Grado en Ingeniería en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Portable data acquisition and representation system for a VLF receptor

Autor: Fernando Montoya Andúgar

Tutor/es: Consuelo Cid Tortuero
Antonio Guerrero Ortega

2019

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Portable data acquisition and representation system for a VLF receptor

Autor: Fernando Montoya Andúgar

Tutor/es: Consuelo Cid Tortuero
Antonio Guerrero Ortega

TRIBUNAL:

Presidente: …………………………………………………

Vocal 1º: ……………………………………………………

Vocal 2º: ……………………………………………………

FECHA: ……………………………………………………

“I have noticed even people who claim everything is predestined, and that we can do nothing to change it, look before they cross the road”

- Stephen Hawking

Thank You Note

This note of thanks is directed to all those who have helped me all these years to accomplish this project, this graduation work.

Furthermore, this past year has been especially difficult for me, but this year I found a very strong support in order to complete, in a good manner, this degree. This key person is Mara, who always believed in me and gave me all the help a person can give.

To my family and friends who always knew I could achieve this goal that I planned four years ago.

To Consuelo Cid Tortuero, a tireless fighter who gave me the opportunity to demonstrate all I could do even at the beginning of the degree. She gave me the opportunity to create the first prototype of a VLF receptor, with a blind faith in me and my colleague Alberto, giving us all we needed in order to complete that task. In addition to this, she continued giving us different chances in different occasions like the opportunity to go to European Space Weather Week (ESWW), at Belgium, to show to the community what are we doing in Alcalá de Henares. For that and for many reasons, I have to greatly thank my tutor Consuelo.

To Antonio Guerrero Ortega, for helping me these four months in all I have needed. He gave me total liberty to face the graduation work with the resources of my choice, giving me help both in the technical part and in the stuff. I really liked this freedom in my work. For all that reasons, thanks Antonio.

To Alberto García Merino, my companion fatigue and colleague that believed in us as a team and put with me all we could do to achieve our goals. I hope, Alberto, you have a promising future.

To the rest of people that faced me once and, in a few minutes, told me they were completely sure I could get to the end in these four years of degree.

To you all, thank you.

Index

Resumen

Este trabajo consta de dos microcontroladores y una serie de periféricos que complementan su funcionalidad. En uno de estos periféricos, la pantalla TFT, se muestra un menú donde se pueden realizar diferentes funcionalidades.

El microcontrolador principal recibe una señal y es capaz de descomponer la misma en sus componentes de frecuencia. Muestra en pantalla, bien sea el espectro de la señal, o la señal en el dominio del tiempo, pudiendo elegir 4 frecuencias de interés que posteriormente podrán ser almacenadas en una tarjeta SD y enviadas al servidor correspondiente mediante el segundo microcontrolador.

Palabras clave

Arduino, Transformada Rápida de Fourier, Osciloscopio, HTTP, Conversor analógico-digital

Summary

This work is based on two microcontrollers and some peripherals that complement their functionality. In one of these peripherals, the TFT screen, shows a menu where you can select different tasks.

The main microcontroller receives a signal and it is capable of decompose the signal in its frequency components. The screen shows both the signal spectrum and the signal in the time domain, depending on the user choice, being able to select 4 of these components that will be saved in a SD card memory and uploaded to the corresponding server through the second microcontroller.

Key words

Arduino, Fast Fourier Transform, Oscilloscope, HTTP, Analog to Digital Converter

Extended Summary

Today and increasingly, the society depends on technology. Our day-to-day needs technology and communications to carry out the work we have to do and the normal course of our lives. In this sense, a failure in the communications system could be as nefarious as the absence of this technology.

Since the Carrington event, the solar storm of 1859, which caused the whole telegraph system failure in Europe and in North America, space weather has gradually become more and more relevant.

Space weather is the denomination of the different phenomena of the interaction between the Sun and the Earth. It studies the varying conditions within the Solar System. Space weather is influenced by the solar wind and the interplanetary magnetic field (IMF) carried by the solar plasma. One of the space weather phenomena are solar flares.

Solar flares have caused communications interruptions on many occasions. We have to mention the event occurred on September 2017 in the Caribbean Area [1], where the communication failed when being crucial to save many lives. This gives us an idea of how important is to understand our System Solar. In order to study the Sun, there are several geostationary satellites as GOES, which measure, at different soft X-ray wavelengths, the flux from the Sun. These data can be downloaded by the scientific community to carry out different research-related studies.

In this End-of-Grade work, we will take advantage of the effects of the solar flares in our atmosphere to identify and quantify them. In particular, we can indirectly measure the sudden ionospheric disturbances (SID) to monitoring in real time the solar activity.

This project will cover the half of a VLF receptor. More specifically, the data acquisition and representation system of the radio signal which is considered already characterized. The antenna, preamplifier and signal conditioning as well as the characterization of the signal are part of another End-of-Grade work.

Firstly, we will discuss the specific objective of this End-of-Grade work and an introduction to the system developed. We comment that our goal is to measure indirectly solar flares by monitoring the Ionosphere with the whole VLF receptor built.

Then, we will explain all the applicable theory to implement the system. The analog to digital conversion theory, the DFT and its improved FFT version, SPI, UART, I2C communication protocols, are some of the points explained for this purpose. In addition to this, we will see different aspects about the C programming language and the use of memory in order to understand how the sketch coded for the system is implemented in our microcontroller.

At this point we will cover the hardware used for the system. As we said, the system is implemented digitally in a microcontroller. We will go over the one used for this purpose: Arduino Due. In this microcontroller we found good balance between quickly productivity and power. It is backed up by the Arduino community and implements a powerful Cortex-M3 processor, which peripherals can be configured by the manufacturer libraries, written in C language, in order to set them as user prefers. The connection between them will be seen lately in the part related to the schemes.

After the microcontroller overview, we will see the rest of the hardware used in this project. We discuss about the internal ADC, TFT screen, RTC and the microcontroller based on the ESP8266 chip.

Then, we will discuss about the software aspects. Arduino IDE is the one chosen for our system. We will talk about the different parts of a sketch coded in this IDE. After this, we have to talk about the libraries used for the schematics. We will cover all the libraries that the Open Source Community has developed, showing the more relevant aspects of the libraries selected. As an important review, the library developed for Arduino Due for the TFT screen controller, uses some features present in the Due microcontroller, such as DMA, that are not present in others Arduino microcontrollers. It gives us more speed than other basic libraries rendering the screen.

After the description of the whole system, in a hardware and a software point of view, we will explain the structure of the code. The sketch, which is about 880 lines of code, implements two state machines in order to control the options and features that our system is capable of. The state machine number one basically checks the screen to know which button was pressed. Knowing the button, there is a set of variables, which only can take the *true* or *false* value*,* that leads the system to the state machine number two. This second state machine checks these variables in order to execute the corresponding part of the code.

We will comment different tests that were made with the purpose of establishing the value of some variables or to demonstrate the applicable theory. Also, we will show the different sketches and examples implemented, for each device used on the system, to demonstrate their functionality. The ADC tests were, probably, the most important part of the system analysis. As a complement to this part, there is a MATLAB script created for the signal analysis of the samples obtained with our system. With an oscilloscope, we input a signal with amplitude and frequency known to obtain in MATLAB the corresponding signal and its frequency. MATLAB is a programming platform designed specifically for engineers and scientists. The heart of MATLAB is the MATLAB language, a matrix-based language allowing the most natural expression of computational mathematics.

With all the tests made, we will explain all the parts of the code implemented. We will go through each part explaining all the concepts that, in advance, were explained before in the corresponding section. The explanation of the code will be better understood if the corresponding 6.-*Tests and results* section has been read. We will see the global variables declared, the functions implemented, and of course, each state of the state machines. We recommend following all the corresponding section before analyzing the code entirely. We explain the purpose of each variable, the meaning of each statement in the code, and go through each function implemented step by step with references to the applicable theory and documentation.

The code basically has four main features. One is a kind of oscilloscope where the user can see the input signal in the time domain. There it is showed part of the signal because the purpose is to notice the maximum and minimum values of the input signal and the form of the wave to show the aspect of the signal. Another function is to analyze in real time the spectrum of the input signal. When we see the signal frequency domain, we can see the relevant harmonic present in the signal. This pretends to see the frequencies of interests in which the user may want information over time. Also, this function can be used as an indicator of the antenna orientation. We can measure, at first glance, if the current orientation of the input signal generator is correct or not. This is one of the main goals that this End-of-Grade work intends to achieve.

The others two goals are described in the following. In order to extract the information over time respect for a certain frequency, there had to be a frequency selection capability. There is a screen in the system implementation where the user can see the spectrum of the input signal and select, over it, 4 frequencies of interest. Once the frequencies are selected, we go into the main purpose of the project. This goal aims to log the data of the desired frequencies selected before. When this capability is set, the system takes the magnitude of the corresponding frequency and store it in memory. This is repeated each 5 seconds to store the values over a minute. When a minute passed, the system takes the average of all the data collected and store them in the SD card, in the form of a csv file. There is a file each day in question. In addition to his, there is implemented another extra feature with the intention to test the possibilities of the system. When the minute passed, the average of the values is computed and stored in the SD card, also the main microcontroller sends to the secondary, the first frequency selected and its average magnitude. This second microcontroller establishes an HTTP connection with the server of the SPACE WEATHER GROUP of University of Alcalá, with the purpose to make a GET method and store in the server those values. The user can access to a specific web page to see these results online.

After the code explanation, we show the scheme connection of the system. Also, there are diagrams and photos of the system in its case.

To end the memory of this End-of-Grade Work, we made a User Manual to manage the system.

In this User manual we show the corresponding screen with examples of the input signals to make an idea of how the system works at high level point of view. We will see a guide to understand the system behavior and with that manual, any user can interact with the system understanding the procedure.

Glossary of abbreviation

ADC – Analog to Digital Converter
CME – Coronal Mass Ejection
CPU – Central Processing Unit
CSV – Coma Separated Values
DAC – Digital to Analog Converter
DFT – Discrete Fourier Transform
DFS – Discrete Fourier Series
DMA – Direct Memory Access
DTFT – Discrete-Time Fourier Transform
FFT – Fast Fourier Transform
GPIO – General Purpose Input-Output
HTTP – Hypertext Transfer Protocol
I/O – Inputs and Outputs
I2C – Inter-Integrated Circuit
IDE – Integrated Development Environment
IP – Internet Protocol
LCD – Liquid Crystal Display
LF – Low Frequency
LSB – Least Significant bit
MSB – Most Significant Bit
PCB – Printed Circuit Board
RAM – Random Access Memory
RTC – Real Time Clock
SAR – Successive Approximation Register
SCL – Serial Clock Line
SD – Secure Digital
SDA – Serial Data Line
SID – Sudden Ionospheric Disturbance
SoC – System on Chip
SPI – Serial Peripheral Interface
SWE-UAH – Space Weather Group of University of Alcalá
S/H – Sample and hold
TCP – Transmission Control Protocol
TFT – Thin Film Transistor
UART – Universally Asynchronous Receiver/Transmitter
VLF – Very Low Frequency
µC – Microcontroller
µs – microseconds

Memory

1.- Introduction and objectives

A solar flare is a sudden flash of brightness variations on the Sun. They produce electromagnetic radiation across the electromagnetic spectrum at all wavelengths, from radio waves to gamma rays. They occur in active regions, sometimes labelled as sunspots, where intense magnetic fields penetrate the photosphere to link the corona to the solar interior. The energy released in a flare may produce a coronal mass ejection (CME) although the relationship between CMEs and flares is still not well understood. Powerful flares disturb the ionosphere interrupting communications.

The Very Low Frequency (VLF 3-30 kHz) and Low Frequency (LF 30-300 kHz) waves are propagated through the surface-ionosphere following a waveguide model. The lowest level of the ionosphere (D layer, at 60-90 km of altitude) conforms a conductive layer formed by electrons and ions that reflect these kinds of waves forming a Zig-Zag long range transmission. These transmissions are often used to air radio navigation, time radio signals (radio clocks) and military communications.

The objective is, therefore, been capable of measure a communication of an VLF or LF emitter in order to monitoring the power of the signal. When a solar flare occurs, the power of the received signal changes and, for that reason, we can measure indirectly the moment and magnitude of a solar flare.

From the Space Weather Group of University of Alcalá (SWE-UAH), it has been intended to build an antenna capable of measure the power of these kind of signals and monitoring the ionosphere situation. There is a prototype in the facilities of SWE-UAH that measured several solar flares on September 2017. This proved that the system works. The prototype consists on a loop antenna with a preamplifier and a computer.

The analog to digital converter used in the prototype is the sound card of the PC, and the signal processing is done with a Python algorithm programmed by Eric Gilbert called "SuperSID" [2]. This leads to the system having to be in a specific place with some characteristics allowing to set the computer and the antenna. Also, you have to see the computer screen with the working algorithm to see the variation of the signal received with changes of the antenna position.

The SWE-UAH has, in addition to this final degree work, another student in charge of a new antenna characterization and the signal offered to the data acquisition system. This work is done by Alberto García Merino and it complements my work, in order to build a complete solar flare detector based on the sudden ionospheric disturbances.

This work consists on the data acquisition and the representation of the signal of that antenna.

Different microcontrollers are available in the market with a good analogue to digital converter (ADC) like ST Microelectronics family, or a simple microcontroller with its own external ADC designed to this kind of signal. After different approaches to different products, the choice was a system based on the Arduino Due board.

Arduino is an open-source electronics platform based on easy-to-use hardware and software. The flexibility of its Integrated Development Environment (IDE), the accessible libraries and the hardware built to work together with it, could facilitate the work of integrating all the components of the system. About the available boards, the requirements of the project made the choice difficult because the microprocessors offered by this platform usually have low capabilities and are used for low requirements. The typical boards are Arduino UNO, with an 8-bit processor with 20 MHz, and Arduino Mega, with more input and outputs ports but with an 8-bit processor with

16 MHz also. On the other hand, Arduino has a board based on the well-known 32-bit Cortex-M3 processor, called Arduino Due. This board implements the AT91SAM3X8E microcontroller with 32-bit bus, clock speed of 84 MHz, Direct Memory Access (DMA) hardware, 12-bits ADC up to 1 msps (mega samples per second), and so on.

With this board, and after some tests with other microcontrollers, the system is based on this Arduino environment with some additional hardware that I will comment on this End-of-Grade work.

So, the heart and brain of the system is this board. The rest of the hardware that pretends to be the complement is an TFT Screen, a Real Time Clock (RTC), SD memory card and another microcontroller called ESP8266.

The TFT screen shows a menu when the system is turned on. You can select four options that is, basically, to view in real time the signal that the board is processing, view the spectrum of that signal, select four frequency components to store in both the SD card and in the SWE-UAH server, and the option to starts logging the data of this four frequency components selected.

To store the data, the system starts to sample the input and stores in a buffer in memory all the digital values. When the buffer is empty, a Fast Fourier Transform is applied to decompose the signal in its frequency components and then store in other buffer the amplitude of the four frequencies selected. When a minute passes, the microcontroller takes the average of the values stored in this set of time and, with this final value, the system stores the data. The data are stored as a csv file in the SD card that shows in each row the four frequencies and its corresponding amplitudes with the timestamp at the end of the row. In addition to this, the system sends this average value through a serial communication to the other microcontroller (ESP8266), which will upload the data to the SWE-UAH server.

The ESP8266 is a low-cost Wi-Fi chip that implements a TCP/IP complete stack and a processor called Tensilica Xtensa with 32 bits and 80MHz. Although the processor is nearly the same as AT91SAM3X8E, the poor Input/Output (I/O) capability and the low memory capacity do not allow us to implement the entire project in it. This chip can be programmed through Arduino IDE as well, and for this project there is a simple program stored in its memory that receive a communication from Due to receive the data and then upload them through a GET command (HTTP protocol) to the server. For this project, and in order to show the capability of the system, there is only one frequency implemented. Although in the SD card the file is, as described before, in the server, there is a web? page that receives one frequency and its amplitude and saves it in a text file which can be viewed through the internet.

The screen used is a 2.8-inch SPI module based on ILI9341 controller. This screen also has a touch panel that lies under the display. Furthermore, the PCB of the screen counts also with a SD slot and the corresponding tracks to implement the SPI protocol also with this memory hardware. For these reasons, and because there is a library written for Arduino Due to handle this controller using the DMA, we consider? that this screen was perfect for our purpose.

The RTC is based on the DS323 chip. This clock pretends to store the timestamp of the samples in order to know when the signal is processed. Also, the system takes into account the day in order to create in the SD card the file corresponding with the current day, which is labeled according to the year, month and day.

Another important section would be the implementation of the Fast Fourier transform. Many tests were made to characterize the signal and to check the correct functioning of the system. It has even been possible to check the complexity of the algorithms of the Discrete Fourier Transform (DFT) and the FFT. In the following parts of this memory we will discuss the tests as well as the peculiarities of them and hardware limitations.

The main program is a state machine written in C programming language that calls different functions to draw in the screen, implement the FFT, configure peripherals, etc. There are several libraries used written by the open-source community, that I will mention at the end of this document, which helped me to save time developing the protocols and controllers' communications.

2.- Applicable theory

2.1.- Analog to digital conversion theory [3]

ADC is the device in charge of generating a digital output signal of n bits from an analogue voltage signal. The n bits are fixed and intrinsic to the ADC device. There are many internal conversion technologies designed for this purpose, all of them with different characteristics.

The time used for digitalization is called conversion time. The first task the ADC performs is to sample and then convert to digital in the retention stage.

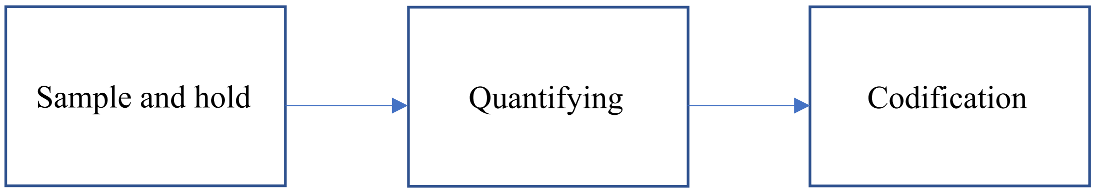


Figure 1: Steps of an ADC

2.1.1.-Sample and hold

The sample and hold (S&H) parts are not mandatory, but necessary for most applications. These two steps are executed by the S&H device which can be inside or outside of the ADC. The sample is considered equally-spaced and constant in order to characterize the signal properly. While sampling step is executing, there is no digital output in the ADC.

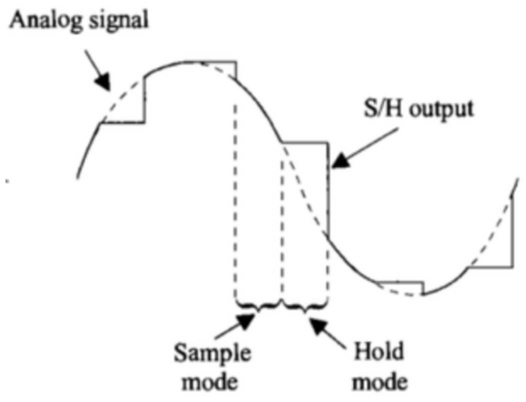


Figure 2: Real S/H process

We can define the sample step as the stage in which the value of an analogue input signal is extracted, every Ts seconds. To obtain enough information of the analogue signal, we have to meet the Nyquist theorem [4]. This means that the sampling frequency has to be at least twice the highest frequency involved in the analogue signal. But in reference to this, this criterion is sometimes poor in real applications and the criteria used is to sample with a frequency ten times higher the highest frequency component.

In the hold process is where the analog to digital conversion process is carried out. It is important that the input remains constant in this step because it is consulted continuously, and variations in this value causes a bad conversion. In addition to this, the hold time is intended to be as small as possible.

2.1.2.- Quantifying

It is performed during the retention interval (hold mode). This process reduces the set of infinite values, at the input, to a finite range of digital values at the output. The ADC reduces a set of values in a time interval into the same digital value, depending on the span configured. The span is the difference between the maximum and minimum voltage that our ADC can handle. In our case, we have a span of 3.3V. In order to obtain the digital value, the transfer function used could be that shown in Figure 3, called quantification by rounding.

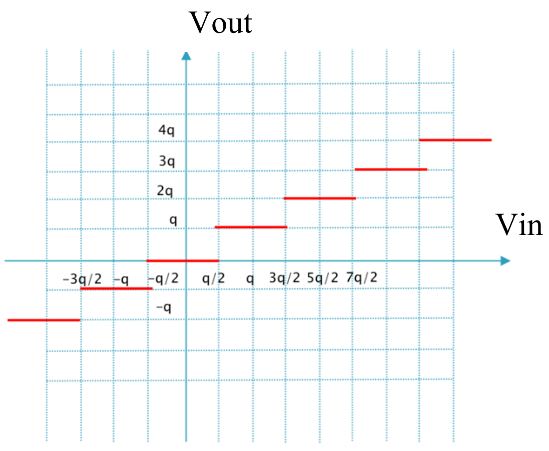


Figure 3: Transfer function of a quantifying process

We can appreciate in Figure 3 that for different values of an input we can obtain the same digital output value. Each section depends on the denominated quantification step ‘q’. This factor depends on the span and the number of bits. We can see this dependence in following expression:

$$q = \frac{V_{ref}\big|_{max} - V_{ref}\big|_{min}}{2^n - 1}$$

In our system, q is fixed as the numerator is 3.3 V, and the denominator only depends on the number of bits of our ADC which is fixed. In this sense, we have a quantification step q of:

$$q = \frac{3.3}{2^{12} - 1} = 805.86\ \mu V$$

The fact that the rounding is necessary, leads us to make an uncertainty. This error can be seen in Figure 3 for example if Vin is equal to $^{q}/_{2}$. In this hypothetical case, if Vout is equal to 0, we will make a $^{-q}/_{2}$ mistake, and by counterpart, if Vout is equal to $^{q}/_{2}$ the error would be $^{q}/_{2}$. For that reason, for each point of discontinuity we have an error of:

$$Error = \frac{\pm q}{2}\ V$$

2.1.3.- Codification

This process is basically the vertical axis of Figure 3. For each value of this axis corresponds a digital value in bits starting with the smallest:

Quantification output	Output Code (12 bits)
0·q	0000…0000
1·q	0000…0001
2·q	0000…0010
3·q	0000…0011
…	…

Figure 4: Codification table

2.2.- Types of ADC

Depending on the way the input signal is sampled, there are several types of ADC with different characteristics.

2.2.1.- Successive approximation ADC

The most commonly used.

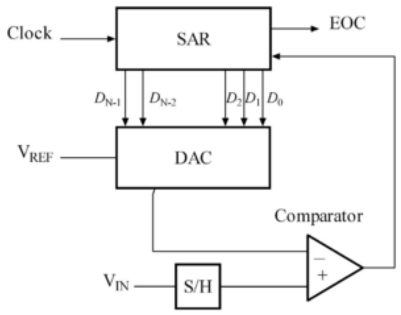


Figure 5: Block diagram of a successive approximation ADC

This kind of ADC converts the input signal via a binary search through all possible quantization levels before finally converging upon a digital output for each conversion [5]. The input signal is acquired by the S/H and led to the comparator. The comparator subtracts the input with the signal coming from the internal DAC, which converts the signal represented by the Successive Approximation Register (SAR). With this procedure, in ‘n’ clock edges the final value is obtained in SAR. Its serial nature limits its operating speed (kHz range) and get slower for high resolutions (n bits growing).

2.2.2.- Flash ADC

Also known as a direct-conversion ADC is the fastest ADC at the market (few GHz range).

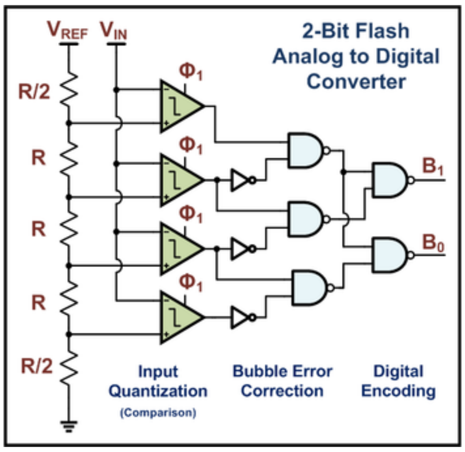


Figure 6: Block diagram of a flash ADC

The speed of the ADC means that there is no need for a S/H because the comparator are the sampling devices. As we can see in Figure 6, a flash converter needs $2^n - 1$ comparators for an n-bit conversion. Without go deeper in the digital encoding, the response time of this device is basically the propagation delay of the digital gates.

2.2.3.- Pipelined ADC

This is the kind of ADC implemented in our µC. It uses a flash converter internally. It is faster than SAR ADC but slower than flash ADC. A possible block diagram of a 12-bit pipelined ADC would be:

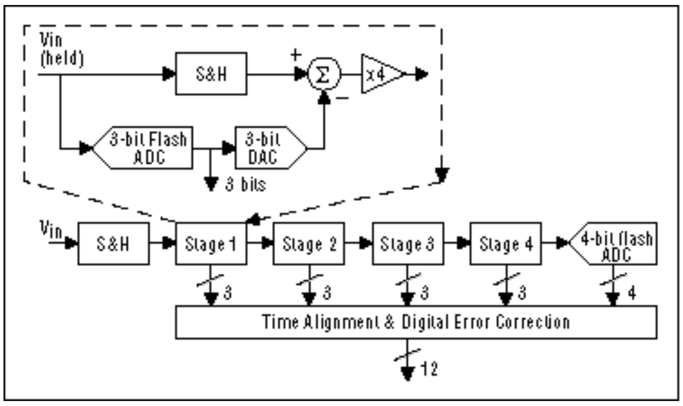


Figure 7: Block diagram of a pipelined ADC [6]

Here, the analog input Vin is first sampled and held steady by a S&H, while the flash ADC, the one seen before, in stage one quantizes it to 3 bits. The 3-bit output is then fed to a 3-bit DAC, and the analog output is subtracted from the input. This "residue" is then gained up by a factor of 4 and fed to the next stage. This gained-up residue continues through the pipeline, providing 3 bits per stage until it reaches the 4-bit flash ADC, which resolves the last 4LSB bits.

Although each stage generates 3 raw bits, we can see that the interstage gain is only 4. This is because each stage resolves effectively only 2 bits. The extra bit is to reduce the size of the residue by one half.

Because the bits from each stage are determined at different points in time, all the bits corresponding to the same sample are time-aligned with shift registers before being fed to the digital-error-correction logic. Note that as soon as a certain stage finishes processing a sample, determining the bits and passing the residue to the next stage, it can start processing the next sample due to the sample-and-hold embedded within each stage. This pipelining action accounts for the high throughput [7].

2.2.- Discrete Fourier Transform [8]

The information within a signal as well as its characteristics are difficulty extractable and manageable in time domain. This is why the introduction to transformed domains supposes a great advantage to understand and to interpret signals and systems. There are two useful aspects that justify the domain change: in one hand, frequency components analysis of the signal, and on the other hand, because the change domain allows us to simplify the computational complexity. As an example of the second part we know that convolution in the time domain corresponds to a simple multiplication in the frequency domain. The more useful transformation tool for the study of digital signals is the Fourier Transform of discrete time signals: Discrete-Time Fourier Transform (DTFT) [9], trough whose properties the theory of frequency analysis is developed.

As we need a microcontroller, we need to implement an algorithm to extract the previous information. We know that that a continuous time signal x(t) admits a numerical interpretation through the sampling theorem, which application give us a numerical sequence x[n] in discrete time. This sequence then, can be processed by numerical algorithms. A similar problem appears when we see that, resulting from DTFT as analysis tool, $X(e^{j\Omega})$ is a function of the continuous variable Ω. As a consequence, it is needed to develop a numerical procedure that allow us to use the theoretical capabilities of DTFT. This technique is the DFT.

The DFT is a numerical sequence obtained from sampling the $X(e^{j\Omega})$ spectrum, allowing us to represent, in a unique way, the signal x[n] in a transformed domain. As a consequence, the DFT keeps a close relationship with the Fourier Transform of x[n].

The inability to process signals of infinite length, mostly because the finite amount of memory, supposes another limitation in digital signal processing. This situation is also handled by applying the DFT as a medium to obtain the real spectrum of the signal. Another advantage is the computational cost. The DFT calculus is solved, in a very efficient way, by fast algorithms called Fast Fourier Transform (FFT). These algorithms justify the use of DFT, especially in those applications, that for their characteristics as consumption, size or time response, need a real time operation.

The DFT can be studied from different point of view. On one hand, it can be raised as a result from sampling the spectrum, and on the other hand it also supports an interpretation based on the relationship with Discrete Fourier Series, with periodic sequences.

2.2.1 Sampling in frequency domain

The $X(e^{j\Omega})$ spectrum of a signal in discrete time x[n] contains the whole frequency information about itself and it is computed by the DTFT:

$$x[n] \overset{\mathcal{F}}{\rightarrow} X(e^{j\Omega})$$

$$X(e^{j\Omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\Omega n} = \mathcal{F}\{x[n]\}$$

But it is a function of continuous variable and, for that reason, computationally intractable. Because of that, a possible strategy is applying the sampling technique to the spectrum, with the objective of maintain the frequency information of x[n] sequence in a finite set of data.

Be $X(e^{j\Omega})$ the periodic spectrum of period 2π of a discrete signal x[n], the DFT of x[n] is defined as the sequence X[k], which results from uniformly sampling $X(e^{j\Omega})$ in the interval [0, 2π]:

$$X[k] = X\left(e^{j\Omega}\right)\Big|_{\Omega=\frac{2\pi}{N}k}, \qquad k = 0, 1, 2, \ldots, N-1$$

From this expression we see that X[k] is a finite sequence composed of N samples equally-spaced from the DTFT of x[n] in the interval [0, 2π]. As X[k] is a sampling of $X(e^{j\Omega})$, a lot of its properties will be reflected, specially the next three ones:

- X[k] is inherently periodic with period N, since the sampling of a period of $X(e^{j\Omega})$ implies the implicit sampling across Ω-axis.
- The low frequencies correspond with the values of *k* around 0 or N.
- The high frequencies correspond with the values of *k* around the half of N.

From the DFT definition it is observed that it can provides a good description of the signal frequency components, at least in those values where DTFT and selected points of DFT coincide. Now, beyond that, it is intended that the transformation is biunivocal, which means that x[n] can be recovered without errors from X[k], in which case, X[k] will define x[n]. In these conditions, it can be defined the inverse transform $x[n] = \mathcal{DFT}^{-1}\{X[k]\}$. Similar to sampling analog signals selecting the correct sample rate, in DFT case is the same as to select the number of samples N needed.

To analyze the consequences of the spectrum sampling, we apply the previous definition

$$X[k] = X\left(e^{j\Omega}\right)\Big|_{\Omega=\frac{2\pi}{N}k} = \sum_{n=-\infty}^{\infty} x[n]e^{-j\frac{2\pi}{N}kn}, \qquad k = 0, 1, 2, \ldots, N-1$$

This expression of infinite terms is divided in groups formed by N addends:

$$\begin{aligned} X\left(e^{j\frac{2\pi}{N}k}\right) &= \cdots + \sum_{n=-N}^{-1} x[n]e^{-j\frac{2\pi}{N}kn} + \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn} + \sum_{n=N}^{2N-1} x[n]e^{-j\frac{2\pi}{N}kn} + \cdots \\ &= \sum_{r=-\infty}^{\infty} \sum_{n=rN}^{rN+N-1} x[n]e^{-j\frac{2\pi}{N}kn} \end{aligned}$$

If we substitute *n* by *n-rN* and, taking into account that $e^{-j\frac{2\pi}{N}k(n-rN)} = e^{-j\frac{2\pi}{N}kn}$:

$$X\left(e^{j\frac{2\pi}{N}k}\right) = \sum_{n=0}^{N-1}\left[\sum_{r=-\infty}^{\infty} x[n-rN]\right]e^{-j\frac{2\pi}{N}kn} = \sum_{n=0}^{N-1} x_p[n]e^{-j\frac{2\pi}{N}kn}, \qquad k = 0, 1, 2, \ldots, N-1$$

Where $x_p[n] = \sum_{r=-\infty}^{\infty} x[n-rN]$.

This is called the analysis equation of the DFT.

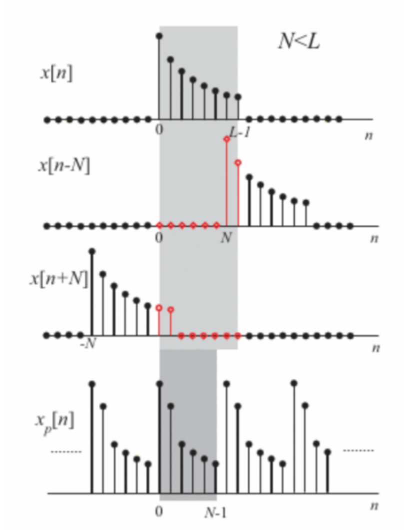


Figure 8: Periodic extension with temporary overlap

As we can see in Figure 8?, if the period is lower than the signal length, the overlap between some samples appears, and the original signal is affected by the final sum.

As it is demonstrated before, the temporary overlap is avoidable exceptionally if x[n] is finite in time. Therefore, if it is wanted to preserve the original signal:

$$x[n] = \begin{cases} x_p[n]\,, & 0 \leq n \leq N-1 \\ 0\,, & rest \end{cases}$$

The condition is N ≥ L, which means that the number of samples to be taken from the spectrum has to be greater than the length of the signal.

