University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

Multi-Channel Acquisition Module for EMG Signals

Diploma thesis

Submitted in partial fulfillment of the requirements
for the degree of *Engineer*
in the domain of *Electronics, Telecommunications and Information Technology*
study program *Applied Electronics*

Thesis Advisor(s)
Prof.Dr.Ing. Corneliu Burileanu,
As.Dr.Ing. Ana-Antonia Neacșu

Student
Andrei-Cătălin Dăescu

Year 2021

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology
Study program **ELAen**

Anexa 1

DIPLOMA THESIS
of student **DĂESCU V.C. Andrei-Cătălin , 441F-ELA**

1. Thesis title: Multi-Channel Acquisition Module for EMG Signals

2. The student's original contribution will consist of (not including the documentation part) and design specifications:
The project focuses on developing a multi-channel EMG Signal Acquisition module, which will be a portable circuit, placed on the forearm. This Acquisition module will communicate via Bluetooth to an external device (PC, Phone).
The system will have the following main components:
microcontroler (at student's choice),
Multi-channel SigmaDelta ADC,
HC-05 Bluetooth Module.
The ensemble will be integrated on a PCB, designed by the student. A GUI will be implemented on the receiving device, where the EMG data can be displayed as a waveform in time. Additionally, other contributions may include a processing module for the raw EMG data (filtering, FFT, etc.). An accelerometer can be added to the circuit to correlate the EMG activity with intertial data.
This acquisition module like this can be used in a variety of applications, ranging from biomedical devices up to entertainment applications.

3. Academic courses the thesis is based on::
AMP, MC, PDS, SDA

4. Thesis registration date: 2020-11-24 12:49:50

Thesis advisor(s),
As. drd. Ing. Ana-Antonia NEACȘU

Prof. Corneliu Burileanu

Student,
DĂESCU V.C. Andrei-Cătălin

Departament director,
Ș.L. dr. ing Bogdan FLOREA

Dean,
Prof. dr. ing. Mihnea UDREA

Validation code: **1b21be4b69**

Statement of Academic Honesty

I hereby declare that the thesis *Multi-Channel Acquisition Module for EMG Signals*, submitted to the Faculty of Electronics, Telecommunications and Information Technologies, University POLITEHNICA of Bucharest, in partial fulfillment of the requirements for the degree of *Engineer* in the domain Electronics and Telecommunications, study program *Applied Electronics* is written by myself and was never before submitted to any faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand that plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, July 2021.
Student: Andrei-Cătălin Dăescu

Table of Contents

List of figures

List of abbreviations

ADC = Analog to Digital Converter
A/D = Analog to Digital
ARM = Advanced RISC Machine
AIN = Analog INput
AINP = Analog INput Positive
AINM = Analog INput Negative
ASCII = American Standard Code for Information Interchange
COM = Computer On Module
CPU = Central Processing Unit
CS = Chip Select
DIN = Data INput
DOUT = Data OUTput
DMA = Direct Memory Access
EMG = Electro-Myo-Graphic
ENIG = Electronless Nickel Immersion Gold
HAL = Hardware Abstraction Layer
IDE = Integrated Development Environment
JTAG = Joint Test Action Group
LL = Low Level
MCU = Microcontroller Unit
MISO = Master In Slave Out
MOSI = Master Out Slave In
RAM = Random Access Memory
RDY = ReaDY
RISC = Reduced Instruction Set Computing
SCK = Serial Clock
SDK = Software Development Kit
SoC = System-on-Chip
SPI = Serial Peripheral Interface
SWD = Single Wire Debug
TDFN = Thin Dual Flat No leads
UART = Universal Asynchronous Receiver Transmitter
USB = Universal Serial Bus

Introduction

Motivation

As the technology becomes more advanced, compact and also much more accessible, areas of science in which only research institutes and laboratories could work started to become accessible to almost everyone interested in.

One of the main goals of present computing technology is facilitating the Human-Computer Interaction beyond the standard technologies that are commonly used. Interaction based on gestures and speech is much more natural and intuitive, although one major advantage is that it can provide disabled people to ineract with computers easier, significantly improving the quality of life as computers are now used in most daily activities.

Electromyography is a technique that involves human-computer interaction based on electrical activity of the skeleton muscles. Human body muscles are controlled by small electrical signals transmitted through nerves. The muscles by themselves are generating electrical signals during contractions, which can be acquired and interpreted by electronic systems in order to identify the gesture performed and turn it into a command for a computer, or even a bionic prothesis.

EMG equipment used to be accessible only in medical applications, being expensive and requiring consistent knowledge of operating it. In time, as the embedded systems like Arduino began to gain popularity in the makers' space, various modules were becoming available as well, enhancing the possibility of adding more funtionality to a project.

Surface ElectroMyography became much more popular thanks to the available sensors that can be directly connected to any microcontroller which includes an ADC. This way, signal acquisition can be achieved by any hobbyist easily.

Objectives

An EMG acquisition module should be able to communicate with a host computer, and, at the same time, it should be also a wearable device connected through a wireless channel to the computer.

One accessible interface that accomplishes most of the requirements is Cyton from OpenBCI. It is an acquisition module that comes with a bluetooth dongle, which inserts into the PC's USB port. It is created to be used with gelled EMG electrodes (the classic type of surface EMG sensors), which comes with some disadvantages:

- They are hard to reposition, as the electrodes lose some of the adherence with every movement;

- They are not reusable. Most Gelled EMG sensors are cheap, altough for repeated measurements new pairs must be used, which does add up to the cost.

Last disadvantage is related to the OpenBCI module by itself. It was not designed to be fully attached to the subject's limb, rather it sits on a table while the electrodes are sticked to the muscles, which does limit the overall mobility of the subject.

The thesis focuses on the following objectives:

- testing a newer type of EMG sensor, commonly known as "Dry Surface EMG";
- designing a compact acquisition module that can be interfaced with the sensors;
- creating a computer Python script that can plot the data acquired by the module;
- compare the results with an existent EMG analysis solution.

The mentioned Dry EMG sensors are made with electrodes which overcome the two major issues of gelled electrodes, as they only consist of stainless steel or Ag contacts that are placed directly on the skin. The do allow easy reposition and can be always reused.

Applicability

One of the main applications of electromyography is the control of bionic prothesis for disabled persons. One acquisition module would be used on the remaining part from the amputee limb. This would be used in combination with a gesture classification procedure in order to transform the acquired signals into commands for the prothesis. Other applications may involve remote control of a robotic arm based on the gestures of the hand, as well as a means of controlling a computer. EMG gesture recognition may be used as well in entertainment application, in conjuction with the Virtual Reality technologies.

Chapter 1
Basics of ElectroMyographic Signals

In order to be able to do a proper acquisition, it is mandatory to understand the origins of the signal, as well as some of its properties. The muscle hierarchical structure will be presented in order to understand what is the source of these signals.

The basic idea behind Electromyography, putting aside the type of sensor used, is that any muscle contraction does produce electrical currents. The human body, from the electrical point of view, is neutral. This does not mean that there are no positive and negative charges at cellular and macrocellular level[1].

Skeleton muscles are the major group of interest, as many EMG applications are focused on rehabilitation of persons whose body integrity was severely affected by diseases or accidents. Most bionic arm or leg extensions acquire raw signals from the limb muscles and replicate the movement of the natural limb.

Skeleton muscles, also know as stray muscles, have the greatest proportion of the human body weight. Analysis of a single muscle group will not reveal all the details regarding the specific movement, since it is the result of coordination of multiple muscle groups. Apart from the groups which are directly involved in the specific movement, body's postural muscles are involved as well. Even for a simple action, such as forearm raising, the body's center of mass changes, and the postural muscles need to adjust their contractions in order to bring the body back into the balance state [2].

For every movement, the muscle groups will be assigned 3 main roles:

- The agonist - it is responsible for the power of the movement;
- The synergist - it assists the agonist in the movement by performing fine corrections;
- The antagonist - it opposes to the movement of the agonist, stabilizing the move performed.

The roles briefly described above are not permanently assigned to each of the muscle groups that are involved in a movement. For example, for a typical palm extension, some groups will be agonists and synergists, while the others will be the antagonists. For the opposite movement, the roles will swap. [3]

1.1 Muscle Anatomy

The muscles are by themselves a hierarchical structure. A muscle group is composed of several compartments, known as fascicles. Each fascicle has in its componency multiple muscle fibers,

which are isolated one from each other by means of a membrane named sarcolemma. Inside the sarcollema are found the myofibrils which are covered by Sarcotubules and Sarcoplasmatic reticulum These are channels that contain ions of Calcium with strong positive charges. The release of ions is controlled by neurons located near them.

The muscle fibers are also composed of smaller threads known as myofibril. It is a structure that is further decomposed into smaller interlacing myofilaments, which are composed of two types: myosin and actin. The muscle structure is accurately described in Figure x.x, showing the hierarchical structure of a typical skeleton muscle group.

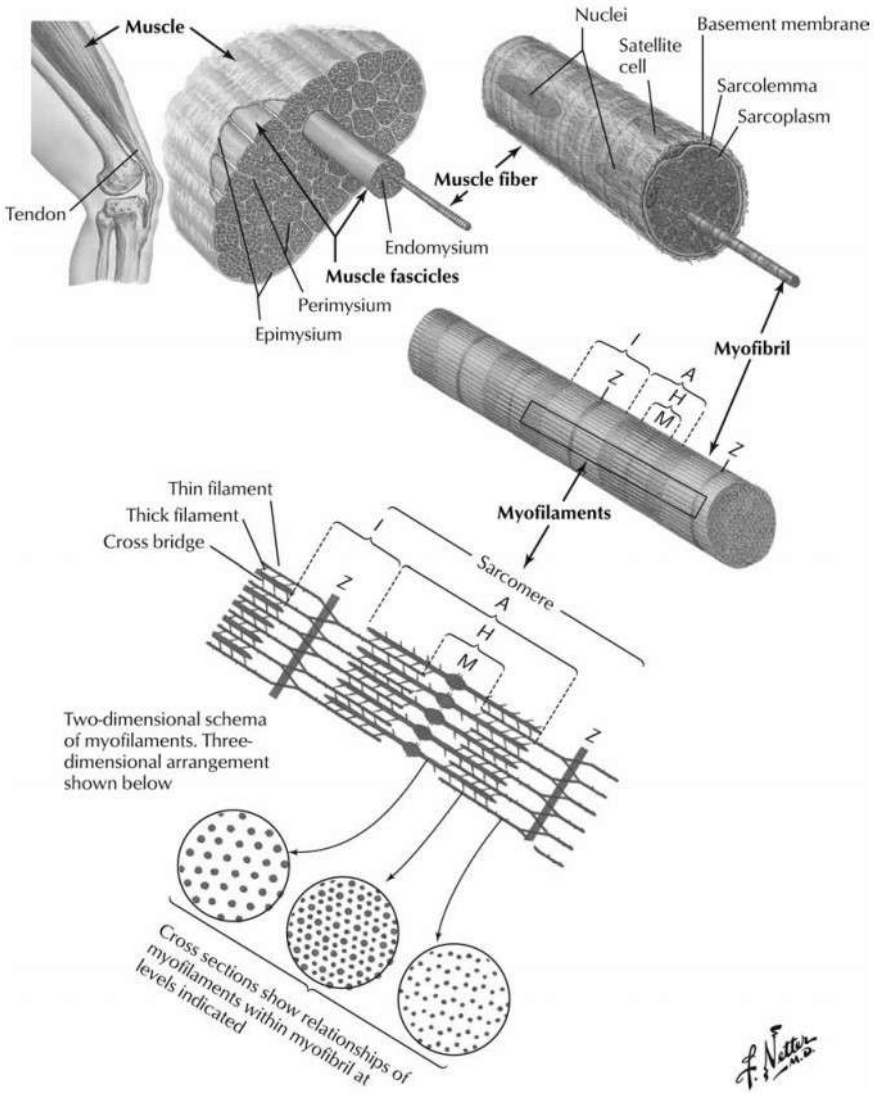


Figure 1.1: Hierarchical Structure of Muscles.

Myosin and actin structures are special due to the fact that their degree of overlapping determines the force generated by the muscle. The existence of dark and light bands on a myofibril is justified by the overlapping of thin filaments (actin) over the thicker ones (myosin). Dark regions represent the areas where actin and myosin filaments are connected. Also, actin filaments are interconnected other groups by means of "Z" disks. A region delimited by two such Z-disks is named Sarcomere. One special section inside the Sarcomere, as the figure illustrates, is the "A" region, where myosin and actin overlap. A-region is the active area of the muscle. The figure does illustrate it in two dimensions, although the overall myofibril is a three-dimensional structure, with complex interlacing [4]. The muscle contraction can be compared, to a certain extent, with the action of hydraulic piston, in which only the piston

pulling force does produce useful mechanical work. The actin filaments are equivalent to the piston axis.

1.2 Electromyographic signal source

Another important aspect that must be known is the chemical process that determines a muscle contraction. The overlapping of myosin and actin fibers determines the force of contraction. When muscle is in the relaxed state, both types of filaments are weakly negative charged, so there is a reciprocal repel force between them. When a triggering signal arrives from the brain, the neurons located nearby give a command to the sarcoplasmatic reticulum, which releases the Calcium ions to the myofilaments. Since a positive and a negative charge attract to each other, the Calcium ions will be attracted in the space between the actin and myosin. The myosin will be attracted to the Calcium, and the Calcium will be attracted to the actin. In this manner, a contraction is generated, and the charge movement inside the muscle fiber will generate an electric current [4].

1.3 Noise in signal acquisition

Noise in EMG signals can be caused by a variety of factors, and its presence is an issue as the EMG signals have a peak-to-peak amplitude of about 10mV, centered around 0V. In many cases, noise can be greater than the signal. Some of the most common causes of noise are:

1. Intrinsec noise of the equipment.

   This kind of noise is the result of the functioning of equipment. All components generate a form of noise. The noise can come from both analog or digital circuits, especially the last one as most modern digital circuits use higher frequency clocks. There is also noise generated by the components themselves[5].

2. Static electricity.

   As the electrodes are placed directly on the skin, part of the noise present on it will be received as well. Static electricity is mostly generated by friction forces, friction with objects, even from friction with air. Human body is a small battery of static electricity, and this aspect may cause issues with the acquisition.

3. Movement artefacts.

   The sensors are positioned on the skin and they are mostly rigid. Skin is elastic and, as a consequence of some moves, the sensors may move as well from their position, generating movement artefacts which appear as high amplitude spikes in the recorded signal[5].

4. EMG intrinsec instability.

   There are multiple groups of muscles involved in a specific movement. As the muscle fibers are split into categories, depending on the force and speed of contractions, the same movement can be generated in a number of ways. This is one of the reasons for the random nature of an EMG signal[5].

Chapter 2
Acquisition Interface

The acquisition interface involves usage of two modules:

1. The EMG Acquisition Module, which is the wearable device placed on the forearm
2. The Adapter, which connects to the Host PC.

2.1 Sensors

The type of sensor used in any data acquisition application determines directly the quality of the acquired signals. The choice of a sensor is strongly influenced by the application purpose. EMG acquisition sensors are classified in:

- `Invasive`;
- `Non-Invasive`.

The `Invasive` type of sensors consist mainly of needles inserted into the muscle, and inside a needle can be a single electrode or a group of them. Depending on the technique involved, the needle can contain the electrode and remain inserted during the acquisition, or it can be used to insert an electrode (a wire) and then taken out, leaving the standalone electrode inserted into the muscle.

The major advantage of invasive sensors is that they offer the highest signal quality. Having the sensor placed close to the signal source, a single muscle group can be accurately monitored in different regions, thus avoiding the overlapping of signals generated by other nearby muscles and also the noise from the skin surface. An advantage of wire electrodes is that they are comfortable for the subject and allows strong contractions over extended periods of time. Being easy to analyse, the EMG signals acquired by means of invasive sensors are suitable to medical applications such as diagnosis. The only downside of the invasive sensors are the inability to change position across different groups of muscles and the necessity of qualified personnel to implant them.

The `Non-Invasive` sensors, commonly referred as Surface EMG sensors (or sEMG), come as a complement for the drawbacks of the previously discussed type. They are easily placed on the skin, the subject can single-handedly equip them, but their placement position does influence the acquisition results, since the signal is a combination of signals collected from multiple muscle groups, over which noise from the skin surface is added, making the classification more difficult in comparison with the invasive method[1].

The surface EMG sensors split up into two categories as well, as they are: gelled EMG and dry EMG sensors.

Gelled EMG sensors consist of an Ag electrode which is placed on the skin by the help of `AgCl` sticky gel pads. They do have the advantage of a greater stability as the sticky gel prevents them from moving and generating artifacts in the acquired signal.

