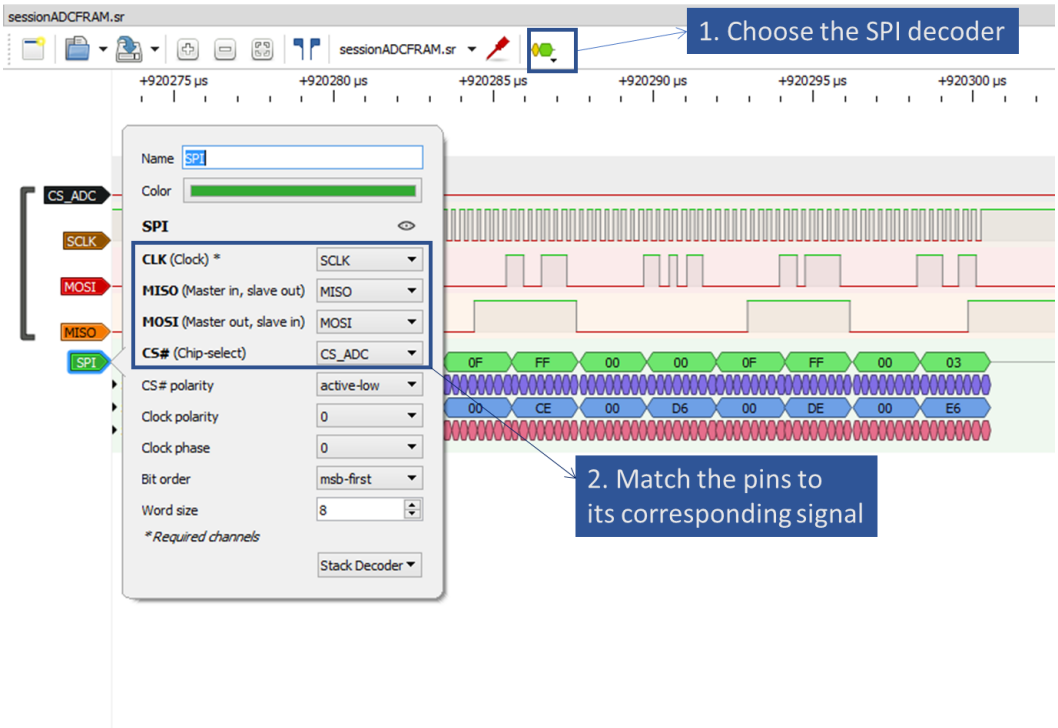

Figure 8.1: Illustration of how the SPI decoder can be set in Pulseview

The interesting part is that Pulseview allows to export decoded MISO line and MOSI line in a txt file. This can be done following the steps in Figure 8.2.