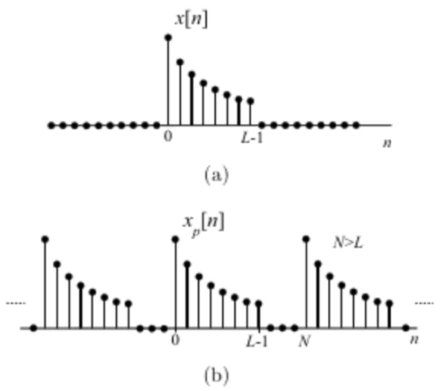


Figure 9: (a) x[n] sequence of length L with (b) its periodic extension without temporary overlap

In compliance with the requirements, we can define the DTFT expression in the form:

$$X\left(e^{j\Omega}\right) = \sum_{n=0}^{N-1} x[n]e^{-j\Omega n}$$

Where the limits of the summation have been reduced for convenience to the interval [0, N-1], as the signal is null outside the interval if N > L. This leads us to the analysis equation:

$$X[k] = X\left(e^{j\frac{2\pi}{N}k}\right) = \sum_{n=0}^{N-1} x_p[n]e^{-j\frac{2\pi}{N}kn}\,, \qquad k = 0, 1, 2, \dots, N-1$$

By counterpart, we have the synthesis equation of DFT:

$$x[n] = x_p[n] = \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}kn} \ , \qquad n < L$$

2.2.2 Relationship between DFT and Fourier Series development

Let $x_p[n] = x_p[n - lN] \ \forall \ l \ \in \ \mathbb{Z}$, the periodic extension of x[n] with N > L period, the DFS serves to characterize $x_p[n]$ and establishes that every periodic signal can be expressed as a lineal combination of a set of harmonically-related exponential functions.

For convenience, we are going to express the synthesis equation of the Fourier Series in the way:

$$x_p[n] = \frac{1}{N} \sum_{k=0}^{N-1} a_k e^{j\frac{2\pi}{N}kn}$$

Where a_k are the coefficients of the Fourier Series. Usually, this formula appears without the constant 1/N, but this does not affect to the approach because the results with one or another constant would be proportional.

For discrete time signals, the number of different coefficients is equal to the signal period, which is N. The coefficient's N values a_k are determinates through the analysis equation:

$$a_k = \sum_{n=0}^{N-1} x_p[n] e^{-j\frac{2\pi}{N}kn}$$

As this is an example without temporary overlap, it verifies that the expression is equal to:

$$a_k = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}$$

The relationship between transformed domains of the original signal and its periodic extension is stablished comparing the equations:

$$X(e^{j\Omega}) = \sum_{n=0}^{N-1} x[n] e^{-j\Omega n}$$

$$a_k = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}$$

Where the difference is on the exponent. It is deduced then that:

$$a_k = X(e^{j\Omega})\big|_{\Omega = \frac{2\pi}{N}k} \ , \ \ k = 0, 1, \dots, N-1$$

Where the DFS coefficients a_k match with spectrum samples of x[n] and, by definition, it is the DFT. Whit that, the DFT also can be defined as:

$$X[k] = \begin{cases} a_k\,, & 0 \le k \le N-1 \\ 0\,, & rest \end{cases}$$

To correctly interpret this section, we have to take into account that a_k coefficients represent the periodical extension $x_p[n]$, whether or not temporary overlap occurs. On the other hand, the number of spectrum samples of x[n] determines the $x_p[n]$ period. If no temporary overlap occurs, the a_k coefficients describe x[n] in a univocal way, and the original signal can be recovered by the expression $x_p[n] = \frac{1}{N}\sum_{k=0}^{N-1} a_k e^{j\frac{2\pi}{N}kn}$.

This case is illustrated in Figure 8 with N > L. It is easily deduced that the limit case is imposed by the length L. While the condition N ≥ L is fulfilled, there is no temporary overlap. As a consequence, the minimum number of points necessary to DFT in order to obtain a univocal operation is N = L.

To accomplish this point, if we meet the specified requirements explained before, with no temporary overlap, the usual way to express the analysis and synthesis equations of DFT are, respectively:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn} \quad , \quad k = 0, 1, \dots, N-1$$
$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j\frac{2\pi}{N}kn} \; , \quad n = 0, 1, \dots, N-1$$

In both equations it is implicitly assumed that x[n] is a finite sequence of length N. For that, all the operations involved in obtaining the DFT imply the complete set of samples.

2.2.3 Simple example of DFT

To show how DFT works, we will see an example of a 1 Hz sinusoid with amplitude equal to 1 and a sampling frequency equal to 8 Hz.

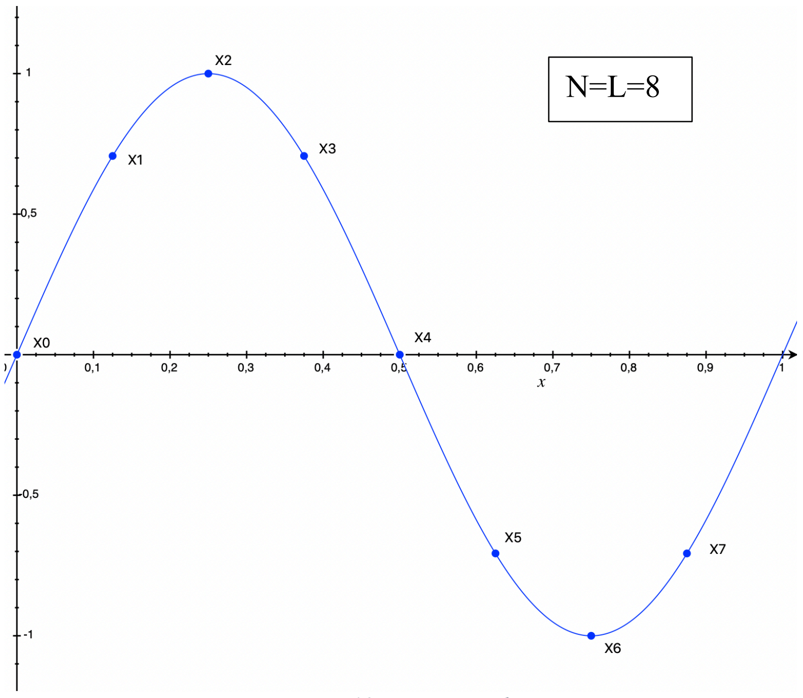


Figure 10: DFT example

With a sampling frequency of 8 Hz, we obtain the following points in our input signal sequence x[n]:

x[0]	0	x[4]	0
x[1]	0,707	x[5]	-0,707
x[2]	1	x[6]	-1
x[3]	0,707	x[7]	-0,707

Figure 11: Input signal of DFT example

In order to obtain the spectrum, we apply the analysis equation for each frequency bin:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn} \quad , \quad k = 0, 1, \dots, N-1$$

$$X_0 = 0e^{-j\frac{2\pi}{8}0} + 0{,}707e^{-j\frac{2\pi}{8}0} + \cdots = \sum x[n] = 0$$

$$X_1 = 0e^{-j\frac{2\pi}{8}1\cdot 0} + 0{,}707e^{-j\frac{2\pi}{8}1\cdot 1} + 1e^{-j\frac{2\pi}{8}1\cdot 2} + 0{,}707e^{-j\frac{2\pi}{8}1\cdot 3} + \cdots = -4j$$

$$X_2 = 0e^{-j\frac{2\pi}{8}2\cdot 0} + 0{,}707e^{-j\frac{2\pi}{8}2\cdot 1} + 1e^{-j\frac{2\pi}{8}2\cdot 2} + \cdots = 0$$

$$X_3 = 0e^{-j\frac{2\pi}{8}3\cdot 0} + 0{,}707e^{-j\frac{2\pi}{8}3\cdot 1} + 1e^{-j\frac{2\pi}{8}3\cdot 2} + \cdots = 0$$

$$X_4 = \cdots = 0$$

$$X_5 = \cdots = 0$$

$$X_6 = \cdots = 0$$

$$X_7 = 0e^{-j\frac{2\pi}{8}7\cdot 0} + 0{,}707e^{-j\frac{2\pi}{8}7\cdot 1} + 1e^{-j\frac{2\pi}{8}7\cdot 2} + 0{,}707e^{-j\frac{2\pi}{8}7\cdot 3} + \cdots = 4j$$

So, the complex numbers obtained from the spectrum are:

X[0]	0	X[4]	0
X[1]	0-4j	X[5]	0
X[2]	0	X[6]	0
X[3]	0	X[7]	0+4j

Figure 12: Sequence of the spectrum of the DFT example

The frequency resolution of the frequency bins is the sampling frequency divided by the number of samples. In our case, we have 8 Hz of sampling frequency divided by 8 samples, so 1 Hz per frequency bin.

To draw the spectrum, we can obtain from the sequence obtained magnitude and phase. We are going to focus in the magnitude of the signal. For that, we can compute the magnitude of the frequency bins by:

$$|X_k| = \sqrt{Re\{X_k\}^2 + Im\{X_k\}^2}$$

Which give us the same magnitude for X_1 and X_7, that is 4.

If we analyze the result, we can see that we are facing a two-sided frequency plot. We have the Nyquist limit at the half of sampling frequency and for that reason we have a symmetric result. For that reason, in order to maintain the energy, we can twice the values and get rid of those values above this limit. Doing that, we only have one frequency bin with value -8j.

But, even knowing that the first frequency bin corresponds to 1 Hz, that is the frequency of our example, we have a magnitude of 8. This is because we use 8 points to obtain the discrete spectrum of the signal and for that reason, we have to average it out over the 8 samples.

In addition to this, if we analyze the phasor obtained by this first frequency bin, we can see that the phase of -8j is -π/2 or 3π/2. If we delay π/2 the first frequency bin cosine function of amplitude 1 (averaged), what we obtain is exactly the initial input function at 1 Hz.

2.3.- Fast Fourier Transform

There are several algorithms that implements the DFT in an efficient way. These are called Fast Fourier Transform algorithms. In this section, we are going to study the algorithm implemented in the system. As we saw previously, the DFT transforms the input samples of a signal x[n] with length L, into an output samples of that input signal spectrum X[k] with period N or also called of N points. Also, we saw that for avoid temporary overlap we have to meet N ≥ L. We will obtain an algorithm in the way:

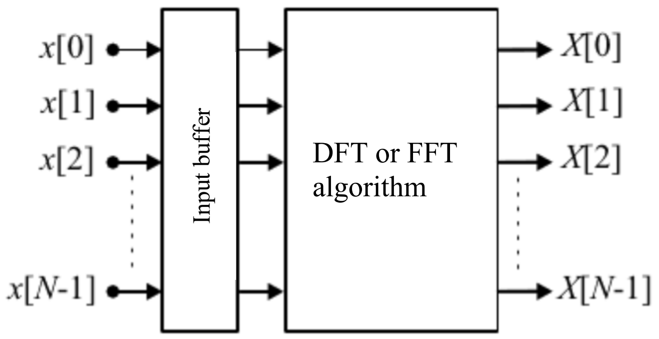


Figure 13: DFT or FFT block diagram

We can express the DFT analysis equation changing the notation to simplify this section:

$$F_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}$$

With k = 0, 1, …, number of samples and also for n = 0, 1, …, number of samples.

If we analyze the algorithm complexity, we can see the number of operations. In the general case, x_n is a complex signal, and to obtaining the N samples of F_k according to the analysis equation we need N^2 complex multiplications and N·(N-1) complex additions. In addition to this, apart from the truncate and rounding operations of microprocessors, they work with real numbers. Each complex multiplication corresponds to 4 real products and 2 real sums, and each complex addition correspond to 2 real sums. The necessary operations are, then, $4N^2$ real multiplications and N·(4N-2) real additions. By counterpart, the objective of the FFT is to achieve a number of operations proportional to $N \log_2 N$, that for great numbers, the numbers of operations are drastically reduced.

The trick to speed up the DFT is to take advantage of the periodic nature of sinusoids. We can firstly divide up the DFT in an even index summation and in an odd index summation. This was discovered by J. W. Cooley and John Tukey and is used in the denominated Cooley-Tukey FFT algorithm [10].

$$F_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}$$

$$\swarrow \qquad\qquad \searrow$$

$$F_k = \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi k(2m)}{N}}}_{\boxed{\begin{array}{c}\text{Even index}\\ (x_0, x_2, x_4, \dots)\end{array}}} + \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi k(2m+1)}{N}}}_{\boxed{\begin{array}{c}\text{Odd index}\\ (x_0, x_1, x_3, \dots)\end{array}}}$$

Now we have the result of DFT in two smaller summations, each one with the half size of the original. Now, we move the two in the numerator that multiplies *m* to denominator.

$$F_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi km}{N/2}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi k(m+1/2)}{N/2}}$$

And we can simplify the odd index term by distributing out the constant that we can move to the front of the odd summation.

$$F_k|_{odd} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi k(m+1/2)}{N/2}} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}-j\frac{\pi k 1/2}{N/2}} = C_k \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}}$$

With $C_k = e^{-j\frac{2\pi k}{N}}$.

If we take a look at the exponentials of even and index terms, they look identical:

$$F_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi km}{N/2}} + C_k \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}} = E_k + e^{-j\frac{2\pi k}{N}} O_k$$

There is something tricky about this particular exponential if we expand it applying the Euler identity, and taking into account that *k* has integers values from 0 to N:

$$e^{-j\frac{2\pi km}{N/2}} = \cos\left(\frac{-2\pi km}{N/2}\right) - j\sin\left(\frac{-2\pi km}{N/2}\right)$$

When the *k* value is larger than N/2 we observe that:

$$\cos\left(\frac{-2\pi(N/2 + r)m}{N/2}\right) \; ; \; r \coloneqq k - N/2 \quad ; \quad r: 1, 2, \dots, N/2$$

And we can distribute the numerator:

$$\cos\left(\frac{-2\pi m\, N/2 - 2\pi mr}{N/2}\right) = \cos\left(-2\pi m - \frac{2\pi mr}{N/2}\right) \quad ; \quad m: 0, 1, 2, \dots, N/2$$

Every time you add 2π to the operand of the cosine, we obtain simply the cosine function without the 2π multiple. So, every time the *k* value is larger than N/2 it is simply the that *k* value minus N/2. This is the called symmetry identity, and it applies to the cosine and sin.

Symmetry identity:

$$\cos\left(-\frac{2\pi km}{N/2}\right) = \cos\left(-\frac{2\pi(\frac{N}{2}+k)m}{N/2}\right)$$
$$\sin\left(-\frac{2\pi km}{N/2}\right) = \sin\left(-\frac{2\pi\left(\frac{N}{2}+k\right)m}{N/2}\right)$$
$$k: 0, 1, \dots, N$$

If we observe the periodicity of the exponential, we can express $X_{k+\frac{N}{2}}$ in terms of E_k and O_k:

$$X_{k+\frac{N}{2}} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi\left(k+\frac{N}{2}\right)m}{N/2}} + e^{-j\frac{2\pi}{N}\left(k+\frac{N}{2}\right)} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi\left(k+\frac{N}{2}\right)m}{N/2}} =$$
$$= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi km}{N/2}} e^{-j2\pi m} + e^{-j\frac{2\pi}{N}k} e^{-j\pi} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}} e^{-j2\pi m} =$$
$$= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi km}{N/2}} - e^{-j\frac{2\pi}{N}k} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}} =$$
$$= E_k - e^{-j\frac{2\pi}{N}k} O_k$$

So, we can rewrite the frequency bins in the form:

$$X_k = E_k + e^{-j\frac{2\pi}{N}k} O_k$$
$$X_{k+\frac{N}{2}} = E_k - e^{-j\frac{2\pi}{N}k} O_k$$
$$E_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi km}{N/2}} \quad ; \quad O_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}}$$

Everything repeats after k=N/2, and for that reason the total number of operations effectively reduces in one half. We can repeat this process and divide each of the even and odd index summations into their own even and odd index summations. Every time we divide the summations, we halve the number of operations in the algorithm.

These final expressions will be called again in the system algorithm explanation. The way the code is implemented is related to this expression of the Cooley-Tukey Fast Fourier Transform.

2.4.- Microcontroller [11]

A microcontroller is just a small computer on a single integrated circuit. It could contain one or more CPUs, along with memory and some I/O peripherals. Program memory is often included in the system as well as small amount of memory RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessor used in personal computers.

They are usually used in automatically controlled applications. By reducing the size and cost compared to a design that uses separate microprocessors, memory, and I/O devices, microcontrollers make economical to digitally control even more applications, processes and devices. Nowadays is usually to find a mixed microcontroller, which has an analog part as well as digital one.

2.4.1.- Embedded design

A microcontroller can be considered a self-contained system with processor, memory and peripherals. While some embedded systems are very sophisticated, many have minimal requirements for memory and program length, with no operating system, and low software complexity.

Microcontrollers must provide real-time response to several events. When certain event occurs, a signal can activate some part of the program at the very moment when it happens. Possible interrupt sources are device dependent, and often include events such as an internal timer overflow, completing an analog to digital conversion, a button being pressed, or a data received on a communication link as UART.

Typically, microcontroller programs must fit in the available on-chip memory. Compilers and assemblers are used to convert the high-level programming languages into a compact machine code for storage in the microcontroller memory. There are several types of memory where the microcontroller can store the program. Usually, they have an EEPROM and flash memory which are easy to use and cheap to manufacture.

2.4.2.- Direct Memory Access (DMA)

Direct Memory Access is a method that allows an input and output device to send or to receive data directly to or from memory, bypassing the CPU to speed up memory operations. The process is handled by the chip named DMA controller (CDMA). There can be several DMA channels, allowing to some peripheral to send or receive data from or to memory in parallel. The DMA channel allow to the CPU to do another task instead of waiting the transfer operation. Once the transfer to memory is completed, an interrupt occurs saying to the CPU that the transfer is done.

2.5.- The C programming language

The C programming language is general purpose programming language developed by Dennis Ritchie between 1969 and 1972 in Bell Laboratories. It is a programming language oriented to the implementation of Operative Systems, UNIX specifically. This programming language is appreciated due to its efficiency in the code it produces, and it is the most popular programming language to create system's software.

It has the typical structures of a high-level programming language as well as constructions that allow us the control in a low level. Compilers usually have extensions that allow mix assembler code wit C code or direct access to memory or peripherals devices.

C is a common language to programming embedded systems. The light code that a C compiler generates, with the capability to access software layers near to the hardware are the causes of its popularity. A C characteristic that justify its use convenience in embedded systems is the bits manipulation. The systems contain memory mapped registers (MMR), through which peripherals are configured. These registers mix many configurations in the same memory address, tough different bits. In C you can change easily one of those bits without change the rest ones.

A C library is a set of functions in C programming language. The most common used libraries are the C standard library and the ANSI C library, which provides the specifications of the standard that are widely shared between libraries, as input and output files functions, memory hosting and common date operations.

As many programs are written in C language, there is a wide variety of libraries available. Some of these libraries are written in C due that C generates object code quickly, and then programmers generate library interfaces to interact with them with routines coded in a higher-level language such as Java, Perl or Python.

2.5.1.- Use of memory

We have to distinguish two types of memory: dynamic memory and static memory. The first one, is the memory reserved in execution time. Its main advantage is that its size can vary during the program execution. This use of memory is needed when the programmer does not know the exact amount of data to handle. The second one, is the amount of memory created at declaring any type of variable. The amount of memory occupied by these variables cannot change in execution time neither be released manually.

The dynamic memory usually is at heap, and the static memory in the stack or in a specifically part of the memory. The static memory has a size fixed known in compile time.

All the objects created in C have a limited time in memory. There are three types of duration: static, automatic and assigned. The global variables and local variables declared with the keyword "*static*" have static duration. They are created before the program starts its execution and they are destroyed when the program ends. The local variables have an automatic duration. They are created when the program goes inside the memory block where they were created, and they are eliminated when the program returns back from this program block. Assigned duration refers to the memory are created una dynamic way with the mechanism that C has.

2.5.2.- Recursivity [12]

A function is "recursive" when in its execution it makes a call of itself. Each time the function calls itself, the previous calling remains not ended and the "recursion level" increases in one unit. Every recursive function needs an input argument or an internal variable whose value varies in each entry. This happens until the "end of recursion" condition is reached. At this moment, all the pendant calls of the function begin to finish.

Each new call to the function generates in the stack new instances of the local variables and input arguments of the function. If too many of this called are executed, the stack may get full and an error might occur. There are compilers that allow to the user the configure of the stack, facilitating tools to have a bigger space for the stack. It is convenient to apply recursive solutions only when the problem to solve can be clearly defined in a recursive way. In most of the cases, it is convenient to use iterative solutions with loops, which do not demand so much amount of memory.

2.6.- Communication protocols [13]

2.6.1.- UART

The asynchronous serial communication is one of the simplest ways to communicate one microcontroller to any peripheral device with only two communication wires and the voltage reference. There are only needed the transmission line (Tx) and the reception line (Rx).

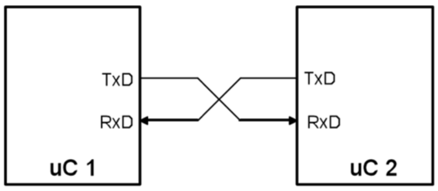


Figure 14: Interconnection scheme between two µC for UART communication

This type of communication is characterized by the not needing of a clock signal. The transmission of a data is realized sending and receiving a frame through the previous ports. Assumed that the resting state is at high level, the Tx/Rx of each frame starts with the START bit (logic ‘0’). Then, the data is sent beginning with the least significant bit (LSB), that could have a variable length, but usually it is worked with 8 bits. After this, it is possible to working with a parity check, sending the parity bit, and ending the transmission with the STOP bit with a high logical level. For example, to send the ASCII code of the letter ‘A’, with an even parity, we could have the format shown in Figure 15.

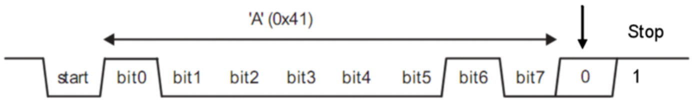


Figure 15: UART example with ASCII code of 'A'

The Figure 15 0 corresponds to the even parity because it represents the numbers of logical ‘1’ present in the frame. In the example there is a low-level active bit because character ‘A’ has two logical highs. In order to send more frames in the same communication, after the STOP bit, the process repeats itself.

2.6.2.- I2C

The I2C protocol has a more complex architecture. The I2C bus was developed in the late 1970’s for Philips consumer products. It is composed by two wire bus: Serial Data Line (SDA) and Serial Clock Line (SCL). It is important to mention that the device need to have an open-drain or open-collector output stages, implementing the wired-AND function.

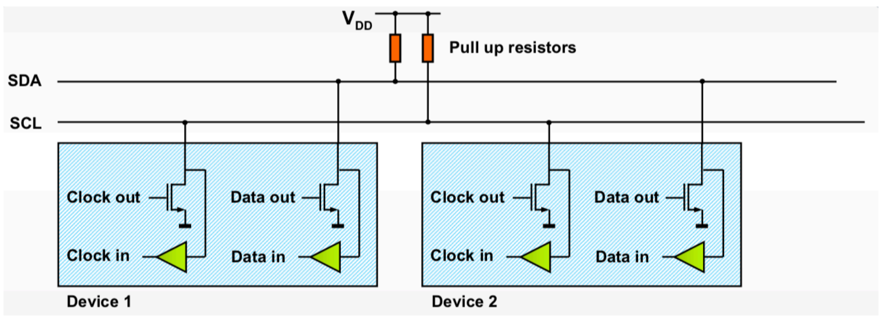


Figure 16: I2C hardware architecture

In the architecture, it could be multiple masters and slaves, implementing a bi-directional communication between them:

- Master-transmitter
- Master-receiver
- Slave-transmitter
- Slave-receiver

Data collision is taken care off. The Master and Slave roles have to configure the architecture. The master is the device that starts the communication between peripherals. Each device is addressed individually by software. The address is unique per device, it can be fully fixed or with a programmable part through hardware pins.

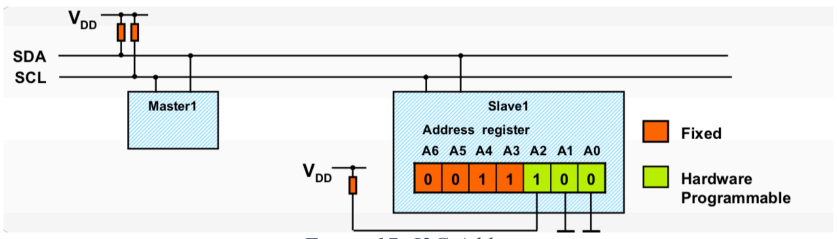
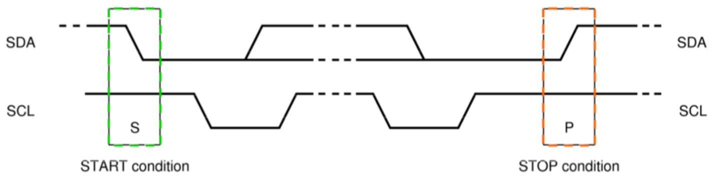


Figure 17: I2C Address

The communication must start with the START condition, and ends with the STOP condition:

- Start condition - a HIGH to LOW transition on the SDA line while SCL is HIGH
- Stop condition - a LOW to HIGH transition on the SDA line while SCL is HIGH



The start bit is always followed by the slave address. The slave address is followed by READ or NOT-WRITE bit that indicates if the master is going to send or to receive data. The receiving device of the frame, after the READ bit must send an ACKNOWLEDGE bit. After all this process, the communication starts sending the corresponding data.

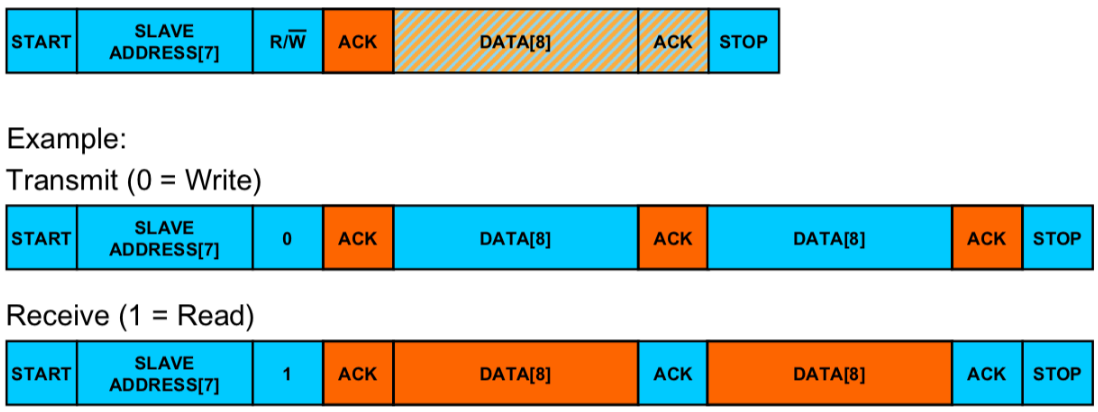


Figure 18: I2C example

During the transfer, SDA must be stable when SCL is high. Each byte has to be followed by an acknowledge bit. The number of data bytes transmitted per transfer is unrestricted. If a slave cannot receive or transmit another complete byte of data because an interrupt for example, it can hold the clock line SCL in low (clock stretching) to force the master into a wait state.

Data transfer with acknowledge is obligatory. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse.

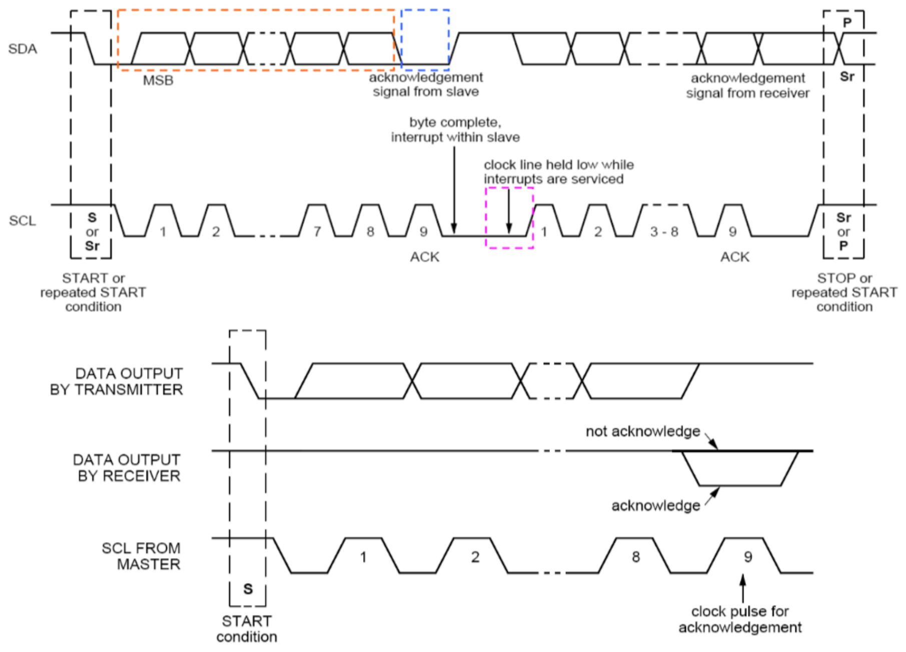

SDA
SCL
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
byte complete, interrupt within slave
clock line held low while interrupts are serviced
S or Sr
P
Sr
Sr or P
ACK
START or repeated START condition
STOP or repeated START condition
DATA OUTPUT BY TRANSMITTER
DATA OUTPUT BY RECEIVER
not acknowledge
acknowledge
SCL FROM MASTER
START condition
clock pulse for acknowledgement

Figure 19: Clock stretching and ACK bit in I2C communication example

2.6.3.- SPI

Serial Peripheral Interface (SPI) is a 4-wire full-duplex synchronous serial data link. The wire lines are Serial Clock (SCLK), Master Out Slave In (MOSI) which means a data from master to slave, Master In Slave Out (MISO) which means data from slave to master, and Slave Select (SS).

It was originally developed by Motorola. It is used for connecting peripherals to each other and to microprocessors. It is composed by shift register that serially transmits data to other SPI devices. The difference between the previous protocols is that here, we need "3+n" wire interface with 'n' number of devices. This means that the SCLK, MOSI and MISO wires are connected in parallel between all the devices in the net, and from the master, there are as many wire lines as devices, each one corresponding to the chip select of each one of them.

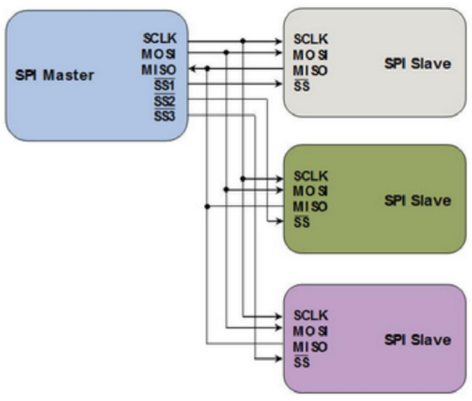


Figure 20: SPI scheme connection

The data frame is sent synchronously with the clock pulses, as in I2C. Before the transmissions starts, the master puts in slow the chip select line of the corresponding device.

Four communication modes are available. Basically, the SCLK edge on which the MOSI line toggles, the SCLK edge on which the master samples the MISO line and the SLCK signal steady level (that is the clock level, high or low, when the clock is not active). It is important to mention that the clock signal only toggles when a data has to be sent. Each mode is formally defined with a pair of parameters called 'clock polarity' (CPOL) and 'clock phase' (CPHA).

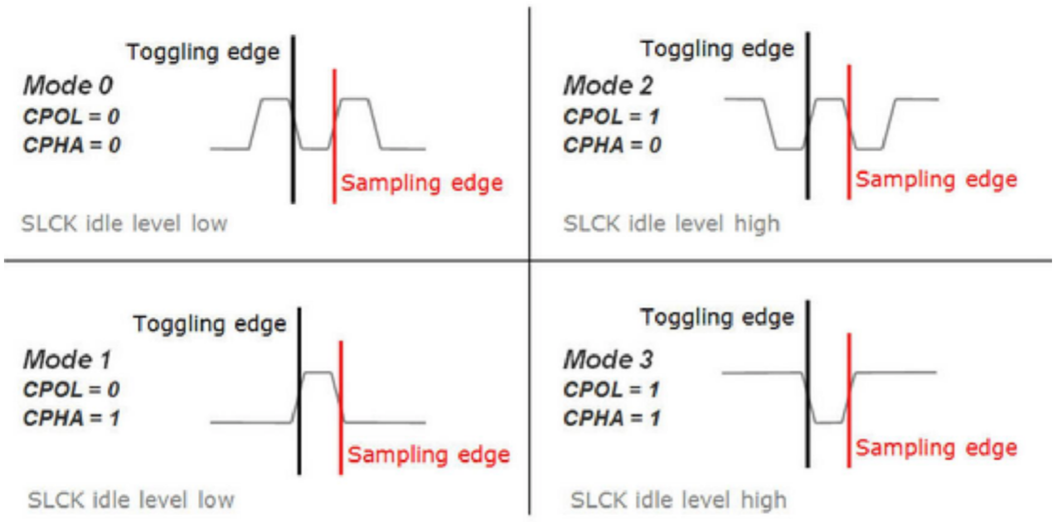


Figure 21: SPI clock polarity and clock phase example

3.- Hardware overview

3.1.- Microcontroller used

Arduino Due is a microcontroller board based on the Atmel SAM3X8E ARM Cortex-M3 CPU. With an 84 MHz core clock, the microcontroller has 54 digital I/O where we use only 9, 12 analogue inputs where we use only one for the antenna signal, 4 UARTs where we use one for debugging and another one to communicate with ESP8266 module, SPI header used for communications between SD card and screen and 2 I2C where we use one for RTC communication. The board presents more characteristics, but they will not be covered in this memory.

It runs at 3.3 V, so the maximum voltage that I/O pins and analog inputs can tolerate is 3.3 V. This means that the signal of the antenna cannot exceed 3.3 V. A safety circuit should be applied in the preamplifier to avoid this undesirable situation.

Figure 22: Arduino Due board [14]

The microcontroller has a flash memory of 512 Kbytes, composed of two blocks of 256 Kbytes each, and 64 + 32 Kbytes of SRAM.

It can be powered by a USB cable or through the 5 V DC Jack input. In a debug mode, it can be powered through the USB cable and also make a UART communication with the computer in order to see the commands created for this purpose.

3.2.- Internal ADC

The ADC of the µC is a 12-bit ADC managed by an ADC Controller. It integrates 16-to-1 analog multiplexer, that makes possible the conversion of 16 channels. One channel is reserved for internal temperature sensor. The ADC supports a 10-bit or 12-bit resolution mode, and conversion results are reported in a common register for all channels, as well as in a channel-dedicated register. There are many ways to trigger it, but for this project, we will trigger it through software. The ADC also integrates a Sleep Mode and a conversion sequencer and connects with a PDC (DMA) channel. These features can reduce both power consumption and processor intervention, but for our system, these features are not used.