Dry EMG sensors have wider Ag electrodes which are placed directly on the skin without any gel. They do have the advantage of easy relocation and reusability, which make them the preferred choice for the presented application.

This aspect does not exclude the possibility of doing a side-to-side comparison of both types of surface electrodes.

As there are many vendors of EMG sensors on the market, some decisions must be taken regarding the characteristics and features they have. Many sensors come with a signal conditioning circuit on the electrode board, while others have the electrode and the signal conditioning circuit as separate modules, linked together by a cable. Other types of sensors (mostly gelled EMG) come only as standalone electrodes. Other issue is the output signal. The EMG signals can be raw or a simple transformation can be applied to it.

The first prototype for the acquisition unit was a sensor that comes with a signal conditioning circuit. This approach will have increased costs, but it will ensure a certain degree of signal quality, especially if the physical distance between the electrode and the circuit is smaller. This way, less additive noise will be present in the raw EMG signal.

One option of dry EMG sensors would be OyMotion Gravity from DfRobot. They do include both the electrode board and the signal conditioning circuit. It only needs an ADC (or a microcontroller with A/D capabilites). One OyMotion Gravity sensor is illustrated in Figure 2.1.

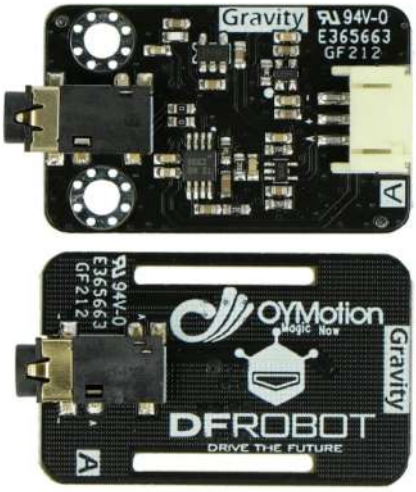


Figure 2.1: OyMotion Gravity Dry EMG Sensor.

Other options may be Advancer MyoWare, which is a gelled EMG sensor that comes under the form of a single board which contains all the necessary circuit. It has the advantage of providing both the raw EMG signal and the processed signal. Just like OyMotion, it needs an external ADC to work. An example of MyoWare sensor is illustrated in Figure 2.2.

Figure 2.2: Advancer MyoWare EMG Sensor (without gel pads).

2.2 Analog to Digital Converter

In order to create a complete acquisition module, a number of 4 or 8 sensors would be needed to record the entire activity of the forearm muscles. Since each of the presented sensors has one channel of analog output, an ADC with 8 channels is mandatory in this situation.

There are many ADCs with multiple channels on the market, with different architectures. Before choosing a specific device, it is important to know the properties of the measured signal, which can be: single-ended or differential, the bandwidth, the amplitude and the signal-to-noise ratio.

Usually, an EMG signal has:

- amplitude up to 10mV (in case of strong contractions);
- useful bandwidth up to 250Hz;

The signal parameters will determine which ADC architecture shall be chosen.

The resolution of the ADC determines directly the quantization noise, which cannot be eliminated, but it can be brought to a tolerable level so that it won't affect the measurement. Both sEMG sensors discussed in the previous section provide single-ended output signal between 0 and 3.3V, with 1.5V offset. For a typical reference voltage of 3.3V, the integrated 10 bit ADC of a microcontroller would achieve a resolution of 3.22 mV/bit, sufficient for a coarse acquisition. If more precision is desired, an ADC with 12 bits or more would be ideal.

The sampling rate is another factor that determines the quality of acquired signal. Considering that up to 8 channels must be supported, each having an analog bandwidth of about 150 Hz, up to 250 Hz maximum, this translates into a sampling rate of at least 300 Sa/s for one channel. Combined for all channels, the ADC must be able to acquire at least 2400 Sa/s for 8 channels, or 1200 Sa/s for 4 channels.

Considering these specification, the EMG module would require an ADC that is capable of providing good resolution and a moderate acquisition speed. The most suitable architecture of ADC that would fulfil the requirement is the `Sigma-Delta Modulator`.

If it was required to design an acquisition module that can capture raw EMG signal directly from the electrodes and convert it into digital, the circuit would need to have:

- bipolar supply;
- differential input amplifier;
- analog filter.

The market offers ADCs which are actually fully integrated analog front end, meaning that they are equipped with amplifiers and filters by default.

For a beginning prototype, having a fully integrated analog front-end allows for a greater design flexibility, as well as a more compact circuit. These are the reasons for choosing `AD7124-8` from Analog Devices.[6]. For easier prototyping, a breakout board like the one from Figure 2.3 will be used.

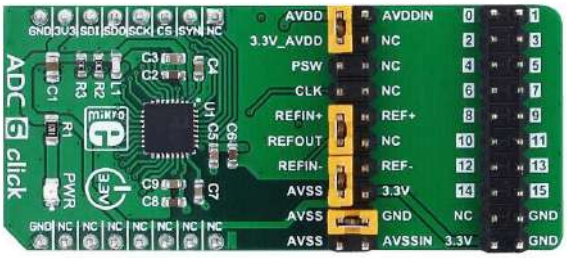


Figure 2.3: AD7124-8 on ADC-6 Click Board.

Key features of the device are:

- Sigma-Delta Architecture, suitable for low frequency signal sampling;
- max. 24 bits precision;
- 8 channels, differential or single-ended;
- internal temperature-compensated 2.5V reference;
- external reference, either on dedicated input pins or analog VDD.

2.3 Microcontroller

The EMG module performs one action: acquiring signals from sensors and transmitting them by means of a Bluetooth transceiver to an external device, such as PC or laptop computer, which can optionally perform classification on it.

The microcontroller choice and the programming interface influence the following aspects:

- core features of the CPU;
- processing power;
- peripheral set;
- number of I/Os;

- physical size and power consumption;
- programming and debugging support;
- the required development tools.

There are many microcontrollers and development board manufacturers. There are devices that excel at price-to-performance ratio, while others have the advantage of extensive support thanks to the communities of developers, as well as organised and documented development tools. The following sub-chapter will present and analyse a few options available on the market, as well as which one will be included in the project.

1. Arduino

`Arduino` is a widely spread concept in the embedded computing. It is a popular platform with support for most of the embedded modules found on the market. Arduino offers the possibility of assembling a project quickly and with a good learning curve. Arduino also offers a simple and user-friendly IDE. There are Arduino Boards that suit most requirements for prototyping. Any Arduino compatible development board can be programmed in Arduino C++. An Arduino Nano board is shown in Figure 2.4.

Figure 2.4: Arduino Nano Board.

2. Espressif

`Espressif` is a chinese manufacturer of wireless SoCs that use a 32 bit RISC CPU architecture named "Tensilica" and can be equipped with both Wi-Fi and Bluetooth technologies, making them suitable for wireless communications. They are Arduino compatible, but the manufacturer does offer some packages for other IDEs such as VisualStudio or Eclipse based IDEs. Some of their SoCs come as dual core devices, which have more processing power in a compact package. There is support offered for both C/C++ and MicroPython. One ESP32 avaliable as a breakout board is shown in Figure 2.5.

Figure 2.5: ESP32 Development Board.

3. STM32

STM32 microcontrollers are Arduino Compatible as well, but they are based on ARM Cortex 32 bit RISC architecture. They can be programmed in Keil MDK, Atollic, ARM Mbed and more. ST Microelectronics does provide the users with Cube IDE, a powerful and free development toolset that is used for both standalone microcontrollers and development boards.

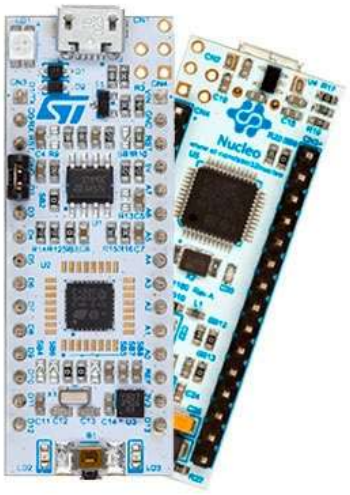


Figure 2.6: STM32 Nucleo Board.

ST offers a series of development boards named Nucleo (Figure 2.6), whose main feature is that they come equipped with `STLink-V2` along with the microcontroller, being also budget-friendly.

4. Raspberry Pi Pico

The name "Raspberry Pi" is usually associated with single board computers, which are not microcontrollers. The perception changed when `RP2040` was introduced. RP2040 is the first microcontroller designed by Raspberry Pi and features dual Core ARM, together with a faster Programmable Input Output peripheral. Raspberry Pi offers SDKs for larger IDEs, just like Espressif, and the microcontroller can be programmed in C/C++, as well as MicroPython. Raspberry Pi Pico can be debugged by means of JTAG/SWD compatible debuggers, or by another Pico (Figure 2.7) programmed as debugger.

Figure 2.7: RP2040 on Raspberry Pi Pico Board.

The microcontrollers presented above are just a few popular options offered in the embedded industry. Choosing a microcontroller for a project is influenced both by the capabilities of the device itself and also by their support, such as available IDEs, frameworks, libraries and updates. Although the project will focus on only one microcontroller, the options listed here may be reconsidered again in a future revision of the EMG Module.

Considering all the presented options, the family chosen for the development of EMG acquisition module is STM32, which offers reasonable performance, support across the entire range of microcontrollers and two C/C++ frameworks that facilitate firmware development.

STM32 splits up into several sub-families, depending on their characteristics and power optimization. For the first stage of prototyping, a `Nucleo-32` Board is used. The module itself does not require a large number of I/O pins, and, for a wearable device, microcontroller should have a small footprint.

The specific part used in the project is `STM32L432KC`[7], whose key features are:

- ARM Cortex M4 with floating point unit
- 256 KB of flash memory, expandable through QSPI interface
- 64 KB of RAM
- 26 I/O
- 2 × USART, 1 x SPI, 1 x USB 1.1
- DMA controller

2.4 Bluetooth Module

Bluetooth by itself is a complex wireless communication protocol, with multiple layers. Although simpler transceivers can be used with microcontroller as well, the final purpose of the project is to have a device capable of sending the acquired data to a computer that can display it under the form of multiple plots.

As listed above, there are many SoCs on the market that integrate both programmable microcontroller and wireless capabilities. As a starting point, a Bluetooth transceiver that is easy to configure and capable of offering a simple communication with a PC will be enough.

Considering these aspects, the module chosen is `HC-05`, from Itead Studio. HC-05 is well known in combination with Arduino Systems, thanks to its ease of operation with the existent libraries. The module has the following characteristics:

- transparent UART Communication between microcontroller and other Bluetooth device;
- Bluetooth Master/Slave capabilities;
- UART Baud Rate up to 921600 bits per second;
- encryption and authentication by default;
- configurable by AT commands through UART;

A typical Bluetooth module on breakout board is shown in Figure 2.8

HC-05 implements Bluetooth 2.0 + EDR (Enhanced Data Rate) and it is available in many electronic parts stores. It can be mounted on breakout module with LEDs and voltage regulator, or as standalone module[8].

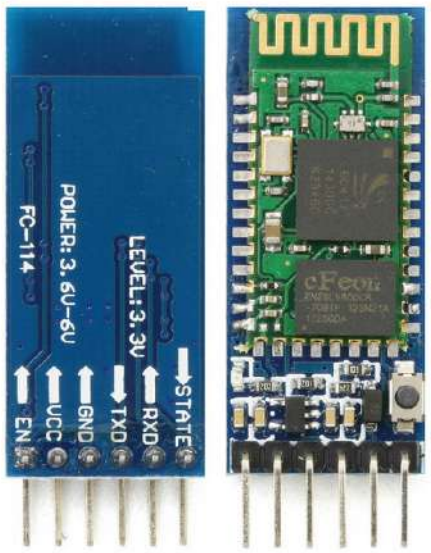


Figure 2.8: HC-05 on Breakout Board.

2.5 Power Supply

The EMG module will be powered by a Li-Ion 3.7V battery. A fully charged Li-Ion battery can have a voltage of maximum 4.2V.

Due to the fact the HC-05 module needs more than 3.5V on the supply line, it can be directly connected to Li-Ion voltage. The microcontroller and the ADC are 3.3V systems, meaning that a voltage regulator must be used. The noise on the power supply must be kept at low values, reason for which a linear voltage regulator will be used. The chosen LDO regulator is *MCP1801*[9], from Microchip, which has a fixed output of 3.3V.

The final aspect worth mentioning here is the battery charging circuit. The acquisition module will have the possibility of charging by micro-USB connection. The USB standard uses a 5V power supply, meaning that a dedicated charger circuit must be included. Also, it is considered best practice to allow the microcontroller to control the charging operation of the circuit, when needed. The choice in this case is `MCP73830L` from Microchip [10]. It is a single-cell Li-Ion charger with programmable current limit, a status LED indicator and an On/Off charge control input, available in a compact TDFN-6 package.

2.6 The complete block diagram

Figure 2.9 describes the summary of the EMG acquisition chain, starting from the analog raw EMG output of the sensors to the plot of the time domain signal.

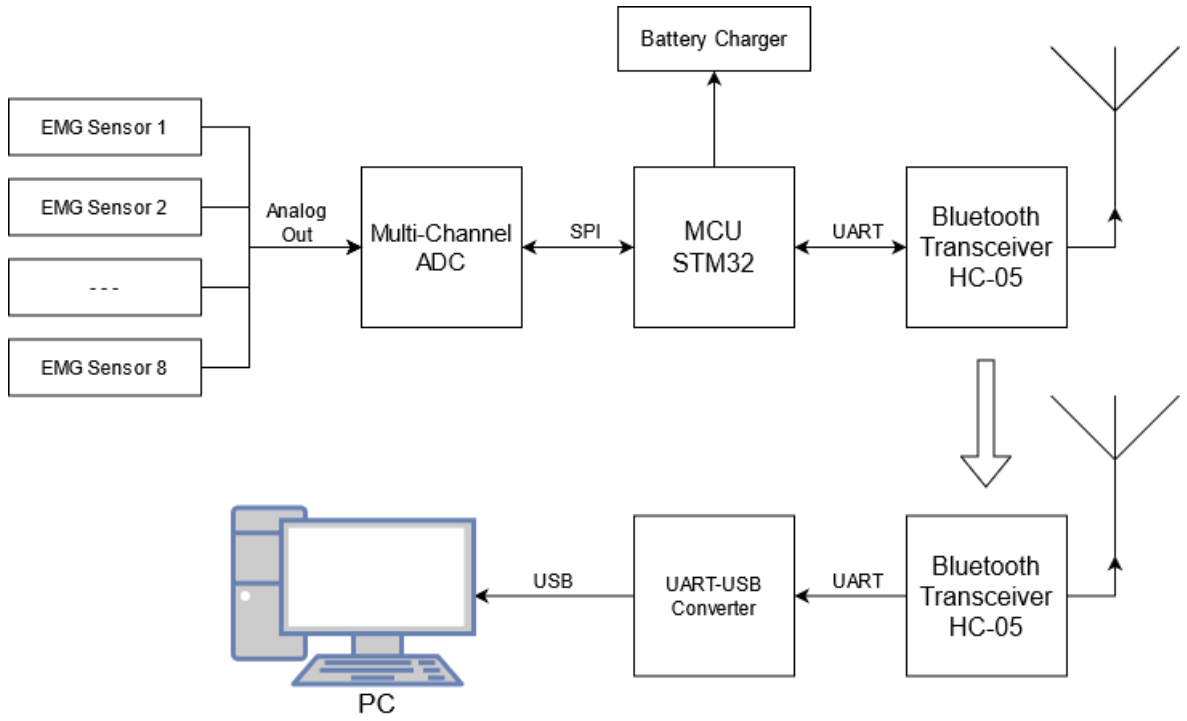


Figure 2.9: HC-05 on Breakout Board.

The EMG acquisition interface is made out of two major blocks:

1. The acquisition module itself
2. The Bluetooth adapter module

The acquisition module is built around a few main components: sensors, multi-channel ADC, microcontroller and transceiver. In this case, the data acquisition starts from left to right. The EMG sensors give an amplified raw signal, which is then converted into numerical form by ADC. The samples are further transmitted to the microcontroller in order to be encoded for UART transmission. Then the samples are transmitted over UART to the Bluetooth module, which further encapsulates and transmits them to the adapter module.

The adapter module is made out of two components connected together: a Bluetooth HC-05 module and a UART-USB converter. The two Bluetooth modules are configured so that they form a permanent, fixed wireless link, in order to decrease latency. As mentioned before, HC-05 communicates through UART, and since the Serial Port on most user machines is non-existant, a USB-UART converter such as CH340G (Figure 2.10) is used.

The complete Setup of the EMG Bracelet soldered on custom PCB, together with the Adapter Circuit, is shown in Figure 2.12.

Both schematic and the PCB of the project were developed in KiCAD, a free and Open-Source design suite that allows projects of any size to be developed without limitations. The manufacturer of the Circuit Boards is OSHPark.