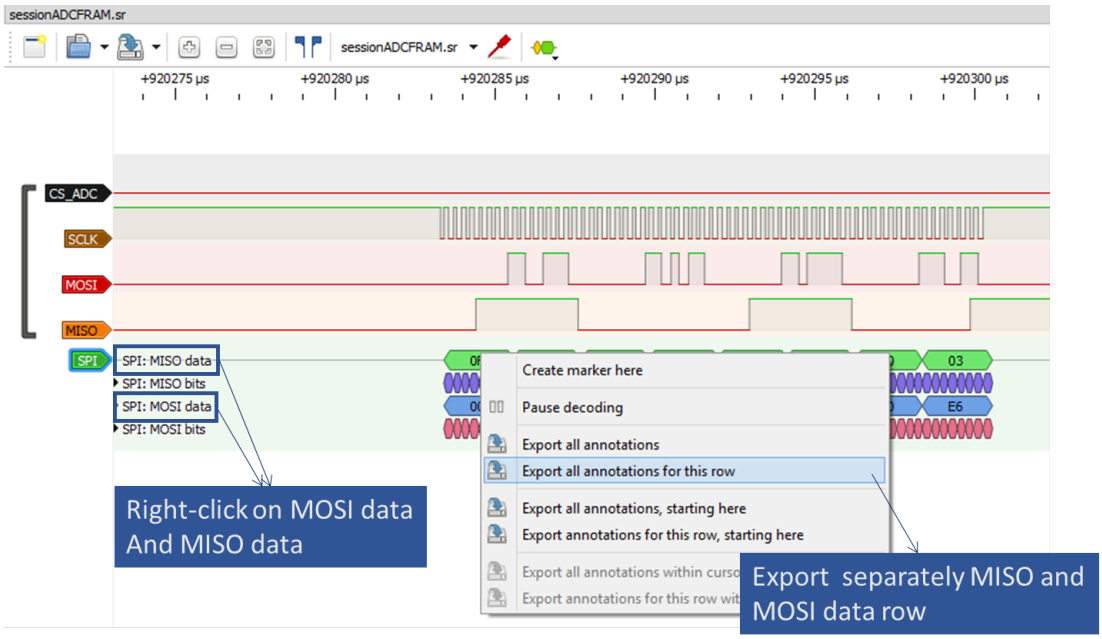

Figure 8.2: Illustration of how exporting decoded data in a file. This must be done separately for MISO and MOSI.

Each line is saved in a different file and will have the same structure as in Figure 8.3.

```
58457188-58457230 SPI: MOSI data: 00
58457230-58457274 SPI: MOSI data: 0F
58457273-58457317 SPI: MOSI data: FF
58457316-58457358 SPI: MOSI data: 00
58457359-58457401 SPI: MOSI data: 00
58458052-58458096 SPI: MOSI data: 0F
58458095-58458137 SPI: MOSI data: FF
58458138-58458180 SPI: MOSI data: 00
58458180-58458224 SPI: MOSI data: 03
58458223-58458265 SPI: MOSI data: 0F
58458266-58458308 SPI: MOSI data: FF
58458308-58458352 SPI: MOSI data: 00
58458351-58458393 SPI: MOSI data: 00
58459044-58459088 SPI: MOSI data: 0F
58459087-58459129 SPI: MOSI data: FF
58459130-58459172 SPI: MOSI data: 00
```

Figure 8.3: Illustration of the structure of the exported decoded data in a file.

Now the goal is to transform this text file so that it has the expected input structure. Typically, everything that is prefixed must disappear and be replaced by the ID of the module with which the DPC communicates. Then, the two separate file must be merged into a new one. And for this task, I wrote a Python script that reads the txt file provided by Pulseview and transforms it so that it has the expected input structure:

```
...
1i: byte1  #MISO
1o: byte2  #MOSI
1i: byte3
1o: byte4
...
```

The advantage is that in the end it is more reliable because it avoids the mistakes that would have been made by hand. But the main advantage is that the exchanged values are the ones that are really

sent and received by DPC, which allows to check the Java applications in the same situations as the data acquisition in real time.

Chapter 9

Status of the EGSE and alternatives

9.1 Data acquisition

For the hardware part that is in charge of data acquisition, everything still needs to be done. However, I am convinced that elements of my work will serve the next student who will take over, such as the explanations on the IPCs that I didn't have the opportunity to study during my studies. The signal characterization part, the explanation and decoding of the SPI protocol will certainly save time.

9.2 Data processing

Most of the validation was done on real data exchanged with the DPC using the methodology explained above. The purpose is to test if the decoder processes the data as requested. Here are the different things validated for each module :

- ADC1, ADC2, ADC3 :
  1. Deduce which channel sends a measurement.
  2. Recover the measurements read by channels.
  3. Display of the measurements in the corresponding channel in the GUI.
- FRAM
  1. Deduce the type of data among the 6 existing ones.
  2. Deduce the content of what is written or read according to the type.
  3. Display in the GUI.
- DSTAR
  1. Deduce the current mode.
  2. Display the result in the GUI.
- Non-SPI data
  1. Deduce the type of data.
  2. Recover the measurements values of the voltage or the state of the digital signal
  3. Display of the values in the GUI.

All the elements implemented for the moment in OUFTI-2 have therefore been tested. So, the back end is up to date with the modules implemented for the moment. With the other modules, more element will have be displayed. Therefore, the GUI will have to evolve. But as explained, by using SceneBuilder, this will not affect the work done on the back end. Of course additional elements can be implemented. We could imagine a more in-depth analysis on the data collected, for example.

9.3 Comments on alternatives solution for data acquisition and observations

As a reminder, the choice of the logic analyzer was made because they had enormous advantages including:

1. No hardware problems
2. Thanks to Sigrok-cli, a relatively simplified PC interface.

And this second point should not be underestimated. No matter how powerful the hardware is, it is useless if it is not combined with an easy-to-implement solution to interface it to the PC. Unfortunately, the data acquisition was a failure. But, after making progress with the JAVA application in charge of data decoding, I had the opportunity to look for possible alternatives for data acquisition.