The ADC has a selectable single-ended or fully differential input and benefits from a 2-bit programmable gain. In our case, we use a single-ended input with no gain (gain = 1). We have to mention that, as we said in microcontroller part, it runs at 3.3. V, so the span is internally configured with an input Vref of 3.3 V. We can see as an example Figure 23from the µC manual:

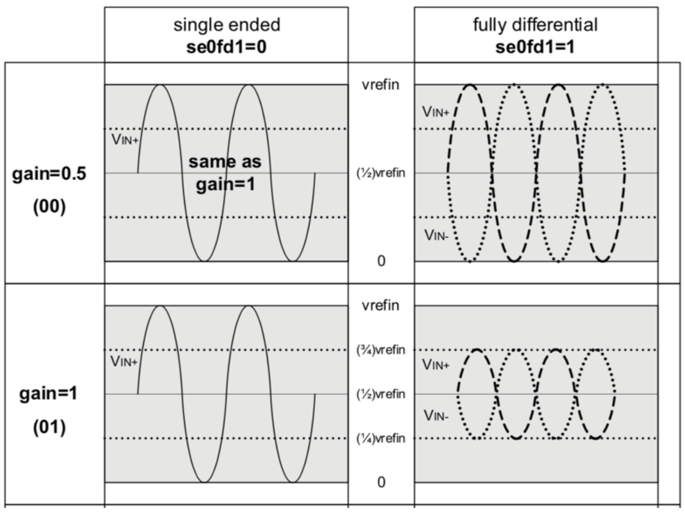


Figure 23: Single ended and fully differential ADC input modes [15]

In the single ended mode, the conversion is made with the voltage that the µC finds at the corresponding analog pin, referenced to ground. For that reason, we have to connect the µC ground to the antenna signal ground.

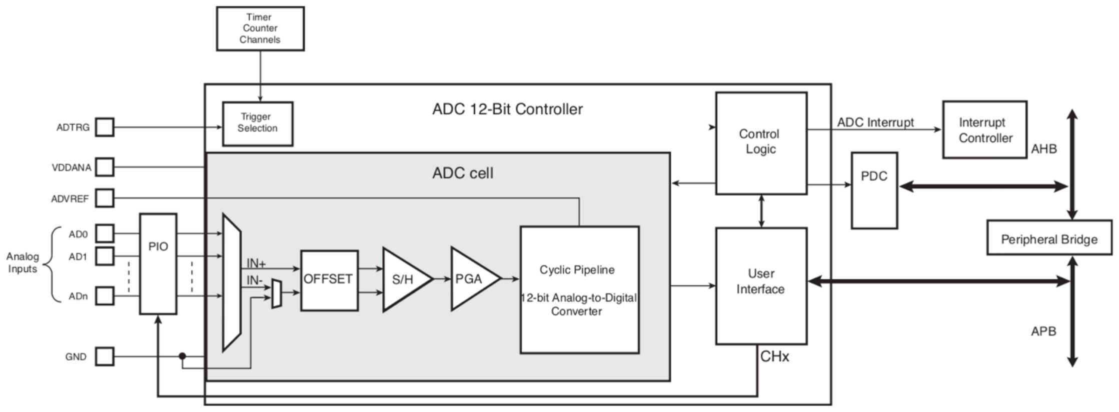


Figure 24: ADC of the microcontroller board used [15]

The use of the ADC is made with a library that the manufacturer gives to programmers. This library is written in C language and handle all the necessary signals that a conversion requires.

This ADC is a pipelined ADC seen in the applicable theory part as we can see in Figure 24. Without going deep in all the signals required to a conversion, we can see an example extracted for the manual to see all the steps in order to realize a conversion in each channel.

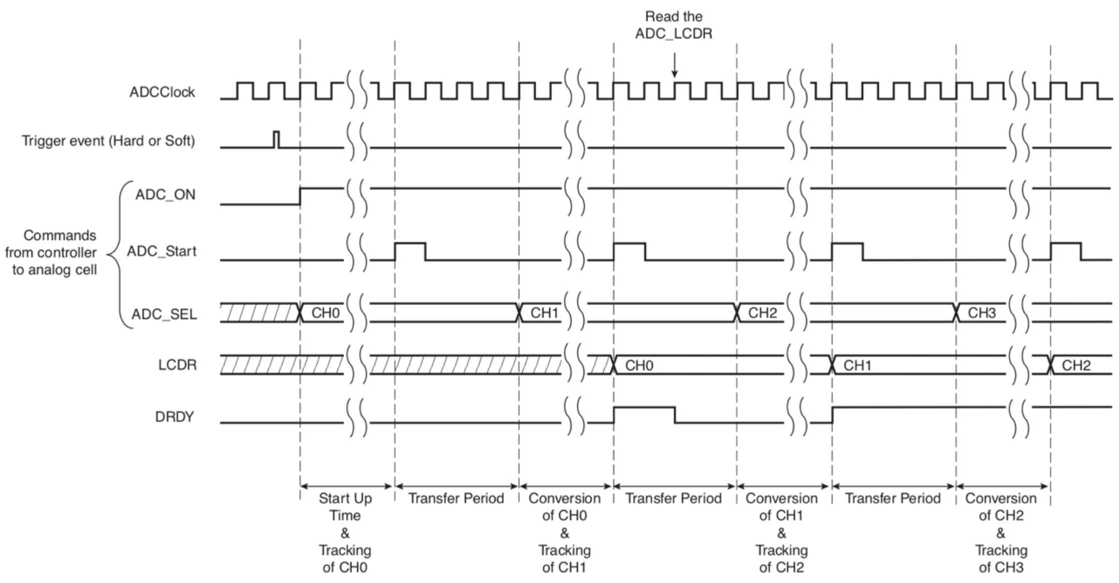

Read the ADC_LCDR
ADCClock
Trigger event (Hard or Soft)
Commands from controller to analog cell
ADC_ON
ADC_Start
ADC_SEL
CH0
CH1
CH2
CH3
LCDR
CH0
CH1
CH2
DRDY
Start Up Time & Tracking of CH0
Transfer Period
Conversion of CH0 & Tracking of CH1
Transfer Period
Conversion of CH1 & Tracking of CH2
Transfer Period
Conversion of CH2 & Tracking of CH3

Figure 25: Signals involved in an ADC conversion [15]

3.3.- TFT screen

The screen used is a 2.8-inch SPI module based on chip ILI9341, SKU: MSP2807. ILI9341 is a 262,144-color single-chip SOC driver for a-TFT liquid crystal display with resolution of 240RGBx320 dots, comprising a 720-channel source driver, a 320-channel gate driver, 172,800 bytes GRAM for graphic display data of 240RGBx320 dots, and power supply circuit. ILI9341 supports parallel 8-/9-/16-/18-bit data bus MCU interface, 6-/16-/18-bit data bus RGB interface and 3-/4-line serial peripheral interface (SPI).

The moving picture area can be specified in internal GRAM by window address function. The specified window area can be updated selectively, so that moving picture can be displayed simultaneously independent of still picture area.

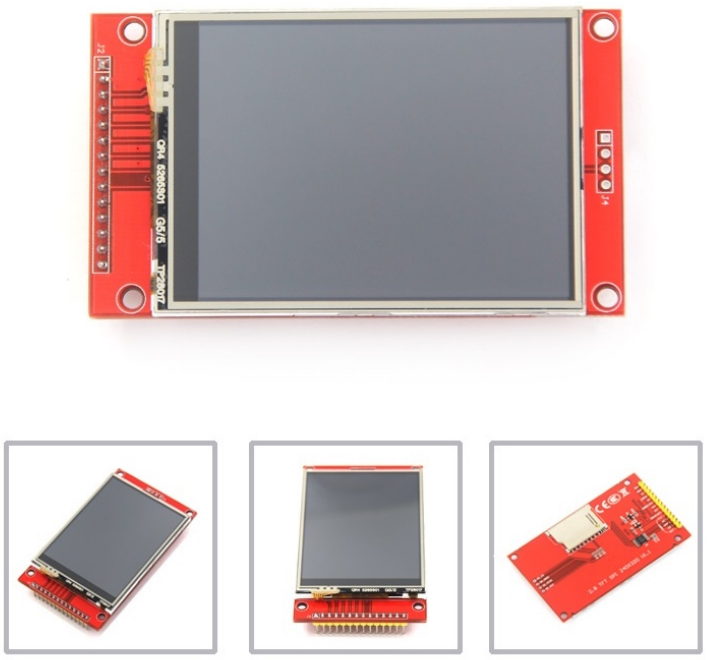
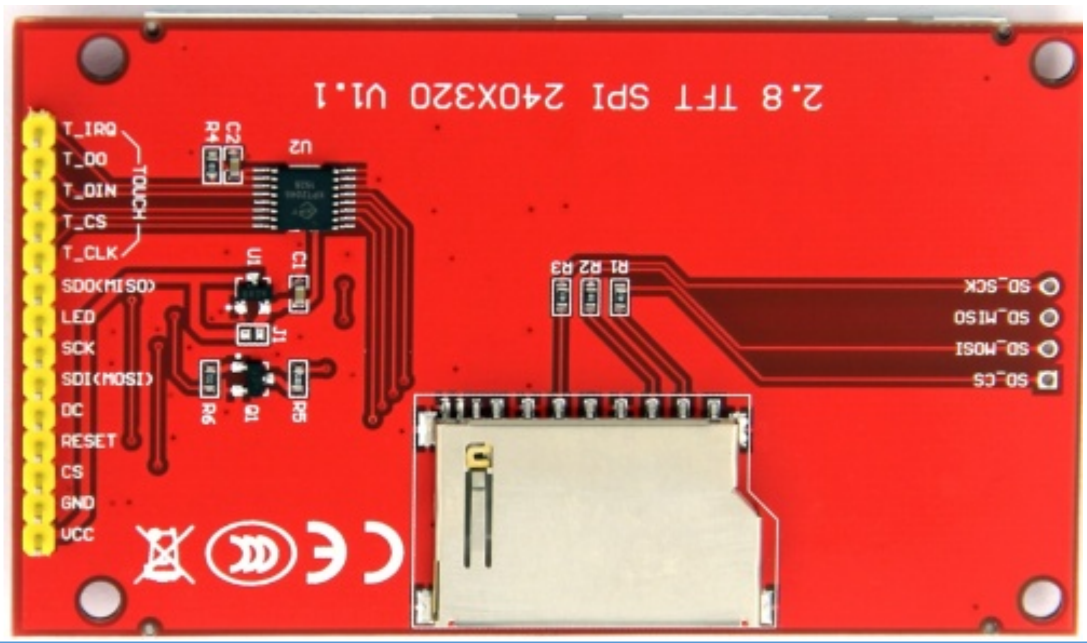


Figure 26: Screen of the system [16]

This screen is handled by a C-library that will be explained in the software part of this memory. The PCB of the screen has a SD slot in order to use SD card with SPI protocol also. This screen also counts with a touch screen which lies under the TFT screen. From the manufacture, we have the following figure and table which resumes all the pins on the screen and their use. At the end of the memory, we will see the actual schematic of the system, with all the pins connected with all the hardware. On the right side of the figure, we can see the SD card connections.



Number	Pin Label	Description
1	VCC	5V/3.3V power input
2	GND	Ground
3	CS	LCD chip select signal, low level enable
4	RESET	LCD reset signal, low level reset
5	DC/RS	LCD register / data selection signal, high level: register, low level: data
6	SDI(MOSI)	SPI bus write data signal
7	SCK	SPI bus clock signal
8	LED	Backlight control, high level lighting, if not controlled, connect 3.3V always bright
9	SDO(MISO)	SPI bus read data signal, if you do not need to the read function, you cannot connect it
(The following is the touch screen signal line wiring, if you do not need to touch function or the module itself does not have touch function, you can not connect them)		
10	T_CLK	Touch SPI bus clock signal
11	T_CS	Touch screen chip select signal, low level enable
12	T_DIN	Touch SPI bus input
13	T_DO	Touch SPI bus output
14	T_IRQ	Touch screen interrupt signal, low level when touch is detected

Figure 27: Screen connections

3.4.- Real-Time Clock (RTC)

Based on the DS3231, the DS3231 is a low-cost, extremely accurate I2C real-time clock with an integrated temperature compensated crystal oscillator (TCXO) and crystal. The device incorporates a battery input and maintains accurate timekeeping when main power to the device is interrupted. The integration of the crystal resonator enhances the long-term accuracy of the device as well as reduces the piece-part count in manufacturing line.

The PCB used incorporates the DS3231 chip and EEPROM ATC24C32 to supply a 32 Kbytes EEPROM to store data. In this project, this memory is not used because the system memory is the SD card in order to facilitate the physically data transport.

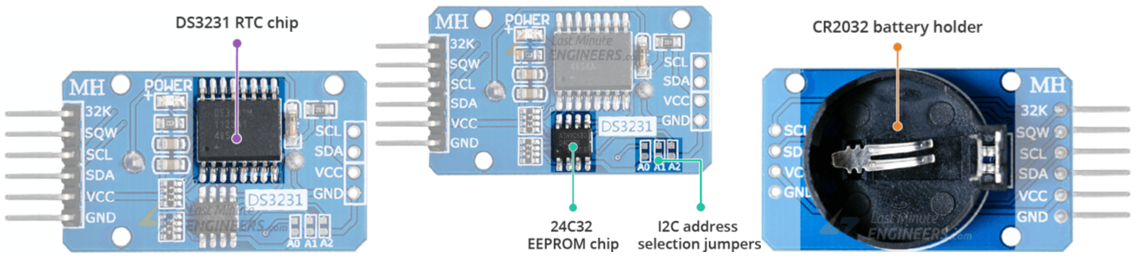


Figure 28: RTC PCB used in the system

3.5.- NodeMCU ESP8266

NodeMCU is a name that involves both Open Source firmware and ESP8266 based PCB. Nowadays, NodeMCU refers to the development board. This board is based on ESP12E.

Figure 29: NodeMCU ESP8266

The ESP12E is a miniature Wi-Fi module present in the market and is used for establishing a wireless network connection for microcontroller or processor. The core of ESP12E is ESP8266EX, which is a high integration wireless System on Chip (SoC). It features ability to embed Wi-Fi capabilities to systems or to function as a standalone application. It is a low-cost solution for developing IoT applications.

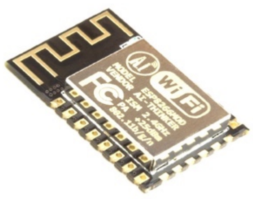


Figure 30: ESP12E

It has a serial communication interface in which the UART communication between the µC is performed, programmable GPIO, an SPI interface and, apart from this, is powered with 3.3 V also.

This board has been programmed through Arduino IDE and it will be covered in the software section of this memory.

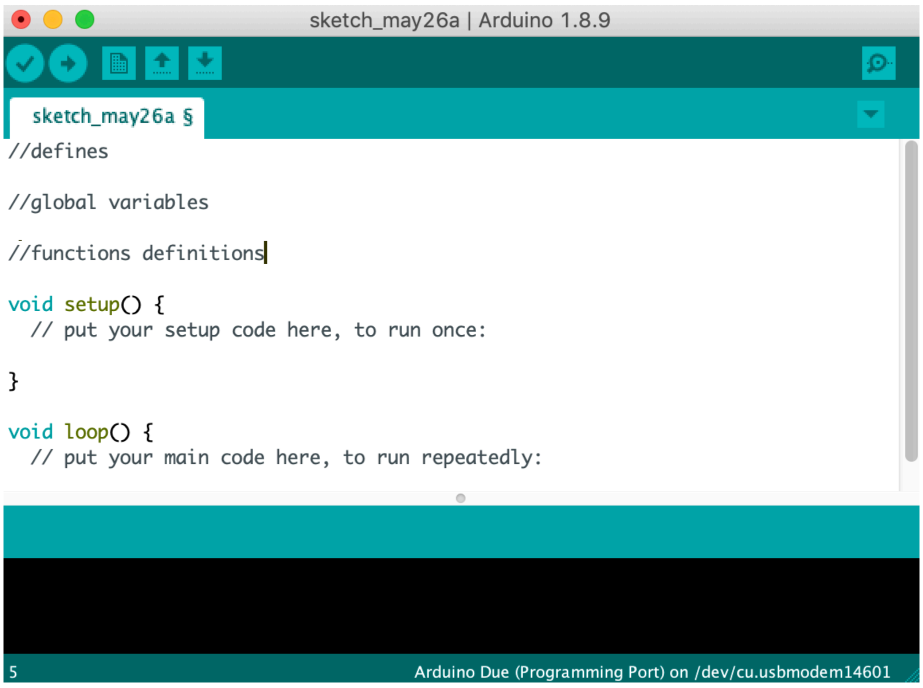
4.- Software overview

4.1.- Arduino IDE

The Arduino IDE is a cross-platform application that is written in the programming language Java. It is used to write and upload programs to Arduino compatible boards, but also, with the help of 3rd party cores, other vendor development boards. It supports the languages C and C++ using special rules of code structuring [17].

The Arduino IDE supplies a software library from the Wiring [18] project, which provides many common input and output procedures. User-written code only requires two basic functions, for starting the sketch and the main program loop, that are compiled and linked with a program stub *main()* into an executable cyclic executive program with the GNU toolchain, also included with the IDE distribution.

If we take a look at a typical program in Arduino, we have:


sketch_may26a | Arduino 1.8.9
sketch_may26a §

```
//defines

//global variables

//functions definitions

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
```

5
Arduino Due (Programming Port) on /dev/cu.usbmodem14601

Figure 31: Typical program structure in Arduino IDE

Where we can see, according to the Wiring project, the basics functions of *setup()* and *loop()*. The setup function is executed once, while the loop functions is iterative. This means that the loop function is executed over and over again while the microprocessor is powered on.

4.2.- Libraries used

4.2.1.- ILI9341_due [19]

This is an Arduino Due library for interfacing with ILI9341 SPI TFTs. Although this library can be used for other microcontroller thanks to its implementation, it can take advantage of the Arduino Due DMA. It can be configured in DMA SPI optimized mode which provides a very faster way to transfer data in comparison with the use without DMA. The library is based on 3 libraries:

- *Ili9341_t3* library from Paul Stoffregen [20]
- *SdFat* from Bill Greiman [21]
- *GLCD* from Michael Margolis and Bill Perry [22]

The first one was used as a base. This *ili9341_t3* library has various optimization for Adafruit's ILI9341 and GFX libraries. One class from *SdFat* library is used for utilizing Due's DMA in SPI transfers which provides the main speed boost. This library is also used to handle the file inside de SD card. From the *GLCD* library the *ili9341_due* library takes the *gText* class as a base for rendering custom fonts. The documentation of the library can be seen in the webpage of the author [16]. It is important to mention that the LCD is measured in pixels. The pixels have an x-coordinate and a y-coordinate. The pixel [0,0] is in the upper left corner of the screen.

4.2.2.- URTouch [23]

This library provides touch functionality to the system. It also uses the SPI library to communicate with the TFT. The library implements some functions that read the register of the TFT controller and handle the data to simplify the use of it. The documentation is within the library and it can be obtained from the author webpage. In our case, we will see the functions used for this project.

4.2.3.- ILI9341_due_Buttons [24]

Created by Graham Lawrence, it is an add-on library which allows to easily add buttons to the user interface. It is based on the *UTFT buttons* library, that is the predecessor of the *URTouch* library commented before. Its documentation is within the library downloaded.

4.2.4.- RTClib [25]

This library is a fork of the original Jeelab's RTClib library. It is implemented by Adafruit and it is a lightweight date and time library for JeeNodes and Arduino. It uses I2C communication between microcontroller and the device. The use of the library is very simple, and with the example we can know how to handle it.

4.2.5.- Including and using the libraries

At the beginning of the sketch, we have to include the libraries

```
#include <SPI.h>
#include <SdFat.h>
#include <ILI9341_due.h>
#include <URTouch.h>
#include <ILI9341_due_Buttons.h>
#include "SmallFont.h"
```

```
#include "BigFont.h"
```
The Fonts are needed in order to write text in the LCD screen. These are basically for an automation and simplicity in the code.

Also, we have some *defines* to simplifies the code.

```
#define Y_MAX 240                //Maximum pixels in Y axis
#define X_MAX 320                //Maximum pixels in X axis
#define Xo 53                    //Pixels for axis (0,0)
#define Yo 10

// LCD
#define TFT_RST 8
#define TFT_DC 9
#define TFT_CS 11
//SD
#define SD_CS 10
//Touch pannel
#define T_CLK 30
#define T_CS 28
#define T_DIN 26
#define T_DOUT 24
#define T_IRQ 22
```

The last 3 lines are the pins where RESET and DC pins of the LC screen are connected in Arduino Due, while the last one is the chip select of the SD card.

Once all of this is declared, we can start to create the objects needed to work with the peripherals in the Arduino IDE. We create the filesystem SD for the SD card and create the file that for the program it is called *logSIDSWAP*. Then, for the TFT we use the initialization function provided in the library with the pins necessary for the correct use of it. We do the same with the touch panel using all the pins connected in the initialization function. With these to objects created we can use the buttons library passing the memory address of the objects commented and for the last, we create the *rtc* object to handle the RTC device.

```
//File that we will save in the SD
SdFat sd; // set filesystem
SdFile logSIDSWAP;

// Use hardware SPI
ILI9341_due tft = ILI9341_due(TFT_CS, TFT_DC, TFT_RST);

URTouch myTouch(T_CLK, T_CS, T_DIN, T_DOUT, T_IRQ);

// Finally we set up ILI9341_due_Buttons :)
ILI9341_due_Buttons  myButtons(&tft, &myTouch);

RTC_DS3231 rtc;
```

To operate with the times that our device will give us, we have to create a time structure as the example does. We create two structures in order to operate with intervals of time

```
DateTime now;
DateTime after;
```

Also, it is important to mention that, the buttons library works with an integer when works with the button. For that reason, all the buttons in the system have to be declared as an integer. In the code we can see the definition of all of them as a global variable.
```
int backButton, dispButton, fftButton, logButton, selectFreqButton;
int saveButton, leftButton, rightButton;
```

To end with this section of using the libraries, we have to mention the differentiation between two areas in the LCD Screen.

```
gTextArea graphArea{Xo, Yo, X_MAX - Xo, 180};
gTextArea allArea{0, 0, X_MAX, Y_MAX};
```

They are just to select the working area when a graph is rendered. The *graphArea* is a square area smaller than the whole screen (*allArea*), that paints the signal or the FFT of the signal without changing the axis and the labels of the axis.

4.2.6.- Arduino-Core SAM libraries [26]

As we will see in the test section of the memory, it is needed to handle the ADC in a different way than the Wiring does. Using the ADC, just like any peripheral, is done by setting appropriate values to related register. We can see in the ADC section all the signals needed to handle the device, and in the manual, we can see the 32-bit 14 involved registers on its behavior. It can be done directly in program, but it is quite susceptible of errors. For that reason, Atmel provides libraries to make the task easier.

They are bundle in Arduino (\*%arduino%*\hardware\arduino\sam\system\libsam\\), but we can see in the references the webpage where they are allocated in order to see the structure of it. With this library, instead write directly in the registers we can use C functions with the appropriate parameters. There are two functions related with the ADC and we are going to explain them in this part.

```
/*Configure
ADC*****************************************************************
**/

void configureADC() {
  // Setup all registers
  pmc_enable_periph_clk(ID_ADC); // To use peripheral, we must enable clock
distributon to it
  adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST); //
initialize, set sampling frequency
  adc_disable_interrupt(ADC, 0xFFFFFFFF); //disable interrupt of theA ADC
  adc_set_resolution(ADC, ADC_12_BITS); //We use the available resolution of
the ADC
  adc_configure_power_save(ADC, 0, 0); // Disable sleep, always powered
  adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1); // Set timings -
standard values
  adc_set_bias_current(ADC, 1); // Bias current - maximum performance over
current consumption
  adc_stop_sequencer(ADC); // not using it
  adc_disable_tag(ADC); // it has to do with sequencer, not using it
  adc_disable_ts(ADC); // disable temperature sensor
  adc_disable_channel_differential_input(ADC, ADC_CHANNEL_7); // A0 is channel
7 of the ADC
  adc_configure_trigger(ADC, ADC_TRIG_SW, 1); // triggering from software,
freerunning mode
  adc_disable_all_channel(ADC);
  adc_enable_channel(ADC, ADC_CHANNEL_7); // just one channel enabled
}
```

Most of the parameters are by default, but we use them to make sure their status. In every function there are the corresponding comment explaining the intention of it. As resume, we enable the the peripherals clock and we use the maximum ADC working frequency. We will comment some aspects about that with the tests and results obtained. The analog input used for the project is A0, and that input does not correspond with the channel 0 of the ADC, instead is the channel 7. We trigger the ADC in our code and for that reason the trigger is set by software.

```
/*Sampling*****************************************************
*********/

void Sampling(double *sw, double *re, double *im)
{
  adc_start(ADC);
  for (int i = 0; i < N; i++) {
    while ((adc_get_status(ADC) & ADC_ISR_DRDY) != ADC_ISR_DRDY)
    {}; //Wait for end of conversion
    sw[i] = adc_get_latest_value(ADC); // Read ADC
  }

  adc_stop(ADC);

  for (int i = 0; i < N; i++) {
    //To store the voltage value in the array -> value*SPAN_ADC/(2^n-
1)
    sw[i] = sw[i] * 3.3 / 4095;
    re[i] = sw[i];
    im[i] = 0;
  }

}
```

The Sampling function receives the memory address of three variables, which is the destination array, and also two arrays in order to difference the real and imaginary part of the signal. Even this is apparently unnecessary, we use the data with two arrays to operate with complex values when we work with the FFT.

Once the ADC starts, the microcontroller waits actively to obtain the complete conversion of the sample. Once it is done, that is flagged by the corresponding register, we add the value to the destination array. This value is in bits, so, once the array is completed, we do in the real array the adequate operation (multiplying by q) to obtain the voltage value. The active wait has been chosen due to the system requirements. As the main purpose of the system is to graph data corresponding with a day, and the time needed to fulfill the array is 36µs as we will cover in the corresponding section, the needs of the system allow us to make this decision instead of working with interruptions.

In order not to intervene in the sampling, we compute the voltage value of each sample after the array is completed and the ADC stops.

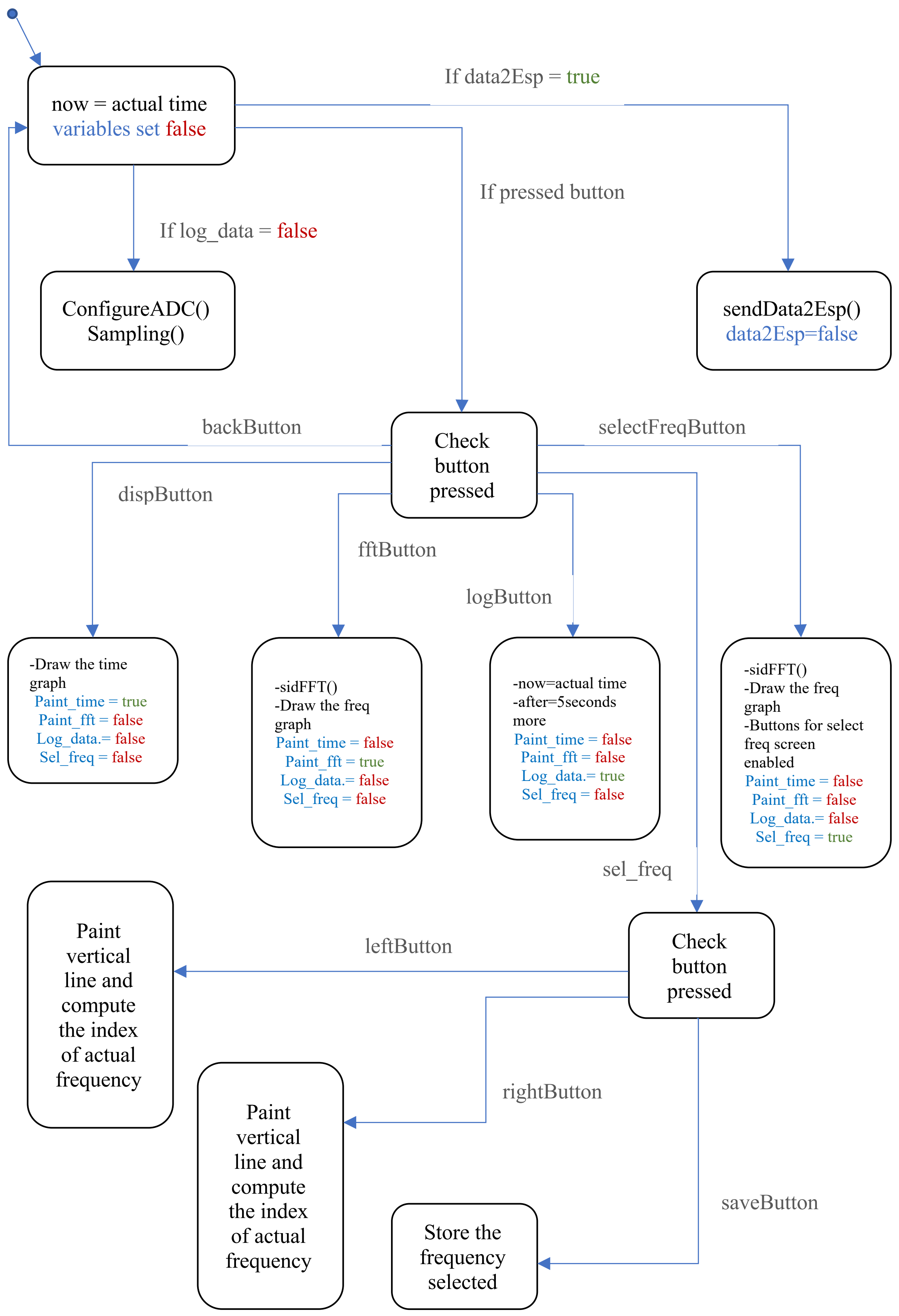
5.- Structure of the code

The structure of the code is basically:

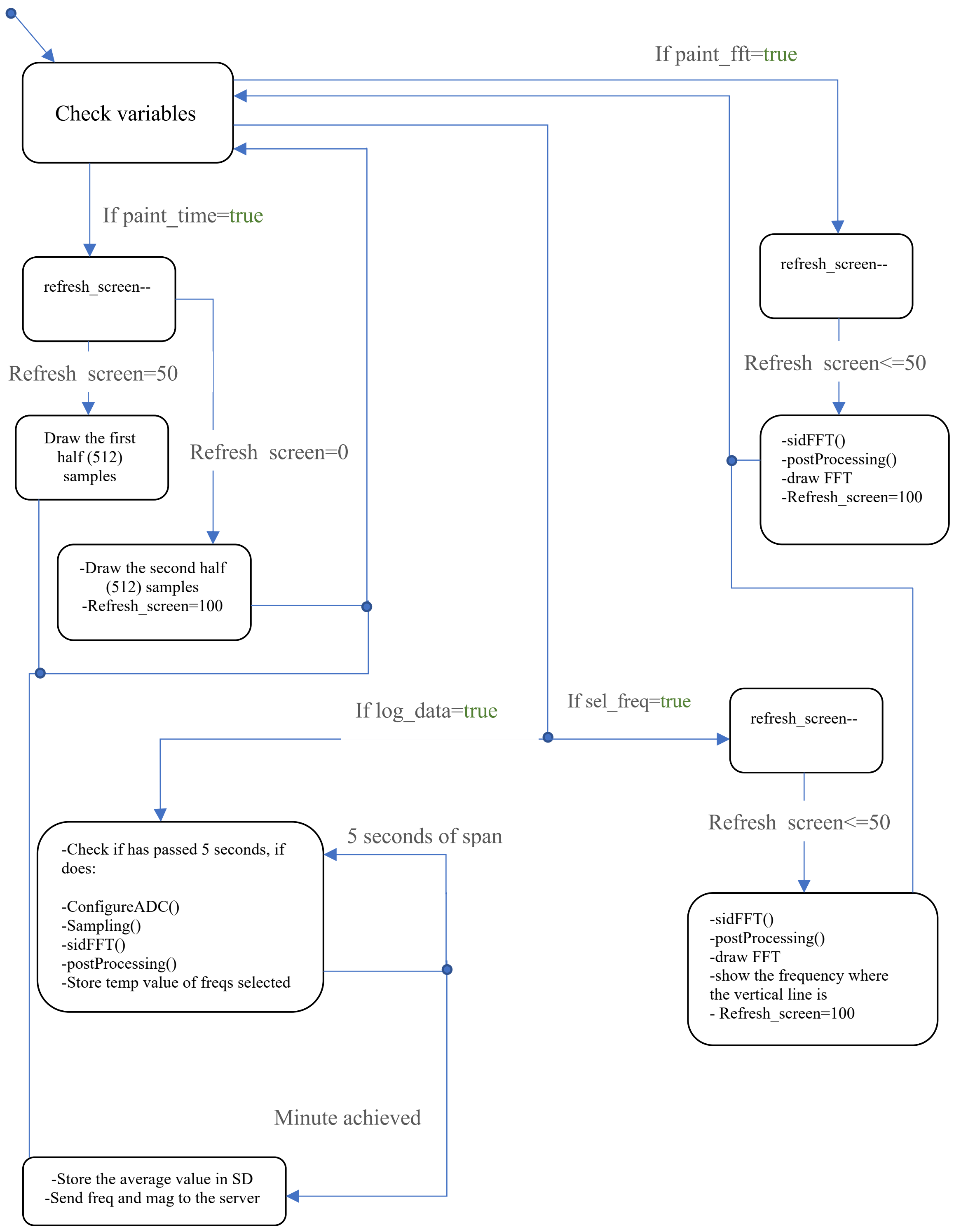
- Includes
- Defines
- Global variables
- Functions declarations
- Setup function that is executed once
- Loop function that implements two states machine to handle the system

To explain the states machine, we are going to use the following diagrams. Each part of the state machine will be covered in the explanation of the code. The first state machine handles the screen and the variables that are the base for the second state machine.

