Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Quadrature synchronous sampling for electrical impedance plethysmography implemented on a MSP432 microcontroller

AUTHOR: José Miguel Sánchez Sanabria

DIRECTOR: Ernesto Serrano Finetti

DATE: February, 21st 2016

Overview

This project describes how to obtain the electric impedance plethysmography ranged in low and high frequencies using the skill of quadrature synchronous sampling without the need of having analog demodulation circuitry.

The system architecture includes a microcontroller unit (MCU), an external analog-to-digital converter (ADC) and an analog-front-end (AFE). The MCU controls the ADC acquisition to accomplish the timing requirements of the QSS and also generates the excitation signal ensuring synchronization. The AFE performs the voltage-to-current conversion and differential signal processing of the captured voltage developed in the impedance under test.

The devices used in this project consist of MSP432 which is a low cost and low power profile microcontroller which drives the analog to digital (ADC) successive approximation ratio (SAR) converter AD7766 that offers up to 24 bits of resolution. The result system is able to obtain a plethysmography at multiple frequencies.

ACKNOWLEDGMENTS

I want to thank Ernesto Finetti Serrano to grant me the possibility of realizing this project.

I thank Ramon Pallars Areny for teaching me the principles of instrumentation and sensors world.

I would like to thank Mr. Francis that helped me with the components soldering and creating some helpful auxiliary tools which I needed.

I want to thank my friends despite having our discrepancies on a lot of things we managed to cheer us up in our sad moments.

And of course I want to especially thank to my family, mother, father and sister that carried me through my worst moments and encouraged me not to give up. Without them any of the things I have achieved in life would have ever been possible.