As discussed, among the existing devices, the solution of a logic analyzer coupled with sigrok-cli is not possible. Therefore, the solutions explored are those based on a microcontroller and a second one based on the use of an FPGA. It should be clear that the purpose of the following sections will not be to provide a complete final solution but rather to share observations and remarks that will be useful for the development of the next hardware solution. Despite the fact that the solution explained in this these is not working, it would be a waste not to share what I found when thinking about an alternative solution

9.3.1 Requirements and tasks

The first minimum hardware requirement is to have the necessary number of pins, i.e. digital inputs (12 SPI, 7 ENABLES + BATT_Full) and 6 analog inputs with a resolution of 10 bits minimum, i.e. 26 in total. Besides the number of pins, it is important to define the tasks to be accomplished in advance to ensure that the device will actually fulfill its role. To ensure that the device does as few tasks as possible so that it only focuses on data acquisition, the natural tasks would be to stipulate:

1. read the signals.
2. send them to the PC in a specific format.

Which specific format? There are 26 pins. For the analog signals, 10 bits will have to be sent for each. The 10 bits will be sent successively one by one. FOr the digital signals, the value read in 1 bit will be sent, which makes 18 bits in total. So, for each pin, one bit will correspond to a signal that would be decided in advance. Therefore, the JAVA application receives the data of 26 bits in this predefined format and manipulates them to transform them so that they correspond to the input format expected by the application.

If we approach the problem this way, we'll have to send the data at a quite high speed. Because it must be remembered that the SPI signals are clocked at 4Mhz. So, it is necessary to sample the data at twice the frequency, 8MHz, or every 0.125μs. Once the data are sampled, it must be able to send them to the PC before the next sampling.

This is based on the assumption that the reading time is zero or at least negligible compared to the sampling period which is 0.125 μs. In concrete terms, this includes the reading of 18 digital pins and the time conversion of the 6 channels for analog data. This assumption is extremely optimistic and unrealistic. It is necessary to take into account the time of each of the tasks to have an idea closer to reality. Therefore, it is necessary to assign to the different tasks a typical average duration. In other words, it is necessary to define a duration for:

1. Reading a digital input
2. Conversion time for an analog data.
3. Data upload rate to the PC

These duration are component dependent and should be considered for both the microcontroller and FPGA-based solution. Taking into account the duration of each task, less time will be available to send the data. Less time to send the same amount of data as before means an increase in upload speed (to the PC). From there, we can think about ways to reduce the upload speed. To do this, there are two

strategies: send less data and /or provide more time to send data.

Regarding the part to send less data, it should be remembered that the sampling rate is only due to the presence of the 3 SPI lines: MISO, MOSI and the clock. For those, there is no choice, you have to satisfy the Nyquist criterion. However, for the other signals, it is useless to sample and send the contents of the pins at such a high frequency. In concrete terms, all non-SPI signals can be sampled by sending them every second without any problems and not every 0.125μs (= 125ns!). For SPI signals, not all select chips need to be sampled at this frequency as well. In the end, there are really only MISO, MOSI and clock that require special attention. In view of this observation, the idea of sending the value of the 26 pins in a predefined 26-bit format is no the best option for our specific application.

To increase the upload time and send the least amount of data related to the SPI, you should not forget the main objective. In our case there are 12 lines linked to the SPI: MISO, MOSI, SCLK and 9 CS. It is no use sending the value of all the pins every 0.125 μs. Indeed, the only information that is interesting are on the one hand the bytes sent on the MISO line and the MOSI line and on the other hand which slave module was involved in the communication. In the end, you have to wait 8 clock cycles and send the 4 bytes that will contain the slave with which the DPC communicates, followed by the bytes sent on the MISO and MOSI line.

For non-SPI signals, the general idea will be to understand that SPI signals have priority and that non-SPI signals should be read when possible. Concretely, the tasks will then be:

1. Read the MISO, MOSI and SCLK signals. Its duration will be 3 digital input readings every 0.125μs.
2. Once the reading of the MISO, MOSI and SCLK signals is complete, it is necessary to determine the CS involved in the communication. Since this information is only necessary at the end of the 8 clock cycles, the reading of the 9 CS can be spread over the 8 clock cycles. Its duration will be 8 digital input readings that can be spread over every 8 clock cycles, i.e. 1μs.
3. The remaining time will be devoted to reading non-SPI signals. This includes 8 digital inputs readings and 6 conversion time of the ADC used for the analogs signal. Its duration will be spread over 1s.