5.1.- State machine 1



5.2.- State machine 2



6.- Tests and results

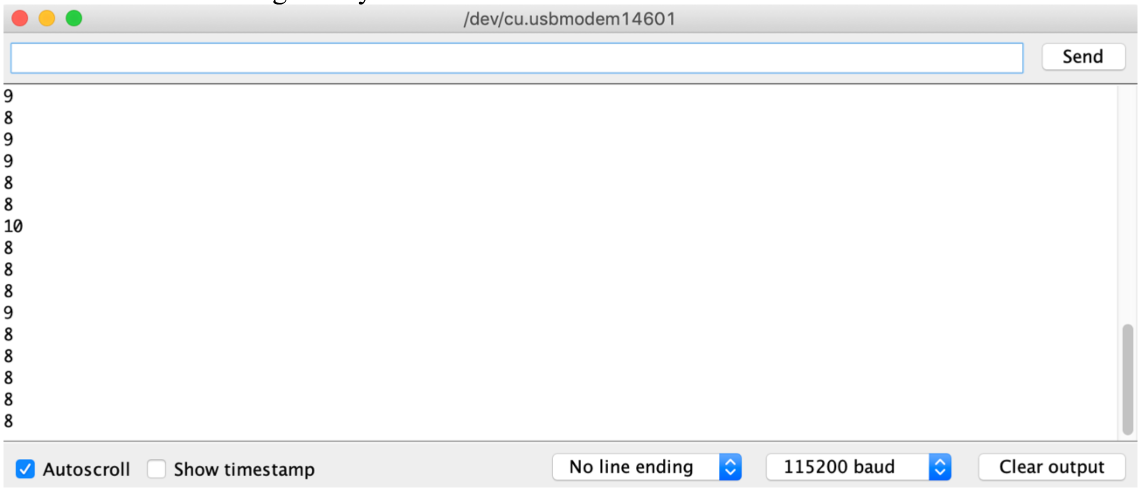
6.1.- ADC

6.1.1.- Test ADC 1

First, the basic test that we made in order to check the performance of the *Wiring* functions, it was coded the following simply sketch:

```
int input = A0;
int led = 13;
int val;
int time1, time2;
int count = 0;
void setup()
{
  pinMode(input,INPUT);
  pinMode(led,OUTPUT);
  Serial.begin(115200);
  delay(1000);
}
void loop()
{
  time1 = micros();
  digitalWrite(led,HIGH);
  val = analogRead(input);
  time2 = micros() - time1;
  digitalWrite(led,LOW);
  Serial.println(time2);
  count++;
  if(count==50) while(1);
}
```

Where we use the UART to see how much time spends the microcontroller in 50 conversions with the standard wiring library. The result was:

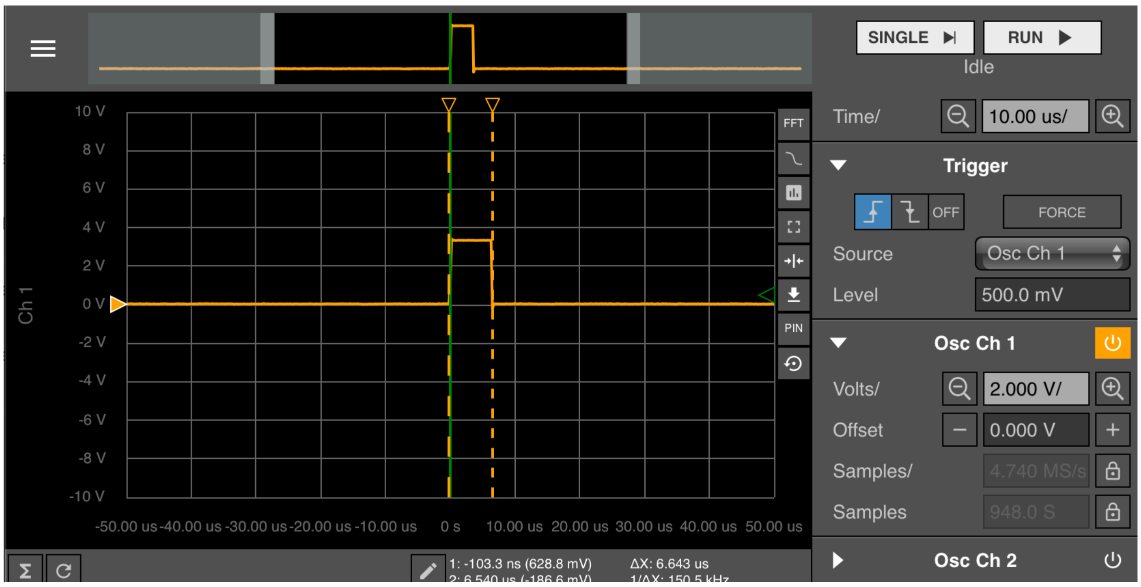


```
9
8
9
9
8
8
10
8
8
8
9
8
8
8
8
8
```

Figure 32: First test of ADC with wiring library

We can see a sampling period of average 8 µs. Which means a sampling frequency of 125 kHz which limits us to signals below 62,5 kHz. It is important to mention that Atmel says in the datasheet of the microcontroller that the ADC has a maximum sampling rate of 1msps, which is a maximum sampling frequency of 1 MHz. This difference between the manufacturer data and the one obtained with the standard Arduino library led us to make another decision about the ADC.

As sending data through the serial port spends some time, without sending data and just see with an oscilloscope [27] the output port 13, we observed:


SINGLE
RUN
Idle
FFT
PIN
Ch 1
10 V
8 V
6 V
4 V
2 V
0 V
-2 V
-4 V
-6 V
-8 V
-10 V
-50.00 us -40.00 us -30.00 us -20.00 us -10.00 us 0 s 10.00 us 20.00 us 30.00 us 40.00 us 50.00 us
Trigger
OFF
FORCE
Osc Ch 1
Osc Ch 2

Time/ 10.00 us/
Source Osc Ch 1
Level 500.0 mV
Volts/ 2.000 V/
Offset 0.000 V
Samples/ 4.740 MS/s
Samples 948.0 S

1: -103.3 ns (628.8 mV) ΔX: 6.643 us
2: 6.540 us (-186.6 mV) 1/ΔX: 150.5 kHz

Figure 33: ADC wiring test 2

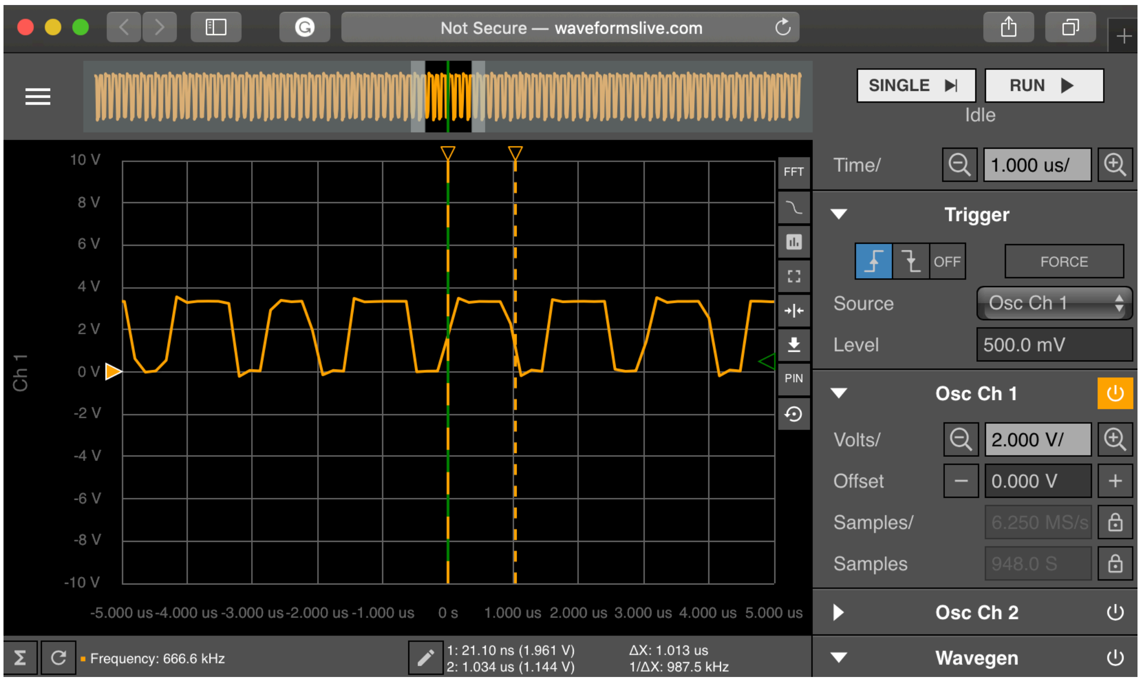
That is a sampling period of 6,643 µs, or 150 kHz of sampling frequency.

6.1.2.- Test ADC 2

In this test we used the Arduino-Core SAM library. The explanation of the code was made in the software part. Here the difference is in the way we put in high or low logical mode the corresponding pin. As the function *digitalWrite()* is slow, we put the pin high or low with the direct register operation through *PIO_Set()* and *PIO_Clear()*:

```
int input = A0;
int led = 13;
int val;

void setup()
{
  pinMode(input,INPUT);
  pinMode(led,OUTPUT);
  // Setup all registers
  pmc_enable_periph_clk(ID_ADC); // To use peripheral, we must enable
clock distributon to it
  adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST); //
initialize, set maximum posibble speed
  adc_disable_interrupt(ADC, 0xFFFFFFFF);
  adc_set_resolution(ADC, ADC_12_BITS);
  adc_configure_power_save(ADC, 0, 0); // Disable sleep
  adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1); // Set timings
- standard values
  adc_set_bias_current(ADC, 1); // Bias current - maximum performance
over current consumption
  adc_stop_sequencer(ADC); // not using it
  adc_disable_tag(ADC); // it has to do with sequencer, not using it
  adc_disable_ts(ADC); // deisable temperature sensor
  adc_disable_channel_differential_input(ADC, ADC_CHANNEL_7);
  adc_configure_trigger(ADC, ADC_TRIG_SW, 1); // triggering from
software, freerunning mode
  adc_disable_all_channel(ADC);
  adc_enable_channel(ADC, ADC_CHANNEL_7); // just one channel enabled
  adc_start(ADC);
}

void loop()
{
  while(1)
  {
    PIO_Set(PIOB,PIO_PB27B_TIOB0);
    while ((adc_get_status(ADC) & ADC_ISR_DRDY) != ADC_ISR_DRDY)
      {}; //Wait for end of conversion
    PIO_Clear(PIOB,PIO_PB27B_TIOB0);
    val = adc_get_latest_value(ADC); // Read ADC
  }
}
```

With this test, what we found on the oscilloscope was:


Not Secure — waveformslive.com
SINGLE
RUN
Idle
10 V
8 V
6 V
4 V
2 V
0 V
-2 V
-4 V
-6 V
-8 V
-10 V
Ch 1
-5.000 us -4.000 us -3.000 us -2.000 us -1.000 us 0 s 1.000 us 2.000 us 3.000 us 4.000 us 5.000 us
FFT
PIN
Trigger
OFF
FORCE
Osc Ch 1
Osc Ch 2
Wavegen

Time/ 1.000 us/
Source Osc Ch 1
Level 500.0 mV
Volts/ 2.000 V/
Offset 0.000 V
Samples/ 6.250 MS/s
Samples 948.0 S
Frequency: 666.6 kHz
1: 21.10 ns (1.961 V)
2: 1.034 us (1.144 V)
ΔX: 1.013 us
1/ΔX: 987.5 kHz

Figure 34: Sampling frequency of ADC with SAM library

In figure 34 we can see that the cursors are more or less in 1 µs of span. However, the oscilloscope says us that the effective frequency is 666,6 kHz. In any case, we can see how the difference between one or other library is perhaps exaggerated.

6.1.3.- Test ADC 3 with MATLAB

To see how we can know the real sampling frequency, we made a script in Arduino and in MATLAB to extract the information through the FFT implemented in MATLAB. The test consists on sampling 1024 samples and send them through UART to the MATLAB script. In order to do this, we create 8 different arrays of 128 samples each for the Arduino compiler to better manage the memory.

The microcontroller is constantly sending character ‘A’ through the UART waiting for the response for the master, which is the MATLAB script. Once the contact is established, and the master returns something in the buffer, the master waits for the command char ‘S’ to start the procedure.

When the user writes ‘S’ in the Command Window, in the microcontroller are performed the *configureADC()* and *sample()* functions. After this, the master sends ‘g’ through the UART that means the microcontroller can send a sample. The communication is performed sample by sample. After few seconds, the 1024 samples are obtained and a FFT is performed in MATLAB to see all the samples in time domain and in frequency domain to test the ADC behavior of our microcontroller.

Is important to mention that, in order to do the FFT correctly, we have to set the sampling frequency in the MATLAB script. This is the key to check the real sampling frequency of the ADC.

The script in Arduino was:

```
#define num_samples  128          // Number of samples in each array of
val
#define num_array_ADC 8           // Arrays of 128 values

uint16_t val[num_array_ADC][num_samples];            // Values from ADC
int input = A0;                  // Analog input for the antenna
//int led = 13;

uint8_t i = 0;                   // Typical general index for the
program
uint8_t ind = 0;               //  val[ind][i]
char inByte = 0;                 // Place holder for incoming character
from Matlab
char Tx_Serial = 'g';          // Character that, when received from
master, will
                                 // trigger a data TX.
char START_sample = 'S';       //Start conversion from ADC

uint16_t response = 0;            // For serial communication with
Matlab

void configureADC(void);
void sample(void);
void establishContact(void);
int getResponse(void);

void setup()
{
  //pinMode(input,INPUT);
  //pinMode(led,OUTPUT);
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(115200);
  establishContact();            // Send byte to establish contact
until master responds
  i = 0;
  //configureADC();
  //sample();
}

void loop() {
  if(Serial.available()) {
    inByte = Serial.read();                // Store the command byte

    if(inByte==START_sample) {
        configureADC();
        sample();
      }

     if(inByte == Tx_Serial) {             // Check to see if it matches 
the command char
        if(i<num_samples && ind<num_array_ADC) {
        response = getResponse(); // Get the response integer
        Serial.println(response); // Send it back to the master
        }
      }

    if(inByte == 'P') {
      i=0;
      ind = 0;
    }
  }
}

void establishContact() {
     while (Serial.available() <= 0) {
       Serial.println('A');   // send a capital A
       delay(300);
     }

     inByte = Serial.read();
 }

int getResponse(){
  int response = 50;
    response = val[ind][i];
    i++;
    if(i==num_samples) {
      ind++;
      i = 0;
    }
  return response;
}

void configureADC() {
  // Setup all registers
  pmc_enable_periph_clk(ID_ADC); // To use peripheral, we must enable 
clock distributon to it
  adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST); // 
initialize, set maximum posibble speed
  adc_disable_interrupt(ADC, 0xFFFFFFFF);
  adc_set_resolution(ADC, ADC_12_BITS);
  adc_configure_power_save(ADC, 0, 0); // Disable sleep
  adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1); // Set timings 
- standard values
  adc_set_bias_current(ADC, 1); // Bias current - maximum performance 
over current consumption
  adc_stop_sequencer(ADC); // not using it
  adc_disable_tag(ADC); // it has to do with sequencer, not using it
  adc_disable_ts(ADC); // deisable temperature sensor
  adc_disable_channel_differential_input(ADC, ADC_CHANNEL_7);
  adc_configure_trigger(ADC, ADC_TRIG_SW, 1); // triggering from
software, freerunning mode
  adc_disable_all_channel(ADC);
  adc_enable_channel(ADC, ADC_CHANNEL_7); // just one channel enabled
}

/*We have to control trough software the sampling frequency!!!!*/
void sample() {
  adc_start(ADC);
  i = 0;
  uint16_t value = 0;
  //Pin 13
  PIO_Set(PIOB,PIO_PB27B_TIOB0);
  while(i<num_samples) {
    while ((adc_get_status(ADC) & ADC_ISR_DRDY) != ADC_ISR_DRDY)
      {}; //Wait for end of conversion
    val[ind][i++] = adc_get_latest_value(ADC); // Read ADC
    //Serial.println(val[i++]);
    if(i==num_samples && ind<num_array_ADC-1) {
      i=0;
      ind++;
    }
  }
  adc_stop(ADC);
  PIO_Clear(PIOB,PIO_PB27B_TIOB0);
  i=0;
  ind=0;
}
```

And the script in MATLAB was:

```
%Number of samples in each array from arduino
num_samples = 128;
%Number of arrays with num_samples data
num_array = 8;

Fs=666600; % Configured sampling frecuency of arduino Due
Ts = 1/Fs;  % Sampling period
L = num_array*num_samples; % Length of the signal
t = (0:L-1)*Ts;% Time vector

delete(instrfind({'Port'},{'/dev/cu.usbmodem14601'}));

%Counter for num_samples
cont_samples1 = 1;
%Counter for num_array
cont_samples2 = 1;
%Array to store the data
y = zeros(num_array, num_samples);

portName = '/dev/cu.usbmodem14601';
Tx_Serial = 'g';

s1 = openSerialPort(portName);     % Open the serial port

START_sample = input("Send char 'S' to star sampling in Arduino Due:
");
fprintf(s1, '%s', START_sample);

while cont_samples1<=num_samples
```

```
    if (cont_samples1<=num_samples) && (cont_samples2<=num_array)
        value = getValue(s1, Tx_Serial)*3.3/4095; % Get a value from 
the device
        y(cont_samples2,cont_samples1) = value;
    %   disp(value);                        % Display it
        cont_samples1 = cont_samples1 + 1;
    end
    if (cont_samples1 > num_samples) && (cont_samples2<num_array)
        cont_samples1 = 1;
        cont_samples2 = cont_samples2 + 1;
    end
end


fprintf(s1, '%s', 'P');
closeSerialPort(s1);
plotData(y, num_array, num_samples, Fs, L);
```

With the functions created:

```
function serialPort = openSerialPort(portName)

%%%%%%%%%%%%%%%%%%%%%%%%%%% OPEN SERIAL PORT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

serialPort = serial(portName);          % define serial port
serialPort.BaudRate=115200;               % define baud rate
set(serialPort, 'terminator', 'LF');    % define the terminator for 
println
fopen(serialPort);

%%%%%%%%%%%%%%%%%%%%%%%%%
 ESTABLISH CONNECTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

w=fscanf(serialPort,'%s');
if (w=='A')
    fprintf(serialPort,'%s','A');      % establishContact just wants 
                                       % something in the buffer
end

return


function value = getValue(serialPort, commandChar)

fprintf(serialPort, '%s', commandChar);
value = fscanf(serialPort, '%d');

return


function closeSerialPort(serialPort)

fclose(serialPort);

return
```

With an input signal of 50kHz, 3,3 V peak to peak with an offset of 1,5 V (remember the limitation of the ADC), we obtained the following figures:

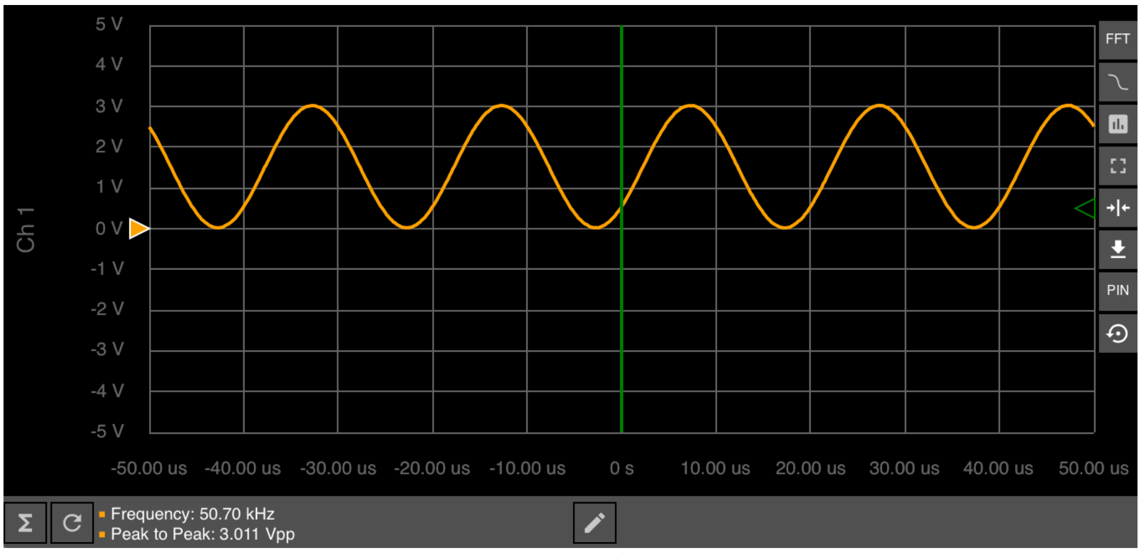


Figure 35: Input signal for ADC test 3

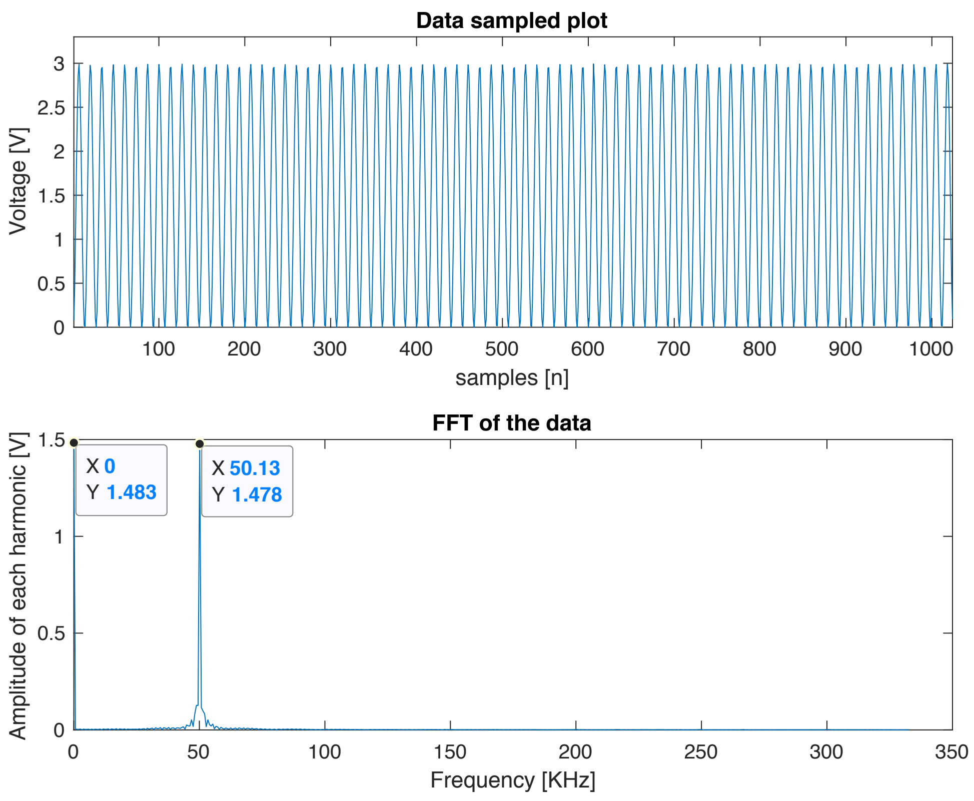


Figure 36: MATLAB result from figure 35

As we can see, using a sampling frequency of 666,6 kHz as the oscilloscope told us, matches with the input signal in our ADC. With these tests we can conclude that the SAM library works great for our purpose. This third test validates the correct sampling of an input signal and clarifies us the real sampling frequency of 66,6 kHz.

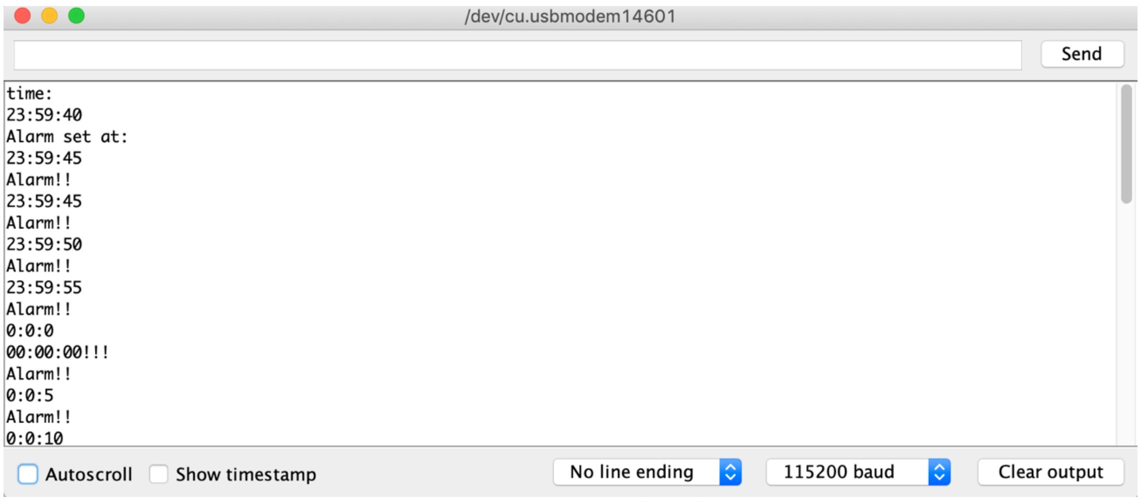
6.2.- RTC

The test of the RTC was extracted from the examples of the library with some modifications in order to test the span of 5 seconds and the start of a new day.

```
#include <Wire.h>
#include "RTClib.h"

RTC_DS3231 rtc;

DateTime now;
DateTime future;

char daysOfTheWeek[7][12] = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday"};
bool print00 = true;

void setup() {
  Serial.begin(115200);

  delay(3000);

  if (! rtc.begin()) {
    Serial.println("Couldn't find RTC");
    while (1);
  }

  if (rtc.lostPower()) {
    Serial.println("RTC lost power, lets set the time!");
    // following line sets the RTC to the date & time this sketch was
compiled
    rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
    // This line sets the RTC with an explicit date & time, for
example to set
    // January 21, 2014 at 23:59:40 you would call:
    //rtc.adjust(DateTime(2014, 1, 21, 23, 59, 40));
  }
  rtc.adjust(DateTime(2014, 1, 21, 23, 59, 40));
  now = rtc.now();
  future = now + TimeSpan(5);
  Serial.println("time:");
  Serial.print(now.hour());
  Serial.print(':');
  Serial.print(now.minute());
  Serial.print(':');
  Serial.print(now.second());
  Serial.println();
  Serial.println("Alarm set at:");
  Serial.print(future.hour(), DEC);
  Serial.print(':');
  Serial.print(future.minute(), DEC);
  Serial.print(':');
  Serial.print(future.second(), DEC);
  Serial.println();

}

void loop() {
  now = rtc.now();

  if((now.second()==future.second()))
```

```
    {
      Serial.println("Alarm!!");
      Serial.print(now.hour(), DEC);
      Serial.print(':');
      Serial.print(now.minute(), DEC);
      Serial.print(':');
      Serial.print(now.second(), DEC);
      Serial.println();
      now = rtc.now();
      future= now + TimeSpan(5); //5 seconds
    }
    if((now.hour() + now.minute() + now.second()) == 0)
     {
      if(print00)
      {
        Serial.println("00:00:00!!!");
        print00=false;
      }
     }
}
```

With this sketch, where the comments clarify the meaning of each line, we obtain:



```
time:
23:59:40
Alarm set at:
23:59:45
Alarm!!
23:59:45
Alarm!!
23:59:50
Alarm!!
23:59:55
Alarm!!
0:0:0
00:00:00!!!
Alarm!!
0:0:5
Alarm!!
0:0:10
```

Figure 37: RTC test output through UART

With this test we can obtain an exact moment to stamp the samples of our system. Also, we can see how the 5 seconds of span works and a section of the code can be executed in the desired moment.

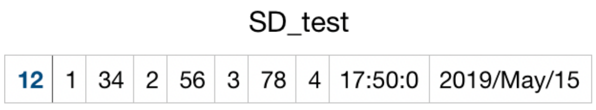
6.3.- SD card

The test for the SD card was codified based on the *SDFat* example called *ReadWrite*. After some test with the code, we created an example that verifies the correct behavior of the microcontroller and the SD card.

To test the handle of the library and the SD, the next code was written. The purpose of this code, is to create the filesystem, create the object that represent the file, initiate the SD card, write the data and close the file. The data to store, is basically a test that indicates two numbers for the four frequencies (12, 34, 56 and 78), and the number of the frequency as its magnitude. Also is fixed a time stamp to check the correct behavior of all the code. In the code we can see a comment that indicates the structure. Also we have to comment that, the frequencies are directly written in kHz.

```
#include <SPI.h>
#include <SdFat.h>

#define SD_SPI_SPEED SPI_HALF_SPEED  // SD card SPI speed
#define SD_CS 10

//File that we will save in the SD
SdFat sd; // set filesystem
SdFile logSIDSWAP;

char nameLogFile[50];

void setup() {
  Serial.begin(115200);

  Serial.print(F("Initiating SD card..."));
  if (!sd.begin(SD_CS, SD_SPI_SPEED))
  {
    Serial.println(F("Card failed, or not present"));
    return;
  }
  Serial.println(F("card initialized."));

  sprintf(nameLogFile, "SD_test.csv");

  if(logSIDSWAP.open(nameLogFile, FILE_WRITE))
  {
//Freq1[kzmag1[v],Freq2[kz];mag2[v];Freq3[kz];mag3[v];hh:mm:ss;yyyy/mm/dd
    logSIDSWAP.print(12);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(1);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(34);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(2);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(56);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(3);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(78);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(4);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(17);
    logSIDSWAP.print(F(":"));
    logSIDSWAP.print(50);
    logSIDSWAP.print(F(":"));
    logSIDSWAP.print(0);
    logSIDSWAP.print(F(";"));
    logSIDSWAP.print(2019);
    logSIDSWAP.print(F("/"));
    logSIDSWAP.print("May");
    logSIDSWAP.print(F("/"));
    logSIDSWAP.println(15);

    logSIDSWAP.close();
    Serial.println(F("Stored data in SD card OK"));
    Serial.println(nameLogFile);
  }
  else
  {
    Serial.print(F("Error opening "));
    Serial.println(nameLogFile);
  }

}

void loop() {


}
```

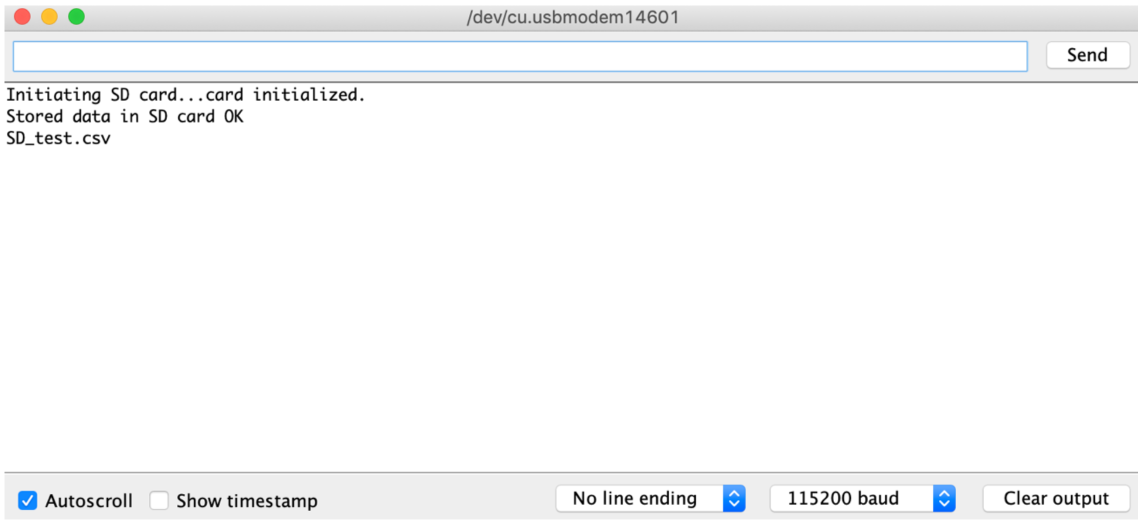
In this example, the function loop is not used because we want to run once. With this sketch compiled and uploaded to the microcontroller, we power down the µC, extract the SD card for the Screen and we can see in the computer the file generated with the following data, indicating that the csv file is separated by semicolon.


SD_test

12	1	34	2	56	3	78	4	17:50:0	2019/May/15

Figure 38: Content of the file SD_test.csv

We can open the Serial monitor to see the debug comment in the example:



```
Initiating SD card...card initialized.
Stored data in SD card OK
SD_test.csv
```

Figure 39: Serial monitor of the example of SD

6.4.- ESP8266 code

The code implemented in the second microcontroller of the system has been developed in Arduino IDE also. At the beginning we set the names of the network we want to connect, the server and the port in which we are going to make a GET method (HTTP protocol). The ESP8266 board has a blue led that are configured in the sketch to toggle if a good connection is achieved. If there is no connection, the led remains off.

For this End-of-Grade work, there is implemented only one frequency with its magnitude to test the correct GET method in the ESP8266 microcontroller.

In the *loop()* function, apart from toggle the led, we check the UART communication with the Due microcontroller. There is a sequence which depends on the character received in one or another side. The ESP8266 is constantly checking the UART in order to receive the character “F”. When it occurs, it sends the character “G” that means the Due can send the frequency. After receiving the character, the ESP8266 sends the character “H” to solicitate the amplitude of the frequency. Once the values are stored, it can try the http GET method with the server.

This try to establish the HTTP connection, is carried out by the function implemented in the *ESP8266WiFi* library, *client.connect()*. If the connection is established, then we put the GET headers in a string to send it to the server. The GET string has the following form:

```
"GET /sid/test/test.php?f=x&v=y HTTP/1.1\r\n
Host: www.spaceweather.es\r\nConnection: close\r\n\r\n"
```

In the example, we have a statement that check the response of the server in order to print it in the serial monitor if it is desired. If there is any problem with the UART communication, in the way the characters and data are exchanged, in the serial monitor we will see “bad command”.