The PCB was manufactured on FR4 substrate, having a number of 2 conductive layers with 2 oz. copper. The Copper surface has ENIG finish. For the passive components, it was chosen the dimension 0805 (or 2012 in metric) as the components are small enough to achieve a greater component density, yet large enough to be easily soldered by hand. The overall dimensions of the board are 62×22 mm, with an overall thickness of 6 mm (excluding the pin headers). The complete schematic of the module is listed in Appendix A.

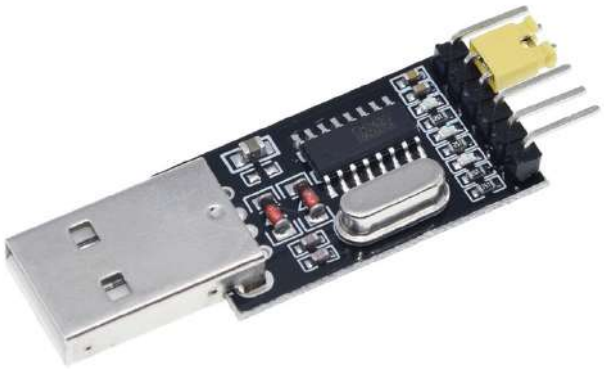


Figure 2.10: CH340G UART-USB Converter

(a) Bluetooth adapter board

(b) The EMG Module (without battery

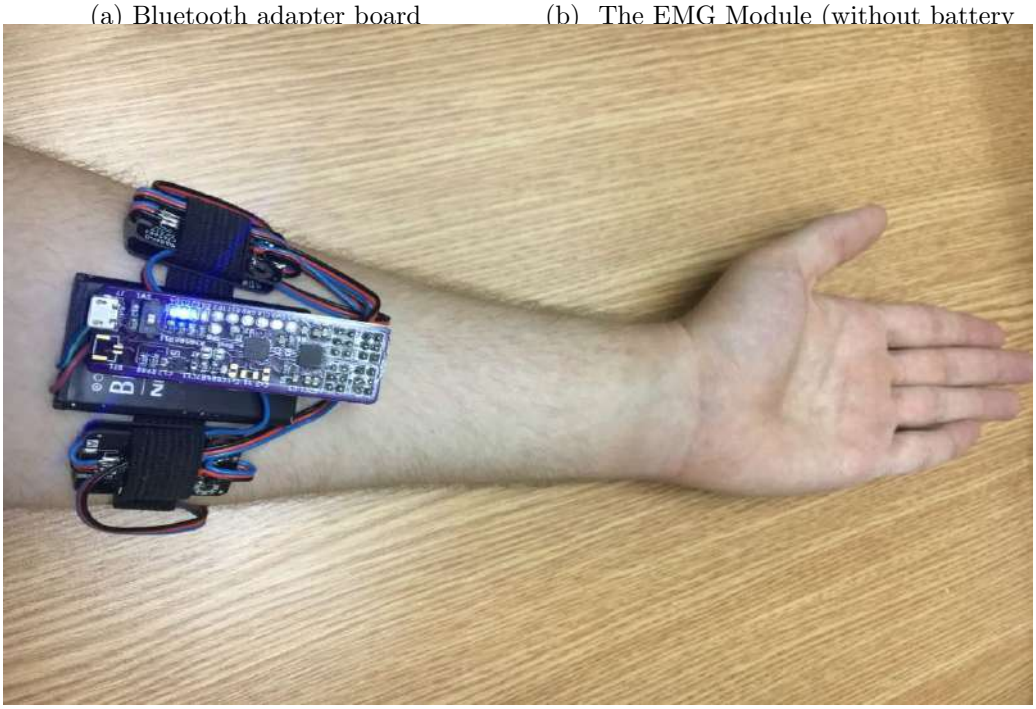


Figure 2.12: The Complete Prototype.

Chapter 3
Firmware and Software

This chapter focuses on two important sides regarding the programmable part: the firmware on the microcontroller, which acquire digital samples and converts them into a format suitable for Bluetooth transmission, and the software, which takes all the samples from Bluetooth and displays them as scrolling plots. The software part will run on an x86/x64 platform with Windows Operating System.

3.1 Firmware

STM32 microcontrollers are based on ARM Cortex architecture, which is optimized for C/C++ programming. Regarding the firmware development, ST provides the users with two frameworks created for 32-bit family:

1. Hardware Abstraction Layer (HAL);
2. Low Level Framework (LL);

`HAL` is best known for the high-level functions that allow for quick and easy prototyping with the microcontroller's advanced core features and peripherals. Only basic knowledge about the device and its peripherals is needed in order to be able to develop a firmware for it. HAL is also a practical introduction to the 32 bit microcontrollers. Just like other higher-level frameworks, HAL utilizes more resources and it is not recommended for applications where performance and memory efficiency are critical.

`LL` drivers are the complementary part of HAL, since they operate at a level much closer to the actual hardware. As expected, they require more knowledge about the hardware's associated registers. However, the advantage of using LL over HAL is the enhanced flexibility and efficiency of the firmware. LL drivers are basically register access functions that are labeled in a much more user friendly manner, removing the need of knowing absolutely all the details about the involved registers, such as bit positions, all configuration values and addresses. Low Level drivers also allow for direct register manipulation without any issues.

Development of the firmware for the EMG module is done by using CubeIDE software, whose most useful feature is CubeMX code generator, which allows for easy peripheral and functions setup. The remaining code will be written by LL drivers.

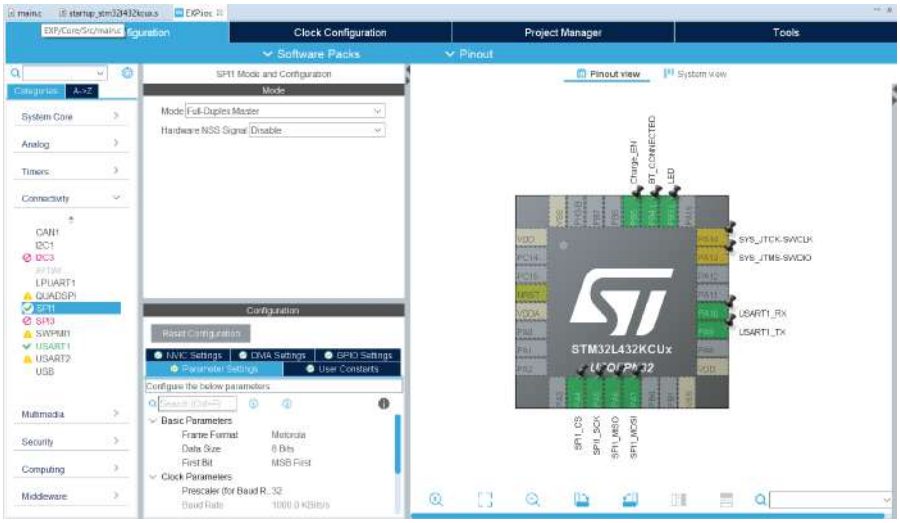


Figure 3.1: CubeMX code configuration tool.

3.1.1 STM32 Operation

The basic flowchart of the code is presented above. The first step of code execution is the initialization of STM32's clock for core and peripherals, followed by the initialization of the peripherals.

The second step involved is waiting shortly after startup so that ADC has time to initialize its function, then the MCU will send the configuration bytes. Bluetooth does not need initialization here since it is preconfigured separately with commands presented in Appendix 4.

After initialization, it comes the part where it is checked if the Bluetooth module is paired with a computer. If yes, proceed to next step, otherwise wait until connection is established.

The next step is the sampling itself, where the microcontroller continuously polls the ADC for finished conversions, then it creates the packet that will be further transmitted through UART. Flowchart from Figure 3.2 shows the microcontroller execution.

The code of the firmware is found in Appendix B.

3.1.2 ADC Operation

As presented in the previous chapter, the `AD7124-8` offers a significant configuration flexibility, which makes it suitable for a wide range of low-bandwidth signals applications. It is important to know which ones are the features of interest and how should they be configured in order to obtain the desired results.

The EMG Module configuration uses the following settings:

- full power mode, for high sampling rate
- internal 2.5V voltage reference
- programmable gain of 1
- no filtering, for lower latency
- up to 8 channels activated, all of them sharing the same filter and gain settings.

To begin with, the communication with ADC is done by means of SPI interface. The ADC is a half-duplex device that works in SPI Mode 3, which means that SPI clock is idle HIGH

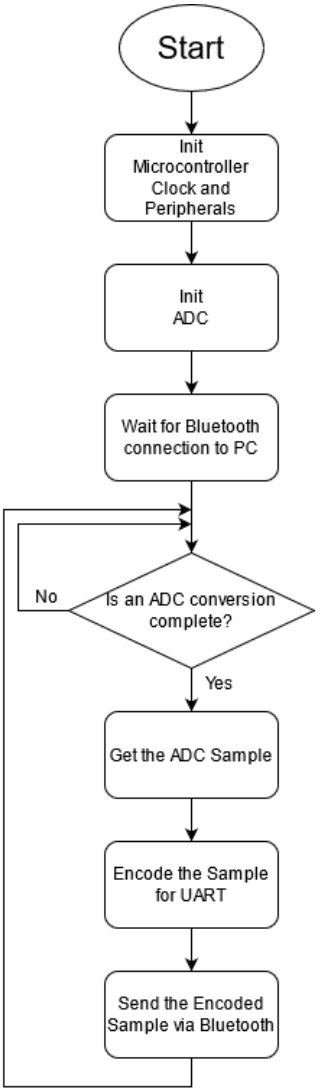


Figure 3.2: STM32 code flowchart.

and the data is clocked out on the second edge. The maximum serial clock frequency that can be applied to the serial interface is about 1 MHz. There are 4 pins to be mentioned in this situation:

1. Chip Select - CS ;
2. Serial Data In - DIN (MOSI);
3. Serial Data Out - DOUT (MISO);
4. Serial Clock - SCK;

AD7124-8 is a device whose operation is controlled by means of registers implemented as Static RAM. Most of them are readable and writable, while others are only readable or writable.

The datasheet includes a map of all the registers used by ADC and which are accessible to the user, as well as their corresponding addresses(Figure 3.3).

Table 64. Register Summary

Addr.	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset	RW
0x00	COMMS	$\overline{WEN}$	R/$\overline{W}$	RS[5:0]						0x00	W
0x00	Status	$\overline{RDY}$	ERROR_FLAG	0	POR_FLAG	CH_ACTIVE				0x00	R
0x01	ADC_CONTROL	0			$\overline{DOUT\_RDY\_DEL}$	CONT_READ	DATA_STATUS	CS_EN	REF_EN	0x0000	RW
		POWER_MODE		Mode				CLK_SEL			
0x02	Data	Data [23:16]								0x000000	R
		Data [15:8]									
		Data [7:0]									
0x03	IO_CONTROL_1	GPIO_DAT4	GPIO_DAT3	GPIO_DAT2	GPIO_DAT1	GPIO_CTRL4	GPIO_CTRL3	GPIO_CTRL2	GPIO_CTRL1	0x000000	RW
		PDSW	0	IOUT1			IOUT0				
		IOUT1_CH				IOUT0_CH					
0x04	IO_CONTROL_2	VBIAS15	VBIAS14	VBIAS13	VBIAS12	VBIAS11	VBIAS10	VBIAS9	VBIAS8	0x0000	RW
		VBIAS7	VBIAS6	VBIAS5	VBIAS4	VBIAS3	VBIAS2	VBIAS1	VBIAS0		
0x05	ID	DEVICE_ID				SILICON_REVISION				0x14/0x16	R
0x06	Error	0				LDO_CAP_ERR	ADC_CAL_ERR	ADC_CONV_ERR	ADC_SAT_ERR	0x000000	R
		AINP_OV_ERR	AINP_UV_ERR	AINM_OV_ERR	AINM_UV_ERR	REF_DET_ERR	0	DLDO_PSM_ERR	0		
		ALDO_PSM_ERR	SPI_IGNORE_ERR	SPI_SCLK_CNT_ERR	SPI_READ_ERR	SPI_WRITE_ERR	SPI_CRC_ERR	MM_CRC_ERR	ROM_CRC_ERR		
0x07	ERROR_EN	0	MCLK_CNT_EN	LDO_CAP_CHK_TEST_EN	LDO_CAP_CHK		ADC_CAL_ERR_EN	ADC_CONV_ERR_EN	ADC_SAT_ERR_EN	0x000040	RW
		AINP_OV_ERR_EN	AINP_UV_ERR_EN	AINM_OV_ERR_EN	AINM_UV_ERR_EN	REF_DET_ERR_EN	DLDO_PSM_TRIP_TEST_EN	DLDO_PSM_ERR_EN	ALDO_PSM_TRIP_TEST_EN		
		ALDO_PSM_ERR_EN	SPI_IGNORE_ERR_EN	SPI_SCLK_CNT_ERR_EN	SPI_READ_ERR_EN	SPI_WRITE_ERR_EN	SPI_CRC_ERR_EN	MM_CRC_ERR_EN	ROM_CRC_ERR_EN		
0x08	MCLK_COUNT	MCLK_COUNT								0x00	R
0x09 to 0x18	CHANNEL_0 to CHANNEL_15	Enable	Setup			0		AINP[4:3]		0x8001[1]	RW
		AINP[2:0]			AINM[4:0]						
0x19 to 0x20	CONFIG_0 to CONFIG_7	0				Bipolar	Burnout		REF_BUFP	0x0860	RW
		REF_BUFM	AIN_BUFP	AIN_BUFM	REF_SEL		PGA				
0x21 to 0x28	FILTER_0 to FILTER_7	Filter			REJ60	POST_FILTER			SINGLE_CYCLE	0x060180	RW
		0				FS[10:8]					
		FS[7:0]									
0x29 to 0x30	OFFSET_0 to OFFSET_7	Offset [23:16]								0x800000	RW
		Offset [15:8]									
		Offset [7:0]									
0x31 to 0x38	GAIN_0 to GAIN_7	Gain [23:16]								0x5XXXXX	RW
		Gain [15:8]									
		Gain [7:0]									

[1] CHANNEL_0 is reset to 0x8001. All other channels are reset to 0x0001.

Figure 3.3: AD7124-8 register map. Source: Analog Devices AD7124-8 datasheet

The Register Map can be accessed only by means of a "Communications Register"(Figure 3.4). It is write-only and it has 3 major fields:

- WEN - Write ENable: When a zero is written, the next 7 bits will be clocked in;
- R/W - Read/Write: It specifies if a read or write operation will be performed on the ADC registers;
- RS[5:0] - Register Select: The address of the requested register is contained in this field.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$\overline{WEN}$ (0)	R/$\overline{W}$ (0)	RS[5:0] (0)					

Figure 3.4: Communication Register.

The registers that need to be modified in order to achieve the desired settings are:

- Channel Registers (Figure 3.5)
- Filter Registers (Figure 3.6)
- Configuration Registers
- ADC Control Register (Figure 3.8)

The device has all 8 channels connected to a CrossBar Network to the Sigma-Delta Modulator, enhancing the flexibility of input configurations. There are 16 input pins, starting with AIN0 (Analog INput 0), up to AIN15, which justifies the existence of 16 Channel Registers. Each channel has 2 terminals called AINP (AIN Positive) and AINM (AIN Negative), which can be connected to any of the 16 pins mentioned, or to a fixed potential such as GND.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable (1)	Setup (0)			(0)	0 (0)	AINP[4:3] (00)	
AINP[2:0] (000)			AINM[4:0] (00001)				

Figure 3.5: Channel Register.Source:Analog Devices

The register presented above has a field for Enable, another filed specifying one of the 8 possible configuration settings, as well as selecting which one is AINP and which one is AINM.

Next step in configuration of the ADC is writing to the configuration register. There are 8 such registers, but since all the EMG sensors involved are the same, one configuration for all active channels is enough. Here are the most important settings:

- Bipolar or Unipolar Conversion;
- Reference Voltage Selection;
- Programmable Gain;

It will be selected Unipolar Conversion on all channels, with Internal 2.5V reference and Programmable Gain of 1.

Filter registers are the ones that determine the type of filter applied, as well as the conversion rate. Filter word will be set to 1, while other bits in the registers will be set to zero to achieve a higher sampling rate.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Filter (0)			REJ60(0)		POST_FILTER(0)		SINGLE_CYCLE(0)
0(0)					FS[10:8](0)		
FS[7:0](0)							

Figure 3.6: Filter Register. Source:Analog Devices

The last register to be configured is "ADC Control". It enables the most important functions of the device. The functions that can be selected are:

- Operation Mode;
- Continuous Read Enable;
- Data Status Output;
- Power Mode.

The operation mode for the ADC will be Continuous conversion. The device will convert samples with a steady rate, as long as it is powered.