From now on, since on the one hand less data is needed to the PC and on the other hand more time is available to send the data, the upload speed is greatly reduced in this case. To get an idea of the bottom of the upload speed range, let's consider the case where only the 4 bytes that are the bytes sent to MISO, MOSI and the slave concerned. These 4 bytes must be sent before the next 4 bytes are ready i. e. before 8 clock cycles. Assuming a zero reading time, which is still an unrealistic assumption, the upload speed will be 4 bytes/1μs = 4MB/s. 4MB/s is a very high speed.

But it should be remembered that so far, the case we are in is very challenging because of the relatively fast speed of SPI communication. Indeed, for the moment, we are considering the case of continuous clock cycles without any interruption. In reality, this is not the case. Indeed, small interruptions are present in the order of several μs. Indeed, an interesting observation is that there is some delay during SPI communication. To observe it, let's analyze the case of a measurement writing of an ADC in the FRAM shown in the Figure 9.1. We recognize the structure that effectively corresponds to the writing of a measure: opcode followed by the address where to write the measure and finally the structure of the measure.

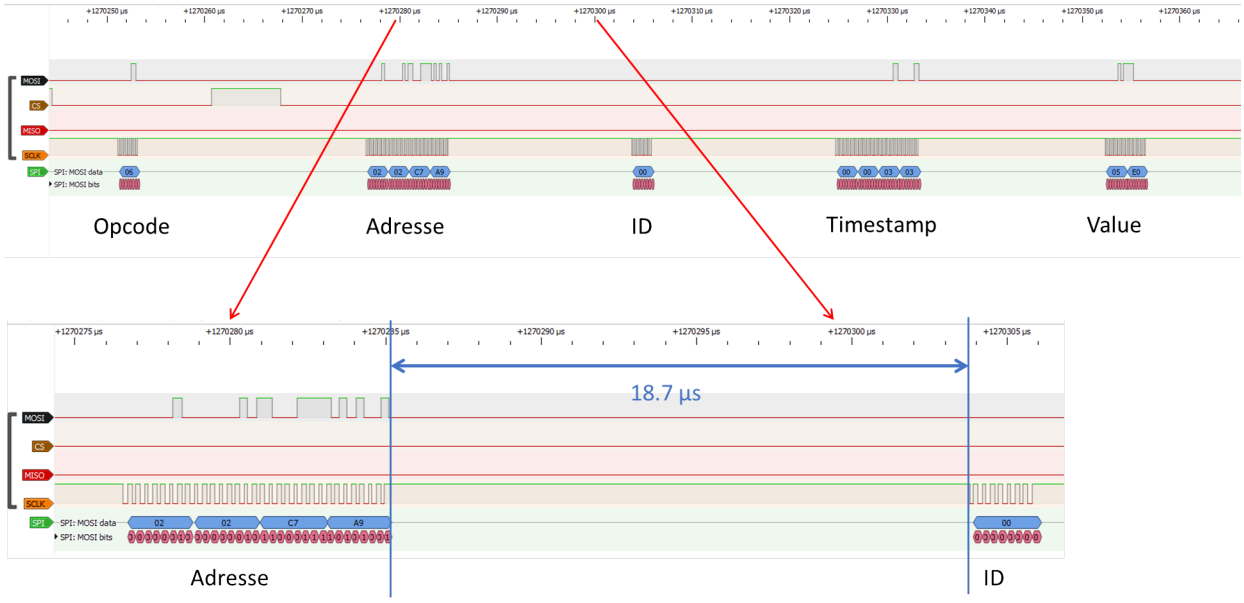

Figure 9.1: Illustration of the delay present when writing measurement obtained by the ADCs in FRAM.

The maximum number of consecutive bytes that can be sent is 4 bytes. After 4 consecutive bytes, there is a delay in the order of 18.7 μs before the next 4 bytes. In concrete terms, the time to send a byte is more or less 8 x 1/3.75 μs $\sim=$ 2μs. If you want to send 16 bytes, the time required is not 32μ (16 x 2µs) but 88μs (16 x 2μs + 3 x 18.7μs) because of the delay that occurs every 4 bytes. It must be verified that this is also the case for FRAM reading, which I did not have the opportunity to do. However, for ADCs, there is also a delay every 8 consecutive bytes, of about 32 μs. The purpose of this observation is simply that the upload speed will probably be lower than the case where the communication is uninterrupted. Because, in the end, there would be more time to send the data, which implies a lower upload speed.