The code is the following one:

```
//esp8266wifi
#include <ESP8266WiFi.h>
#define LED_BUILTIN 2

const char* ssid     = "iPhone de Fernando";
const char* password = "k28ku3tcd1894";
//const char* ssid     = "MOVISTAR_D5C0";
//const char* password = "EUVKJFCH74H9M9EKMWMM";
const char* host = "www.spaceweather.es";
const int httpPort = 80;
String freq = "0";
String volt = "0";
WiFiClient client;

void setup() {
  WiFi.mode(WIFI_STA);//Client mode
  WiFi.begin(ssid, password);

  Serial.begin(115200);
  pinMode(LED_BUILTIN, OUTPUT);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");

  }
  Serial.println("");
  Serial.println("IP:");
  Serial.println(WiFi.localIP());

  if (!client.connect(host, httpPort)) {
    Serial.println("connection failed");
    return;
  }
delay(5000);
}
void loop() {
  // Toggle the led
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  //Check the serial communication to send data to the server
  if (Serial.available()>0) {
    String inByte = Serial.readString();
    if (inByte == "F") {
      Serial.print("G");
      delay(100);
      while(!Serial.available());
      freq = Serial.readString();
      delay(100);
      Serial.print("H");
      delay(100);
      while(!Serial.available());
      volt = Serial.readString();

      if (!client.connect(host, httpPort)) {
        Serial.println("connection failed");
        return;
      }
      // This will send the request to the server
      client.print(String("GET /sid/test/test.php?f=") + freq + "&v="
+ volt + " HTTP/1.1\r\n" +
                   "Host: " + host + "\r\n" +
                   "Connection: close\r\n" +
                   "\r\n"
                  );
      while(client.available()) {
        String line = client.readStringUntil('\r');
        Serial.print(line);
      }

    }
    else {
      Serial.println("bad command");
    }

    client.stop();
  }
}
```

7.- Main code explanation

7.1.- Includes

This part has been covered in the software explanation section. We have to include all the libraries covered and the standard libraries <*Wire.h*> and <*stdin.h*> to use some defaults functions in the sketch.

```
//Use of Screen
#include <SPI.h>
#include <SdFat.h>
#include <ILI9341_due.h>
#include <URTouch.h>
#include <ILI9341_due_Buttons.h>
#include "SmallFont.h"
#include "BigFont.h"

//Use of RTC
#include "RTClib.h"

//Some necessary libraries
#include <Wire.h>
#include <stdint.h>
```

7.2.- Defines

After the includes, we have to define some variables that make the future ampliations and changes easier. We can change the sampling frequency, the center of coordinates of our graphs, and pins where the devices are connected.

```
#define Y_MAX 240                //Maximum pixels in Y axis
#define X_MAX 320                //Maximum pixels in X axis
#define Xo 53                    //Pixels for axis (0,0)
#define Yo 10

// LCD
#define TFT_RST 8
#define TFT_DC 9
#define TFT_CS 11
//SD
#define SD_CS 10
//Touch pannel
#define T_CLK 30
#define T_CS 28
#define T_DIN 26
#define T_DOUT 24
#define T_IRQ 22

#define Esp8266 Serial2
```

The names of the variables in the defines are the same that are put in the PCB of the device to simplify the understanding of the algorithm. We can see also, how we named the UART 2 of the microcontroller as the second microcontroller. This is because the UART communication with them are implemented in the UART 2 of Arduino Due and is easy to implement in the code the serial communication with the ESP8266. Also, if it is connected in other serial port, we just need to change this define without change any part of the code.

7.3.- Global variables

First, we create the global objects to use the libraries as we could see in the software section. We can see that we pass the variables of the defines.

```
//File that we will save in the SD
SdFat sd; // set filesystem
SdFile logSIDSWAP;

// Use hardware SPI
ILI9341_due tft = ILI9341_due(TFT_CS, TFT_DC, TFT_RST);

URTouch myTouch(T_CLK, T_CS, T_DIN, T_DOUT, T_IRQ);

// Finally we set up ILI9341_due_Buttons :)
ILI9341_due_Buttons  myButtons(&tft, &myTouch);

RTC_DS3231 rtc;
```

Then, the global variables itself are declared with some parts differentiated:

```
/*** GLOBAL VARIABLES*************************/
int pressed_button;
//check if a button is already pushed
boolean pressed = false;
//in order to paint values in the graph
boolean paint_time = false;
boolean paint_fft = false;
boolean log_data = false;
boolean sel_freq = false;
boolean showFrequencies = false;
```

The Booleans are for identify what button was pressed and establish the state of the second state machine. The Boolean *pressed* is just to know if the pulse in the screen has ended or not.

```
//Constant that extrapolates the voltage of the signal to adequate it
to pixels in the graph
uint16_t pixel_mag_factor = 52; //54.54 (180/3.3)
uint16_t pixel_magfreq_factor = 52;
```

These two variables are constants to adequate the actual values of the voltage in memory with the pixels of the LCD to paint correctly the graphs of the signal. This mean that, if a 3,3 V is obtained, we have to paint in the 3,3 level of the screen marked with the axis, this pixel.

```
//To store the values in time of x-axis in time draw
char xlabeltime[3][10];
char xlabelfreq[3][10];
```

These two variables are for to write in the axis the corresponding labels, either seconds or Hertz.

```
//Variable to do a passive wait to refresh the draw of the time signal
in the graph
uint32_t refresh_screen = 100;
```

This variable is to paint the graphs without keeping the CPU in a *while()* statement. Because the user can touch the touch panel in any moment, the course of the program cannot be stopped for a long in some section. For that reason, the CPU goes inside the paint section, evaluates the variable and continue with the *loop()* statement. This can be seen in the state machine section.

```
int backButton, dispButton, fftButton, logButton, selectFreqButton;
int saveButton, leftButton, rightButton;
```

As we covered in the library section, these are the buttons to manage the corresponding functions of them.

```
/*FFT stuff**********************************************/
const uint16_t N = 1024;
double input_wave[N];
double re[N], im[N], Xr[N], Xi[N];
double freqsFFT[N / 2], magFFT[N / 2], freqsSampled[4];
double preMax = -100.0;
//String to contain the max value of FFT and print it in the screen
char spreMax[20] = "0";
char dcComponent[20] = "0";
```

As we work with a processor, we decided to work with real numbers. To do that, we created two arrays to store the corresponding values of the signals. We can see that we are to sample 1024 samples, which will be stored in the *input_wave* array. The array *re[N]* is to store the real part of the input signal, that is, the signal itself, while *im[N]* stores the imaginary part of the signal, that is 0. The *Xr[N]* stores the real part of the FFT computed and the *Xi[N]* stores the imaginary part of the FFT computed.

The variable *freqsFFT[N/2]* stores the frequencies of the FFT according with the sampling frequency and the number of points of the FFT. The variable *magFFT[N/2]* stores the magnitude of each frequency store in the previous variable. The variable *freqsSampled[4]* stores the four frequencies we want to save in the system to log the data.

The variable *preMax* stores the maximum magnitude value of the FFT computed. The variables *spreMax[20]* and *dcComponent[20]* stores the string of the maximum value of FFT and DC component of it respectively, in order to write its values on the screen. It is important to say, that the library cannot write the number in the screen and we have to convert it to string.a

```
/*RTC stuff**********************************************/
DateTime now;
DateTime after;
```

These two objects are for store the time value at a given time and the time some seconds after. This is done because we want to check an interval of time and compute the corresponding operation periodically to log the data. We will see the procedure later.

```
/*LOG stuff**********************************************/
//Array containing the indexes of frequencies to sample in FFT array
(freqsFFT[N/2])
uint16_t logFFTindexes[4];
//to store the index temporary of FFT to save frequency
uint16_t indFx = 32;
//to print frequency 1, frequency 2, etc
uint8_t logFQIndexes = 0;
char logFQIndexesStr[4][3];
//To store the values of frequencies sampled
char freqSampledString[4][25];
//to show in sel frequencies screen the fq to store
char freqSampledStringTemp[4 + 1][25];
//Store the value every 5 seconds. 12 values every minute and 4
frequencies to store
double magValues5sec[4][60 / 5];
//index to store data in the above array
```

```
int logindex5sec = 0;
//value to store in SD card
double valueSD_minute[4][1];
//Name of the file
char nameLogFile[50];
//Name of timestamp
char timestamp[12] = "00:00:00";
```

These are the variables related to the log operation. We can read at the comments that *logFFTindexes[4]* store the indexes of the whole array where we have the frequencies computed in order to select correctly the wanted frequency on this array. For example, with N points of FFT, and with a sampling frequency of F kHz, we might have in *freqsFFT[N/2]* the corresponding frequencies and maybe in index 3 we have the frequency bin of 2 kHz. This index 3 is stored in *logFFTindexes[4]*.

The variable *indFx* is an auxiliary variable to do certain operations to traverse the array of frequencies. We will see the use of it in the corresponding part of the algorithm. The variables *logFQIndexes* and its corresponding string version *logFQIndexesStr[4][3]* are just to know which number of the 4 possible frequencies to save are selecting the user in the frequency selection screen, and the string version of it with the purpose of print it on the screen.

The variable *freqSampledString[4][25]* is to store the string value of the corresponding frequency of the 4 selected. That is, according with the previous example, the 2 kHz itself. The variable *freqSampledStringTemp[4+1][25]* is just to store the temporary string value of the previous variable in order to show in the screen the frequency selected before save its value (with *saveButton*) in the *freqSampledString[4][25]* variable.

The variable *magValues5sec[4][60/5]* stores the magnitude values of the corresponding saved frequency every 5 seconds per minute. The variable *logindex5sec* is just to save in the corresponding index position the correct magnitude. The variable *valueSD_minute[4][1]* stores the values to save in the SD card, which are computed every minute. The variable *nameLogFile[50]* is just the variable that store the name of the csv file which contains all the data corresponding with the day. The variable *timestamp[12]* is used to store the current time in a string format.

```
/*Esp8266 Stuff***********************************************/
String freq2ESP = "0";
String volt2ESP = "0";
String inStr = "nothing";
bool data2Esp = false;
```

These variables are the frequency and the magnitude to send to ESp8266 for the server and auxiliary variables to check the process and the state of the state machine.

```
/*Areas differentation*******************************************/
gTextArea graphArea{Xo, Yo, X_MAX - Xo, 180}; //170 available pixels
for draw signals in y-axis
gTextArea allArea{0, 0, X_MAX, Y_MAX};
```

These two structs are for select a definited area in the screen to work with the graphs. The struct *graphArea* is the area inside the axis to clean and paint constantly the signal we have sampled.

7.4.- Statement of functions

After the declaration of all the variables our program will share between all parts of the code, we declare all the functions used, in order to say to the compiler that it has to look for a specific part of the memory where a function is written. These functions will be explained at the end of this section.

```
/*** FUNCTIONS*************************/
void drawScreen1(ILI9341_due &d);
void timeAxis(ILI9341_due &d);
void freqAxis(ILI9341_due &d);
void drawGraph(ILI9341_due &d);

void sidFFT(double *X_real, double *X_im, double *xreal, double
*ximag, int freq_bin, int N, int h, int h_interval);
double postProcessing(double *Xr, double *Xi, double premax);

void Sampling(double *sw, double *re, double *im);
void configureADC(void);

void sendData2ESP(void);
```

7.5.- Setup() function

With all the declarations, we have the *setup()* function and the *loop()* function. Analyzing firstly the *setup()* function we have:

```
void setup()
{
  //For PC communication
  Serial.begin(9600);
  //For Esp8266 communication we use Serial1,2 or 3 of Arduino Due
(Baudrate=115200)
  Esp8266.begin(115200);
  delay(2000);
  Serial.println(F("Initiating system..."));
```

Where we initiate the two UART communications, one with the computer to debug the system and other with the ESP8266 microcontroller.

```
// Initial setup
  tft.begin();
  tft.setRotation(iliRotation270);  // landscape
  tft.fillScreen(ILI9341_BLACK);

  tft.setFont(SmallFont);

  myTouch.InitTouch();
  myTouch.setPrecision(PREC_MEDIUM);

  myButtons.setTextFont(BigFont);
```

Here we initiate the screen with the *ILI9341_due* library, put the presentation in landscape mode, paint all the LCD in black, select the small font initially to write in the screen, initiate the touch panel with medium precision that is enough according with the experience, and select for the buttons of the *ILI9341_due_Buttons* library the big font. The use of the libraries has been covered according to the documentation of each one. The documentation, as we explain before, are mention in the software section of the memory and in the Bibliography part.

```
/*Buttons to be used in the LCD menu ***************************/
```

```
  /* Main Menu Buttons***************************/
  dispButton = myButtons.addButton( 10,  20, 300,  30, "Signal
scope");
  fftButton = myButtons.addButton(10, 60, 300, 30, "FFT");
  logButton = myButtons.addButton(10, 160, 300, 30, "LOG data");
  selectFreqButton = myButtons.addButton(10, 200, 300, 30, "Sel
Frequencies");
  backButton = myButtons.addButton( 10, 218, 75, 20, "BACK");

  /*To select the frequencies to sample*/
  leftButton = myButtons.addButton( 95, 218, 70, 20, "<-");
  rightButton = myButtons.addButton( 245, 218, 70, 20, "->");
  saveButton = myButtons.addButton(168, 210, 74, 30, "Save");
```

These are the instantiation of all the buttons object to handle its corresponding methods as we will see in the code. The parameters are the coordinates in the screen and the labels of themselves.

```
if (! rtc.begin()) {
    Serial.println(F("Couldn't find RTC"));
    //while (1);
  }
```

Then we check if there is communication with the RTC.

```
if (rtc.lostPower())
  {
    Serial.println("RTC lost power, lets set the time!");
    // following line sets the RTC to the date & time this sketch was
compiled
    rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
    // This line sets the RTC with an explicit date & time, for
example to set
    // January 21, 2014 at 23:59:00 you would call:
    // rtc.adjust(DateTime(2014, 1, 21, 23, 59, 40));
  }
    Serial.println(F("RTC set:"));
    now = rtc.now();
    sprintf(timestamp, "%02d:%02d:%02d", now.hour(), now.minute(),
now.second());
    Serial.println(timestamp);
```

And if there is an error with the RTC or it lost the time, we set its time according with the momentum when the sketch is compiled. In order to see the current time, we print in the monitor the time set.

```
Serial.print(F("Initiating SD card..."));
  if (!sd.begin(SD_CS, SD_SPI_SPEED))
  {
    Serial.println(F("Card failed, or not present"));
    return;
  }
  Serial.println(F("card initialized."));
```

We start, in a similar way, the SD card, checking if it is initialized correctly or not.

```
sprintf(freqSampledString[0], "%.2f", 0);
sprintf(freqSampledString[1], "%.2f", 0);
sprintf(freqSampledString[2], "%.2f", 0);
sprintf(freqSampledString[3], "%.2f", 0);
```

```
Serial.println(F("Done"));
  tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
  drawScreen1(tft);
}
```

Then we save the 0 value in the string to initiate those variables. With all of this done, we print in the serial monitor that the system is initiated with de command “Done” and select for the text in the screen the color black in a white background.

7.6.- Loop () function

7.6.1.- State machine 1 in loop () function

With the system initialized, we only have to run the state machines.

```
void loop()
{
  //to store the maximum value of FFT
  double preMax = -100.0;

  now = rtc.now();

  //Always sampling the signal if we are not loging the data
  if (!log_data)
  {
    configureADC();
    Sampling(input_wave, re, im);
  }

  if (data2Esp) {
    if(Esp8266.available()) inStr = Esp8266.readString();
    sendData2Esp();
    data2Esp = false;
  }
```

First, we set the maximum FFT value to an impossible value to check the correct functioning of the algorithm. Then, in every iteration of the code we check the current time requesting it to the RTC.

If we are not in the log state, we sample the input signal in each iteration in order to use the system as an oscilloscope whether in time domain or frequency domain. This is because we can use the system to analyze the input signal in almost real time, and so check the frequencies where most of the power receiver is allocated. Also, we check the variable to send data to the server, cancelling it to leave its activation to the corresponding part of the code.

With this, we start the state machine 1 that checks the buttons.

```
/*State machine that manages the behaviour of the project*/
  if (myTouch.dataAvailable() == true)
  {
    pressed_button = myButtons.checkButtons();

    if ((pressed_button == dispButton) && (!pressed))
    {
      tft.setTextArea(allArea);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(fftButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      myButtons.disableButton(leftButton);
      myButtons.disableButton(rightButton);
      myButtons.disableButton(saveButton);
      pressed = true;
      paint_time = true;
      paint_fft = false;
      log_data = false;
      sel_freq = false;
      drawGraph(tft);
      timeAxis(tft);
      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);
    }
```

If the touch panel detects a pulse, we check which button is pulsed. This means that the pixels pressed are in the area of the coordinates of the corresponding object button. The structure of the code of each button is more or less the same. With the pulse of the *dispButton*, we disable all the buttons because the pulse can activate the buttons, even if they are not drawn. The pulse of the user maybe lasts longer than an iteration lasts, and for that reason we have to disable all the buttons. According to the button, we activate or deactivate the corresponding Boolean variables. The button *dispButton,* activate the time domain screen, viewing the input signal as an oscilloscope, and for that reason, we call the functions *drawGraph(tft)* and *timeAxis(tft)* to draw in the screen the axis and its labels in time domain. We pass as an argument the *tft* object in order to pass the address memory of it and handle correctly the TFT screen. Once it is done, we activate and draw the *backButton* that leads us the main screen again.

```
if ((pressed_button == fftButton) && (!pressed))
    {
      tft.setTextArea(allArea);
      myButtons.disableButton(fftButton);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      myButtons.disableButton(leftButton);
      myButtons.disableButton(rightButton);
      myButtons.disableButton(saveButton);
      pressed = true;
      paint_time = false;
      paint_fft = true;
      log_data = false;
      sel_freq = false;
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      drawGraph(tft);
      freqAxis(tft);
      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);
    }
```

If the user presses the *fftButton*, we do more or less the same as before. In this case, we also compute the FFT and call to *postprocessing* function that, among other things, returns the maximum value of the computed spectrum. We will cover these functions at the end of the section. Then, we draw the graph and we put the corresponding frequency labels. We also enable the *backButton* to return back to the main screen.

```
if ((pressed_button == logButton) && (!pressed))
    {
      myButtons.disableButton(fftButton);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      myButtons.disableButton(leftButton);
      myButtons.disableButton(rightButton);
      myButtons.disableButton(saveButton);
      pressed = true;
      paint_time = false;
      paint_fft = false;
      log_data = true;
      sel_freq = false;
```

When the user pulses the log button, there are many things that our system does. First, it does the same than the previous buttons.

```
tft.fillScreen(ILI9341_BLACK);
      //Show the screen displaying that data are stored
      tft.setTextScale(2);
      tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
      tft.printAt("Loging data", Xo, Yo);
      tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
      tft.setTextScale(1);

      tft.printAt(F("Frequencies sampled:"), Xo, Yo + 50);

      tft.setTextColor(ILI9341_GOLD, ILI9341_BLACK);
      tft.printAt(F("Frequency 1: "), Xo + 10, Yo + 80);
      tft.printAt(freqSampledString[0], Xo + 140, Yo + 80);
      tft.printAt(" kHz", Xo + 190, Yo + 80);

      tft.printAt(F("Frequency 2: "), Xo + 10, Yo + 110);
      tft.printAt(freqSampledString[1], Xo + 140, Yo + 110);
      tft.printAt(" kHz", Xo + 190, Yo + 110);

      tft.printAt(F("Frequency 3: "), Xo + 10, Yo + 140);
      tft.printAt(freqSampledString[2], Xo + 140, Yo + 140);
      tft.printAt(" kHz", Xo + 190, Yo + 140);

      tft.printAt(F("Frequency 4: "), Xo + 10, Yo + 170);
      tft.printAt(freqSampledString[3], Xo + 140, Yo + 170);
      tft.printAt(F(" kHz"), Xo + 190, Yo + 170);
```

After this, print in the screen the corresponding frequencies that the system will log in the SD card. We can see that the code repeats in order to print the 4 frequencies and its corresponding values. For better clarification, see the global variables section to understand the *freqSampledString* variable. The values that the function *tft.printAt* receives, in addition to the string value, are the coordinates where the text will show in the screen.

```
      now = rtc.now();
      //Alarm set 5 seconds later.
      after = now + TimeSpan(5);
```

Then the current time and a span of 5 seconds are computed.

```
     sprintf(nameLogFile, "%04d-%02d-%02d.csv", now.year(),
now.month(), now.day());
```

Then, we store the name of the file that will contain the values of the day, in the corresponding variable. We can see that the name of the file is just the current day. This is done because the user can enter in the log screen as many times as the user wants, independently of the desire to store data or not, and if the system is logging data or not. Each time the user presses the button, we work with the corresponding file of the day.

```
     Serial.println(F("Starting with data loging..."));
      sprintf(timestamp, "%02d:%02d:%02d", now.hour(), now.minute(),
now.second());
      Serial.println(timestamp);

      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);

      tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
    }
```

After that, we print in serial monitor the time when the system starts to log data and we enable and draw the *backButton* as usual.

```
    if ((pressed_button == selectFreqButton) && (!pressed))
    {
      myButtons.disableButton(fftButton);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      tft.clearTextArea(ILI9341_BLACK);
      pressed = true;
      paint_time = false;
      paint_fft = false;
      log_data = false;
      sel_freq = true;
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      drawGraph(tft);
      freqAxis(tft);
      myButtons.enableButton(leftButton);
      myButtons.drawButton(leftButton);
      myButtons.enableButton(saveButton);
      myButtons.drawButton(saveButton);
      myButtons.enableButton(rightButton);
      myButtons.drawButton(rightButton);
      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);
    }
```

This button is more or less the same than *fftButton.* The difference is in the Boolean variables. This is because, to store the desired frequencies in memory, it is useful to see the current spectrum in the screen. We will see soon, in this own state machine, which affects the variable *sel_freq*, that is the difference between *paint_fft*. We can see, that in addition to this, we draw three new buttons that is the *leftButton*, *rightButton* and *saveButton*. These buttons aim to select the frequency of the four possible, which we want to store. With these buttons, as we will see in the part that check the *sel_freq* variable, we can see the frequency selected in the spectrum to know if the current frequency selected is the desired one or not.

```
   if ((pressed_button == backButton) && (!pressed))
    {
      myButtons.deleteAllButtons();
      dispButton = myButtons.addButton( 10,  20, 300,  30, "Signal
scope");
      fftButton = myButtons.addButton(10, 60, 300, 30, "FFT");
      logButton = myButtons.addButton(10, 160, 300, 30, "LOG data");
      selectFreqButton = myButtons.addButton(10, 200, 300, 30, "Sel
Frequencies");
      backButton = myButtons.addButton( 10, 218, 75, 20, "BACK");
      leftButton = myButtons.addButton( 95, 218, 70, 20, "<-");
      rightButton = myButtons.addButton( 245, 218, 70, 20, "->");
      saveButton = myButtons.addButton(168, 210, 74, 30, "Save");

      pressed = true;
      paint_time = false;
      paint_fft = false;
      log_data = false;
      sel_freq = false;
      tft.setTextArea(allArea);
      myButtons.disableButton(backButton);
      drawScreen1(tft);
    }
```

The *backButton* aims to return back to the main screen. After several tests with the system, we discovered the necessity of clear all the buttons instead activate or deactivate them. This is because the use of memory. The system needs a certain amount of dynamic memory, and the library of the buttons take advantage of it. When we delete the buttons, we are releasing dynamic memory, and when we want to create again the buttons objects, we use the dynamic memory management, finding a new place in memory where all the corresponding data could be allocated.

```
 if (sel_freq)
    {
      if ((pressed_button == leftButton) && (!pressed))
      {
        pressed = true;
        //paint white line before paint the new one
        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_WHITE);
        indFx -= 1;
        if (indFx < 0) indFx = 0;
        if (indFx > (N / 2) - 1) indFx = (N / 2) - 1;

        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_DARKGOLDENROD);
        sprintf(freqSampledStringTemp[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
      }
```

We can see that with the *sel_freq* activated, we draw a vertical line in the middle of the spectrum. We paint a white vertical line when the leftButton is pressed because we want to “erase” the “old” vertical one and paint a “new” one at the left of this frequency. The handle of the frequencies is also covered, we can see that we subtract in one the *indFx* variable because it marks the index of the frequencies of the FFT computed. In the drawing of the vertical line, we can see that in X axis we multiply by two the value. This will be covered in the explanation of the funcitons but, basically, is because the pixels in the screen do not correspond with the absolute values of the arrays. We draw each two pixels, the value of the array. This pretends to amplify the view to make it easier for the user to view the signal. The *rightButton* behaves in a similar way as we can see.

```
if ((pressed_button == rightButton) && (!pressed))
      {
        pressed = true;
        //paint black line before paint the new one
        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_WHITE);
        indFx += 1;
        if (indFx < 0) indFx = 0;
        if (indFx > (N / 2) - 1) indFx = (N / 2) - 1;

        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_DARKGOLDENROD);
        sprintf(freqSampledStringTemp[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
      }
```

The difference is that we add instead of subtracting one to the variable *indFx*.

```
if ((pressed_button == saveButton) && (!pressed))
      {
        pressed = true;
        logFFTindexes[logFQIndexes] = indFx;
        sprintf(freqSampledString[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
        tft.setTextColor(ILI9341_WHITE, ILI9341_RED);
        tft.printAt("SAVED!", Xo + 150, Yo + 10);
        tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
        logFQIndexes ++;
        if (logFQIndexes > 3) logFQIndexes = 0;
      }
    }
  }
```

When the user pulses the *saveButton*, we can see the difference between the variables *freqSampledString* and *freqSampledStringTemp*. The second one is for show in the screen the frequency where the vertical line is, and the first one is to store this current value because is what the user wants to save. Also, we store the index of the frequency spectrum array in *logFFTindexes* and print the word "SAVED!" in the screen during a frame. As we have 4 frequencies to store, the variable *logFQIndexes* adds one.

```
 else
  {
    pressed = false;
  }
```

With this we end the state machine 1. This else statement is to handle a long pulsation in the screen by the user. We go into the state machine 1 code only once with one pulse.

7.6.2.- State machine 2 in loop() function

As we saw, the state machine 1 establishes the variables that are the input of the state machine 2. These two state machines are implemented in parallel, and we are going to see how the second one is implemented in the code.

```
if (paint_time)
  {
    tft.setTextArea(graphArea);
    refresh_screen--;

    if (refresh_screen == 50)
    {
      tft.clearTextArea(ILI9341_WHITE);
      for (int i = 0; i < 128; i++)
      {
        tft.drawLine(Xo + i * 2, 189 - input_wave[i]*pixel_mag_factor,
Xo + (i + 1) * 2, 189 - input_wave[i + 1]*pixel_mag_factor,
ILI9341_STEELBLUE); //147 vs 189
      }
    }

    if (refresh_screen == 0)
    {
      tft.clearTextArea(ILI9341_WHITE);
      for (int j = 128; j < 255; j++)
      {
        tft.drawLine(Xo + ((j - 128) * 2), 189 -
input_wave[j]*pixel_mag_factor, Xo + ((j + 1 - 128) * 2), 189 -
input_wave[j + 1]*pixel_mag_factor, ILI9341_STEELBLUE);
      }
      refresh_screen = 100;
    }
  }
```

When *paint_time* is activated, we go throw the oscilloscope screen. As we draw actually the axes, we select the *graphArea* to update the signal only in the corresponding area. We use the *refresh_screen* variable to update not too fast the graph, but with the real time sensation. We can see in the code, concretely in the for loop statemen, that we draw 128 samples of the array. As we have two *for loops* statements, we draw only 256 samples of the input signal. This is because the system does not pretend to analyze the signal in time domain, but to analyze the signal in frequency domain. Also, we notice that we add 189 to the Y coordinate. This is because the center of the axes is in the top left corner. The purpose of this screen is to see certain aspect of the sampled signal, analyzing the maximum and minimum values mainly.

Also, we can see how we draw a sampled value every two pixels in X axis. The purpose of this is to find an equilibrium between simplicity in the code and legibility in the signal sampled and showed. We do the same for the frequency domain graph.

```
 if (paint_fft)
  {
    tft.setTextArea(graphArea);
    refresh_screen--;

    if (refresh_screen <= 50)
    {
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      preMax /= 1000;
      tft.clearTextArea(ILI9341_WHITE);
      //We draw until 84kHz because the frequencies of interest are
there
      for (int i = 0; i < 128; i++)
      {
        tft.drawLine(Xo + i * 2, 190 - magFFT[i]*pixel_magfreq_factor,
Xo + (i + 1) * 2, 190 - magFFT[i + 1]*pixel_magfreq_factor,
ILI9341_DARKSLATEBLUE);
      }
      //DC Component
      tft.setTextColor(ILI9341_DARKVIOLET, ILI9341_KHAKI);
      sprintf(dcComponent, "DC component: %.2f V", magFFT[0]);
      tft.printAt(dcComponent, 50, 210);
      sprintf(spreMax, "More Power at %.2f  kHz", preMax);
      tft.printAt(spreMax, 5, 0);

      refresh_screen = 100;
    }
    tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
  }
```

If we see the corresponding frequency domain option, we can see at first glance that there is only one for loop statement. This is because when we analyze the spectrum of the signal, with 1024 points and with the sampling frequency of 666,6 khz, in the index 128 we have 83,32 kHz. As we can see the frequencies below 100 kHz, for this End-of-Grade work it has been considered enough. With a few lines more, the spectrum can be observed entirely.

In every iteration, taking into account the *refresh_screen* variable, we compute the FFT of the input signal and call for the *postprocessing* function. In addition to this, we show the DC component of the spectrum and the maximum value of the spectrum in text mode.

```
 if (log_data)
  {
    if (now.second() == after.second())
    {
      now = rtc.now();
      after = now + TimeSpan(5); //5 seconds of span
      sprintf(timestamp, "%02d:%02d:%02d", now.hour(), now.minute(),
now.second());
      Serial.println(timestamp);

      configureADC();
      Sampling(input_wave, re, im);
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      preMax = postProcessing(Xr, Xi, preMax);

      magValues5sec[0][logindex5sec] = magFFT[logFFTindexes[0]];
      magValues5sec[1][logindex5sec] = magFFT[logFFTindexes[1]];
      magValues5sec[2][logindex5sec] = magFFT[logFFTindexes[2]];
      magValues5sec[3][logindex5sec] = magFFT[logFFTindexes[3]];

      logindex5sec++;
```

When the user pulses *logButton*, we saw that, a span of 5 seconds is set. Also, at the beginning of the state machine 1 we saw that there is no constant sampling when this variable is activated. When the *now* and *after* variables match, *configureADC* and *sampling* functions are activated.

Then, is computed the FFT and executed the *postProcessing* function. Once all of this is done, we store the values of the corresponding frequencies in the 5-seconds array *magValues5sec*. As we want to measure over every 5 seconds, in order to average the value of a minute to store it in the SD card, we add these 5-seconds values in the array *magValues5sec*.

```
     /*if we achieve the minute.... Store in memory the data*/
      if (logindex5sec == (60 / 5))
      {
        logindex5sec = 0;
        for (int i = 0; i < 60 / 5; i++)
        {
          valueSD_minute[0][0] += magValues5sec[0][i];
          valueSD_minute[1][0] += magValues5sec[1][i];
          valueSD_minute[2][0] += magValues5sec[2][i];
          valueSD_minute[3][0] += magValues5sec[3][i];
        }
        //we compute the average value of the minute
        valueSD_minute[0][0] /= (60 / 5);
        valueSD_minute[1][0] /= (60 / 5);
        valueSD_minute[2][0] /= (60 / 5);
        valueSD_minute[3][0] /= (60 / 5);

        freq2ESP = String(freqsFFT[logFFTindexes[0]]);
        volt2ESP = String(valueSD_minute[0][0]);
        data2Esp = true;
```

When it is been a minute, we compute the average value of all the samples in the 4 frequencies. Also, we see that we activate the ESP8266 procedure in order to send the first frequency and its magnitude to the server.

```
// Check if its 00:00:00-00:00:05
        if (((now.hour() + now.minute()) == 0) && (now.second() < 5))
        {
          //Create a new file
          sprintf(nameLogFile, "%04d-%02d-%02d.csv", now.year(),
now.month(), now.day());
        }
```

We have to check also the day in which the system works. This pretends to generate a new file in a necessary case. When the system knows the file it has to handle, the csv file is generated.

```
//Freq1[kz] ; mag1[v]; Freq2[kz]; mag2[v]; Freq3[kz]; mag3[v];
hh:mm:ss ; yyyy/mm/dd.csv
          logSIDSWAP.print(freqsFFT[logFFTindexes[0]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[0][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(freqsFFT[logFFTindexes[1]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[1][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(freqsFFT[logFFTindexes[2]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[2][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(freqsFFT[logFFTindexes[3]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[3][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(timestamp);
          logSIDSWAP.print(";");
          logSIDSWAP.println(nameLogFile);


          logSIDSWAP.close();
          Serial.println(F("Stored data in SD card OK"));
        }
        else
        {
          Serial.print(F("Error opening "));
          Serial.println(nameLogFile);
        }
```

We can see how we store the data in the file. The way to do that is the same as we explained before in the tests section. Also, we add the debug option reading in the serial monitor if the file has been created correctly or not.

```
       valueSD_minute[0][0] = 0;
       valueSD_minute[1][0] = 0;
       valueSD_minute[2][0] = 0;
       valueSD_minute[3][0] = 0;

      }

    }

  }
```

After the file is properly generated, we restart the variables in order to start a new minute of values. It is important to mention that all the procedure has to spend less than five seconds. At the beginning of this state, the span of 5 seconds was set. This is important in order to make improvements on the system.

```
 if (sel_freq)
  {
    //freqsSampled[0] = selectFreqFunction(1);
    tft.setTextArea(graphArea);
    refresh_screen--;

    if (refresh_screen <= 50)
    {
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      tft.clearTextArea(ILI9341_WHITE);
      //We draw until 84kHz because the frequencies of interest are
there
      for (int i = 0; i < 128; i++)
      {
        tft.drawLine(Xo + i * 2, 190 - magFFT[i]*pixel_magfreq_factor,
Xo + (i + 1) * 2, 190 - magFFT[i + 1]*pixel_magfreq_factor,
ILI9341_STEELBLUE);
      }

      tft.setTextColor(ILI9341_BLACK, ILI9341_KHAKI);
      tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_DARKGOLDENROD);

      //Frequency %d: %.2f kHz
      tft.printAt(F("Frequency "), Xo - 30, 0);
      sprintf(logFQIndexesStr[logFQIndexes], "%d:", logFQIndexes + 1);
      tft.printAt(logFQIndexesStr[logFQIndexes], Xo + 65, 0);
      sprintf(freqSampledStringTemp[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
      tft.printAt(freqSampledStringTemp[logFQIndexes], Xo + 90, 0);
      tft.printAt(F(" kHz"), Xo + 140, 0);

      refresh_screen = 100;
    }
    tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
  }

}
```

The last state of the state machine 2 is the *sel_freq* one. In this state, the system behaves nearly to the FFT screen. The main difference is that the current frequency showed is updated every time the system goes inside this state. In the state machine 1, we saw that when the user touches the *leftButton* or *rightButton*, a vertical white line is drawn in the current frequency value and a new yellow one is drawn in the correct position. This line position matches the frequency showed when the system is inside this piece of code.