Contents

1. INTRODUCTION ---------- 1

1.1. Background and Motivation ---------- 1

1.2. State of the Art ---------- 2

1.3. Goals of Present Work ---------- 2

2. PRINCIPLES OF THE BIOIMPEDANCE ---------- 3

2.1. The Bioimpedance ---------- 3

2.2. The IPG ---------- 5

3. PRINCIPLES OF QUADRATURE SYNCHRONOUS SAMPLING ---------- 7

3.1. Basis ---------- 7

3.2. Frequency Requirements ---------- 8

3.3. Timing Requirements ---------- 9

4. ANALOG FRONT END ---------- 11

4.1. Generating Circuit ---------- 11

4.1.1. Requirements ---------- 11

4.1.2. Design ---------- 11

4.1.3. Implementation ---------- 12

4.1.3.1. Active Filtering ---------- 12

4.1.3.2. Howland Current Pump ---------- 14

4.2. Acquiring Circuit ---------- 19

4.2.1. Requirements ---------- 19

4.2.2. Design ---------- 19

4.2.3. Implementation ---------- 20

4.2.3.1. Differential Electrode Buffers and High Pass Filter ---------- 20

4.2.3.2. Differential Amplifier and Low Pass Filter ---------- 21

4.3. Power Circuit ---------- 23

4.3.1. Requirements ---------- 23

4.3.2. Design and Implementation ---------- 23

5. DIGITAL BACK END ---------- 24

5.1. Generating ---------- 24

5.1.1. Requirements ---------- 24

5.1.2. Design ---------- 24

5.2. Acquiring ---------- 25

5.2.1. Requirements ---------- 25

5.2.2. Design ---------- 26

5.3. Processing ---------- 28

5.3.1. Requirements ---------- 28

5.3.2. Design ---------- 28

6. SOFTWARE IMPLEMENTATION ---------- 29

6.1. MCU algorithm ---------- 29

6.2. The Configuration ---------- 29

6.2.1. System's Clock ---------- 29

6.2.2. GPIO ---------- 29

6.2.3. NVIC ---------- 30

6.2.4. SPI ---------- 30

6.2.5. DMA ---------- 31

6.2.6. Timer A & Timer 32 ---------- 31

6.3. The Start-Sleep-End ---------- 32

6.4. The Storing ---------- 32

6.5. The Active phase: Sample ---------- 33

6.6. Matlab Algorithm ---------- 38

7. EXPERIMENTAL RESULTS ---------- 39

7.1. Calibration ---------- 39

7.2. IPG ---------- 44

8. CONCLUSIONS AND FUTURE WORK ---------- 47

8.1. Conclusion ---------- 47

8.2. Future Work ---------- 47

9. BIBLIOGRAPHY ---------- 48

ANNEX A: MCU CODE ---------- 50

ANNEX B: MATLAB CODE ---------- 83

List of Figures

2.1: Simplified Cole-Cole equation representation.	3
2.2: Electrical Circuit equivalent of a cell.	4
2.3: Wave travelling in cell medium as function of its frequency.	5
2.4: IPG waveform and its main features like Systolic peak used for synchronism of cardiovascular detection system [13].	6
3.1: Sampling times of in phase and quadrature pairs. [14].	7
3.2: Frequency response of a Sample & Hold operation.	8
4.1: Active low pass filter and improved howland current pump design for generating circuit stage.	11
4.2: Active Low pass Filtering circuit for 10 kHz.	12
4.3: Frequency responses of active low pass filtering circuit.	13
4.4: Left: MSP432 output wave at the input of the filter. Right: Wave at the output of the filter.	13
4.5: Experimental frequency response at the output of the filter.	14
4.6: Non-ideal current pump model.	15
4.7: Dual configuration at negative feedback Howland current pump circuit.	15
4.8: Howland current pump output impedance characterization.	17
4.9: Acquiring circuit design for a differential input / output.	19
4.10: Differential buffer and high pass filter.	20
4.11: ADC differential input voltage range.	21
4.12: Differential signal at the output of differential high pass filter.	21
4.13: Differential Amplifier circuit.	22
4.2: Powering circuit design with voltage regulators.	23
5.1: Generating Wave digital module design.	25
5.2: Acquiring software module design.	27
5.3: Processing software module design.	28
6.1: GPIO Input and Output.	29
6.2: Storing interrupt into the NVIC.	30
6.3: Start Sleep End Phase procedure.	32
6.4: Left: SDHC Card initialization of SPI procedure. Right: SDHC Card data.	33

6.5: Active phase: Sampler program flowchart.	34
6.6: Active Phase, green - Data Ready; Purple - Differential signal; yellow - ADC SAR Sample Clock.	35
6.7: 4 Mini active phases, Green - Data Ready; Purple - Differential signal; Yellow - ADC SAR Sample Clock.	36
6.8: Active Phase, ADC sample process, Green - Data Ready; Purple - Differential signal; Yellow - ADC SAR Sample Clock.	37
6.9: Single sample transmission: Red: Chip Select. Blue: Received 24 bits. Green: Bit Clock. Yellow: Sampler Clock.	38
7.1: Experimental simulation electrode scenario plus load using capacitors and resistors.	39
7.2: Impedance Module of a known load. Up 10: kHz. Down: 1 MHz.	40
7.3: Parallel Resistance of a known load. Up 10: kHz. Down: 1 MHz.	41
7.4: Parallel Capacitance of a known load. Up 10: kHz. Down: 1 MHz.	41
7.5: Noise Impedance. Up 10: kHz. Down: 1 MHz.	42
7.6: FFT of Noise Impedance of known load. Up 10: kHz. Down: 1 MHz.	43
7.7: Captured Voltage Module (for Noise calculation). Up 10: kHz. Down: 1 MHz.	43
7.8: Normalized Voltage module of a modulated 2Hz Sine at 10 kHz using 1 % modulation index.	44
7.9: FFT captured of a modulated 2Hz Sine at 10 kHz using 1 % modulation index.	44
7.10: Normalized Voltage module of a modulated 2Hz Sine at 10 kHz using 0.1 % modulation index.	45
7.11: FFT captured of a modulated 2Hz Sine at 10 kHz using 0.1 % modulation index.	45
7.12 IPG at 10 kHz.	46
7.13: FFT of IPG at 10 kHz.	46
7.14: Normalized and Filtered IPG at 10 kHz.	46

List of Tables

3.1: Required uncertainty times for a given frequency and resolution of both resistive and reactive components. 10