9.3.2 Microcontroller based

To find the microcontroller, the same approach as with what we did to choose the Arduino. From a material point of view, the additional constraints will be the number of digital inputs that will be more important. If we choose an Arduino for the simplicity of development and prototyping, one model corresponds to the material characteristics: the Arduino Mega 2560. However, it must be ensured that it is powerful enough to sample quickly.

First of all, the first thing to check is the time it takes to read a digital pin. To find this information, one must look at the Arduino library that implements the DigitalRead function. However, this function has the disadvantage of being very slow, in the order of a few μs. There are alternatives to this function that allow to make a digital pin reading in only 2 clock cycles. However, since the Arduino is clocked at 16 Mhz, i. e. a period of 62.5ns, the reading time is at least 2 x 62.5 ns = 0.125 μs. Which is still too slow. So,typically thanks to the fact that the tasks to be performed and their duration have been determined in advance, this solution can be eliminated. So, developing an alternative solution on Arduino doesn't make sense because timings constraints are not met.

However, the choice is not limited to the Arduino. Indeed, other microcontrollers exist, like the STM32 microcontroller series.[22] In this series, all are far superior to the Arduino, in every aspects like clockrate, upload biterate,... So,as mentioned above, I didn't implement or test this solution. But before anything, the first step would be to list the tasks and make a timing diagram to make sure that the STM32 microcontroller meets the requirements. If the task list is feasible, it will be necessary to find a way to interface it with the PC. The STM32 offers a solution to communicate data via USART, USB or even Ethernet for some of them. What is interesting is that a lot of documentation is available. For example, Many projects exist and some ideas can be taken up such as avoid the USB protocol and uses a

USART to USB to send data to the PC or which java class can be used to access data received by USB.[23]

9.3.3 FPGA Based

If it turns out that using a microcontroller is not powerful enough, one solution would be to use the FPGA. Indeed, if processing speed is not enough, it may be necessary to switch to an FPGA. However, one of the concerns will be to interface it with the PC. By doing research, 2 solutions emerge. The first one is familiar because it uses the FX2 chip that was present in the Dslogic [25]. Indeed, development boards with an FX2 and drivers exist in the market[24]. Another relatively well documented solution is to use not a USB port but an Ethernet link to interface with the PC [26].

Chapter 10

Conclusion

In this thesis, the objective was to build an EGSE. The EGSE is an essential tool to check the proper operation of OUFTI-2, a step that is essential for any satellite. Thus, a number of signals were chosen because of their importance in the proper operation of OUFTI-2. The objective of the EGSE is to acquire these signals and analyze them. Therefore, two main components are essential in the design of the EGSE.

First, it was necessary to build a hardware device that could acquire these signals and interface with a PC. After distinguishing the types of data that need to be monitored, a data acquisition strategy was discussed. This is based on the use of a logic analyzer, the DSlogic and an Arduino. To interface with the PC, an open-source software, Sigrok-cli, allowed access to the data acquired by the DSlogic. Thanks to numerous tests, and despite concessions, it turned out that the solution found to acquire the data could not work. These early tests avoided implementing a hardware solution that was doomed to fail.

Despite the failure in data acquisition, it was possible to progress in the software part of the EGSE. The EGSE software must allow data processing and display in a graphical interface. To move forward in this part, it was necessary to define the structure of the data acquired by the data acquisition part of the EGSE. Once this structure was well defined, a JavaFX-based program was implemented. In concrete terms, the value of voltages, the content of digital signals and the data sent by SPI buses can be decoded and displayed on a graphical interface.

As far as SPI buses are concerned, the program is updated with the subsystems already implemented in OUFTI-2: D-STAR, FRAM, ADC1, ADC2, ADC3. Tests on actual exchanged data between the DPC and the subsystem were done to validate the implementation of the application.

In this work, an effort has been made to be as clear as possible to allow the student who will continue the project to start on a good basis. Indeed, to finalize the EGSE, it will be necessary to find a data acquisition solution that allows interfacing with the PC. And on the software side, when the SPI communication between the DPC and the BCN, RAD, IMU and BATT subsystems is defined, the JavaFX application must be updated. Fortunately, it has been designed to allow easy integration of the decoding of the SPI communication specific to each subsystem.