Whit this, all the features and the state machines of the system have been covered. In the next section we are going to describe and to explain the implementation of the functions called from the main code.

7.7.- Implemented functions in the system

7.7.1.- FFT function

To understand the code, the reader is recommended to read the recursivity section. The FFT function is recursively implemented to take advantage of the intermediate results.

We also put here the call for the function in order to understand better the input arguments:

```
sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
```

```
void sidFFT(double *X_real, double *X_im, double *xreal, double
*ximag, int freq_bin, int N, int h, int h_interval)
{
  uint32_t k;
  double Xre_temp, Xim_temp, Xre_temp_kplusN2, Xim_temp_kplusN2,
exp_Oddk_re, exp_Oddk_im;

  if (N == 1)
  {
    X_real[freq_bin] = xreal[h];
    X_im[freq_bin] = ximag[h];
    return;
  }
  else
  {
    //Even part of the DFT subsection. X0,x2,x4….
    sidFFT(X_real, X_im, xreal, ximag, freq_bin,      N / 2,
h,          h_interval * 2);
    //Odd part of the DFT subsection computed through X[k+N/2]. X1,x3…
    sidFFT(X_real, X_im, xreal, ximag, freq_bin + N / 2, N / 2, h +
h_interval, h_interval * 2);
```

This first part of the function is the recursivity one. The recursivity is used to the split of the even and odd summation of the original one. It is important to remark that the odd index summation is computed from the values of X[k+N/2]. This is because we take the advantage of the recursivity again. We store the values of the odd index summation at the same time that we work with X[k+N/2] in the next part of the code. Thanks to this, we can use the equations of the FFT, shown in applicable theory section, using X[k] and X[k+N/2] instead of each even and odd index summation. In resume, we obtain the even index of the FFT through X[k] and, thanks to the symmetry identity, the odd index of FFT through X[k+N/2]. We can see that the call of the function needs some input arguments. These arguments are pointer to the address memory of the corresponding global variable. With these, we only generate in the stack some *double* pointers instead of a *double* array in each recursive call.

- X_real → is the destination real part. This array will contain the real part of the FFT implemented
- X_im → is the imaginary part of the FFT array.
- xreal → is the real part of the input signal.
- ximag → is the imaginary part of the input signal
- freq_bin → is the index of X[k], where the summation is split. This is the recursive index in the destination array that will set the limits of the even or odd summation to compute the DFT. This input argument is the start of the summation.
- N → the number of points to compute the DFT
- h → the index of x[n] where the DFT has to compute its value.
- h_interval → Is the separation between samples to compute the summation $(x_1, x_3, \dots)$

The first statement is if N is equal to 1. If we have a DFT of only one point, that it will be the "end of recursion" condition, the 1-point DFT is equal to the 1-sample input signal. When we divide the summation in an even an odd index summation, we can continue until only a summation of one value is achieved. We can see that in the applicable theory.

One the "tree of recursivity" has been established in memory, and the "end of recursion" condition is achieved, each call of the function will go inside the following part of the code, where we are going to explain line by line.

```
for (k = 0; k < (N / 2); k++)
    {
```

First of all, we have the limits of the summation in the form of a *for loop*. We have to remember that this N is not the global variable, but the input argument received in the recursivity process. Each call of the function passes the half of N, generating a summation of even and odd indexes of each summation. Once established the limits of the corresponding summation, and knowing the corresponding interval of the variable *k* we have

```
        // t <- X_k
        // t_temp <- X[k]
        Xre_temp = X_real[k + freq_bin];
        Xim_temp = X_im[k + freq_bin];
```

Which means that we extract E_k of X[k], and we store its real and imaginary part in *Xre_temp* and *Xim_temp*.

```
      // t_temp_kplusN2 <-X[k+N/2]
```

```
        Xre_temp_kplusN2 = X_real[k + freq_bin + N / 2];
        Xim_temp_kplusN2 = X_im[k + freq_bin + N / 2];
```

We repeat the previous process storing the real and imaginary part of the odd index summation through X[k+N/2].

If we remember the equations of FFT, to compute each frequency bin X_k

$$X_k = E_k + e^{-j\frac{2\pi}{N}k} O_k$$
$$X_{k+\frac{N}{2}} = E_k - e^{-j\frac{2\pi}{N}k} O_k$$
$$E_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi km}{N/2}} \quad ; \quad O_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi km}{N/2}}$$

And now that we have the interval of *m* set, due to the corresponding recursivity call, we can compute each frequency bin according with the expressions.

First of all, we comput the exponential term in terms of sin and cosine.

```
  // Calculation temp
        exp_Oddk_re = cos(-2 * PI * k / N) * Xre_temp_kplusN2 - sin(-2 *
PI * k / N) * Xim_temp_kplusN2;
        exp_Oddk_im = cos(-2 * PI * k / N) * Xim_temp_kplusN2 + sin(-2 *
PI * k / N) * Xre_temp_kplusN2;
```

This part of the code is for extract the real and imaginary part of the multiplication between the exponential and the odd term. We can see it through the Euler identity. If we take $-j\frac{2\pi}{N}k = \alpha$, we can see that:

$$e^{\alpha} O_k = \left(\cos(\alpha) + j\sin(\alpha)\right) \cdot (Re\{O_k\} + j \cdot Im\{O_k\}) =$$
$$\cos(\alpha) \cdot Re\{O_k\} + j \cdot \cos(\alpha) \cdot Im\{O_k\} + j\sin(\alpha) Re\{O_k\} + j \cdot j \cdot \sin(\alpha)\, Im\{O_k\} =$$
$$(\cos(\alpha)\, Re\{O_k\} - \sin(\alpha)\, Im\{O_k\}) + j(\cos(\alpha)\, Im\{O_k\} + \sin(\alpha)\, Re\{O_k\})$$

So, at the end, we have the real part and imaginary part separated of the factor $e^{-j\frac{2\pi}{N}k} O_k$. This is what we do with the *varibles exp_Oddk_re* and *exp_Oddk_im*.

```
      // X_k <- t + exp(-2*pi*i*k/N) X_(k+N/2)
        X_real[k + freq_bin] = Xre_temp + exp_Oddk_re;
        X_im[k + freq_bin] = Xim_temp + exp_Oddk_im;
        // X_(k+N/2) <- t - exp(-2*pi*i*k/N) X_(k+N/2)
        X_real[k + freq_bin + N / 2] = Xre_temp - exp_Oddk_re;
        X_im[k + freq_bin + N / 2] = Xim_temp - exp_Oddk_im;
      }
    }
}
```

With this, what we so is just compute the FFT equation using the temporary variables computed before.

7.7.2.- ConfigureADC and Sampling function

```
void configureADC() {
  // Setup all registers
  pmc_enable_periph_clk(ID_ADC); // To use peripheral, we must enable
clock distributon to it
  adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST); //
initialize, set sampling frequency
  adc_disable_interrupt(ADC, 0xFFFFFFFF); //disable interrupt of theA
ADC
  adc_set_resolution(ADC, ADC_12_BITS); //We use the available
resolution of the ADC
  adc_configure_power_save(ADC, 0, 0); // Disable sleep, always
powered
  adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1); // Set timings
- standard values
  adc_set_bias_current(ADC, 1); // Bias current - maximum performance
over current consumption
  adc_stop_sequencer(ADC); // not using it
  adc_disable_tag(ADC); // it has to do with sequencer, not using it
  adc_disable_ts(ADC); // disable temperature sensor
  adc_disable_channel_differential_input(ADC, ADC_CHANNEL_7); // A0 is
channel 7 of the ADC
  adc_configure_trigger(ADC, ADC_TRIG_SW, 1); // triggering from
software, freerunning mode
  adc_disable_all_channel(ADC);
  adc_enable_channel(ADC, ADC_CHANNEL_7); // just one channel enabled
}


void Sampling(double *sw, double *re, double *im)
{
  adc_start(ADC);
  for (int i = 0; i < N; i++) {
    while ((adc_get_status(ADC) & ADC_ISR_DRDY) != ADC_ISR_DRDY)
    {}; //Wait for end of conversion
    sw[i] = adc_get_latest_value(ADC); // Read ADC
  }

  adc_stop(ADC);

  for (int i = 0; i < N; i++) {
    //To store the voltage value in the array -> value*SPAN_ADC/(2^n-
1)
    sw[i] = sw[i] * 3.3 / 4095;
    re[i] = sw[i];
    im[i] = 0;
  }

}
```

These two functions are explained before in the tests section. We configure the ADC peripheral of Arduino Due using the libsam library and the, we activate the ADC waiting actively each sample of the input signal. After this, the ADC is stopped and we compute the real and imaginary part of the voltage input.

7.7.3.- Postprocessing function

This function does some important things.

```
double postProcessing(double *Xreal, double *Xim, double preMax)
```

```
{
  double maxk = -1.0;
  double maxvalue = -1.0;
  int j = N / 2;
  for (int i = N / 2; i < N; i++) {                    //we check the
half of the FFT result because is symetric
    double nowFre = abs((i - N) * Fs * 1.0 / N);     //actual
frequency.We start with the highest one!!!!!
    double temp = sqrt(Xreal[i] * Xreal[i] + Xim[i] * Xim[i]);
//temporal magnitude to compute the maximum
```

Firstly, goes through the FFT array obtaining the frequency and its magnitude value.

```
   if (j > -1)
    {
      freqsFFT[j] = nowFre;        //We take the frequencies in
ascending order
      magFFT[j] = temp * 2 / N;
      j--;
    }
```

With this data, we store the frequencies in ascending order to short the frequencies and magnitudes.

```
   if (temp > maxvalue) {
      maxk = nowFre;
      maxvalue = temp;
    }
  }
```

And at the end of the *for loop* statement, we store the maximum value discovered in the FFT and the frequency where it is.

```
//DC Component
  for (int k = 0; k < N; k++)
  {
    Xreal[0] = Xreal[0] + input_wave[k];
  }
  //magFFT contains the half of the samples! The average is then 2*N
  magFFT[0] = Xreal[0] / 2 / N;

  return maxk;
}
```

Then, we compute de DC component of the signal that is only the sum of all the averaged terms of the input signal. With this, we have the FFT finished and also we have the maximum value of it.

7.7.4.- sendData2Esp function

```
void sendData2Esp(void) {
    Serial.println("Starting communication with ESP:");
    delay(50);
    Esp8266.print("F");
    while (!Esp8266.available());
    inStr = Esp8266.readString();
    if (inStr == "G") {
      Esp8266.print(freq2ESP);
      Serial.print("Sended as freq: ");
      Serial.println(freq2ESP);
      delay(50);
      while (!Esp8266.available());
      inStr = Esp8266.readString();
      if (inStr == "H") {
        Esp8266.print(volt2ESP);
        delay(50);
        Serial.print("Sended as voltage: ");
        Serial.println(volt2ESP);
      }
    }
    if(inStr=="H")
    Serial.println("Packet Send");
}
```

We saw that this function is activated when it is been a minute. Also, we saw in the tests section that the protocol starts with the character “F”. Then, the microcontroller waits actively to the character “G”. This wait is active because the ESP8266 code is also implemented in a continuous way. After that, we can read in above code the rest of the protocol explained before.

7.7.5.- drawScreen1 function

```
/*Functions to draw the screen******************************/
void drawScreen1(ILI9341_due &d)
{
  d.setTextScale(1);
  d.fillScreen(ILI9341_BLACK);
  myButtons.enableButton(dispButton);
  myButtons.drawButton(dispButton);
  //delay(200);
  myButtons.enableButton(fftButton);
  myButtons.drawButton(fftButton);
  //delay(200);
  myButtons.enableButton(logButton);
  myButtons.drawButton(logButton);
  //delay(200);
  myButtons.enableButton(selectFreqButton);
  myButtons.drawButton(selectFreqButton);
}
```

This function just erase the screen in black and draw the buttons of the main screen.

7.7.6.- drawGraph function

```
void drawGraph(ILI9341_due &d)
{
  /* Function that draw a graph in the TFT screen. To compute the 
values, we take
      into account that the axis are Y(10-200, 190 pixels of spam) 
X(54-310, 256 of spam)
  */
  tft.fillScreen(ILI9341_WHITE);
  d.drawFastHLine(Xo - 2, 191, 256, ILI9341_DARKRED);
  d.drawFastVLine(Xo - 2, Yo + 1, 180, ILI9341_DARKRED);
}
```

The function basically paints the axes. We take into account the pixels available in the screen. We have to take into account that the center of the axis is in the upper left corner on the screen.

7.7.7.- timeAxis function

```
void timeAxis(ILI9341_due &d)
{
  d.drawFastVLine(Xo + 64, 191, 2, ILI9341_BLACK);
  sprintf(xlabeltime[0], "%.2f", (float)64000 / (2 * Fs));
  d.printAt(xlabeltime[0], Xo + 41, 193);

  d.drawFastVLine(Xo + 128, 191, 2, ILI9341_BLACK);
  sprintf(xlabeltime[1], "%.2f", (float)128000 / (2 * Fs));
  d.printAt(xlabeltime[1], Xo + 105, 193);

  d.drawFastVLine(Xo + 192, 191, 2, ILI9341_BLACK);
  sprintf(xlabeltime[2], "%.2f", (float)192000 / (2 * Fs));
  d.printAt(xlabeltime[2], Xo + 170, 193);

  d.drawFastVLine(Xo + 256, 191, 2, ILI9341_BLACK);
  d.printAt("ms", Xo + 243, 193);

  //Horizontal lines to voltage reference
  d.printAt("V", Xo - 30, 0);

  d.drawFastHLine(Xo - 5, 105, 5, ILI9341_BLACK); //1,65 volts
reference
  d.printAt("1,65", Xo - 46, 97);

  d.drawFastHLine(Xo - 5, 149, 5, ILI9341_BLACK); //0,825 volts
reference
  d.printAt("0.83", Xo - 46, 146);

  d.drawFastHLine(Xo - 5, 61, 5, ILI9341_BLACK); //2,475 volts
reference
  d.printAt("2,48", Xo - 46, 58);

  d.drawFastHLine(Xo - 5, Yo + 7, 5, ILI9341_BLACK); //3.3 volts
reference
  d.printAt("3,3", Xo - 36, Yo + 4);

  d.printAt("0", Xo - 5, 195);
}
```

This function paints the labels of the axes. We can see that the value showed is a value obtained from the sampling frequency. The procedure to obtain the exact pixel is basically divide the pixels painted as axis, and then divide the sampling frequency in the same proportion. For the Y axis the procedure is the same. We divide the axis and establish the correct position in pixels of the value.

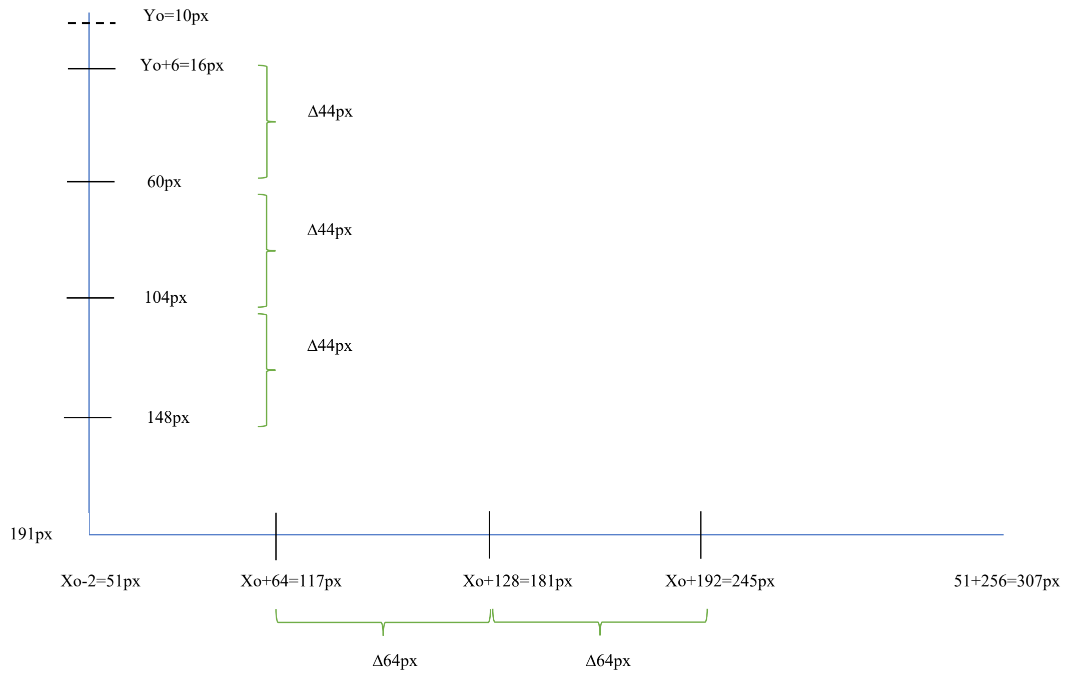
7.7.8.- freqAxis function

```
void freqAxis(ILI9341_due &d)
{
  //Horizontal lines to voltage reference
  d.printAt("V", Xo - 30, 0);
  d.drawFastHLine(Xo - 5, 104, 5, ILI9341_BLACK); //1,65 volts
reference
  d.printAt("1,65", Xo - 46, 96);

  d.drawFastHLine(Xo - 5, 148, 5, ILI9341_BLACK); //0,825 volts
reference
  d.printAt("0.83", Xo - 46, 145);

  d.drawFastHLine(Xo - 5, 60, 5, ILI9341_BLACK); //2,475 volts
reference
  d.printAt("2,48", Xo - 46, 57);

  d.drawFastHLine(Xo - 5, Yo + 6, 5, ILI9341_BLACK); //3.3 volts
reference
  d.printAt("3,3", Xo - 36, Yo + 3);

  d.printAt("0", Xo - 5, 195);

  //Vertical marks to reference the frequencies
  d.drawFastVLine(Xo + 64, 191, 2, ILI9341_BLACK);
  //As we draw a sample every two pixels Xo+64 -> freqsFFT[64/2]
  sprintf(xlabelfreq[0], "%.2f", freqsFFT[64 / 2] / 1000);
  d.printAt(xlabelfreq[0], Xo + 41, 193);

  d.drawFastVLine(Xo + 128, 191, 2, ILI9341_BLACK);
  sprintf(xlabelfreq[1], "%.2f", freqsFFT[128 / 2] / 1000);
  d.printAt(xlabelfreq[1], Xo + 105, 193);

  d.drawFastVLine(Xo + 192, 191, 2, ILI9341_BLACK);
  sprintf(xlabelfreq[2], "%.2f", freqsFFT[192 / 2] / 1000);
  d.printAt(xlabelfreq[2], Xo + 170, 193);

  d.drawFastVLine(Xo + 254, 191, 2, ILI9341_BLACK);
  d.printAt("kHz", Xo + 235, 193);
}
```

This function works in same way than the previous one. It divides the Y axis to establish the voltage value, and then establishes vertical marks to put the frequencies as a reference to the user.

As we draw each sample every two pixels, we know that, as we see in the code, 64 pixels mean the sample number 32.

We can see the graph area in pixels in the following way. Using the same procedure for the time graph.



```
#define Y_MAX 240
#define X_MAX 320

gTextArea graphArea{Xo, Yo, X_MAX - Xo, 180}
```

As we can see, the graph will be painted starting from Xo, that is 53 pixels, so the difference between the first frequency label is also 64 pixels. The same for the Y axis, that reaches a length of 190, discarding the line of the x axis.

8.- Conclusions and future work

After all the parts covered, we can see that the system works as expected. In the User Manual we can see how to handle the system and how the system responds correctly with a known input signal.

We saw that the libraries used work properly, but we do not know entirely the behavior of them. The libraries use the dynamic memory to reach their objectives, but this left to us an amount of memory unknown to write the code. This has given us several problems. The *Serial.print()* function, for example, uses dynamic memory to store the information to send or to receive. Also, the buttons need some amount of memory to exist. All of these took us to situations in where the behavior of the system was unexpected. We saw for example shift displacement in the FFT results if we do not erase the buttons when we return back to the main screen, also the timestamp of the sampled data was a meaningless string when we can see in other part of the code that it was right, etc.

The use of the libraries and also the Arduino environment have showed us that for complex project might be not recommended unless you are an experience user in embedded system programming.

On the other hand, we have seen how easy the Arduino programming can be. The use of the functions implemented in Wiring project allow us to work without spend too much time in details and certain aspects as the bit handle. The protocols used in the microcontroller are very well implemented in the libraries and for that reason we do not take care of them in the code. In addition to this, the use of the DMA in the screen library allow us to paint quickly the signal, that is basically in projects where the user see the signal.

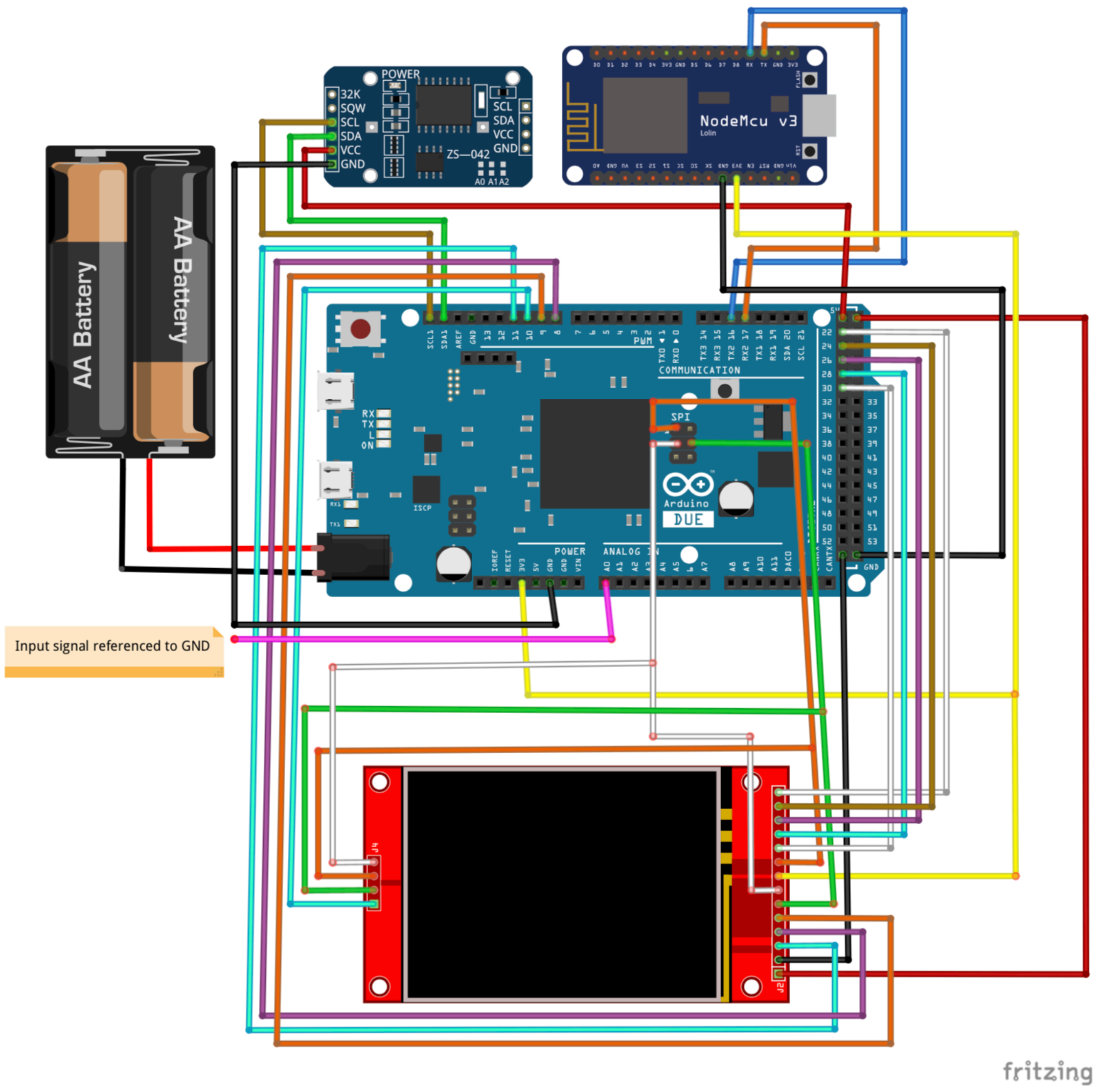
We can say too, that the Cortex M3 has been a good choice. With other microcontrollers in the market, the price of the Cortex M3 and its implementation in an Arduino compatible board, has made it possible to implement the End-of-Grade work in the semester time.

Also, we have seen that the second microcontroller has much more potential than the use of the GET method. This opens to us new possibilities to improve the system due to the capabilities of them. An embedded server, new peripherals, stand by modes or new features are just some examples that can be implemented in our system.

As a future work, there are several things to do with this project. A design in another IDE with other compilers would be necessary in order to improve the system and have more control over the code and the microcontroller behavior. The handle of the memory begins necessary in large projects. Furthermore, the energy of the system would be improved also. The system could be in a place without internet and without power supplying. With the SD card there is no problem to put the system far away from these electric requirements. In order to do that, the system could have a green energy suppling which charges the system over the day, for example, and does not run out of power. Also, a new design in PCB with a proper case would be much more professional in the presentation of the system. Although these features are not too much complicated to implement, the limited amount of time to develop the project left us these features in the future work section.

After all the work done, and this is my opinion, we have learned a lot of embedded designs. We have developed a digital system that can be modified without any change in the hardware and this is the key to establish a foundation for a future project complex, elaborated and with a robust and efficient aspect.

Scheme of the project



Budget

Hardware resources

Components	Price (€)
Arduino Due	35,00
TFT Screen 2.8-inch ILI9341	8,30
4 GB SD card	4,29
RTC DS3231	6,79
ESp8266	
8 x Battery AA Rechargeable	10,00
AA Battery charger	5,00
Battery case	1,50
Breadboard wires	6,39
Acrylic case, screw and DIY material	20,22
Total	**97,49**

Software resources

Concept	Euros/hour	Hours	Amount (€)
Microcontroller programming	36	80	2.880
Algorithms developed to test the project	36	60	2.160
Study about other embedded platforms	36	80	2.880
Total		**7.920**	

Writing and typing

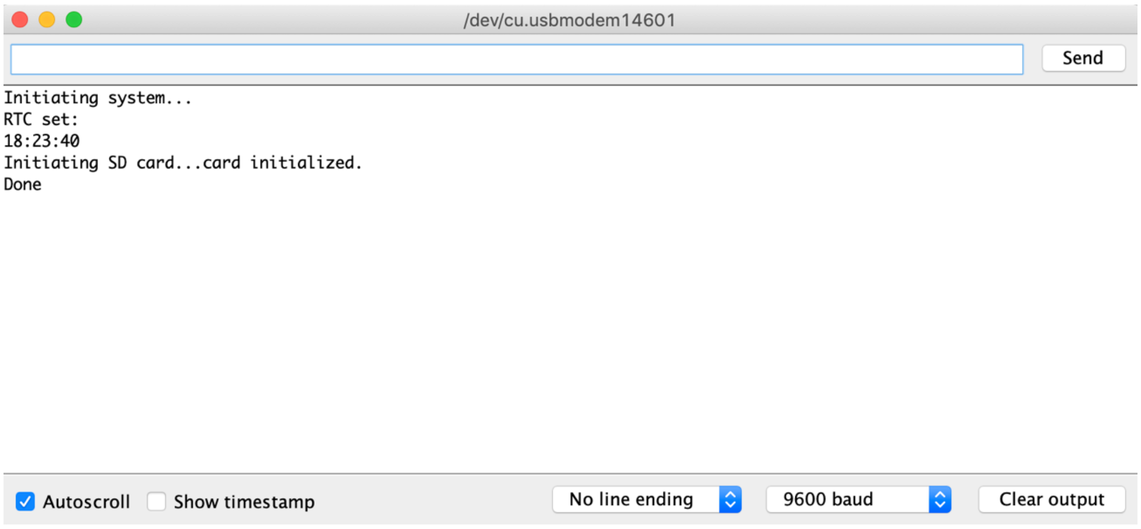
Concept	Euros/hour	Hours	Amount (€)
Telecommunication engineer	44	300	13.200
Memory typing	9	70	630
Total		**13.830**	

TOTAL

Concept	Amount (€)
Hardware resources	97,49
Software resources	7.920
Writing and typing	13.830
Total	**21.847,49**

User manual

We are to explain the use of the system in debug mode, powering the microcontroller with the USB cable. When we power the system, and we open the serial monitor at 9600 baud, we see



```
Initiating system...
RTC set:
18:23:40
Initiating SD card...card initialized.
Done
```

Figure 40: Initiating system in serial monitor

This means that the system initiates correctly. Once the system is initialized, we can see the main screen in the LCD.

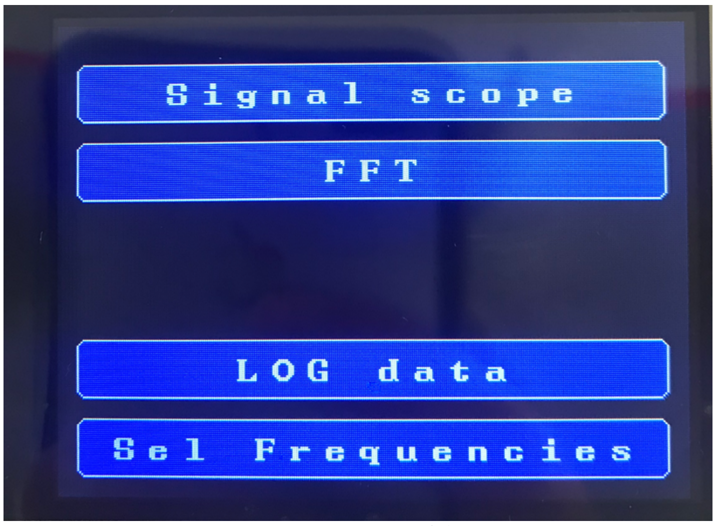
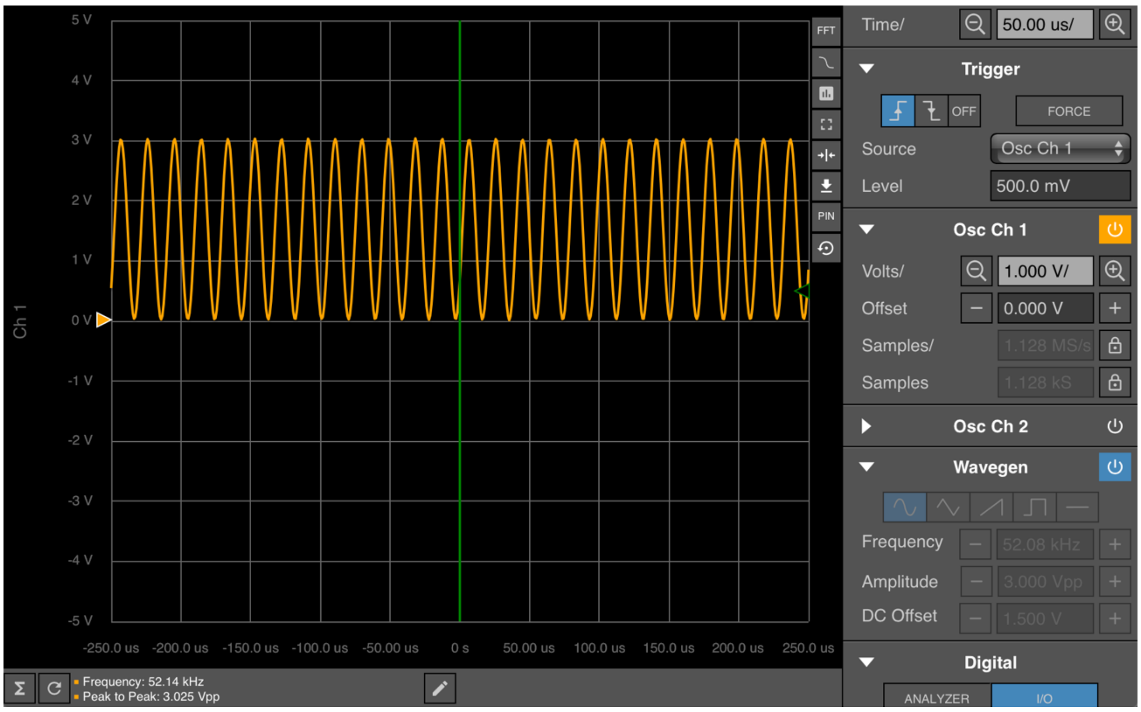


Figure 41: Main screen of the system

Then, we can select any of the 4 features available. The "*Signal scope*" to see the input signal in time domain, the "*FFT*" to see the spectrum of the input signal below 87 kHz, "*LOG data*" to start logging the data in the SD card and sending to the server the frequency number one with its magnitude, and "*Sel Frequencies*" to save the 4 frequencies of interest for the user in order to log data.

As an example, we are going to set as an input a sinusoidal signal of 50 kHz, with 3 V peak to peak with an offset of 1,5 v. The aspect of the input signal, according to the oscilloscope used is:



Frequency: 52.14 kHz
Peak to Peak: 3.025 Vpp

Time/ 50.00 us/

Trigger
Source Osc Ch 1
Level 500.0 mV

Osc Ch 1
Volts/ 1.000 V/
Offset 0.000 V
Samples/ 1.128 MS/s
Samples 1.128 kS

Osc Ch 2

Wavegen
Frequency 52.08 kHz
Amplitude 3.000 Vpp
DC Offset 1.500 V

Digital

Figure 42: Input signal for the example

We can see at the bottom right corner of Figure 42, that we set the wave generator with a signal of 52,08 kHz. The values of the bottom left corner are values computed by the oscilloscope according to the data sampled.