Continuous Read Enable is not required for the moment, but it can be used in a further revision of the project. It basically allows reading of a sample just by setting CS LOW and apllying the required number of SPI SCK pulses, without any prior read command. The only disadvantage is that available data can be read only once until the next conversion result.

Data Status Output is a useful feature for multi-channel conversion, as the Status Register contains the number of the converted channel and it will be sent together with the conversion result when a Read Command is issued.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$\overline{\text{RDY}}$ (0)	ERROR_FLAG (0)	0 (0)	POR_FLAG (0)	CH_ACTIVE (0)			

Figure 3.7: Status Register.Source:Analog Devices

The last option determines the conversion rate, as well as the power consumption of the converter. There are three power modes: Full Power, Middle Power and Low Power. The first one will be selected since the conversion rate is more important for the EMG Module.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0 (0)	0 (0)	0 (0)	DOUT_$\overline{\text{RDY}}$_DEL (0)	CONT_READ (0)	DATA_STATUS (0)	$\overline{\text{CS}}$_EN (0)	REF_EN (0)
POWER_MODE (0)		Mode (0)				CLK_SEL (0)	

Figure 3.8: ADC Control Register.Source:Analog Devices

Once the AD7124 has been configured, the only details to be mentioned in its functioning are verifying when an A/D conversion is finished and the effective number of bits. There are two possible methods:

1. Read the Status Register's RDY Flag. Reading the Status Register is pretty straight forward. A specific byte is sent to the Communications register, and the ADC returns the Status Register content. A mask with the requested byte is applied to see if a conversion is ready. The only drawback of this method is that it is time inefficient, because there are a total of 16 Clock Pulses applied for the polling alone, not to mention the effective transfer of conversion result, which requires another register read.

2. Poll for the DOUT/RDY pin transition. DOUT pin has a double role in the communication with the microcontroller. It is used to output the content of the requested ADC register, as well as indicating the end of a conversion. High emphasis is put on this feature, as it significantly increases the output data transfer rate on the SPI interface. When the CS Pin is Logic LOW, the DOUT line stays HIGH during a conversion, then it is set to logic LOW to mark the end of conversion. This method is much faster compared to the Status register Polling, and it is the recommended approach if Continuous Reading is enabled.

The conversion result is composed of 3 bytes, transmitted from most significant byte to the least significant byte, as well as from most significant bit to least significant bit. The Read Command, which is 1 byte, plus the conversion result, 3 bytes, plus the Status Register Contents appended after every second, this would result 5 bytes/sample.

There is one final detail about the conversion result. The datasheet from Analog Devices specifies what effective resolutions can be achieved with different filter settings and sampling rates. The filter type used is the default one.

Table 9. Effective Resolution (Peak-to-Peak Resolution) vs. Gain and Output Data Rate (Bits), Full Power Mode

Filter Word (Dec.)	Output Data Rate (SPS)	Output Data Rate (Zero Latency Mode) (SPS)	Gain = 1	Gain = 2	Gain = 4	Gain = 8	Gain = 16	Gain = 32	Gain = 64
2047	9.4	2.34	24 (21.7)	24 (21.4)	23.7 (21)	23.1 (20.5)	22.7 (20.2)	22.3 (19.8)	21.6 (19)
1920	10	2.5	24 (21.7)	24 (21.4)	23.7 (21)	23 (20.5)	22.6 (20.1)	22.3 (19.7)	21.6 (19)
960	20	5	23.9 (21.2)	23.5 (20.8)	23.2 (20.4)	22.5 (20)	22.1 (19.5)	21.8 (19.2)	21.1 (18.4)
480	40	10	23.5 (20.7)	23 (20.3)	22.6 (19.8)	22.1 (19.3)	21.7 (18.9)	21.2 (18.6)	20.5 (17.8)
384	50	12.5	23.3 (20.5)	22.9 (20.2)	22.5 (19.6)	21.9 (19.1)	21.5 (18.7)	21.1 (18.5)	20.4 (17.7)
320	60	15	23.2 (20.3)	22.8 (20)	22.4 (19.5)	21.8 (19)	21.4 (18.6)	21 (18.3)	20.2 (17.6)
240	80	20	23 (20)	22.6 (19.7)	22.1 (19.3)	21.6 (18.9)	21.2 (18.5)	20.7 (18.1)	20 (17.3)
120	160	40	22.5 (19.5)	22.1 (19.2)	21.7 (18.9)	21 (18.3)	20.6 (18)	20.1 (17.5)	19.5 (16.9)
60	320	80	22 (19.1)	21.6 (18.6)	21.2 (18.2)	20.6 (17.8)	20.2 (17.4)	19.7 (17)	19 (16.3)
30	640	160	21.5 (18.5)	21.1 (18.1)	20.7 (17.7)	20.1 (17.2)	19.7 (16.8)	19.2 (16.3)	18.5 (15.6)
15	1280	320	21 (18)	20.5 (17.6)	20.2 (17.2)	19.5 (16.7)	19.1 (16.3)	18.7 (15.9)	18 (15.1)
8	2400	600	20.5 (17.5)	20.1 (17.2)	19.7 (16.7)	19 (16.2)	18.6 (15.7)	18.2 (15.3)	17.5 (14.6)
4	4800	1200	20 (17)	19.5 (16.5)	19 (16)	18.3 (15.6)	17.9 (15.1)	17.5 (14.6)	16.8 (14)
2	9600	2400	19.1 (16)	18.5 (15.3)	18 (15.1)	17.2 (14.5)	16.9 (13.9)	16.5 (13.5)	15.9 (12.7)
1	19,200	4800	16.1 (13.3)	16 (13.2)	15.9 (13)	15.5 (12.7)	15.4 (12.4)	15.1 (12.2)	14.6 (11.5)

Figure 3.9: Resolution (in bits) as function of output data rate (in Sa/s).Source:Analog Devices

Considering the Fig 3.9 from ADC datasheet, it results that conversion rates greater than 4000 Sa/s, at a gain of 1, with default filter enabled, result in an effective peak-to-peak resolution of only 17 bits.

This leads to the conclusion that the third bit from the conversion result will be mostly noise, and if taken into account the fact that the EMG signal is already amplified by the sensor, a resolution greater than 16 bits would not improve significantly the results. In order to save bandwidth, the third bit will not be further encoded in the UART data frame.

From the number of bytes per sample and the number of samples per second, there can be established the optimal SPI operating speed for the ADC. For an optimal sampling rate of 4000 Sa/s, the ADC would give a throughput of 4000 samples $\times$ 4 bytes/channel = 16 KB/s. Translated into bits, this means 16KB/s $\times$ 8 = 128 kbps. Although it appears that a SPI clock speed of 250 kbps would be more than enough, the signal sample passes through SPI transmission, followed by signal encoding, then a UART transmission. The data sent from MCU is then encapsulated in the Bluetooth Protocol stack, which is then sent to the Adapter module, which decodes the radio packet, converts into UART and transmits it through USB to the PC program. All these elements from the chain do increase the overall latency of the acquisition module, without taking into account the latency introduced by the Script running on the host PC.

In order to minimize the latency, both SPI and UART interfaces will use the maximum transmission rates allowed by the involved components. The SPI will have a clock frequency of 1 Mhz.

3.1.3 Sample Encoding

At first glance, it may seem pretty straight-forward to send the received data from ADC directly to the Bluetooth transceiver through UART. There are some issues which need to be addressed, though.

1. First issue is about UART interface and the data received by the host PC. In the communication between an embedded device and a PC, through UART interface are sent alpha-numeric characters that compose the messages that are commonly seen in a terminal emulator. A byte can take 256 values. The ASCII table uses first 128 values for message transmissions. The issue is that the first 32 characters are actually "control characters", which are interpreted differently by the PC, compared to normal characters which are simply displayed. If it is taken into consideration the fact that a 3 byte conversion result can take any value in the range 0-255 for each byte, it results that, in some cases, control characters can be sent instead of normal alpha-numeric symbols, which may lead to unexpected results on the receiving end. When encoding the data, it must be taken into account how it will be decoded by the host program. A simple solution to encode the signal samples is to convert them into plain text hexadecimal numbers. For example, the byte with hexadecimal value 0x27 would be translated into two bits, which hold the alpha-numeric characters '2' and '7'. In case of channel conversion, this would translate into encoding the number into its corresponding character, since it can take value between 1 and 8, so half a byte can be converted into a single character.

2. The second issue is related to the separation of data frames that contain signal samples. Both Bluetooth modules involved in the transmission chain are communicating through UART, which is by default an asynchronous interface. Since the module is sending samples from multiple channels, it is mandatory for the data frame to contain at least the signal sample and the corresponding channel number. In order to delimit a frame, the simplest solution is to add a special character at the end of it. The choice of the delimiter character is dependent on the program running on the PC. The most common character for introducing new strings from a UART console is the new-line character, commonly referred as '\n'.

The two concerning issues were addressed, and now the data frame fields can be defined.

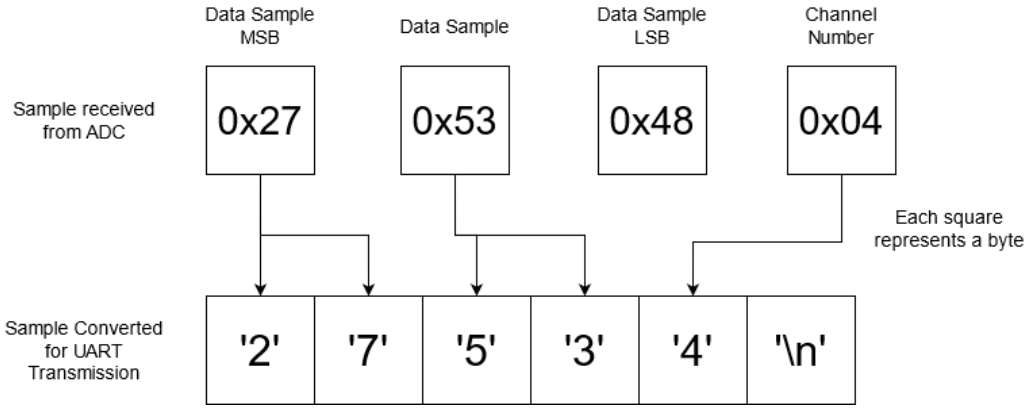


Figure 3.10: SPI to UART data conversion example.

3.1.4 Bluetooth Operation

Unlike the ADC, Bluetooth has a different method of configuration, as all the settings are stored in a flash memory. In order to describe the configuration procedure, the Pinout will be presented in Figure 3.11.

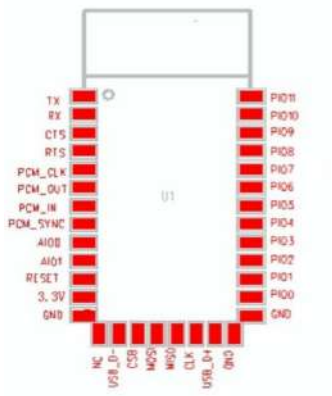


Figure 3.11: HC-05 Pinout.

Although the module presents a high number of pins, only a few of them will be used. These are the Power supply pins, UART TX and UART RX, PIO8 and PIO11.

PIO8 is an output which indicates the working state of the module by the blinking frequency:

- Unpaired - flashes at 2Hz;
- AT Configuration -flashes at about 2 seconds.

PIO11 is input. It will be used during AT configuration.

Entering the AT command mode with a HC-05 involves usage of a USB-UART converter connected to it. The necessary steps are the following:

1. Keep pressed the AT switch/jumper and, then, power on the module. The LED will flash slowly indicating AT mode is ON;
2. Open Terminal Emulator and select the appropriate COM Port (or dev tty, if Linux is used). Select the baud rate of 38400, with one stop bit and no parity check selected;

3. Open the Port.

4. Issue the command “AT ”, followed by the Enter character, or ”\r \n”. Each command that is sent must begin with “AT+” and end with the “\r\n” characters;

5. If the Bluetooth responds with “OK”, the configuration mode is working properly;

6. Send the desired configuration commands;

7. When the configuration is complete, power off the module, or issued the command “AT+RESET \r\n”. The module will reset and the LED will flash fast, meaning that is ready to be paired.

The method applies every time when it is desired to modify some of the Bluetooth settings, and it is the easiest one since both the commands and the responses from the device can be seen. If a command is not recognised, or wrong parameters are set, the error can be corrected immediately.

In the Bluetooth communication, there is one device with the role of ”master”, and one device with the role of ”slave”. A master device is able to initiate a communication with a slave device, and it must know the address of the slave. A slave device can only respond to the request of a master. The adapter module will have the master role assigned, and the EMG module will be slave.

The last detail to be mentioned is the baud rate for UART transmission. HC-05 supports baud rates up to 921600 bps. The maximum value will be chosen in order to minimize the latency as much as possible Configuration Commands are found in Appendix C.

3.2 The Software

The software part is focused on displaying the signals in the time domain, in the form of a rolling plot.

The flowchart of Python Program is described as follows (Figure 3.12).

The program described in the flowchart will start by attempting to connect to the adapter module's corresponding COM Port on Windows, with the specific UART parameters. Then it will request the user to input the number of channels to be visualised in the plot. When the number of channels is provided, the program creates a new plot with one subplot per channel. All subplots start with a zero-line signal, in order to define the line that will be continuously updated.

When the above steps were performed, the program enters in a permanent loop in which a batch of samples is acquired, then the plot is updated with the newly acquired samples.

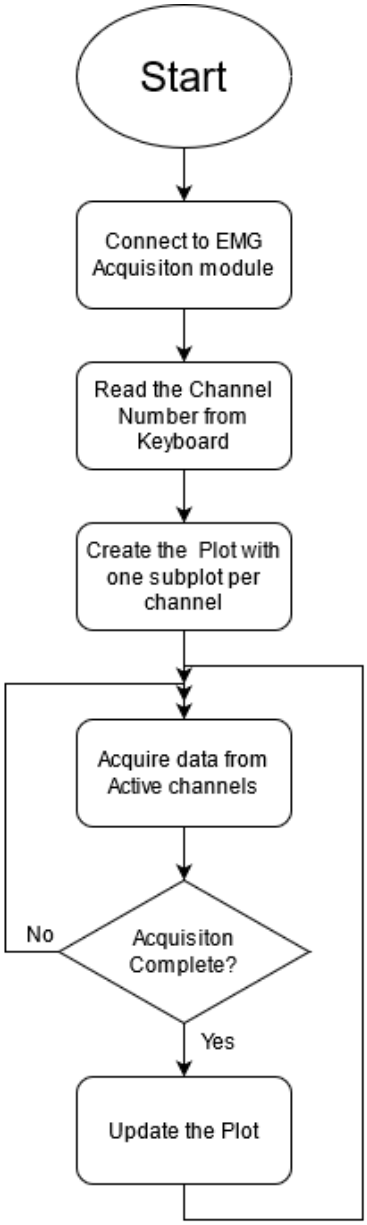


Figure 3.12: Main Program Execution.

The software written on the host PC will be based on Python 3.8 and the associated libraries:

- **Matplotlib** [11] - the library for displaying different types of plots;
- **Numpy** [12] - the numerical library used in Python;
- **PySerial** [13] - the library required to communicate with the virtual COM ports.

3.2.1 Matplotlib

Matplotlib is a well-known library used to create high quality plots, and it is flexible enough to be adapted to the requirements of the project. By default, Matplotlib favors plot quality, rather than plot speed, which means that some workarounds must be performed in order to obtain results closer to a real-time plot. The library does include some animation function, but they will not be used in the project due to the following issues:

1. the functions do update the plot after every single sample collected, which can translate into poor optimization on the running platform. The issue consist in the fact that, for an optimal acquisition on 8 channels, about 4000 samples per second needs to be plotted. Updating the plot by 4000 times per second is computationally intensive, as well as pointless, since a typical display achieves 60 Hz refresh rates. A custom animation function will be used.

2. it is desired to achieve a scrolling plot effect, and most Matplotlib tutorials found in documentation are focused on slower animations, which append a point at a time, while the older points from the graph are not removed. If the scrolling is from right towards left, it results in a visual compression of the graph towards the left of the plot.

The desired effect can be obtained in a much simpler manner, avoiding excessively complex code and achieving a greater plot refresh rate (Figure 3.13).