4.1: Requirements of generating circuit. 11

4.2: Output Impedance required for N bits of resolution. 17

4.3: AD8041 Bandwidth and pole parameters. 18

4.4: Requirements of acquiring circuit. 19

4.5: Requirements of power circuit. 23

5.1: Requirements of acquiring circuit. 24

5.2: Requirements of acquiring software. 25

5.3: Requirements of processing software. 28

6.1: SPI Configuration. 30

6.2: DMA Configuration. 31

6.3: Timers A Configuration. 31

6.4: Timers 32 Configuration. 31

7.1: System precision. 44

1. Introduction

1.1. Background and Motivation

Nowadays, the spread use of mobile devices opens up the possibility of implementing cheap, non-invasive measurements techniques aimed at obtaining information about a person's health status. The state of art of these techniques shows a steady evolution towards different bioelectrical signal analysis in general. One such technique is the measurement of the electrical bioimpedance of living tissues (i.e. the human body).

The Bioimpedance measurement is cheap and non-invasive measurement method. This characteristic is the one that makes current state of art constant evolving towards bioelectric analysis in general.

Many electronic weighing scales give information about the body fat, body water or muscle content by measuring the basal electrical bioimpedance of different body segments. However, they also show weak variations due to the physiological activity of the respiratory and the circulatory systems

The study of such variations is known as impedance plethysmography (IPG). Electrical impedance myography (EIM) is another form of recording changes of the body. In this case, the variations in impedance are due to the geometrical changes induced by the muscular activity of a body limb which enables monitoring of rehabilitation therapy in injured limbs.

These variations of complex impedance of living tissue –real and imaginary parts– exhibit resistive and reactive behaviour. Moreover, this behaviour is frequency-dependent, which makes it interesting to perform multiple frequency measurements.

This kind of studies usually requires more complex systems that include coherent analog demodulators both for the in-phase and quadrature components.

However quadrature synchronous sampling (QSS) is an under-sampling acquisition strategy that enables the direct acquisition of the in-phase and the quadrature signals of an AM signal, alleviating the use of complex hardware hence reducing cost, space and power consumption.

It is desirable then to design a system based on low-cost microcontrollers (MCUs) with low power consumption. Because of the small modulation index of

bioimpedance variations, a high resolution system is required of at least 16 bits [3] in the range of hundreds of kilohertz which requires high CPU clocks.

1.2. State of the Art

The current state of the art includes several methods and designs to obtain the bioimpedance measurements such as IPG and ECG using low frequency carrier and analog demodulators based on FPGA [1], others include the use of a well-known MCU (MSP430) while obtaining the in-phase and quadrature components through synchronous sampling at low frequency [2] and others include multichannel acquisition of in-phase and quadrature component both channels characterized also at low frequency range [3].

1.3. Goals of Present Work

The aim of this project is to implement QSS on an MSP432 MCU to measure cardiac and respiratory IPG signals at frequencies from 10 kHz to 100 kHz or more. The MSP432 operates with a 48 MHz clock, enabling the desired upper frequency range of carriers (100 kHz or more).
Specifically we need to develop a system that fulfills the following requirements:

1. Develop a low power consumption system that includes the necessary analog front-end (AFE) to yield reliable signals, to be built on a wearable object so that health centers and civilians can make disposition of it.
2. Use an MSP432 (Texas Instruments, Inc.) as the core of the system. The new 32 bit MCU which exhibits the required low power consumption profile required by the project
3. Store the data on a High Capability SD Card allowing the measurement to be performed without the need of connecting to a PC.

2. Principles of the Bioimpedance

2.1. The Bioimpedance

Bioimpedance refers to the passive electrical properties of any organic tissue. As any electrical impedance, it has both a resistive (R) and a reactive (X_c) component whose values depend on the excitation frequency, intracellular an extracellular water and on the capacitance of the cells' membrane.

The electrical properties of the living tissue can be considered as an ionic conductor it is based on concentration activity charge and mobility of free ions, positive ions (cations) move to the cathode and negative ions (anions) will move to the anode. At electrodes the transformation between electrical current and ionic current takes place.