If we pulse the "*Signal scope*" button, what we see is:

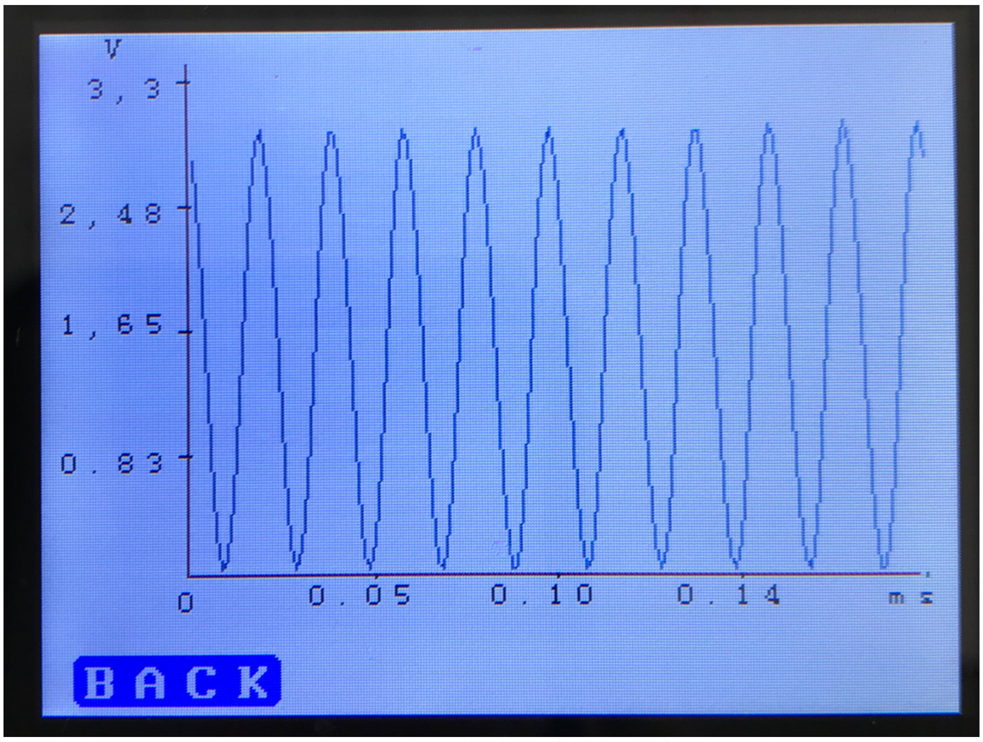


Figure 43: Signal scope of the example

Where we can see that the values correspond with the input parameters, with 3 V of maximum value and average value of 1,5 V.

To measure the signal properly, we use the FFT feature of the system. If we pulse "*BACK*" button, and then we pulse "*FFT*" button, we can see the following figure:

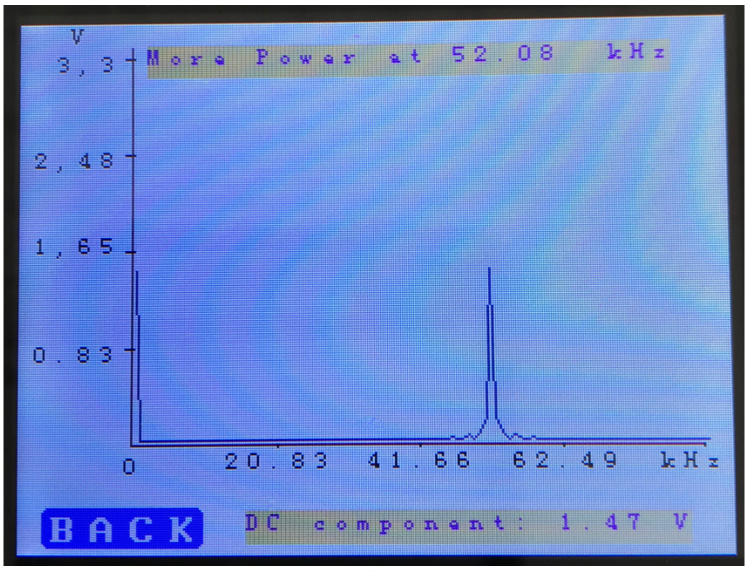
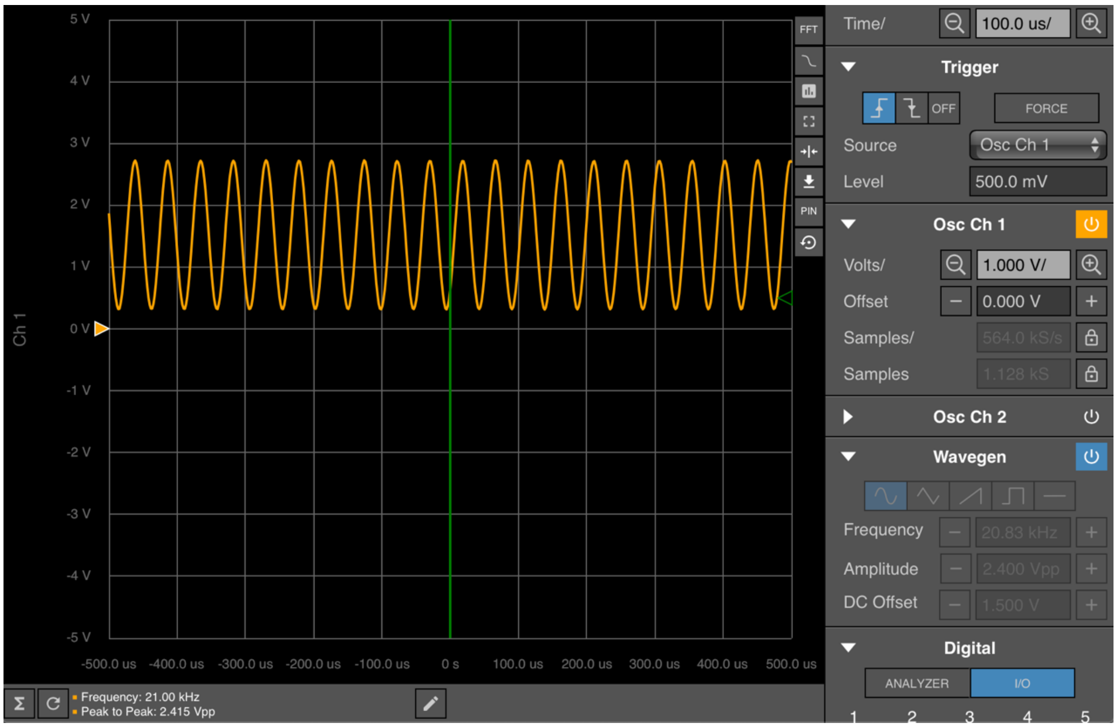


Figure 44: FFT of the input example

The normal lapse of time between the screen updates is 1,20 seconds approximately. This interval of time can be modified through the *refresh_screen* variable. We can see that the DC component corresponds with the input data and also the harmonic. As the input is a pure sinusoidal, we only have one harmonic that correspond with the maximum power of the input signal.

To check again the system, we are going to change the input signal, in order to distinguish between DC component and the harmonic we want to measure. In this case, to match the frequencies in the FFT array and the example, we are going to input the next signal:



Frequency: 21.00 kHz
Peak to Peak: 2.415 Vpp

Time/ 100.0 us/
Source: Osc Ch 1
Level: 500.0 mV
Volts/ 1.000 V/
Offset: 0.000 V
Samples/ 564.0 kS/s
Samples: 1.128 kS
Frequency: 20.83 kHz
Amplitude: 2.400 Vpp
DC Offset: 1.500 V

Figure 45: Second input signal of the example

With these, we should have a pure harmonic in our FFT with an amplitude of 1,2 V and with DC component of 1,5 V.

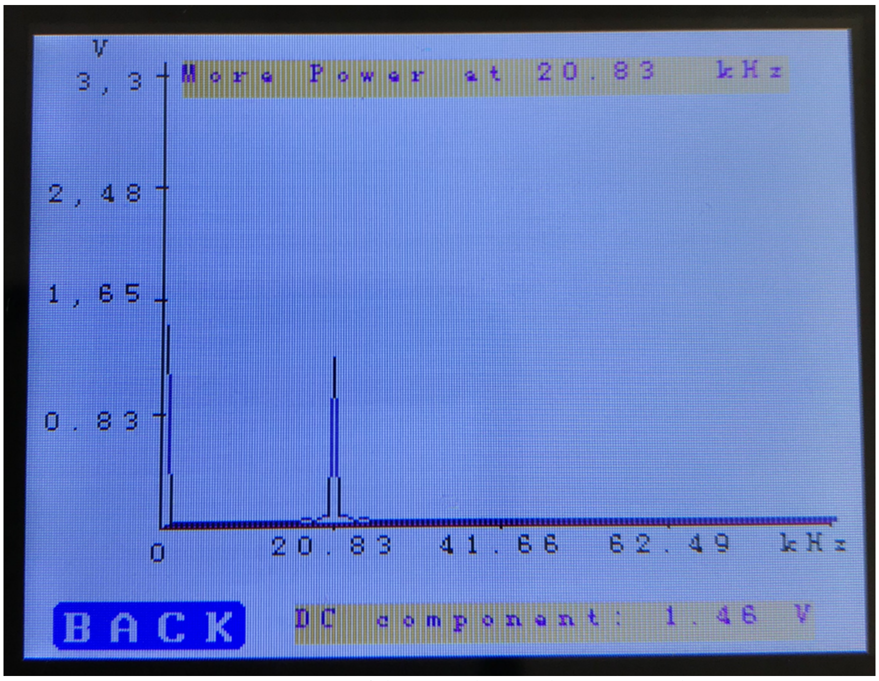


Figure 46: FFT of the second example input signal

Then we go back pressing the "*BACK*" button in order to select the frequencies of interest. For this example, we are going to select 20,83 kHz as frequency 1, and the DC component for frequency 2. When we push "*Sel Frequencies*" we see:

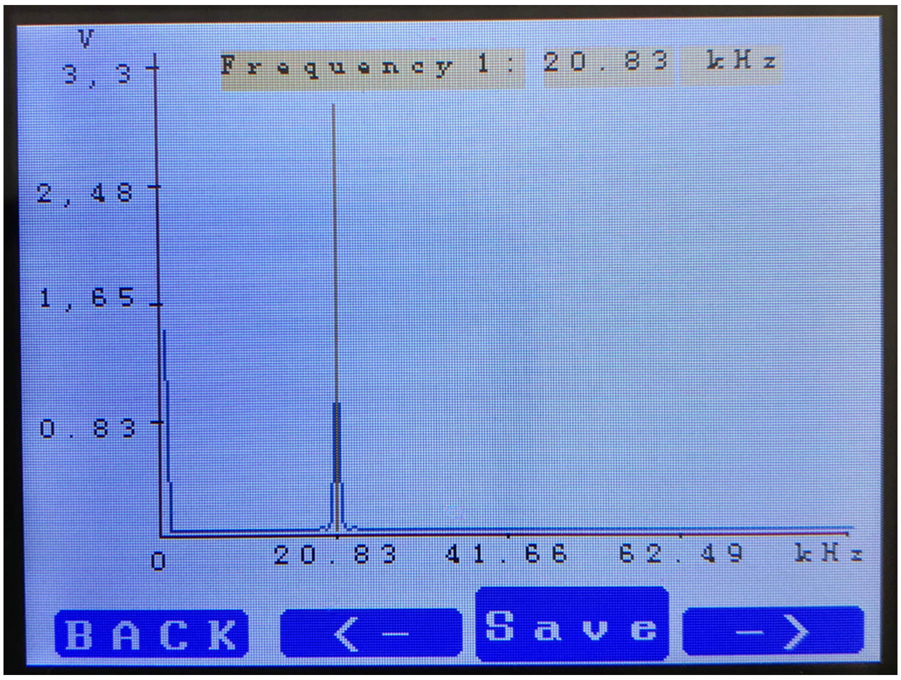


Figure 47: Sel Frequencies screen 1

Where we can see the vertical line and the title indicating us the number of the frequency we are going to save and the frequency itself. By default, the 20,83 kHz frequency is selected. We can go through the spectrum with the arrow's buttons. If we pulse the right button, we can see the change:

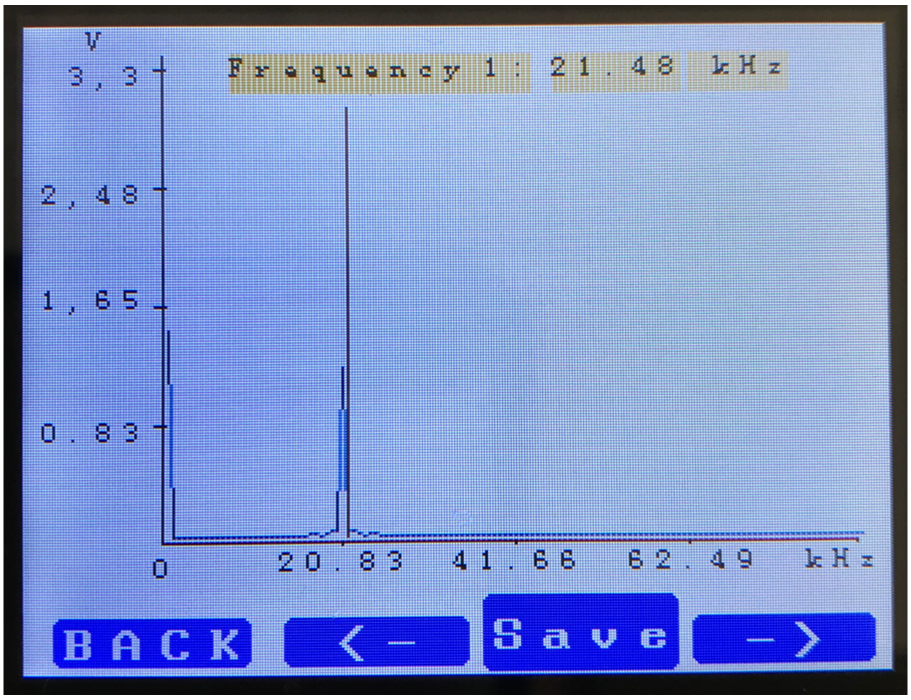


Figure 48: Sel Frequencies screen 2

And we can see the next frequency bin in our FFT array. With this procedure, we can select the frequency of interest. As we want to store the 20,83 kHz in frequency 1, we pulse the left arrow and then pulse the "*Save*" button. When we pulse the button, we can see, during one frame which is approximately 1,2 seconds, the next text on the screen:

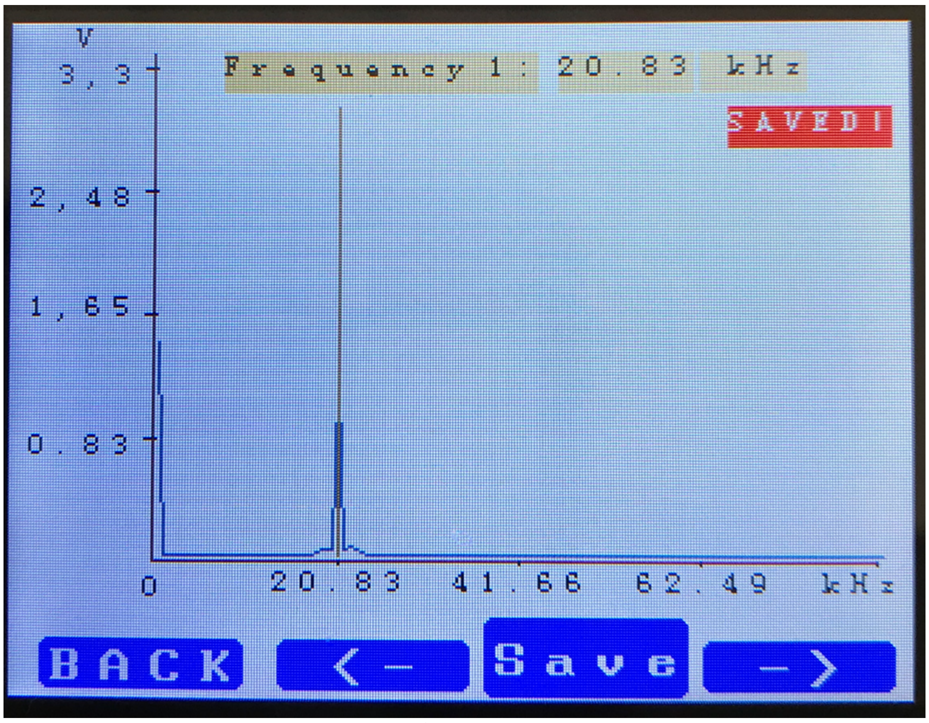


Figure 49: Sel Frequencies screen 3

Which means that the frequency selected when the user presses the "*SAVE*" button is stored in frequency 1. After the frame, the system shows the same screen but with the label frequency 2 in the title.

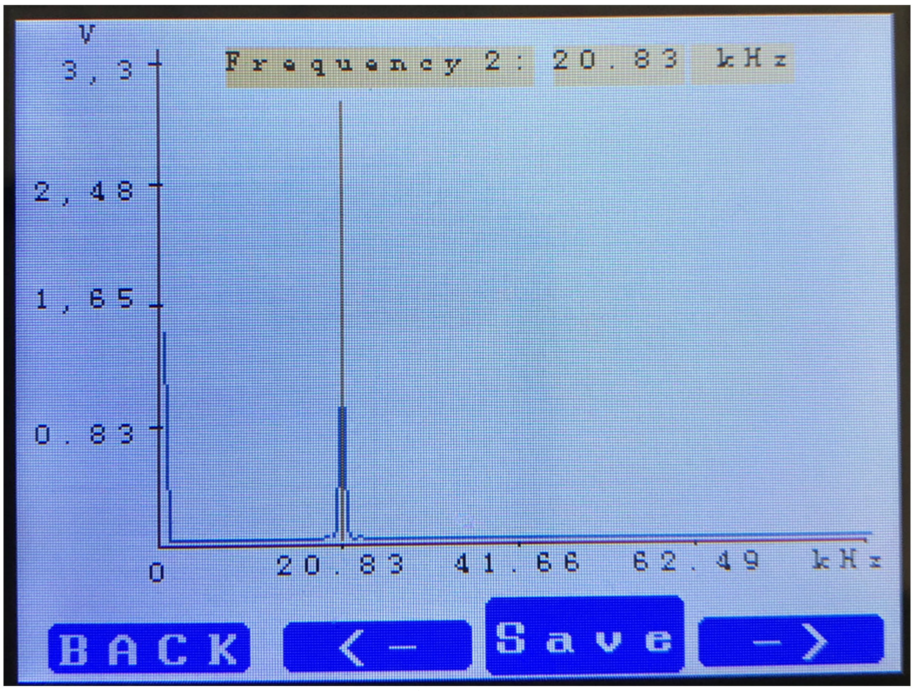
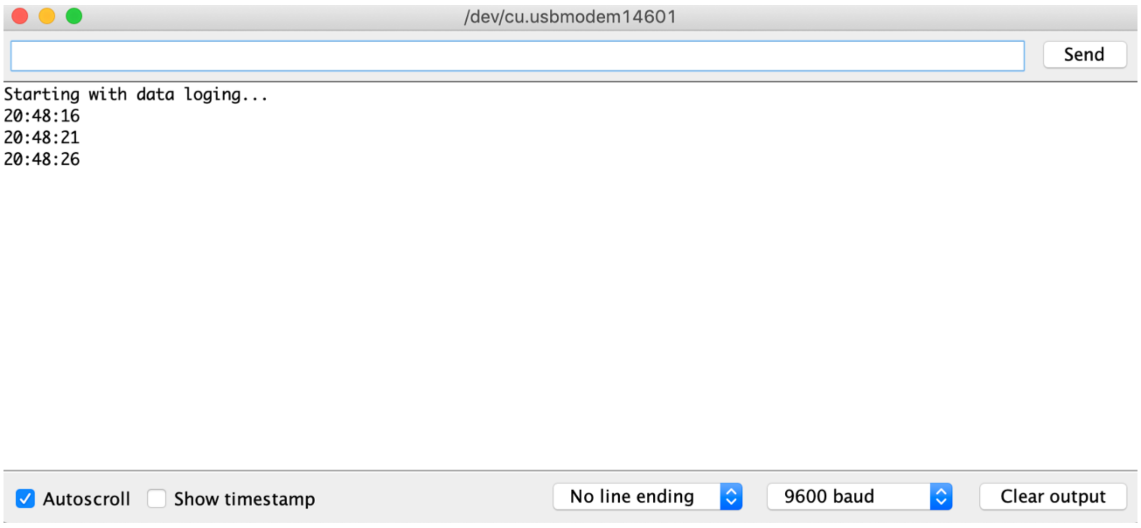


Figure 50: Sel Frequencies screen 4

Then, we can go to the DC component through the left button. Once we are at frequency 0,0 kHz, we pulse the "SAVE" button again. The system, by default, has the DC component as the frequencies to sample in the 4 frequencies available. So, the selection of the DC component in the frequency 2 is not necessary, but it is mentioned because we can only measure one harmonic and the DC component for the example.

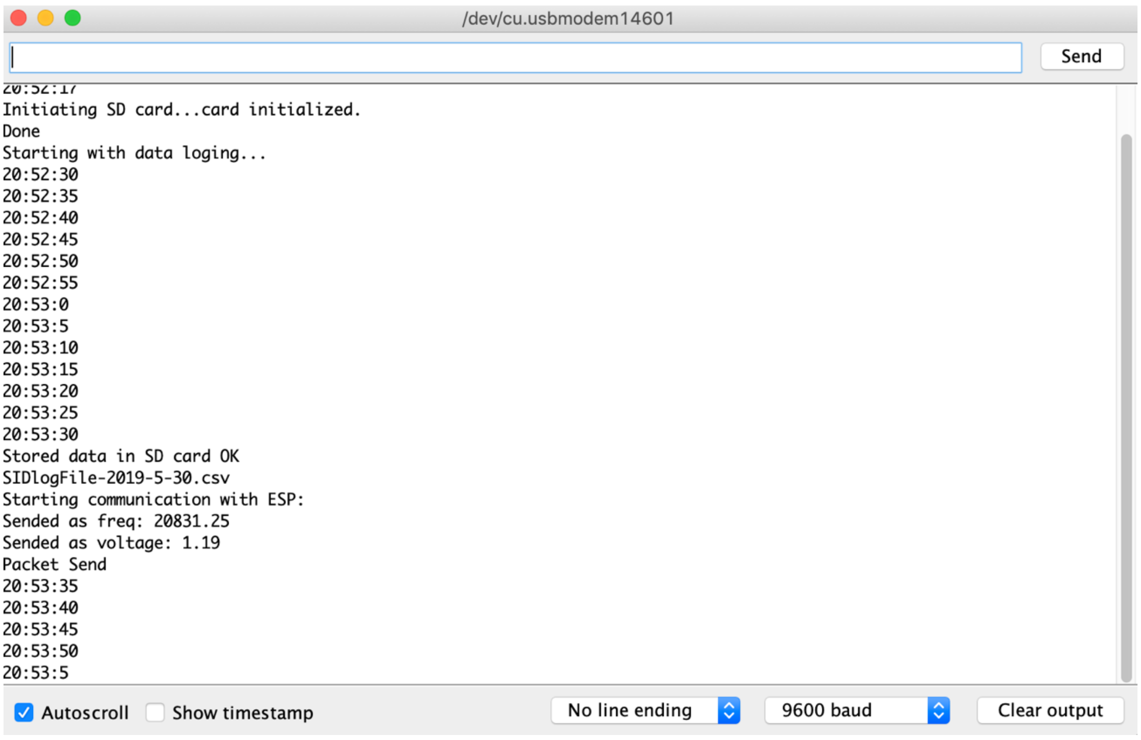
Once we have the desired frequencies stored, we can pulse "*BACK*" button to return back to the main screen. Then, we can start to log the data by the pulse of the "*LOG data*" button. As we are in debug mode, once we pulse the button, what we see is the next figure:



```
Starting with data loging...
20:48:16
20:48:21
20:48:26
```

Figure 51: LOG data debuging 1

That shows us that the system is logging the data each 5 seconds. When it has been a minute, we can see the following debugging comments.



```
20:52:17
Initiating SD card...card initialized.
Done
Starting with data loging...
20:52:30
20:52:35
20:52:40
20:52:45
20:52:50
20:52:55
20:53:0
20:53:5
20:53:10
20:53:15
20:53:20
20:53:25
20:53:30
Stored data in SD card OK
SIDlogFile-2019-5-30.csv
Starting communication with ESP:
Sended as freq: 20831.25
Sended as voltage: 1.19
Packet Send
20:53:35
20:53:40
20:53:45
20:53:50
20:53:5
```

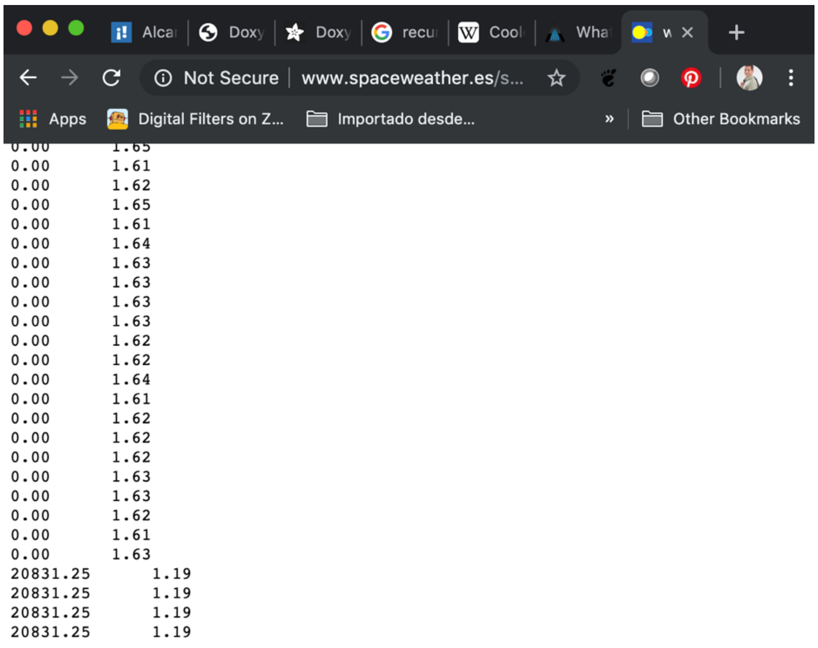
Figure 52: LOG data debugging 2

Where we can see that the system stores in the SD card all the values and start the communication with the ESP8266. Once it sends the packet through the GET method, we can see how the system continues with the sample every 5 seconds.

If we go to the website created for the purpose of this End-of-Grade work:

http://www.spaceweather.es/sid/test/data.txt

We can see all the results for the selected frequency 1 of our system. In the moment we have made this user manual, we can see in that webpage the next results:

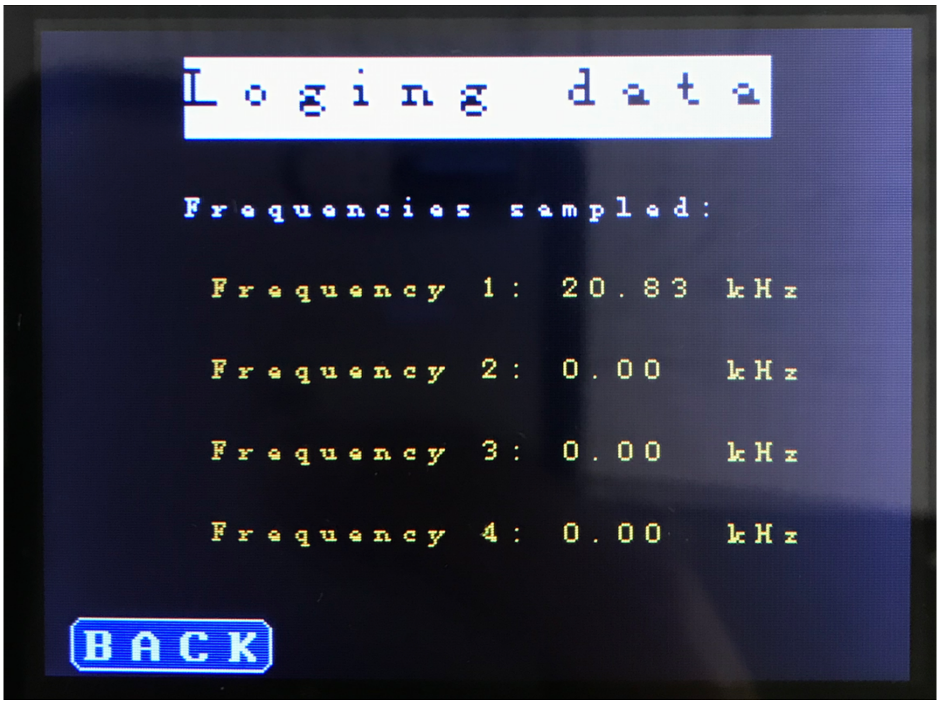


```
0.00        1.65
0.00        1.61
0.00        1.62
0.00        1.65
0.00        1.61
0.00        1.64
0.00        1.63
0.00        1.63
0.00        1.63
0.00        1.63
0.00        1.62
0.00        1.62
0.00        1.64
0.00        1.61
0.00        1.62
0.00        1.62
0.00        1.62
0.00        1.63
0.00        1.63
0.00        1.62
0.00        1.61
0.00        1.63
20831.25        1.19
20831.25        1.19
20831.25        1.19
20831.25        1.19
```

Figure 53: Webpage with the result of frequency 1

Where we can see that we sampled the DC component before the manual, and then, with the 20,83 kHz frequency selected as frequency one and in the LOG data mode, the corresponding results are showed.

The LOG data screen in our system is the following one:



Loging data

Frequencies sampled:

Frequency 1: 20.83 kHz

Frequency 2: 0.00 kHz

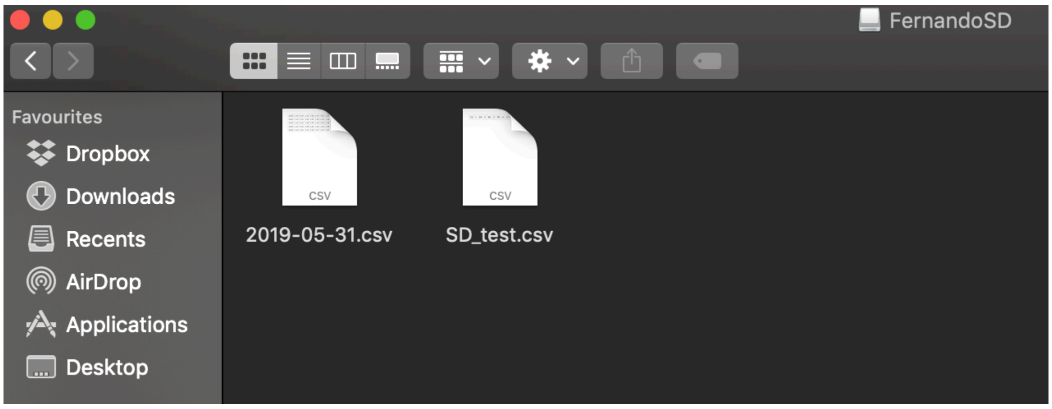
Frequency 3: 0.00 kHz

Frequency 4: 0.00 kHz

Figure 54: LOG data screen

Where we can see that the system is logging the data and it shows us the frequencies that have been selected.

The data is stored in the SD card in the form of files with the name of the day. We can power down the system and extract the SD card from the screen in order to see the files and its contents. To maintain the same procedure than the one shown before, we put the system to sampling the 20,83 kHz frequency some minutes at the next day. This is just an example, and with several days the system will have a file per day. If we check the content of the SD card for this purpose, we can see that:

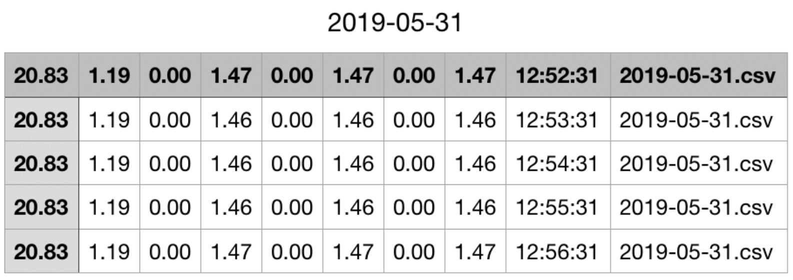


2019-05-31.csv

SD_test.csv

Figure 55: SD card content

The SD card has the file created for the test and also the file created while this user manual has been created. If we open the file corresponding with the day when this manual has been created:



2019-05-31

20.83	1.19	0.00	1.47	0.00	1.47	0.00	1.47	12:52:31	2019-05-31.csv
20.83	1.19	0.00	1.46	0.00	1.46	0.00	1.46	12:53:31	2019-05-31.csv
20.83	1.19	0.00	1.46	0.00	1.46	0.00	1.46	12:54:31	2019-05-31.csv
20.83	1.19	0.00	1.46	0.00	1.46	0.00	1.46	12:55:31	2019-05-31.csv
20.83	1.19	0.00	1.47	0.00	1.47	0.00	1.47	12:56:31	2019-05-31.csv

Figure 56: Results in SD card corresponding to the day of the user manual creation

And we can see the first frequency for this example that is 20,83 kHz, with and amplitude of 1,19 V, (2,44 V peak to peak of the input means 1,2 V of input amplitude), and a DC component sampled in frequencies 2, 3 and for of 1,47 V (1,5 V DC component of the input).

With this, the User manual has been covered. We saw how to manage the system in order to store the frequency components of the input signal in order to study their behavior lately.

Bibliography

[1] R. Redmon, D. B. Seaton, R. Steenburgh, J. He and J. V. Rodriguez, "September 2017's Geoeffective Space Weather and Impacts to Caribbean Radio Communications During Hurricane Response," *Space Weather,* no. 16, pp. 1190-1201, 2018.

[2] E. Gilbert, "GitHub SuperSID," [Online]. Available: https://github.com/ericgibert/supersid.