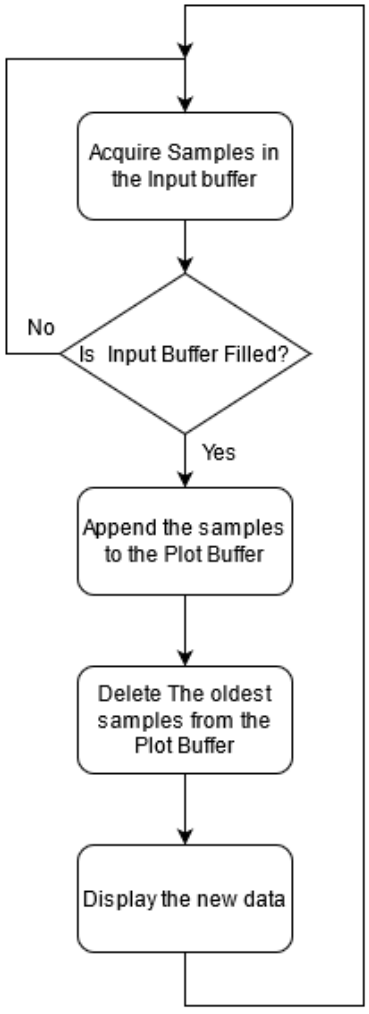


Figure 3.13: Plot refresh method.

The idea is simple: the program needs to acquire a number of samples into a buffer of determined size, and then it must append it to the buffer of the plot (the part which is seen on the screen). In order to obtain the "roll" effect, as seen on a typical digital storage oscilloscope, it is necessary to remove the old samples from the plot buffer. The last step is to plot the new buffer and the code repeats into an infinite loop.

3.2.2 PySerial

PySerial is a Copyright library created by Chris Liechti for serial communication in Python. It can work on Windows, Linux and macOS, but for the purpose of the work it will only be used on Windows.

The library provides the users with easy to use functions, that can establish a software connection with the adapter module, as it appears listed as a "COM" port when plugged into an USB port. The code written with it is simple and easy to understand. Strings of any length can be read from UART if they end with the new-Line character.

One critical subroutine in the code is the acquisition and decoding. In order to plot multiple graphs on a single plot, it is mandatory that the buffers of the channels have exactly the same number of samples to be displayed. It is necessary to create a code that ensures all the activated channels buffers are filled before updating the plot. Basically, every channel will have its own buffer, together with a buffer counter which will be incremented after every received character. When a channel has the buffer completely filled, the program will fill the remaining buffers. When all the channel buffers are complete, the program updates the plot. The next flowchart describes the procedure (Figure 3.14).

Another important aspect is the acquisition reliability. In case of incorrect decoding of a sample, or a sample that is corrupted, it is not desirable for the execution to be stopped by an error. Loss of one sample out of 1000 is not a major issue, but if this occurs, a program without error handling mechanisms would have to be restarted, leading to discontinuous operation.

Python does offer a simple mechanism that can provide the code with a degree of reliability, which is known as "try-except". The interpreter executes the code found in the "try" subroutine until an error occurs. When error state is detected, the execution is redirected to an error subroutine where the "exception" can be handled accordingly. In the case of acquisition subroutine, the error subroutine simply resumes the acquisition, which is continued until all the activated channel buffers are filled.

3.2.3 Numpy

Numpy is the standard numerical library included in any Python version. It is required, as the samples that come from the EMG acquisition module are homogeneous, and all the data plotted is also homogeneous. Numpy allows for more efficient vector manipulation.

The combination of all the techniques presented above into a single Python script represents the program that handles the stream of data from sensors. An example of a running program is shown below (Figure 3.15).

The screenshot presented here shows a typical Matplotlib window, in which multiple plots are stacked in order to efficiently use the screen space. It is the proof that the EMG acquisition chain is complete, from the sensor signal, up to the running program result. From this point, experiments on the prototype EMG module can be performed.

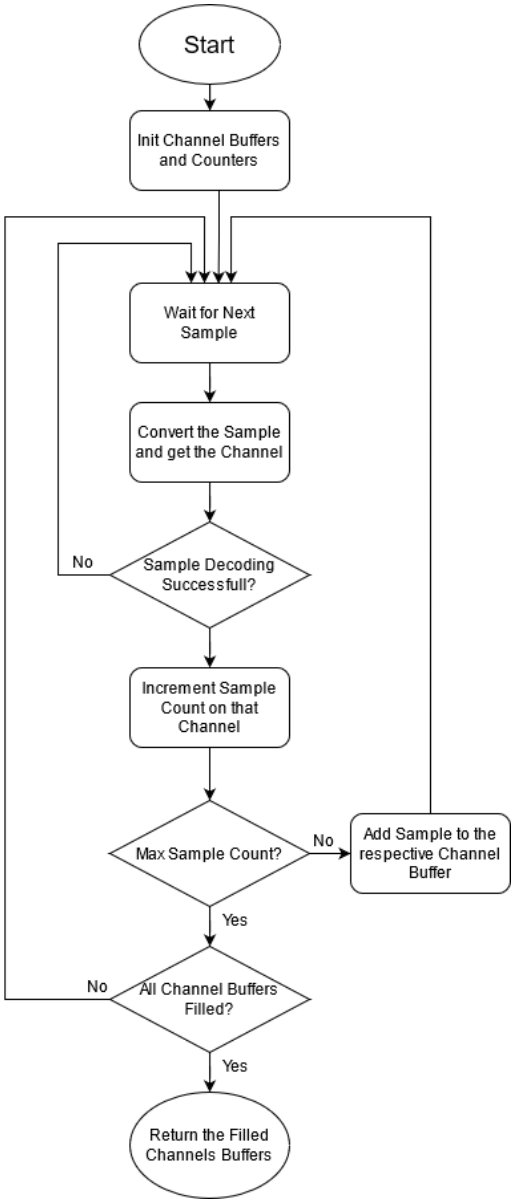


Figure 3.14: Sample acquisition and decode.

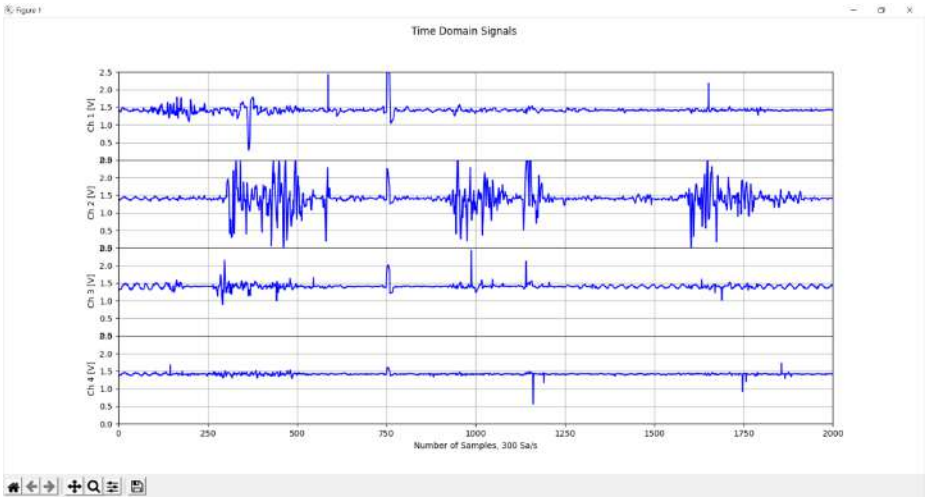


Figure 3.15: Example of signal acquisition.

Chapter 4
Experimental Results

In order to see if the acquired data is correct, a comparison with a reference EMG acquisition interface will be performed. The comparison device used is the OpenBCI from Cyton [3], together a few gelled electrodes for EMG signal acquisition. A few common hand gestures will be performed in order to be able to compare the signals. Also, a comparison between a dry EMG sensor and a gelled EMG sensor will be performed in order to test the performance of each type.

Placement of the sensors on the forearm is crucial, as combined signals from different muscle groups will be recorded, which is the main disadvantage of surface EMG sensors. The highest quality of signal can be achieved in the areas where muscle has the greatest thickness, as the areas closer to the arm joints will have lower peak-to-peak potentials. The experimental results can be influenced to a small degree by the positioning of the sensors, without taking into consideration the type used.

4.1 The Acquisition Modules

4.1.1 The Prototype

The bracelet attached to the custom module has only 4 sensors for the experiment, which are sufficient in order to see some forearm muscle activity. Interpreting and classifying the data is a more complex project for the moment. The 4 sensors are placed at approximately equal distances on the forearm, on the thickest region of the forearm. Placing is important, as it dictates the quality of the measured signal.

The module has 4 Indicator LEDs. Their corresponding functions are:

- D1 - Charging state;
- D2 - Bluetooth Connected;
- D3 - Bluetooth Ready;
- D4 - Status LED;

When Boot-up, the Status LED will flash 4 times, then the Bluetooth Ready LED will indicate EMG Module is ready for pairing with the Adapter. When the Adapter Module is inserted into the USB port of a PC, the Bluetooth module will blink fast a few times. When the

synchronisation is complete, both the EMG module and Adapert will signal that the wireless channel is established and the acquisition can begin.

From this moment, the Multi-Channel Data Plotter program can be run in the interpreter. When run, it will ask the user for the number of channels to be displayed on the plot. The module is set to output samples for 4 channels, so in this situation, any number from 1 to 4 can be written.

The program will automatically adjust the plot sizes, so that the signal can be easily visualized.

4.1.2 The OpenBCI Module

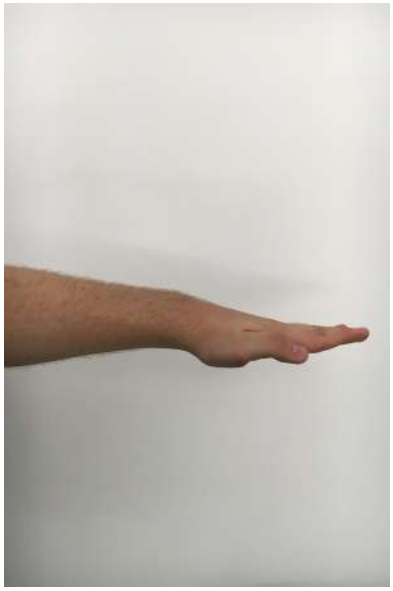
OpenBCI is well known for being an open-source sollution for acquiring different types of biometric signals, such as EEG or EMG signals. The module is used together with a complete acquisition software, which can display the selected channels, as well as interpreting the results and displaying the Fourier Transform.

Figure 4.1: Cyton and the Prototype.

The Figure 4.1 illustrates on the left side the Cyton board and the USB dongle, while in the right is illustrated the prototype custom device.

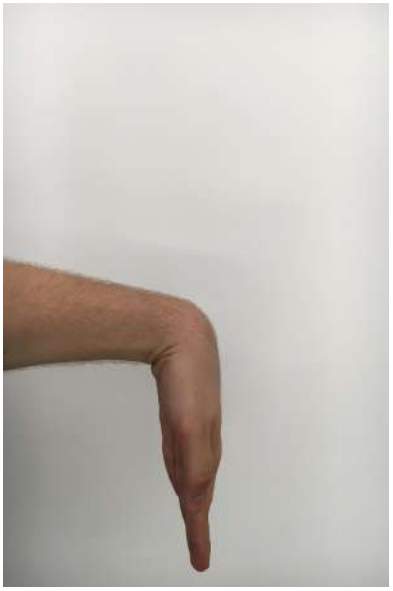
4.2 Hand Gestures

In order to be able to determine if the prototype acquisition system is capable of correctly sampling and displaying the signal, reference signals are needed. These reference signals can be acquired with OpenBCI, as it is a commercially available device, meaning that it was throroughly verified. Once the gestures' reference signals are acquired, the custom EMG module will be tested in the same manner, with electrodes in the same positions. Figure 4.2 shows some of the most common gestures that can be performed in order to test an acquisition system.

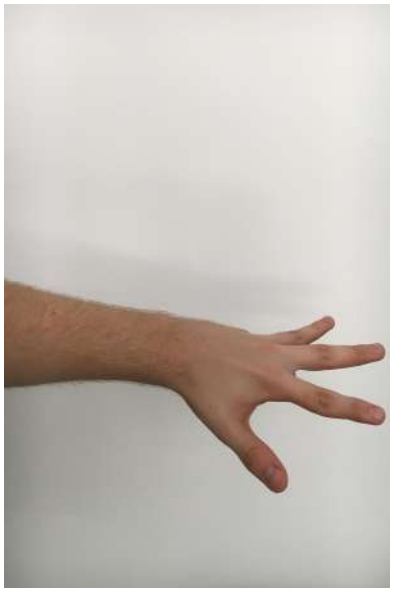


(a) Hand Rest

(b) Wrist Extension

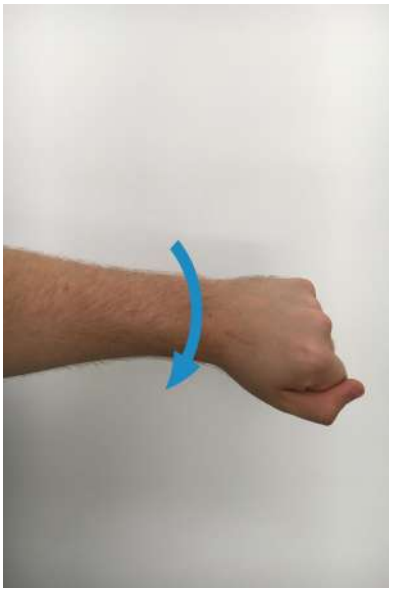


(c) Wrist Flexion

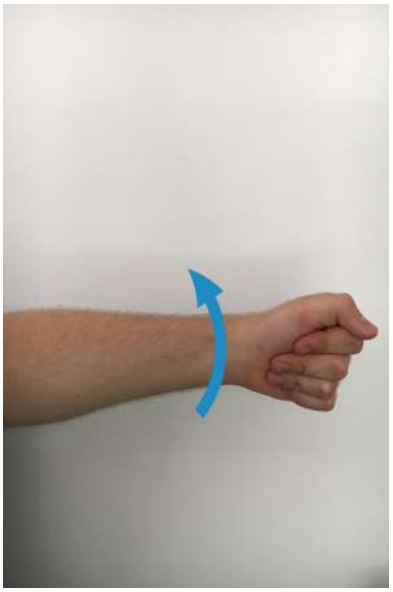


(d) Open Hand

(e) Fist



(f) Pronation



(g) Supination

Figure 4.2: Simple Hand Gestures

4.3 Experimental Results

The testing begins with the measurements of EMG potentials begins by acquiring the reference data with OpenBCI. The test setup is shown below (Figure 4.3).

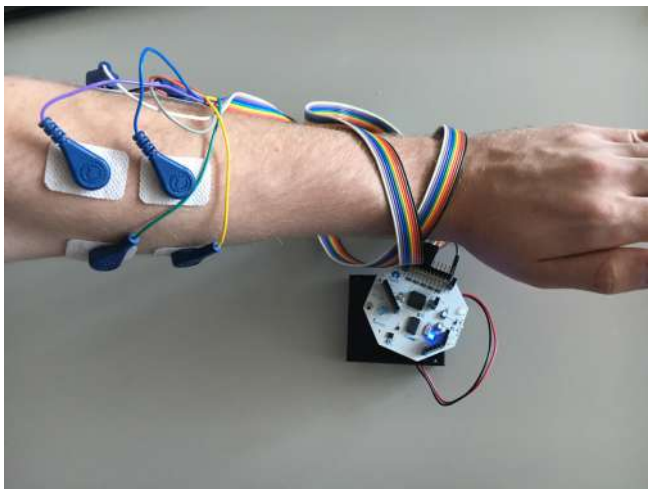


Figure 4.3: OpenBCI Test Setup.

Figure 6.3 illustrates that the electrodes should be placed so that the signal is sampled in the thickest muscle region, like in the case of dry EMG sensors described in Chapter 3. Also, in case of gel electrodes, there is one electrode whose purpose is to offer a reference point in signal measurements. That electrode is Reference Ground and it is placed on the forearm bone end, due to the fact that EMG action potential in that spot is close to zero.

To easily compare the signals acquired from both devices, the acquisition results, which are under the form of screenshot, will be placed together.

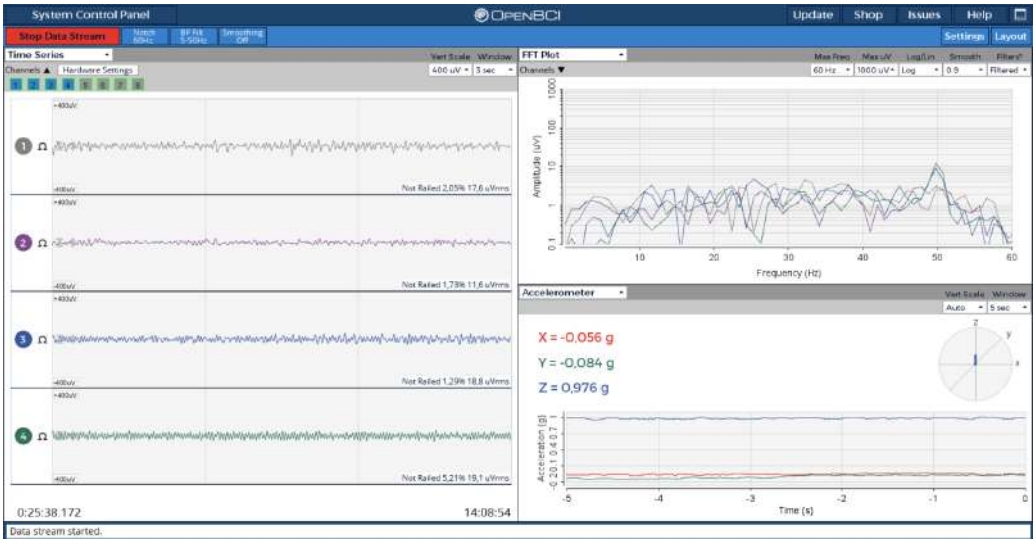


Figure 4.4: Hand Rest (OpenBCI).