Cells are the basic structural elements of living tissues hence they have an important role in the determination of electrical impedance. Their membrane has the ability to store capacitive energy (acting as a dielectric insulator) so that living tissue is considered as a dispersive medium. Cole [11] introduced the first mathematical expression able to describe the measurements in living tissues.

It is known as the Cole-Cole equation:

$$Z = R_\infty + \frac{\Delta R}{1 + (j\omega\tau)^\sigma} \quad (2.1)$$

$$\Delta R = R_0 - R_\infty \quad (2.2)$$

Where Z is the impedance value at frequency ω, *j* is the complex number, R_∞ is the impedance at infinite frequency, R_0 is the impedance at zero frequency, τ is the characteristic time constant and σ is a dimensionless parameter with a value between 0 and 1 adjusted empirically to fit the observations.

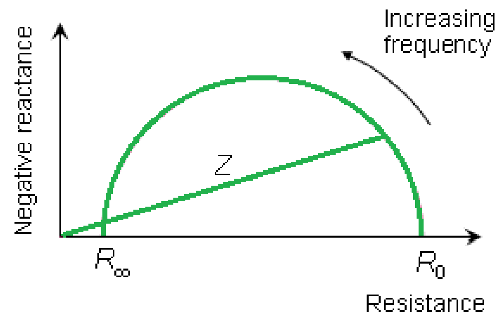


Figure 2.1: Simplified Cole-Cole equation representation.

Note that σ is closely related with the spectral width of the dispersion; the minimum spectral width corresponds to σ = 1 and the dispersion is broadened as σ tends to lower values.

Since we are studying the model with a minimal spectral width dispersion [12] we are evaluating the living tissue containing cell as a system with σ = 1.

The Cole-Cole equation with σ = 1 is equal to the Randles circuit model described as:

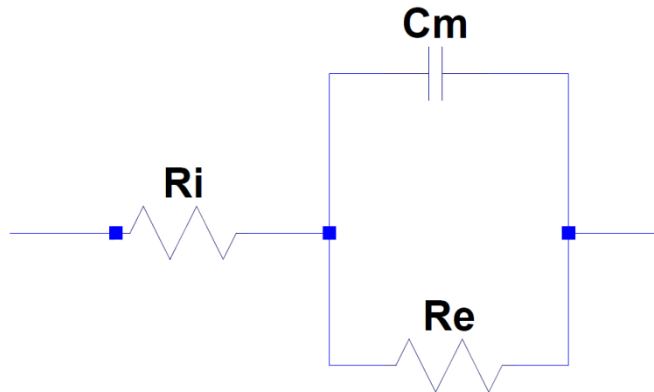


Figure 2.2: Electrical Circuit equivalent of a cell

Being C_m the cell membrane capacity and R_i the intracellular resistance and R_e the extracellular resistance:

- R_i depends on the saline solution: $H_2O - Na$
- R_e depends on the saline solution: $H_2O - Cl$

This means that for low frequencies the cell membrane acts as an open circuit and therefore the current flows the path of R_i+ R_e but for high frequencies the cell membrane will act as an shorted circuit so the impedance will be R_i.

As a consequence of this behaviour measurement at low frequencies yield information about the extracellular fluid while at high frequencies yield information about the intracellular media.

Usually, the range of frequencies of interest is that of the so-called beta dispersion found in a frequency range from few kilohertz to few megahertz. It is a common practice to work at frequencies above 5 – 10 kHz.