Bibliography

[1] NASA, *CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers*, https://www.nasa.gov/sites/default/files/atoms/files/nasa_csli_cubesat_101_508.pdf

[2] Maxim Integrated, *MAX14575A: 250mA to 2.5A Adjustable Current-Limit Switches*, https://datasheets.maximintegrated.com/en/ds/MAX14575A-MAX14575C.pdf.

[3] DIGILENT, *Lab 4c: Communications - SPI Serial Protocols*, https://reference.digilentinc.com/learn/courses/unit-4-lab4c/start (accessed August 2019).

[4] National Instruments, *DATASHEET NI9205*, http://www.ni.com/pdf/manuals/378020a_02.pdf.

[5] Pico Technology, *PicoLog 1000 Series Multipurpose data acquisition datasheet*, https://www.picotech.com/download/datasheets/picolog-1000-series-data-sheet.pdf.

[6] Tektronix, *The XYZs of Logic Analyzers*, https://www.ece.ubc.ca/~robertor/Links_files/Files/xyzlogicanalysis.pdf.

[7] CYPRESS, *EZ-USB FX2LP USB Microcontroller High-Speed USB Peripheral Controller*, https://www.cypress.com/file/138911/download.

[8] Sigrok, *Sigrok main page*, https://sigrok.org/. (accessed November 2018).

[9] Sigrok, *Supported hardware*, https://sigrok.org/wiki/Supported_hardware (accessed November 2018).

[10] DreamSourceLab, *DSView User Guide*, https://www.dreamsourcelab.com/doc/DSView_User_Guide.pdf.

[11] Sigrok, *Dslogic Plus PCB front*, https://sigrok.org/wiki/File:Dslogic_plus_pcb_front.jpg (accessed March 2019).

[12] Arduino, *ATmega328P datasheet*, http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (accessed March 2019).

[13] Texas Instruments, *TXB0104 datasheet*, http://www.ti.com/lit/ds/symlink/txb0104.pdf (accessed April 2019).

[14] Sigrok, *sigrok-cli*, https://sigrok.org/wiki/Sigrok-cli (accessed November 2018).

[15] Sigrok , *Sigrok bug 762*, https://sigrok.org/bugzilla/show_bug.cgi?id=762 (accessed March 2019).

[16] Saleae , *API saleae*, https://support.saleae.com/saleae-api-and-sdk/socket-api (accessed February 2019).

[17] Maxim Integrated, *12-Bit 300ksps ADCs with FIFO, Temp Sensor, Internal Reference*, https://datasheets.maximintegrated.com/en/ds/MAX1227-MAX1231.pdf.

[18] CYPRESS, *FM25V20A: 2-Mbit (256 K × 8) Serial (SPI) F-RAM*, https://www.cypress.com/file/141396/download.

[19] F. Piron, *Optimization of the AX-25 and D-STAR telecommunications systems of the OUFTI-2 nanosatellite*, http://hdl.handle.net/2268.2/6751, Master's thesis, University of Liege, 2019.

[20] G. Martin, *Conception, implémentation et test du logiciel de l'ordinateur de bord du nanosatellite OUFTI-2*, Bachelor's thesis, HEPL, 2019.

[21] Sigrok, *PulseView User Manual*, https://sigrok.org/doc/pulseview/0.4.1/manual.html#overview (accessed November 2018).

[22] STMicroelectronics , *STM32 microcontroller page*, https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html (accessed July 2019).

[23] Embedded Lab, *STM32 SERIAL COMMUNICATION*, http://embedded-lab.com/blog/stm32-serial-communication/ (accessed July 2019).

[24] ZTEX, *SDK for ZTEX FPGA Boards*, https://www.ztex.de/firmware-kit/ (accessed July 2019).

[25] Hridya Valsaraju and Gopalakrishnan Vijayakumar, Cypress Semiconductors, *Implementing high Speed USB functionality with FPGA- and ASIC-based designs*, https://www.eetimes.com/document.asp?doc_id=1279155# (accessed July 2019).

[26] Jean P. Nicolle, *A recipe to send Ethernet traffic*, https://www.fpga4fun.com/10BASE-T0.html (accessed July 2019).