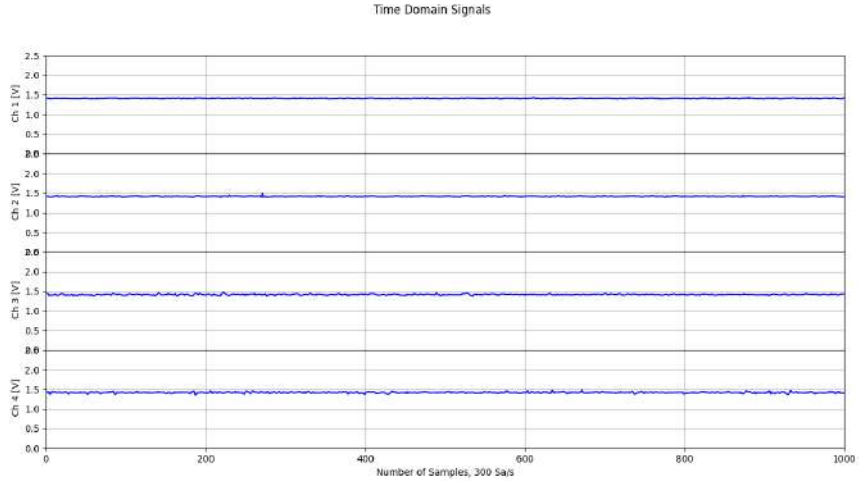


Figure 4.5: Hand Rest (Prototype).

As expected, in Figures 4.4 and 4.5, since the arm is resting, its EMG action potentials will be close to zero.

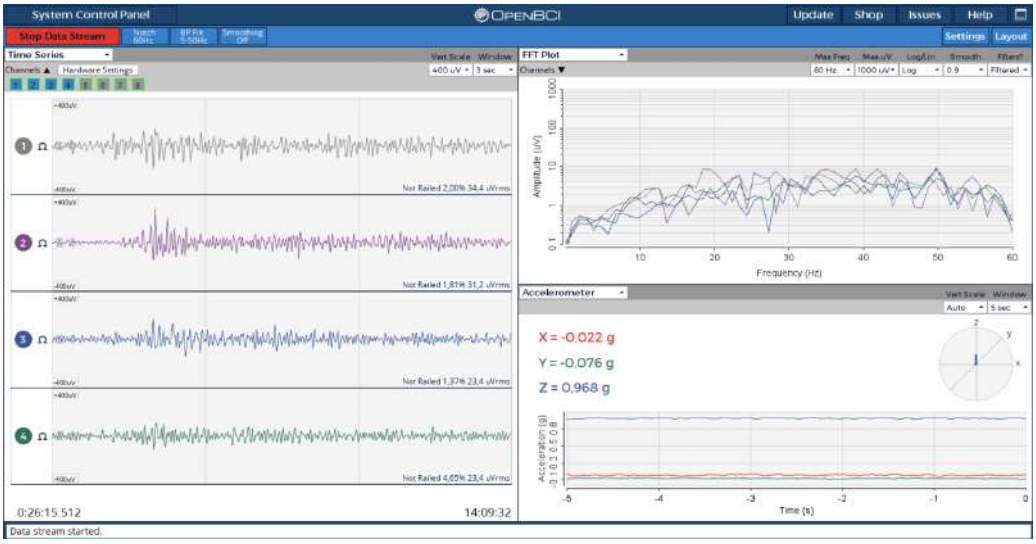


Figure 4.6: Wrist Extension (OpenBCI).

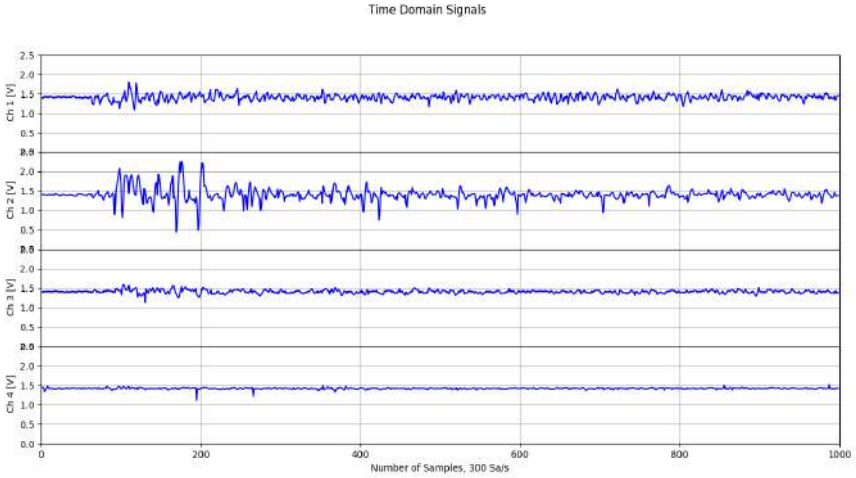


Figure 4.7: Wrist Extension (Prototype).

Figures 4.6 and 4.7 illustrate the Wrist Extension gesture, captured on both devices (consecutive testing was performed).

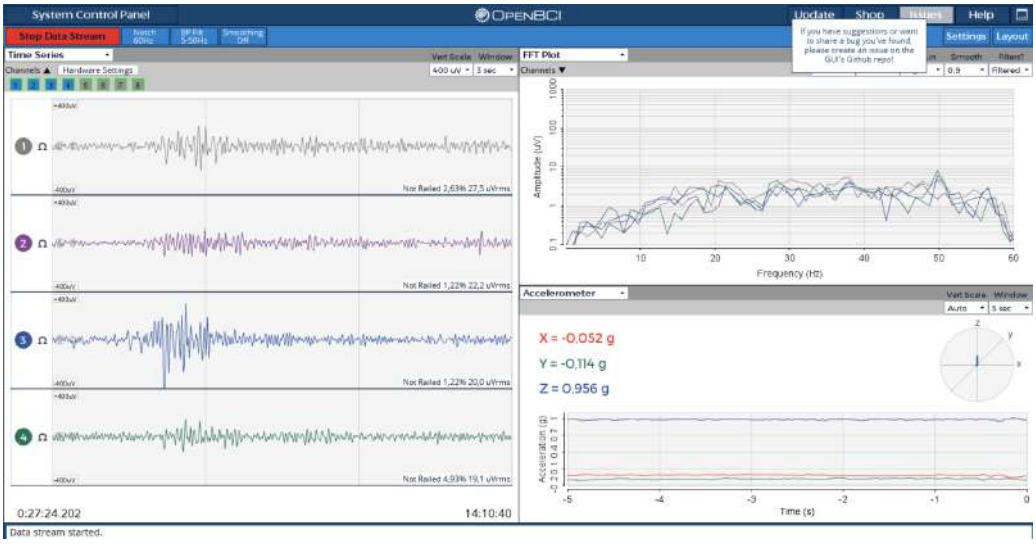


Figure 4.8: Wrist Flexion (OpenBCI).

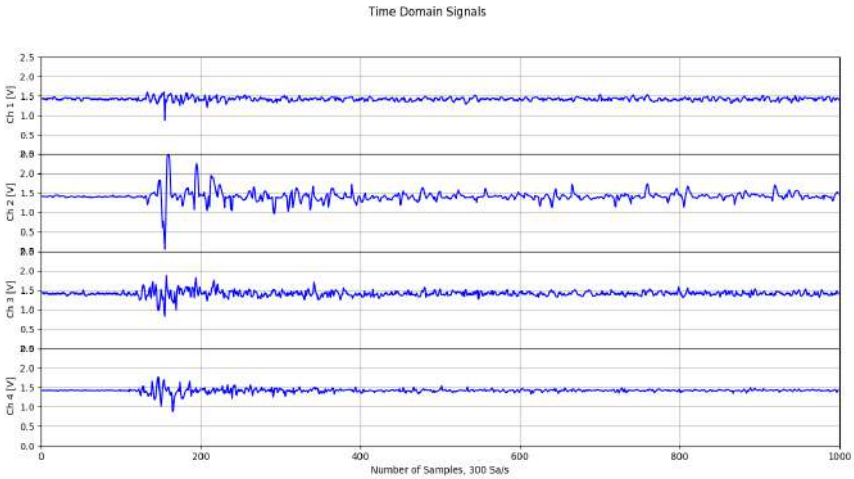


Figure 4.9: Wrist Flexion (Prototype).

The Figures 4.8 and 4.9 illustrate the Wrist Flexion, and as a consequence, muscle groups from the opposite side start to contract in order to perform the movement (they are also aided by gravitational force).

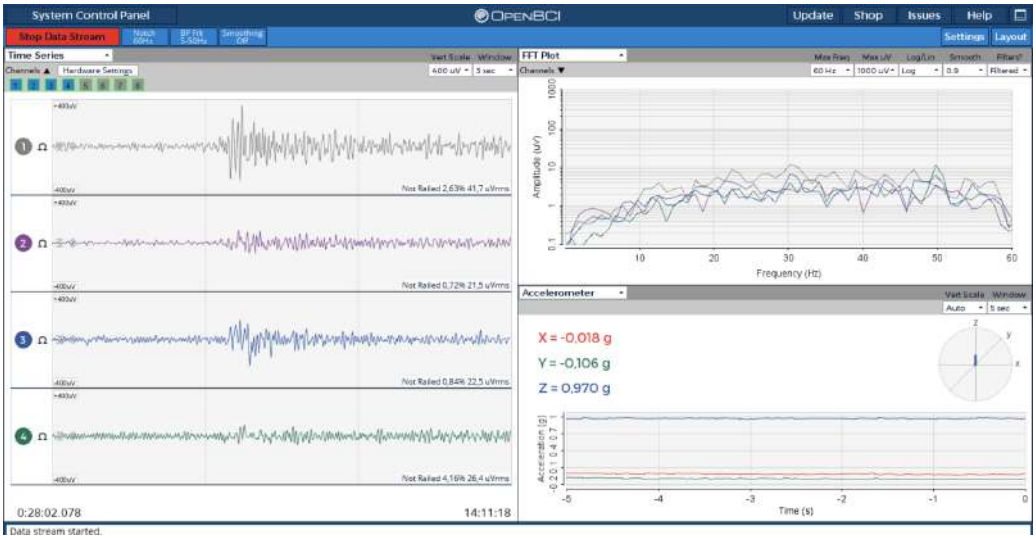


Figure 4.10: Open Hand (OpenBCI).

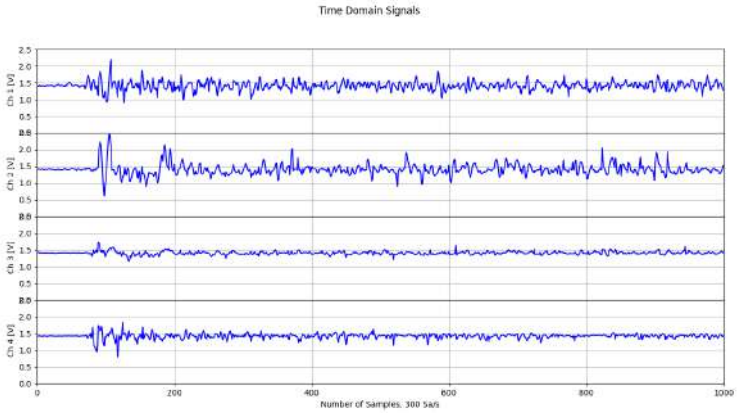


Figure 4.11: Open Hand (Prototype).

Figures 4.10 and 4.11 show how the finger extension gesture is reflected as an EMG signal. As mentioned in Chapter 2, even small movements of the body are a coordinated sequence of larger muscle groups as well.

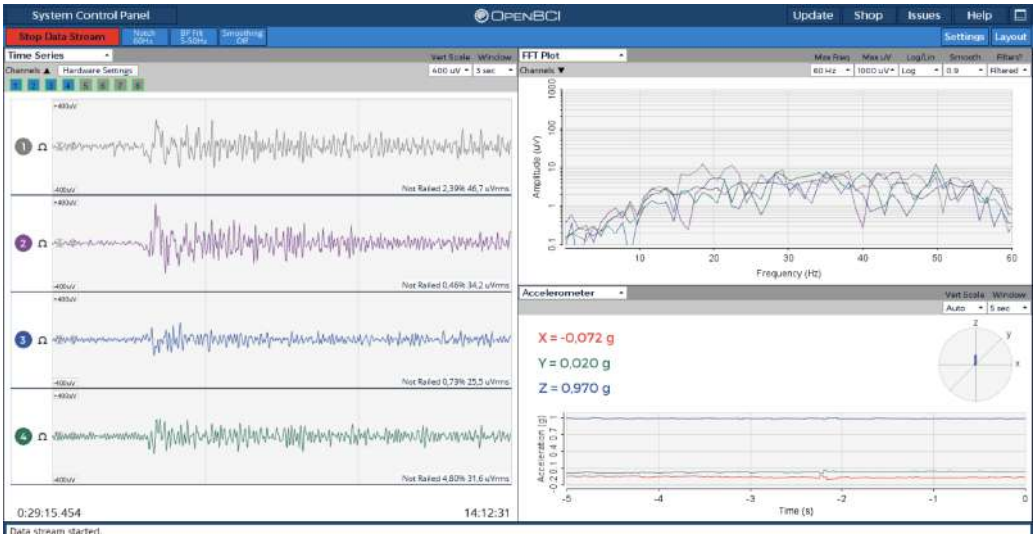


Figure 4.12: Fist (OpenBCI).

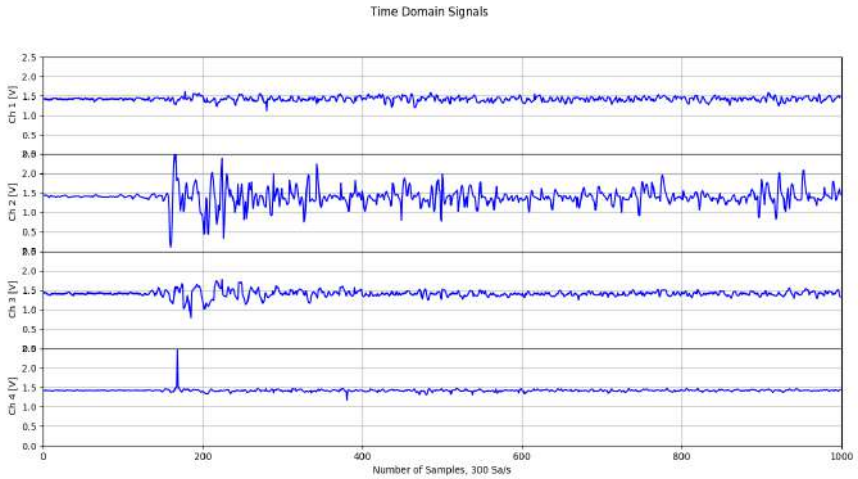


Figure 4.13: Fist (Prototype).

In Figures 4.11 and 4.12, the Fist gesture is shown. As expected, it is a gesture that generates higher EMG signals, as the contraction is more powerful.

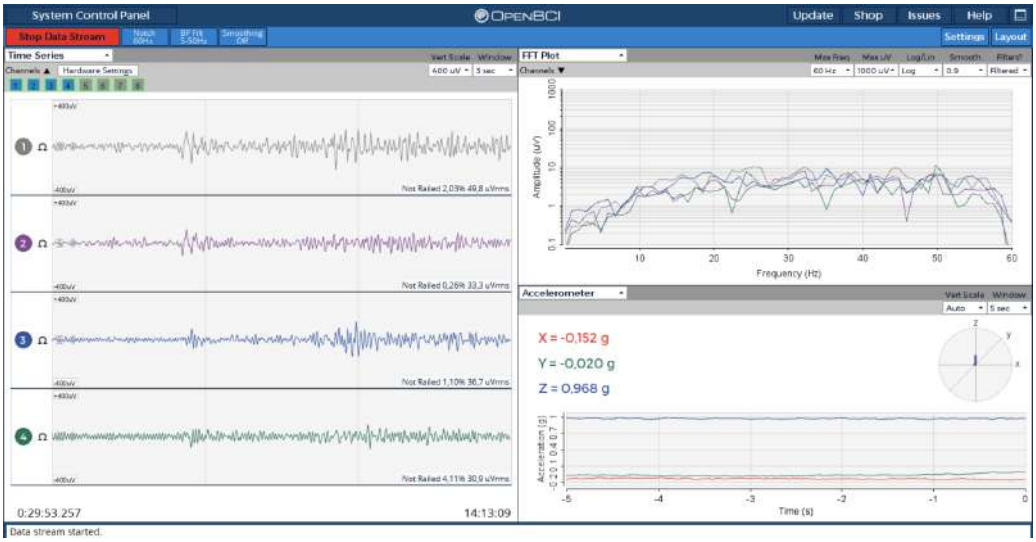


Figure 4.14: Pronation (OpenBCI).