[3] I. b. Muñoz, Subject of Electronic Subsystems, vol. Data conversion, University of Alcalá de Henares, 2018.

[4] H. Nyquist, "Certain topics in telegraph transmission theory," April 1928. [Online]. Available: https://web.archive.org/web/20130926031230/http://www.ieee.org/publications_standards/publications/proceedings/nyquist.pdf.

[5] Wikipedia, "Succesive Approximation ADC," [Online]. Available: https://en.wikipedia.org/wiki/Successive_approximation_ADC.

[6] M. Integrated, "Tutorial 1023," [Online]. Available: https://www.maximintegrated.com/en/app-notes/index.mvp/id/1023.

[7] M. Integrated, "Understanding Pipelined ADCs," March 2001. [Online]. Available: http://materias.fi.uba.ar/6644/info/varios/conversores/basico/Understanding%20pipelined%20ADCs.htm.

[8] M. B. Velasco, "La Transformada Discreta de Fourier," in *Tratamiento Digital de Señales*, Servicio de Publicaciones Universidad de Alcalá, 2013.

[9] A. V. Oppenheim and A. S. Wilsky, Signals and Systems, Pearson Prentice Hall, 1997.

[10] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation,* no. 19, pp. 297-301, 1965.

[11] Wikipedia, "Microcontroller," [Online]. Available: https://en.wikipedia.org/wiki/Microcontroller.

[12] A. G. Baquero, "Recursivity," in *Theory slides of Programming Subject*, 2016.

[13] J. Pastor and J. M. Villadangos, Subject of Advanced Digital Electronic Systems, University of Alcalá de Henares, 2019.

[14] A. Due, "Arduino Store," [Online]. Available: https://store.arduino.cc/due.

[15] ATMEL, "Datasheet SAM3X/SAM3A Series," [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/atmel-11057-32-bit-cortex-m3-microcontroller-sam3x-sam3a_datasheet.pdf.

[16] L. Wiki, "SKU:MSP2807," [Online]. Available: http://www.lcdwiki.com/2.8inch_SPI_Module_ILI9341_SKU:MSP2807.

[17] J. J. Purdum, Beginning C for Arduino: learn C programming for the Arduino, New York: Apress, 2015.

[18] Wiring, "Wiring project," [Online]. Available: http://wiring.org.co/.

[19] M. Buriak, "ILI9341_due library," [Online]. Available: http://marekburiak.github.io/ILI9341_due/.

[20] P. Stoffregen, "ili9341_t3," [Online]. Available: https://github.com/PaulStoffregen/ILI9341_t3.

[21] B. Greiman. [Online]. Available: https://github.com/greiman/SdFat.

[22] M. Margolis and B. Perry. [Online]. Available: https://code.google.com/p/glcd-arduino.

[23] H. Karlsen, "URTouch library," Rinky-Dink Electronics, [Online]. Available: http://www.rinkydinkelectronics.com/library.php?id=92.

[24] G. Lawrence, "ILI9341_due_Buttons," [Online]. Available: https://github.com/ghlawrence2000/ILI9341_due_Buttons.

[25] Adafruit, "RTClib," [Online]. Available: https://github.com/adafruit/RTClib.

[26] Atmel, "SAM libraries," [Online]. Available: https://github.com/arduino/ArduinoCore-sam/blob/master/system/libsam/include/adc.h.

[27] DIGILENT, "OpenScope MZ," DIGILENT, [Online]. Available: https://reference.digilentinc.com/reference/instrumentation/openscope-mz/start.

Annex I Main code

```
//Use of Screen
#include <SPI.h>
#include <SdFat.h>
#include <ILI9341_due.h>
#include <URTouch.h>
#include <ILI9341_due_Buttons.h>
#include "SmallFont.h"
#include "BigFont.h"

//Use of RTC
#include "RTClib.h"

//Some necessary libraries
#include <Wire.h>
#include <stdint.h>

#define SD_SPI_SPEED SPI_HALF_SPEED  // SD card SPI speed

#define Fs 666600 //ADC_FREQ_MAX implies this Fs. See example ADC_Due

#define Y_MAX 240               //Maximum pixels in Y axis
#define X_MAX 320               //Maximum pixels in X axis
#define Xo 53                   //Pixels for axis (0,0)
#define Yo 10

// LCD
#define TFT_RST 8
#define TFT_DC 9
#define TFT_CS 11
//SD
#define SD_CS 10
//Touch pannel
#define T_CLK 30
#define T_CS 28
#define T_DIN 26
#define T_DOUT 24
#define T_IRQ 22

#define Esp8266 Serial2

//File that we will save in the SD
SdFat sd; // set filesystem
SdFile logSIDSWAP;

// Use hardware SPI
ILI9341_due tft = ILI9341_due(TFT_CS, TFT_DC, TFT_RST);

URTouch myTouch(T_CLK, T_CS, T_DIN, T_DOUT, T_IRQ);

// Finally we set up ILI9341_due_Buttons :)
ILI9341_due_Buttons  myButtons(&tft, &myTouch);

RTC_DS3231 rtc;

/*** GLOBAL VARIABLES*************************/
int pressed_button;
//check if a button is already pushed
boolean pressed = false;
```

```
//in order to paint values in the graph
boolean paint_time = false;
boolean paint_fft = false;
boolean log_data = false;
boolean sel_freq = false;
boolean showFrequencies = false;
//Constant that extrapolates the voltage of the signal to adequate it
to pixels in the graph
uint16_t pixel_mag_factor = 52; //54.54 (180/3.3)
uint16_t pixel_magfreq_factor = 52;
//To store the values in time of x-axis in time draw
char xlabeltime[3][10];
char xlabelfreq[3][10];
//Variable to do a passive wait to refresh the draw of the time signal
in the graph
uint32_t refresh_screen = 100;

int backButton, dispButton, fftButton, logButton, selectFreqButton;
int saveButton, leftButton, rightButton;

/*FFT stuff**************************************************/
const uint16_t N = 1024;
double input_wave[N];
double re[N], im[N], Xr[N], Xi[N];
double freqsFFT[N / 2], magFFT[N / 2], freqsSampled[4];
double preMax = -100.0;
//String to contain the max value of FFT and print it in the screen
char spreMax[20] = "0";
char dcComponent[20] = "0";

/*RTC stuff**************************************************/
DateTime now;
DateTime after;

/*LOG stuff**************************************************/
//Array containing the indexes of frequencies to sample in FFT array
(freqsFFT[N/2])
uint16_t logFFTindexes[4];
//to store the index temporary of FFT to save frequency
uint16_t indFx = 32;
//to print frequency 1, frequency 2, etc
uint8_t logFQIndexes = 0;
char logFQIndexesStr[4][3];
//To store the values of frequencies sampled
char freqSampledString[4][25];
//to show in sel frequencies screen the fq to store
char freqSampledStringTemp[4 + 1][25];
//Store the value every 5 seconds. 12 values every minute and 4
frequencies to store
double magValues5sec[4][60 / 5];
//index to store data in the above array
int logindex5sec = 0;
//value to store in SD card
double valueSD_minute[4][1];
//Name of the file
char nameLogFile[50];
//Name of timestamp
char timestamp[12] = "00:00:00";

/*Esp8266 Stuff**************************************************/
String freq2ESP = "0";
String volt2ESP = "0";
String inStr = "nothing";
bool data2Esp = false;

/*Areas
differentation***********************************************/
gTextArea graphArea{Xo, Yo, X_MAX - Xo, 180}; //170 available pixels
for draw signals in y-axis
gTextArea allArea{0, 0, X_MAX, Y_MAX};


/*** FUNCTIONS*************************/
void drawScreen1(ILI9341_due &d);
void timeAxis(ILI9341_due &d);
void freqAxis(ILI9341_due &d);
void drawGraph(ILI9341_due &d);

void sidFFT(double *X_real, double *X_im, double *xreal, double
*ximag, int freq_bin, int N, int h, int h_interval);
double postProcessing(double *Xr, double *Xi, double premax);

void Sampling(double *sw, double *re, double *im);
void configureADC(void);

void sendData2ESP(void);



void setup()
{
  //For PC communication
  Serial.begin(9600);
  //For Esp8266 communication we use Serial1,2 or 3 of Arduino Due
(Baudrate=115200)
  Esp8266.begin(115200);
  delay(2000);
  Serial.println(F("Initiating system..."));

  // Initial setup
  tft.begin();
  tft.setRotation(iliRotation270);  // landscape
  tft.fillScreen(ILI9341_BLACK);

  tft.setFont(SmallFont);

  myTouch.InitTouch();
  myTouch.setPrecision(PREC_MEDIUM);

  myButtons.setTextFont(BigFont);

  /*Buttons to be used in the LCD menu ***************************/
  /* Main Menu Buttons***************************/
  dispButton = myButtons.addButton( 10,  20, 300,  30, "Signal
scope");
  fftButton = myButtons.addButton(10, 60, 300, 30, "FFT");
  logButton = myButtons.addButton(10, 160, 300, 30, "LOG data");
  selectFreqButton = myButtons.addButton(10, 200, 300, 30, "Sel
Frequencies");
  backButton = myButtons.addButton( 10, 218, 75, 20, "BACK");

  /*To select the frequencies to sample*/
  leftButton = myButtons.addButton( 95, 218, 70, 20, "<-");
  rightButton = myButtons.addButton( 245, 218, 70, 20, "->");
  saveButton = myButtons.addButton(168, 210, 74, 30, "Save");

  if (! rtc.begin()) {
    Serial.println(F("Couldn't find RTC"));
    //while (1);
  }
  if (rtc.lostPower())
  {
    Serial.println("RTC lost power, lets set the time!");
    // following line sets the RTC to the date & time this sketch was
compiled
    rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
    // This line sets the RTC with an explicit date & time, for
example to set
    // January 21, 2014 at 23:59:00 you would call:
    // rtc.adjust(DateTime(2014, 1, 21, 23, 59, 40));
  }
  rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
  Serial.println(F("RTC set:"));
  now = rtc.now();
  sprintf(timestamp, "%02d:%02d:%02d", now.hour(), now.minute(),
now.second());
  Serial.println(timestamp);

  Serial.print(F("Initiating SD card..."));
  if (!sd.begin(SD_CS, SD_SPI_SPEED))
  {
    Serial.println(F("Card failed, or not present"));
    return;
  }
  Serial.println(F("card initialized."));

  sprintf(freqSampledString[0], "%.2f", 0);
  sprintf(freqSampledString[1], "%.2f", 0);
  sprintf(freqSampledString[2], "%.2f", 0);
  sprintf(freqSampledString[3], "%.2f", 0);

  Serial.println(F("Done"));
  tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
  drawScreen1(tft);
}

void loop()
{
  //to store the maximum value of FFT
  double preMax = -100.0;

  //data2Esp = false;

  now = rtc.now();

  //Always sampling the signal if we are not loging the data
  if (!log_data)
  {
    configureADC();
    Sampling(input_wave, re, im);
  }

  if (data2Esp) {
    if(Esp8266.available()) inStr = Esp8266.readString();
    sendData2Esp();
    data2Esp = false;
  }

  /*State machine that manages the behaviour of the project*/
  if (myTouch.dataAvailable() == true)
  {
    pressed_button = myButtons.checkButtons();

    if ((pressed_button == dispButton) && (!pressed))
    {
      tft.setTextArea(allArea);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(fftButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      myButtons.disableButton(leftButton);
      myButtons.disableButton(rightButton);
      myButtons.disableButton(saveButton);
      pressed = true;
      paint_time = true;
      paint_fft = false;
      log_data = false;
      sel_freq = false;
      drawGraph(tft);
      timeAxis(tft);
      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);
    }

    if ((pressed_button == fftButton) && (!pressed))
    {
      tft.setTextArea(allArea);
      myButtons.disableButton(fftButton);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      myButtons.disableButton(leftButton);
      myButtons.disableButton(rightButton);
      myButtons.disableButton(saveButton);
      pressed = true;
      paint_time = false;
      paint_fft = true;
      log_data = false;
      sel_freq = false;
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      drawGraph(tft);
      freqAxis(tft);
      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);
    }

    if ((pressed_button == logButton) && (!pressed))
    {
      myButtons.disableButton(fftButton);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      myButtons.disableButton(leftButton);
      myButtons.disableButton(rightButton);
      myButtons.disableButton(saveButton);
      pressed = true;
      paint_time = false;
      paint_fft = false;
      log_data = true;
      sel_freq = false;

      tft.fillScreen(ILI9341_BLACK);
      //Show the screen displaying that data are stored
      tft.setTextScale(2);
      tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
      tft.printAt("Loging data", Xo, Yo);
      tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
      tft.setTextScale(1);

      tft.printAt(F("Frequencies sampled:"), Xo, Yo + 50);

      tft.setTextColor(ILI9341_GOLD, ILI9341_BLACK);
      tft.printAt(F("Frequency 1: "), Xo + 10, Yo + 80);
      tft.printAt(freqSampledString[0], Xo + 140, Yo + 80);
      tft.printAt(" kHz", Xo + 190, Yo + 80);

      tft.printAt(F("Frequency 2: "), Xo + 10, Yo + 110);
      tft.printAt(freqSampledString[1], Xo + 140, Yo + 110);
      tft.printAt(" kHz", Xo + 190, Yo + 110);

      tft.printAt(F("Frequency 3: "), Xo + 10, Yo + 140);
      tft.printAt(freqSampledString[2], Xo + 140, Yo + 140);
      tft.printAt(" kHz", Xo + 190, Yo + 140);

      tft.printAt(F("Frequency 4: "), Xo + 10, Yo + 170);
      tft.printAt(freqSampledString[3], Xo + 140, Yo + 170);
      tft.printAt(F(" kHz"), Xo + 190, Yo + 170);

      now = rtc.now();
      //Alarm set 5 seconds later.
      after = now + TimeSpan(5);

      sprintf(nameLogFile, "%04d-%02d-%02d.csv", now.year(),
now.month(), now.day());

      Serial.println(F("Starting with data loging..."));
      sprintf(timestamp, "%02d:%02d:%02d", now.hour(), now.minute(),
now.second());
      Serial.println(timestamp);

      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);

      tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
    }

    if ((pressed_button == selectFreqButton) && (!pressed))
    {
      myButtons.disableButton(fftButton);
      myButtons.disableButton(dispButton);
      myButtons.disableButton(logButton);
      myButtons.disableButton(selectFreqButton);
      tft.clearTextArea(ILI9341_BLACK);
      pressed = true;
      paint_time = false;
      paint_fft = false;
      log_data = false;
      sel_freq = true;
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      drawGraph(tft);
      freqAxis(tft);
      myButtons.enableButton(leftButton);
      myButtons.drawButton(leftButton);
      myButtons.enableButton(saveButton);
      myButtons.drawButton(saveButton);
      myButtons.enableButton(rightButton);
      myButtons.drawButton(rightButton);
      myButtons.enableButton(backButton);
      myButtons.drawButton(backButton);
    }

    if ((pressed_button == backButton) && (!pressed))
    {
      myButtons.deleteAllButtons();
      dispButton = myButtons.addButton( 10,  20, 300,  30, "Signal
scope");
      fftButton = myButtons.addButton(10, 60, 300, 30, "FFT");
      logButton = myButtons.addButton(10, 160, 300, 30, "LOG data");
      selectFreqButton = myButtons.addButton(10, 200, 300, 30, "Sel
Frequencies");
      backButton = myButtons.addButton( 10, 218, 75, 20, "BACK");
      leftButton = myButtons.addButton( 95, 218, 70, 20, "<-");
      rightButton = myButtons.addButton( 245, 218, 70, 20, "->");
      saveButton = myButtons.addButton(168, 210, 74, 30, "Save");

      pressed = true;
      paint_time = false;
      paint_fft = false;
      log_data = false;
      sel_freq = false;
      tft.setTextArea(allArea);
      myButtons.disableButton(backButton);
      drawScreen1(tft);
    }

    if (sel_freq)
    {
      if ((pressed_button == leftButton) && (!pressed))
      {
        pressed = true;
        //paint black line before paint the new one
        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_WHITE);
        indFx -= 1;
        if (indFx < 0) indFx = 0;
        if (indFx > (N / 2) - 1) indFx = (N / 2) - 1;

        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_DARKGOLDENROD);
        sprintf(freqSampledStringTemp[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
      }
      if ((pressed_button == rightButton) && (!pressed))
      {
        pressed = true;
        //paint black line before paint the new one
        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_WHITE);
        indFx += 1;
        if (indFx < 0) indFx = 0;
        if (indFx > (N / 2) - 1) indFx = (N / 2) - 1;

        tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_DARKGOLDENROD);
        sprintf(freqSampledStringTemp[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
      }
      if ((pressed_button == saveButton) && (!pressed))
      {
        pressed = true;
        logFFTindexes[logFQIndexes] = indFx;
        sprintf(freqSampledString[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
        tft.setTextColor(ILI9341_WHITE, ILI9341_RED);
        tft.printAt("SAVED!", Xo + 150, Yo + 10);
        tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
        logFQIndexes ++;
        if (logFQIndexes > 3) logFQIndexes = 0;
      }
    }
  }

  else
  {
    pressed = false;
  }


  if (paint_time)
  {
    tft.setTextArea(graphArea);
    refresh_screen--;

    if (refresh_screen == 50)
    {
      //        Serial.println("Paint first half");
      tft.clearTextArea(ILI9341_WHITE);
      for (int i = 0; i < 128; i++)
      {
        tft.drawLine(Xo + i * 2, 189 - input_wave[i]*pixel_mag_factor,
Xo + (i + 1) * 2, 189 - input_wave[i + 1]*pixel_mag_factor,
ILI9341_STEELBLUE); //147 vs 189
      }
    }

    if (refresh_screen == 0)
    {
      tft.clearTextArea(ILI9341_WHITE);
      for (int j = 128; j < 255; j++)
      {
        tft.drawLine(Xo + ((j - 128) * 2), 189 -
input_wave[j]*pixel_mag_factor, Xo + ((j + 1 - 128) * 2), 189 -
input_wave[j + 1]*pixel_mag_factor, ILI9341_STEELBLUE);
```

```
      }
      refresh_screen = 100;
    }
  }

  if (paint_fft)
  {
    tft.setTextArea(graphArea);
    refresh_screen--;

    if (refresh_screen <= 50)
    {
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      preMax /= 1000;
      tft.clearTextArea(ILI9341_WHITE);
      //We draw until 84kHz because the frequencies of interest are
there
      for (int i = 0; i < 128; i++)
      {
        tft.drawLine(Xo + i * 2, 190 - magFFT[i]*pixel_magfreq_factor,
Xo + (i + 1) * 2, 190 - magFFT[i + 1]*pixel_magfreq_factor,
ILI9341_DARKSLATEBLUE);
      }
      //DC Component
      tft.setTextColor(ILI9341_DARKVIOLET, ILI9341_KHAKI);
      sprintf(dcComponent, "DC component: %.2f V", magFFT[0]);
      tft.printAt(dcComponent, 50, 210);
      sprintf(spreMax, "More Power at %.2f  kHz", preMax);
      tft.printAt(spreMax, 5, 0);

      refresh_screen = 100;
    }
    tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
  }

  if (log_data)
  {
    if (now.second() == after.second())
    {
      now = rtc.now();
      after = now + TimeSpan(5); //5 seconds of span
      sprintf(timestamp, "%02d:%02d:%02d", now.hour(), now.minute(),
now.second());
      Serial.println(timestamp);

      configureADC();
      Sampling(input_wave, re, im);
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      preMax = postProcessing(Xr, Xi, preMax);

      magValues5sec[0][logindex5sec] = magFFT[logFFTindexes[0]];
      magValues5sec[1][logindex5sec] = magFFT[logFFTindexes[1]];
      magValues5sec[2][logindex5sec] = magFFT[logFFTindexes[2]];
      magValues5sec[3][logindex5sec] = magFFT[logFFTindexes[3]];

      logindex5sec++;

      /*if we achieve the minute.... Store in memory the data*/
      if (logindex5sec == (60 / 5))
      {
        logindex5sec = 0;
        for (int i = 0; i < 60 / 5; i++)
        {
          valueSD_minute[0][0] += magValues5sec[0][i];
          valueSD_minute[1][0] += magValues5sec[1][i];
          valueSD_minute[2][0] += magValues5sec[2][i];
          valueSD_minute[3][0] += magValues5sec[3][i];
        }
        //we compute the average value of the minute
        valueSD_minute[0][0] /= (60 / 5);
        valueSD_minute[1][0] /= (60 / 5);
        valueSD_minute[2][0] /= (60 / 5);
        valueSD_minute[3][0] /= (60 / 5);

        freq2ESP = String(freqsFFT[logFFTindexes[0]]);
        volt2ESP = String(valueSD_minute[0][0]);
        data2Esp = true;

        // Check if its 00:00:00-00:00:05
        if (((now.hour() + now.minute()) == 0) && (now.second() < 5))
        {
          //Create a new file
          sprintf(nameLogFile, "%04d-%02d-%02d.csv", now.year(),
now.month(), now.day());
        }

        if (logSIDSWAP.open(nameLogFile, FILE_WRITE))
        {
          //Freq1[kz] ; mag1[v]; Freq2[kz]; mag2[v]; Freq3[kz];
mag3[v]; hh:mm:ss ; yyyy/mm/dd.csv
          logSIDSWAP.print(freqsFFT[logFFTindexes[0]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[0][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(freqsFFT[logFFTindexes[1]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[1][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(freqsFFT[logFFTindexes[2]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[2][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(freqsFFT[logFFTindexes[3]] / 1000);
          logSIDSWAP.print(";");
          logSIDSWAP.print(valueSD_minute[3][0]);
          logSIDSWAP.print(";");
          logSIDSWAP.print(timestamp);
          logSIDSWAP.print(";");
          logSIDSWAP.println(nameLogFile);

          logSIDSWAP.close();
          Serial.println(F("Stored data in SD card OK"));
        }
        else
        {
          Serial.print(F("Error opening "));
          Serial.println(nameLogFile);
        }

        Serial.println(timestamp);
        Serial.println(nameLogFile);
        valueSD_minute[0][0] = 0;
        valueSD_minute[1][0] = 0;
        valueSD_minute[2][0] = 0;
        valueSD_minute[3][0] = 0;

      }

    }

  }

  if (sel_freq)
  {
    tft.setTextArea(graphArea);
    refresh_screen--;

    if (refresh_screen <= 50)
    {
      sidFFT(Xr, Xi, re, im, 0, N, 0, 1);
      // Post-processing
      preMax = postProcessing(Xr, Xi, preMax);
      tft.clearTextArea(ILI9341_WHITE);
      //We draw until 84kHz because the frequencies of interest are
there
      for (int i = 0; i < 128; i++)
      {
        tft.drawLine(Xo + i * 2, 190 - magFFT[i]*pixel_magfreq_factor,
Xo + (i + 1) * 2, 190 - magFFT[i + 1]*pixel_magfreq_factor,
ILI9341_STEELBLUE);
      }

      tft.setTextColor(ILI9341_BLACK, ILI9341_KHAKI);
      tft.drawFastVLine(Xo + indFx * 2, Yo + 20, 160,
ILI9341_DARKGOLDENROD);

      //Frequency %d: %.2f kHz
      tft.printAt(F("Frequency "), Xo - 30, 0);
      sprintf(logFQIndexesStr[logFQIndexes], "%d:", logFQIndexes + 1);
      tft.printAt(logFQIndexesStr[logFQIndexes], Xo + 65, 0);
      sprintf(freqSampledStringTemp[logFQIndexes], "%.2f",
freqsFFT[indFx] / 1000);
      tft.printAt(freqSampledStringTemp[logFQIndexes], Xo + 90, 0);
      tft.printAt(F(" kHz"), Xo + 140, 0);

      refresh_screen = 100;
    }
    tft.setTextColor(ILI9341_BLACK, ILI9341_WHITE);
  }

}

/*FFT*****************************************************************
**********************************************/

void sidFFT(double *X_real, double *X_im, double *xreal, double
*ximag, int freq_bin, int N, int h, int h_interval)
{
  uint32_t k;
  double Xre_temp, Xim_temp, Xre_temp_kplusN2, Xim_temp_kplusN2,
exp_Oddk_re, exp_Oddk_im;
```

```
  if (N == 1)
  {
    X_real[freq_bin] = xreal[h];
    X_im[freq_bin] = ximag[h];
    return;
  }
  else
  {
    //Even part of the DFT subsection
    sidFFT(X_real, X_im, xreal, ximag, freq_bin,     N / 2,
h,          h_interval * 2);
    //Odd part of the DFT subsection
    sidFFT(X_real, X_im, xreal, ximag, freq_bin + N / 2, N / 2, h +
h_interval, h_interval * 2);

    for (k = 0; k < (N / 2); k++)
    {
      // t_temp <- X_k
      Xre_temp = X_real[k + freq_bin];
      Xim_temp = X_im[k + freq_bin];
      // t_temp_kplusN2 <-X[k+N/2]
      Xre_temp_kplusN2 = X_real[k + freq_bin + N / 2];
      Xim_temp_kplusN2 = X_im[k + freq_bin + N / 2];
      // Calculation temp
      exp_Oddk_re = cos(-2 * PI * k / N) * Xre_temp_kplusN2 - sin(-2 *
PI * k / N) * Xim_temp_kplusN2;
      exp_Oddk_im = cos(-2 * PI * k / N) * Xim_temp_kplusN2 + sin(-2 *
PI * k / N) * Xre_temp_kplusN2;

      // X_k <- t + exp(-2*pi*i*k/N) X_(k+N/2)
      X_real[k + freq_bin] = Xre_temp + exp_Oddk_re;
      X_im[k + freq_bin] = Xim_temp + exp_Oddk_im;
      // X_(k+N/2) <- t - exp(-2*pi*i*k/N) X_(k+N/2)
      X_real[k + freq_bin + N / 2] = Xre_temp - exp_Oddk_re;
      X_im[k + freq_bin + N / 2] = Xim_temp - exp_Oddk_im;
    }
  }
}

/*Sampling***********************************************************
*********/

void Sampling(double *sw, double *re, double *im)
{
  adc_start(ADC);
  for (int i = 0; i < N; i++) {
    while ((adc_get_status(ADC) & ADC_ISR_DRDY) != ADC_ISR_DRDY)
    {}; //Wait for end of conversion
    sw[i] = adc_get_latest_value(ADC); // Read ADC
  }

  adc_stop(ADC);

  for (int i = 0; i < N; i++) {
    //To store the voltage value in the array -> value*SPAN_ADC/(2^n-
1)
    sw[i] = sw[i] * 3.3 / 4095;
    re[i] = sw[i];
    im[i] = 0;
  }
```

```
}

/*Post
processing*************************************************************
*********/

double postProcessing(double *Xreal, double *Xim, double preMax)
{
  double maxk = -1.0;
  double maxvalue = -1.0;
  int j = N / 2;
  for (int i = N / 2; i < N; i++) {                     //we check the
half of the FFT result because is symetric
    double nowFre = abs((i - N) * Fs * 1.0 / N);     //actual
frequency.We start with the highest one!!!!!
    double temp = sqrt(Xreal[i] * Xreal[i] + Xim[i] * Xim[i]);
//temporal magnitude to compute the maximum

    if (j > -1)
    {
      freqsFFT[j] = nowFre;        //We take the frequencies in
ascending order
      magFFT[j] = temp * 2 / N;
      j--;
    }

    if (temp > maxvalue) {
      maxk = nowFre;
      maxvalue = temp;
    }
  }

  j = N / 2;

  //DC Component
  for (int k = 0; k < N; k++)
  {
    Xreal[0] = Xreal[0] + input_wave[k];
  }
  //magFFT contains the half of the samples! The average is then 2*N
  magFFT[0] = Xreal[0] / 2 / N;

  return maxk;
}


/*Configure
ADC********************************************************************
**/

void configureADC() {
  // Setup all registers
  pmc_enable_periph_clk(ID_ADC); // To use peripheral, we must enable
clock distributon to it
  adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST); //
initialize, set sampling frequency
  adc_disable_interrupt(ADC, 0xFFFFFFFF); //disable interrupt of theA
ADC
  adc_set_resolution(ADC, ADC_12_BITS); //We use the available
resolution of the ADC
```

```
  adc_configure_power_save(ADC, 0, 0); // Disable sleep, always
powered
  adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1); // Set timings
- standard values
  adc_set_bias_current(ADC, 1); // Bias current - maximum performance
over current consumption
  adc_stop_sequencer(ADC); // not using it
  adc_disable_tag(ADC); // it has to do with sequencer, not using it
  adc_disable_ts(ADC); // disable temperature sensor
  adc_disable_channel_differential_input(ADC, ADC_CHANNEL_7); // A0 is
channel 7 of the ADC
  adc_configure_trigger(ADC, ADC_TRIG_SW, 1); // triggering from
software, freerunning mode
  adc_disable_all_channel(ADC);
  adc_enable_channel(ADC, ADC_CHANNEL_7); // just one channel enabled
}

void sendData2Esp(void) {
    Serial.println(F("Starting communication with ESP:"));
    delay(50);
    Esp8266.print("F");
    while (!Esp8266.available());
    inStr = Esp8266.readString();
    if (inStr == "G") {
      Esp8266.print(freq2ESP);
      Serial.print(F("Sended as freq: "));
      Serial.println(freq2ESP);
      delay(50);
      while (!Esp8266.available());
      inStr = Esp8266.readString();
      if (inStr == "H") {
        Esp8266.print(volt2ESP);
        delay(50);
        Serial.print(F("Sended as voltage: "));
        Serial.println(volt2ESP);
      }
    }
    if(inStr=="H")
    Serial.println("Packet Send");

}
/*Functions to draw the screen******************************/
void drawScreen1(ILI9341_due &d)
{
  d.setTextScale(1);
  d.fillScreen(ILI9341_BLACK);
  myButtons.enableButton(dispButton);
  myButtons.drawButton(dispButton);
  //delay(200);
  myButtons.enableButton(fftButton);
  myButtons.drawButton(fftButton);
  //delay(200);
  myButtons.enableButton(logButton);
  myButtons.drawButton(logButton);
  //delay(200);
  myButtons.enableButton(selectFreqButton);
  myButtons.drawButton(selectFreqButton);
}

void drawGraph(ILI9341_due &d)
{
  /* Function that draw a graph in the TFT screen. To compute the
values, we take
      into account that the axis are Y(10-200, 190 pixels of spam)
X(54-310, 256 of spam)
  */
  tft.fillScreen(ILI9341_WHITE);
  d.drawFastHLine(Xo - 2, 191, 256, ILI9341_DARKRED);
  d.drawFastVLine(Xo - 2, Yo + 1, 180, ILI9341_DARKRED);
}

void timeAxis(ILI9341_due &d)
{
  d.drawFastVLine(Xo + 64, 191, 2, ILI9341_BLACK);
  sprintf(xlabeltime[0], "%.2f", (float)64000 / (2 * Fs));
  d.printAt(xlabeltime[0], Xo + 41, 193);

  d.drawFastVLine(Xo + 128, 191, 2, ILI9341_BLACK);
  sprintf(xlabeltime[1], "%.2f", (float)128000 / (2 * Fs));
  d.printAt(xlabeltime[1], Xo + 105, 193);

  d.drawFastVLine(Xo + 192, 191, 2, ILI9341_BLACK);
  sprintf(xlabeltime[2], "%.2f", (float)192000 / (2 * Fs));
  d.printAt(xlabeltime[2], Xo + 170, 193);

  d.drawFastVLine(Xo + 256, 191, 2, ILI9341_BLACK);
  d.printAt("ms", Xo + 243, 193);

  //Horizontal lines to voltage reference
  d.printAt("V", Xo - 30, 0);

  d.drawFastHLine(Xo - 5, 105, 5, ILI9341_BLACK); //1,65 volts
reference
  d.printAt("1,65", Xo - 46, 97);

  d.drawFastHLine(Xo - 5, 149, 5, ILI9341_BLACK); //0,825 volts
reference
  d.printAt("0.83", Xo - 46, 146);

  d.drawFastHLine(Xo - 5, 61, 5, ILI9341_BLACK); //2,475 volts
reference
  d.printAt("2,48", Xo - 46, 58);

  d.drawFastHLine(Xo - 5, Yo + 7, 5, ILI9341_BLACK); //3.3 volts
reference
  d.printAt("3,3", Xo - 36, Yo + 4);

  d.printAt("0", Xo - 5, 195);
}

void freqAxis(ILI9341_due &d)
{
  //Horizontal lines to voltage reference
  d.printAt("V", Xo - 30, 0);

  d.drawFastHLine(Xo - 5, 104, 5, ILI9341_BLACK); //1,65 volts
reference
  d.printAt("1,65", Xo - 46, 96);

  d.drawFastHLine(Xo - 5, 148, 5, ILI9341_BLACK); //0,825 volts
reference
  d.printAt("0.83", Xo - 46, 145);
  d.drawFastHLine(Xo - 5, 60, 5, ILI9341_BLACK); //2,475 volts
reference
  d.printAt("2,48", Xo - 46, 57);

  d.drawFastHLine(Xo - 5, Yo + 6, 5, ILI9341_BLACK); //3.3 volts
reference
  d.printAt("3,3", Xo - 36, Yo + 3);

  d.printAt("0", Xo - 5, 195);

  //Vertical marks to reference the frequencies
  d.drawFastVLine(Xo + 64, 191, 2, ILI9341_BLACK);
  //As we draw a sample every two pixels Xo+64 -> freqsFFT[64/2]
  sprintf(xlabelfreq[0], "%.2f", freqsFFT[64 / 2] / 1000);
  d.printAt(xlabelfreq[0], Xo + 41, 193);

  d.drawFastVLine(Xo + 128, 191, 2, ILI9341_BLACK);
  sprintf(xlabelfreq[1], "%.2f", freqsFFT[128 / 2] / 1000);
  d.printAt(xlabelfreq[1], Xo + 105, 193);

  d.drawFastVLine(Xo + 192, 191, 2, ILI9341_BLACK);
  sprintf(xlabelfreq[2], "%.2f", freqsFFT[192 / 2] / 1000);
  d.printAt(xlabelfreq[2], Xo + 170, 193);

  d.drawFastVLine(Xo + 254, 191, 2, ILI9341_BLACK);
  d.printAt("kHz", Xo + 235, 193);
}
```