Appendix A

Appendix

A.1 SPI Decoder Code

```
# Key: (CPOL, CPHA). Value: SPI mode.
# Clock polarity (CPOL) = 0/1: Clock is low/high when inactive.
# Clock phase (CPHA) = 0/1: Data is valid on the leading/trailing clock edge.
spi_mode = {
  (0, 0): 0, # Mode 0
  (0, 1): 1, # Mode 1
  (1, 0): 2, # Mode 2
  (1, 1): 3, # Mode 3
}

class Decoder(srd.Decoder):

## Asign the input singnals to variables: Depends on the langage
## Example
## Miso = Pin 1 , Mosi = Pin 2, SCLK = Pin 3,
## CS 0-> 8 = Pin 4->12

## Specify the options:
        #- Default CPOL & CPHA for the SPI Mode
        self.CPOL = 0
        self.CPHA = 0
        #- The bit order: MSB or LSB first
        self.bo = 'msb-first'
        #- Wordsize
        self.ws = 8

  def __init__(self):
    self.reset()


        ##Define the variables useful for the decoding.
        def reset(self):

                ##is limited between 0 to wordsize (=8)
    self.bitcount = 0

                ##Miso or Mosi Byte
    self.misodata = self.mosidata = 0
```

```
                ##Useful if wordsize>8,
                ##If wordsize = 8, data can be sent with 1 byte
                ##If wordsize > 8, need self.bw bytes to send data
    self.bw = (self.options['wordsize'] + 7) // 8

        ##COnstruct the MOSI and MISO byte and transfer them
def putdata(self):

    bdata = self.misodata.to_bytes(self.bw, byteorder='big')
                #SEND DATA SERIAL PRINT

    bdata = self.mosidata.to_bytes(self.bw, byteorder='big')
                #SEND DATA SERIAL PRINT

def reset_decoder_state(self):
    self.misodata = 0
    self.mosidata = 0
    self.bitcount = 0

def cs_asserted(self, cs):
    active_low = (self.options['cs_polarity'] == 'active-low')
    return (cs == 0) if active_low else (cs == 1)


                ##Each bit is read one by one
                ## Until wordsize is not reached, it is important to continue receiving data.
def handle_bit(self, miso, mosi, clk, cs):
    # If this is the first bit of a dataword, save its sample number.

    # Receive MISO bit into our shift register.
    ##Since self.misodata is reset after each wordsize is reached
    ## to 0, we shift the new bit to misodata.
    ##bitcount = 0: miso = 1, misodata=1
    ##bitcount = 1: miso = 1, misodata=11
    ##bitcount = 2: miso = 0, misodata=110

      if self.bo == 'msb-first':
        self.misodata |= miso << (ws - 1 - self.bitcount)
      else:
        self.misodata |= miso << self.bitcount

    # Receive MOSI bit into our shift register.

      if self.bo == 'msb-first':
        self.mosidata |= mosi << (ws - 1 - self.bitcount)
      else:
        self.mosidata |= mosi << self.bitcount

    self.bitcount += 1

    ## Most important, we receive bit per bit, so until wordsize is not reached
    ## it is important to continue receiving data.
    # Continue to receive if not enough bits were received, yet.
    if self.bitcount != ws:
      return

    ##Once the bitcount equals to the wordsize, we can send the data
```

```
    self.putdata()

    ##bitcount is reset to 0 there
    self.reset_decoder_state()

  def find_clk_edge(self, miso, mosi, clk, cs):
    /!\
    if "clock not changed ":
      return

    # Sample data on rising/falling clock edge (depends on mode).
    mode = spi_mode[self.CPOL, self.CPHA]
    if mode == 0 and clk == 0:  # Sample on rising clock edge
      return
    elif mode == 1 and clk == 1: # Sample on falling clock edge
      return
    elif mode == 2 and clk == 1: # Sample on falling clock edge
      return
    elif mode == 3 and clk == 0: # Sample on rising clock edge
      return

                ##Once the mode meets the clock, we care about the bits.
    self.handle_bit(miso, mosi, clk, cs)

  def decode(self):

    # We want all CLK changes.

                ## 'l': Low pin value (logical 0)
                ## 'h': High pin value (logical 1)
                ## 'r': Rising edge
                ## 'f': Falling edge
                ## 'e': Either edge (rising or falling)
                ## 's': Stable state, the opposite of 'e', the current and previous pin value
                ##                were both low (or both high).
    wait_cond = [{0: 'e'}]

    while True:
      (clk, miso, mosi, cs) = self.wait(wait_cond)
      self.find_clk_edge(miso, mosi, clk, cs)
```