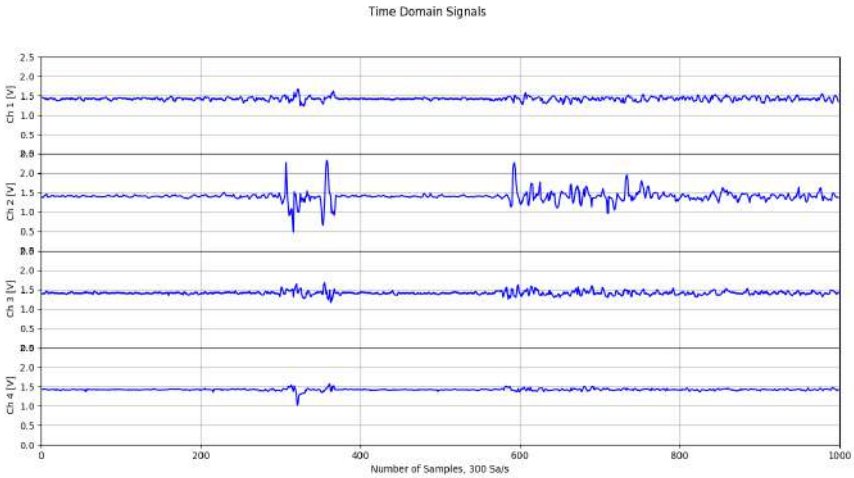


Figure 4.15: Pronation (Prototype).

Figures 4.14 and 4.15 show the pronation gesture.

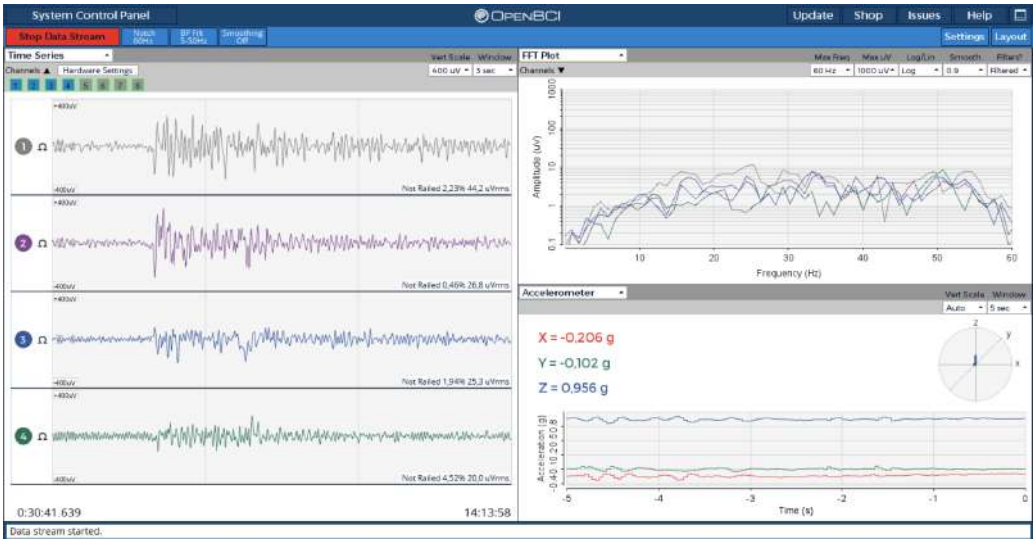


Figure 4.16: Supination (OpenBCI).

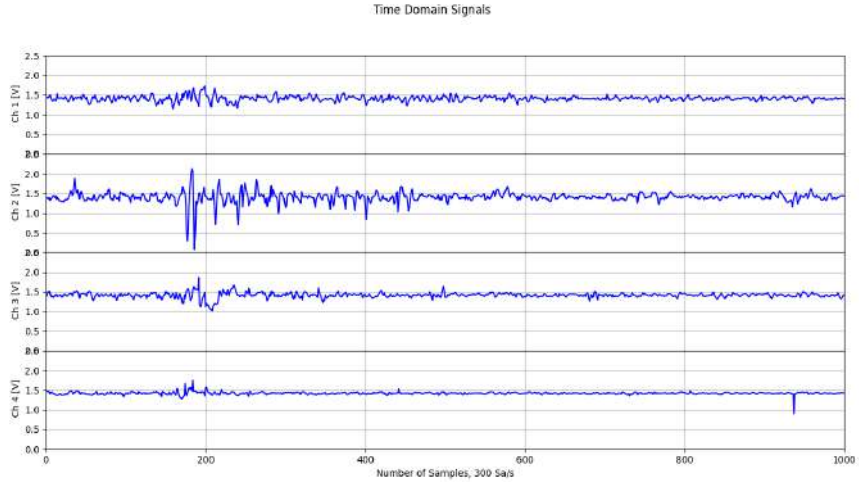


Figure 4.17: Supination (Prototype).

Lastly, Figures 4.16 and 4.17 show the supination gesture.

It is quite hard to replicate exactly the same gestures, as the condition of human body may vary depending on the intermediary activites performed.

Conclusions

Results

As seen in the screenshots from the previous chapters, the signals look pretty similar, but they cannot be called "identical" due to the following considerations:

- It is almost impossible to reproduce exactly the same movement when two acquisition systems are tested sequentially, as the muscles can achieve the same movement in different ways. Muscles cannot be analysed separately from the rest of the body, since what is happening in other body areas can influence directly their activity. Neither brain activity, which is the source of commands for the muscle movement , is the same in different moments of time, as the muscular activity of the body is deeply influenced by the emotions of the person during the measurements. As an example, an anxious person would have a higher average tone, in comparison with a person that feels relaxed. The muscular activity can also be influenced by breathing, as neither breathing activity is always the same.

- Positioning is critical when recording EMG signals, since one of the major disadvantages of this technique is that it records the activity of multiple muscle groups, and it may accumulate noise from the skin surface as well. The signals generated by muscular fascicles can propagate and, thus, the final result is a mixture of signals, whose classification does involve advanced techniques.

- Lastly, there were two types of sensors involved in comparison. Gel sensors and dry surface sensors are different in many aspects. Some of these aspects include the area of contact between the pads, the covered length of the muscle, as well as the amplification of signals, which have a significant influence over the quality of the acquired signal, as well as its characteristics.

For the OpenBCI Cyton module, the sensors are based on solid gel electrodes, with wires that connect directly to the module's ADC. The gel electrodes are thicker, sticky and have a larger contact area with the skin, as well as covering a larger muscle length. Despite the fact that their placement is rather difficult in some cases, they do allow greater flexibility in positioning. Another aspect to consider is that OpenBCI displays the signal as it is, without any amplification, or filtering applied (although filtering can be enabled). Considering the length of the electrode wires, this may result in a much noisier operation in some cases, but the exact raw waveforms can be precisely viewed. Also, OpenBCI has the major advantage of using a more sophisticated ADC, which is specifically designed for biopotential measurements.

For the Prototype EMG module, the sensors are dry electrodes, as they are stainless steel pieces mounted on a PCB. Compared to gel electrodes, their placement is easy, but the flexibility of positioning is limited. One important feature of the used sensors is that each of them comes as a pair of boards, one holding the dry electrode, which would normally connect to a signal amplifier and conditioner circuit, linked through a male-to-male 3.5mm Jack. The output signal comes amplified and also single ended, since an offset of 1.5V is added. This has the advantage that it simplifies the overall complexity of the acquisition module, as well as limiting the noise that is picked up, because the weak EMG signal is amplified close to its source, then, as it travels to the ADC, it has already a greater SNR, which improves the signal quality.

Personal Contributions

The project I worked at is mainly focused on building a simple, complete EMG acquisition system, which does allow for the muscle signals to be viewed in real time on a computer.

For the hardware side, I did illustrate the reasons behind component choices, as well their configuration procedures and settings. The choices were made so that parts would fulfill the requirements of the project, and also made sure they are easy to find and to have support when debugging was necessary. Also, different stages of prototyping were illustrated, in order to make sure the circuit works properly, as PCB designing and manufacturing stage is performed only after a circuit is thoroughly verified.

For the software, I focused on creating a code based on Python, an easy to understand and powerful language, which includes stable and well documented libraries for the functions required by the acquisition chain.

I focused on incorporating all the elements presented above into a system that is comparable, to a certain extent, with an already existent product, built for the same purpose. The advantage of the Prototype EMG module is that it is much easier to equip, allowing for easier and more natural movements of the forearm.

Further Improvements

Further improvements can be brought to the acquisition chain, starting with the actual hardware, continuing with the firmware, up to the PC script that displays the data.

For the acquisition hardware, a significant upgrade would be the replacement of the Bluetooth Transceiver and the microcontroller with a SoC that performs both functions. This modification would decrease the overall dimensions of the PCB, as well as increasing the performance by eliminating the sample encoding technique presented in Chapter 3.

The firmware on the STM32 can be upgraded as well by using DMA transfers with peripherals. It is a technique that allows the CPU to perform both ADC sample acquisition and encoding, while the DMA Controller performs Bluetooth transmission of the previous signal sample. Both the processes would be performed in a pipeline-like stage.

Lastly, the Python script could be rewritten to use multiprocessing technique, as all modern computers are equipped with multi-core processors. One core would perform the update of the time plow while another core would perform signal reception and decoding.

Lastly, and the most useful feature that can be implemented, is the addition of classification methods in order to extract the useful information from the raw EMG signals and transform them into commands for a computer.

Bibliography

[1] Roberto Merletti and PJJEK Di Torino. Standards for reporting emg data. *J Electromyogr Kinesiol*, 9(1):3–4, 1999.

[2] Ana Neacsu, Jean-Christophe Pesquet, and Corneliu Burileanu. Accuracy-robustness trade-off for positively weighted neural networks. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8389–8393. IEEE, 2020.

[3] `https://docs.openbci.com/docs/Welcome.html`, 2019.

[4] Jeffrey R Cram and Glenn S Kasman. The basics of surface electromyography. *to Surface Electromyography*, 1998.

[5] Ioana Bădițoiu. Electrofiziologie – digitalizarea și prelucrarea semnalelor emg asociate. 2020.

[6] `https://www.mikroe.com/adc-6-click`, 2019.

[7] `https://www.st.com/en/microcontrollers-microprocessors/stm32l432kc.html`, 2020.

[8] `https://components101.com/wireless/hc-05-bluetooth-module`, 2019.

[9] `https://www.microchip.com/wwwproducts/en/en530836?utm_source=MicroSolutions&utm_medium=Link&utm_term=FY18Q2&utm_content=APID&utm_campaign=Article`, 2019.

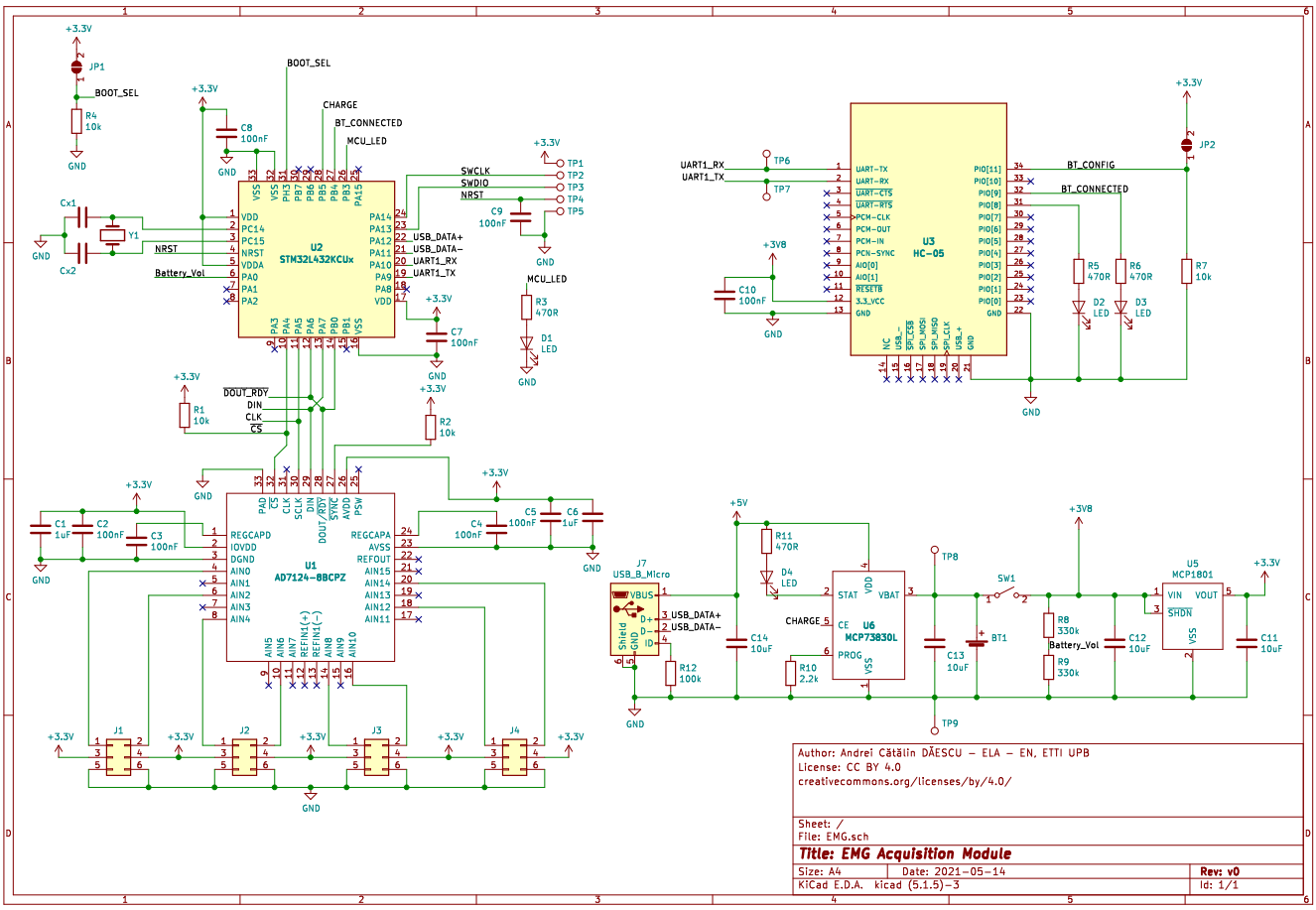
[10] `https://www.microchip.com/wwwproducts/en/MCP73830L`, 2019.

[11] `https://matplotlib.org/`, 2020.

[12] `https://numpy.org/`, 2019.

[13] `https://pythonhosted.org/pyserial/`, 2021.

Appendix A
Complete Electrical Schematic



Appendix B
Microcontroller Firmware

```
1
2  #include "main.h"
3  #include "spi.h"
4  #include "usart.h"
5  #include "gpio.h"
6
7  void SystemClock_Config(void);
8
9  uint8_t ADC_Get_ID();
10
11 uint8_t ADC_Get_Status();
12
13 void ADC_Reset();
14
15 void ADC_Init( uint8_t channel_number );
16
17 void SPI_COMM( uint8_t * tx_buffer, uint8_t * rx_buffer, uint8_t size);
18
19 void UART_TX( uint8_t *tx_buffer, uint8_t size);
20
21
22 int main(void)
23 {
24
25   /* USER CODE BEGIN 1 */
26
27   uint8_t active_channels = 8 ;
28   uint8_t index = 0;
29   uint8_t upper_half, lower_half;
30
31   uint8_t sample[] = {0x42, 0x00, 0x00, 0x00, 0x00};
32   uint8_t packet[] = {'0', 'x', '0', '0', '0','\r', '\n'};
33
34   /* USER CODE END 1 */
35
36   /* MCU Configuration--------------------------------------------------------*/
37
38   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
39
40   LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);
41   LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR);
42
43   NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
44
45   /* Initialize the System Clock with the requested settings */
46
47   SystemClock_Config();
48
49   /* Initialize all configured peripherals */
50   MX_GPIO_Init();
51   MX_SPI1_Init();
52   MX_USART2_UART_Init();
53
```

```
54  /* USER CODE BEGIN 2 */
55
56  LL_SPI_Enable( SPI1);       // Enable SPI1 Peripheral for communication with ADC
57  LL_USART_Enable( USART2);     // Enable USART2 Peripheral for communication with Bluetooth
      transceiver
58
59  ADC_Reset();          // Send the ADC Reset Command
60  LL_mDelay(5);         // wait 5 ms for the ADC to initialise and settle.
61  ADC_Init( active_channels);     // Initialize the ADC for 4 Channel conversion
62  LL_mDelay(5);         // wait 5 ms for the ADC to initialise and settle.
63
64  GPIOA->BRR = SPI1_CS_Pin;     // CS Low to begin Transactions with ADC
65
66
67  /* USER CODE END 2 */
68
69
70
71  /* Infinite loop */
72
73  while (1)
74  {
75
76
77    /* USER CODE BEGIN 3 */
78
79    GPIOA->BRR = SPI1_CS_Pin;         // This time CS is held LOW to monitor MISO line.
80                                      // If a conversion is available, MISO line is pulled LOW
81
82    while( GPIOA->IDR & LL_GPIO_PIN_6);         // Wait here until MOSI line is pulled Low by
      ADC
83
84    SPI_COMM( (uint8_t *) sample, (uint8_t *) sample, 5);   // Get the Conversion result by
      sending the Read command
85    sample[0] = 0x42;           // Reinitialize the Read Command
86
87
88    packet[4] = (sample[4] & 0x07) + 49;        // put an AND mask on bits to extract the
      channel number.
89                                // Add 48 to convert the number into the corresponding
      character value.
90
91
92                  // This part is doing the Sample Encoding
93    for(index = 0; index < 2; index++)        // Encode only the first 2 Most Significant
      Bytes from the conversion result
94    {
95      upper_half = ( sample[index + 1] & 0xF0 ) / 16; // get the upper nibble from the
      converted byte and shift it Right 4 positions
96      lower_half = sample[index + 1] & 0x0F;    // get the lower nibble from the converted
      byte, no shift needed here
97      if(upper_half < 10)         // As the conversion is Hexadecimal
98        packet[2 * index] = upper_half + 48;    // Getting the corresponding symbol for the
      converted upper nibble is done by adding an offset
99      else
100       packet[2 * index] = upper_half + 55;    // if the number is greater than 9, it means
      hexadecimal characters are used, so the offset changes.
101
102     if(lower_half < 10)         // same encoding goes for the lower nibble
103       packet[2 * index + 1] = lower_half + 48;
104     else
105       packet[2 * index + 1] = lower_half + 55;
106
107
108   }
109
110   UART_TX( (uint8_t *) packet, sizeof(packet));     // transmit the encoded sample through
      UART
111
112   sample[1] = 0;
113   sample[2] = 0;
114   sample[3] = 0;
115   sample[4] = 0;
116
117 }
118 /* USER CODE END 3 */
```

```
119 }
120
121 /**
122   * @brief System Clock Configuration
123   * @retval None
124   */
125 void SystemClock_Config(void)
126 {
127   LL_FLASH_SetLatency(LL_FLASH_LATENCY_1);
128   while(LL_FLASH_GetLatency()!= LL_FLASH_LATENCY_1)
129   {
130   }
131   LL_PWR_SetRegulVoltageScaling(LL_PWR_REGU_VOLTAGE_SCALE1);
132   LL_RCC_HSI_Enable();
133
134    /* Wait till HSI is ready */
135   while(LL_RCC_HSI_IsReady() != 1)
136   {
137
138   }
139   LL_RCC_HSI_SetCalibTrimming(16);
140   LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_HSI, LL_RCC_PLLM_DIV_1, 8, LL_RCC_PLLR_DIV_2);
141   LL_RCC_PLL_EnableDomain_SYS();
142   LL_RCC_PLL_Enable();
143
144    /* Wait till PLL is ready */
145   while(LL_RCC_PLL_IsReady() != 1)
146   {
147
148   }
149   LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);
150
151    /* Wait till System clock is ready */
152   while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL)
153   {
154
155   }
156   LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_2);
157   LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
158   LL_RCC_SetAPB2Prescaler(LL_RCC_APB2_DIV_1);
159
160   LL_Init1msTick(32000000);
161
162   LL_SetSystemCoreClock(32000000);
163   LL_RCC_SetUSARTClockSource(LL_RCC_USART2_CLKSOURCE_PCLK1);
164 }
165
166 /* USER CODE BEGIN 4 */
167
168 uint8_t ADC_Get_ID()            // A simple subroutine that tests the communication with the
      ADC
169 {
170   uint8_t comms[] = {0x45, 0x00};       // This is the "read ID" command
171
172   SPI_COMM( (uint8_t *) comms, (uint8_t *) comms, 2); // Send the command
173   return comms[1];          // return the ID byte
174
175 }
176
177 uint8_t ADC_Get_Status()          // A subroutine used to return the contents of Status
      Register
178 {
179   uint8_t comms[] = {0x40, 0x00};
180
181   SPI_COMM( (uint8_t *) comms, (uint8_t *) comms, 2);
182   return comms[1];
183
184 }
185
186 void ADC_Reset()            // A simple subroutine in order to reset the ADC, by writing at
      least 64 bits of 1 on DIN
187 {
188     uint8_t dummy[10] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
189     uint8_t empty[10];
190
191     SPI_COMM( (uint8_t *) dummy, (uint8_t *) empty, sizeof(dummy)  );
```

```
192
193 }
194
195 void ADC_Init( uint8_t channel_number )       // Configuration Subroutine
196 {
197
198     uint8_t i = 0;       // iterator for the number of channels
199     uint8_t buffer[3];       // this vector will hold the configuration bytes
200   uint8_t dummy[4];   // this vector is used as a reception end for SPI routine
201
202
203   // channel configuration bytes, depending on the number of channels selected
204   uint8_t channels[24] = { 0b00001001 , 0b10000000 , 0b00010011 ,
205                0b00001010 , 0b10000000 , 0b01010011 ,
206                0b00001011 , 0b10000000 , 0b10010011 ,
207                0b00001100 , 0b10000000 , 0b11010011 ,
208                0b00001101 , 0b10000001 , 0b00010011 ,
209                             0b00001110 , 0b10000001 , 0b01010011 ,
210                0b00001111 , 0b10000001 , 0b10010011 ,
211                0b00010000 , 0b10000001 , 0b11010011 };
212
213
214
215     uint8_t config_0[3]   = { 0b00011001 , 0b00000000 , 0b00010000 };   // unipolar supply for
       the analog channels, gain of 1.
216     uint8_t filter_0[4]   = { 0b00100001 , 0b01000000 , 0b00000000, 0b00000000 }; // Filtering
       at mininum to ensure maximum sampling rate
217     uint8_t adc_control[3]  = { 0b00000001 , 0b00000101 , 0b11000000 };   // Internal
      Reference, Full Power, output status reg with each conversion
218
219
220
221     for(i = 0; i < channel_number; i++)             // send the channel configuration bytes,
      depending on the number of active channels
222     {
223             buffer[0] = channels[ 3 * i + 0];
224             buffer[1] = channels[ 3 * i + 1];
225             buffer[2] = channels[ 3 * i + 2];
226             SPI_COMM( (uint8_t *) buffer, (uint8_t *) dummy, sizeof(buffer));
227     }
228
229   // send the other configuration bytes
230
231     SPI_COMM( (uint8_t *) config_0, (uint8_t *) dummy, sizeof(config_0));
232     SPI_COMM( (uint8_t *) filter_0, (uint8_t *) dummy, sizeof(filter_0));
233     SPI_COMM( (uint8_t *) adc_control, (uint8_t *) dummy, sizeof(adc_control));
234
235
236 }
237
238 void SPI_COMM( uint8_t * tx_buffer, uint8_t * rx_buffer, uint8_t size)
239 {
240   int i = 0;
241
242   GPIOA->BRR = SPI1_CS_Pin;   // CS Low to begin Transaction
243
244   for(i = 0; i < size; i++)
245   {
246
247      LL_SPI_TransmitData8( SPI1, *(tx_buffer + i));      // place the data in the buffer
248      while( LL_SPI_IsActiveFlag_BSY( SPI1));       // wait until transmission complete
249      *(rx_buffer+i) = LL_SPI_ReceiveData8( SPI1);      // simultaneously with transmission,
     read the input data
250   }
251
252    GPIOA->BSRR = SPI1_CS_Pin;   // CS High to end transaction
253 }
254
255 void UART_TX( uint8_t *tx_buffer, uint8_t size)
256 {
257   int i = 0;
258
259   for(i = 0; i < size; i++)
260   {
261     LL_USART_TransmitData8( USART2, *(tx_buffer + i));    // place the data in the buffer
262     while( !LL_USART_IsActiveFlag_TC( USART2));     // wait until transmission is complete
```

```
263   }
264 }
265
266
267 /* USER CODE END 4 */
268
269 /**
270   * @brief  This function is executed in case of error occurrence.
271   * @retval None
272   */
273 void Error_Handler(void)
274 {
275   /* USER CODE BEGIN Error_Handler_Debug */
276   /* User can add his own implementation to report the HAL error return state */
277   __disable_irq();
278   while (1)
279   {
280   }
281   /* USER CODE END Error_Handler_Debug */
282 }
283
284 #ifdef  USE_FULL_ASSERT
285 /**
286   * @brief  Reports the name of the source file and the source line number
287   *         where the assert_param error has occurred.
288   * @param  file: pointer to the source file name
289   * @param  line: assert_param error line source number
290   * @retval None
291   */
292 void assert_failed(uint8_t *file, uint32_t line)
293 {
294   /* USER CODE BEGIN 6 */
295   /* User can add his own implementation to report the file name and line number,
296      ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
297   /* USER CODE END 6 */
298 }
299 #endif /* USE_FULL_ASSERT */
300
301 /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Multi-Channel Acquisition Script

```
import matplotlib.pyplot as plt
import numpy as np
import serial
import time

#-------------------------------------------------------------------------
# Acquisition Module Software Instance
#-------------------------------------------------------------------------

stm32 = serial.Serial( port = "COM6",                                    # select communications port;
                       baudrate = 921600,                                # baud Rate;
                       bytesize = 8,                                     # byte size;
                       timeout = 2,                                      # timeout in seconds;
                       stopbits = serial.STOPBITS_ONE )                  # one stop bit;

stm32.set_buffer_size( rx_size = 16384, tx_size = 16384 )                # some large enough buffers for our acquisition;
                                                                         # for a max of 8 channels simultaneously activated
                                                                         # and for a sampling freq. of 500Sa/s/channel
                                                                         # about 4000 samples of signal will arrive,
                                                                         # each sample containing 6 bytes.
                                                                         # 4 bytes for the sample, 1 byte for the channel number, one for '\n'.

print( "Acquisition Module Paired" )
print( "Time for some data exchange" )

#-------------------------------------------------------------------------
# Subroutines Space
#-------------------------------------------------------------------------
# Acquisition Subroutine - it receives the number of channels and buffer size1
#-------------------------------------------------------------------------

def acquire( channel_number, max_buffer_size ):                          # the number of active channels and the size of the buffer must be known;

    max_16b_unipolar_value = 65536                                       # Only 16 bits out of 24 are useful data, the remaining LSB is noise;

    ADC_V_Ref = 2.5                                                      # The ADC uses 2.5V reference;

    sample_count = np.zeros( channel_number, dtype = 'uint16' );         # Initialize the multi-channel counter variable;

    max_samples = max_buffer_size * np.ones( channel_number )            # this variable represents the maximum capacity for each channel;

    buffer_dimensions = tuple( [ channel_number, max_buffer_size ] )     # Express the buffer dimensions as tuple for the next instruction;

    channel_buffers = np.zeros( buffer_dimensions )                      # Define the channels buffers;

    active_channel = 0;                                                  # This variable indicates to which channel the sample belongs to;

    error_count = 0;                                                     # In case of failure, error count can be valuable information;


    while( not np.array_equal( sample_count, max_samples ) ):            # as long as not all the channel buffers are full, acquire data;

        try:                                                             # "Try " statement is used to ignore any reception errors and continue acquisition
```

```
54
55              if( stm32.in_waiting > 0 ):                                                 # if
    there is any data in the buffer
56
57                  serial_data = stm32.readline()                                          # read
     until '\n' character store it in program variable;
58
59                  active_channel = serial_data[ len( serial_data ) - 2] - 49              # Get
    the corresponding channel of the received sample;
60
61                  if( sample_count[ active_channel ] < max_buffer_size ):                 # if
    the max_buffer for a channel was not reached, continue acquisition on that channel;
62
63                      channel_buffers[ active_channel ][ sample_count[active_channel ] ] = \
64                                      ( int( serial_data[0:2], 16) * 256 + int( serial_data
    [2:4], 16) )/ max_16b_unipolar_value * ADC_V_Ref
65
66                                                                                          #
    convert to voltage values;
67
68                      sample_count[ active_channel ] += 1;                                #
    increment the count on that channel;
69
70          except:
71              error_count += 1;                                                           #
    Increment the error count in case of failed reception
72
73      return channel_buffers
74
75  #---------------------------------------------------------------------------
76  # MAIN PROGRAM
77  #---------------------------------------------------------------------------
78  # interrogate for the number of channels first
79  #---------------------------------------------------------------------------
80
81  #
82  window_size = 1000                                                                      # This
     states how many samples are displayed on the screen for all channels;
83
84  sliding_window_size = int( window_size / 20 )                                           # This
     variable defines how fast the scroll rate is;
85
86  print( "Choose the number of channels (from 2 to 8):", end=' ' )                        # User
     Input - the number of channels;
87
88  channel_number = int( input() )
89
90  signal = [ None ] * channel_number                                                      #
    Define a signal vector for each channel;
91
92  line = [ None ] * channel_number                                                        # Line
     is the variable that holds channels data samples;
93
94  plt.ion()                                                                               # Set
    Matplotlib non-blocking execution;
95
96  fig = plt.figure()                                                                      #
    Create the figure and the number of subplots;
97
98  grid_spec = fig.add_gridspec( channel_number, hspace = 0 )                              # Add
    grids for each channel plot;
99
100 ax = grid_spec.subplots( sharex = True, sharey = True )                                 # Axis
     share settings for the plots;
101
102
103 if channel_number == 1:                                                                 # In
    case of a single channel, a single plot is needed
104     ls = list()                                                                         # but
    the code works on a list of plots, so a single
105     ls.append( ax )                                                                     #
    element must be converted into a single element list,
106     ax = ls                                                                             #
    which fixes a bug in the code when only one channel is selected;
107
108
```

```
109 fig.suptitle( "Time Domain Signals" )                                    # Draw
     the title of the figure;
110
111 for ch in range( channel_number ):                                       #
    First time initialize the signals for plotting;
112
113     signal[ ch ] = np.zeros( window_size )                               #
    Initialise signal buffers with zero;
114
115 window = range( window_size )                                           #
    Define the range of samples for display;
116
117 for ch in range( channel_number ):
118
119     line[ ch ], = ax[ ch ].plot( window, signal[ ch ], "b-")             #
    Create one subplot for each channel;
120
121     ax[ ch ].set_xlim( [ 0, window_size ] )                              # Time
     domain limits (expressed as number of samples);
122
123     ax[ ch ].set_ylim( [ 0, 2.5 ] )                                      #
    Vertical limits;
124
125     ax[ ch ].grid()                                                      # Add
    Grids for easier reading;
126
127     ax[ ch ].set_ylabel( " Ch " + str(ch + 1) + " [V]" )                 #
    Display the channel number for each suboplt;
128
129 ax[ channel_number - 1].set_xlabel( " Number of Samples, 300 Sa/s " )    #
    Informative subtitle for time Axis;
130
131 plt.show( block = False )                                               # for
    Python IDLE plots to work, this instruction is added;
132
133 while True:                                                             #
    Update the plots in an infinite Loop;
134
135 #    start = time.time()                                                # Used
     only for speed measurement purposes;
136
137     buffer = acquire( channel_number, sliding_window_size )
138
139 #    time_elapsed = time.time() - start                                 # Used
     only for speed measurement purposes;
140 #    print(time_elapsed)
141
142     for ch in range(channel_number):
143         signal[ ch ] = np.append( signal[ ch ], buffer[ ch ])            # Add
    the newly acquired samples
144
145         signal[ ch ] = signal[ ch ][ sliding_window_size: ]              # And
    remove the old ones;
146
147         line[ ch ].set_ydata( signal[ ch ] )                             # Set
    the plot line data;
148
149     fig.canvas.draw_idle()                                               #
    Finally Update the plot;
150     fig.canvas.flush_events()
151
152
153 #------------------------------------------------------------------------
154 # END OF CODE
155 #------------------------------------------------------------------------
```