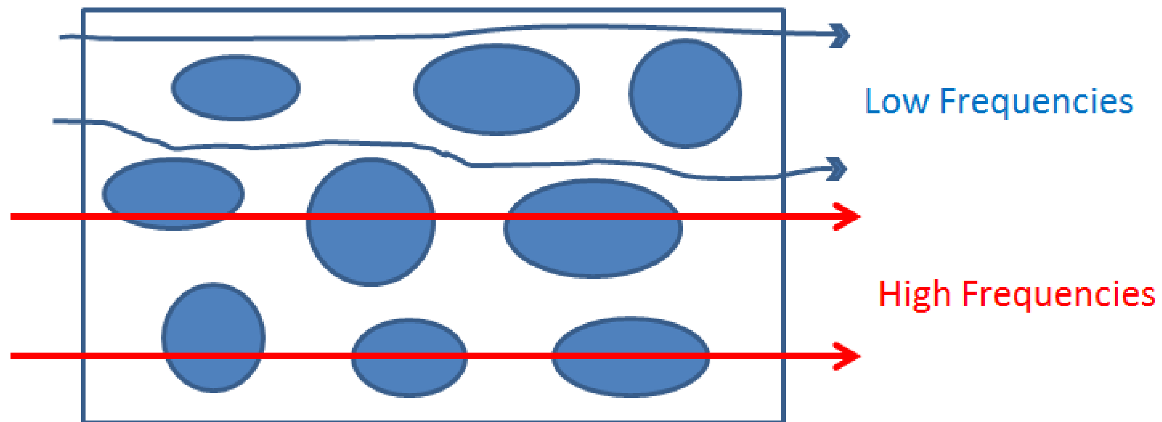


Figure 2.3: Wave travelling in cell medium as function of its frequency

However, the measured impedance is not exclusively dependent of these properties; geometry also plays an important role. Bearing in mind the usual simplified model of a cylindrical conductor, its resistance, depends on the resistivity of the material (an intrinsic property) as well as its shape (section area and length).

Analogously, the electrical bioimpedance will depend on the resistivity of the body segment under study but also on the cell constant, a parameter related to its geometry. This enables recording not only true resistivity chances but also geometrical changes that are linked to physiological activity.

2.2. The IPG

Plethysmography records the variations produced by physiological activity on a given volume conductor like a limb or the human trunk. For example, photo-plethysmography enables to record volume variations at the capillary arteries of fingers yielding information about heart rate.

IPG signals record impedance variations due to physiological activity, for example respiration or heart beats. If we model a body segment as a cylindrical conductor [4] the variations induced by the pumping of arterial blood can be described by:

$$\Delta V = -\frac{\rho L^2}{Z_0^2}\left(\Delta Z_p + \Delta Z_v\right) \quad (2.3)$$

Where:

- ΔV = Arterial Volume change.
- L = Length of the arterial section between voltage electrodes.
- ρ = Blood resistivity.
- Z_o = Basal impedance according to Cole model
- ΔZ_p = Impedance variation due to blood resistivity change
- ΔZ_v = impedance variation due to arterial volume change

Therefore the IPG records the pulsatile impedance changes due to cardiovascular activity caused by pressure pulse-propagations. The typical IPG waveform can be seen in the next figure:

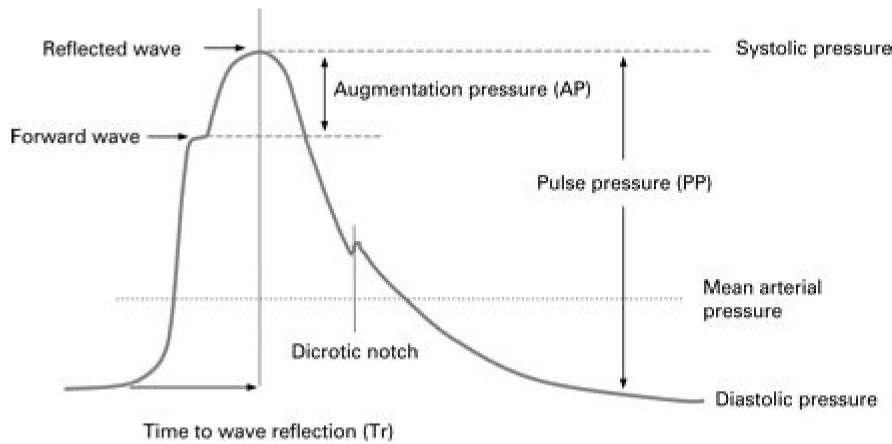


Figure 2.4: IPG waveform and its main features like Systolic peak used for synchronism of cardiovascular detection system [13]

Usually the information of the IPG remains in a bandwidth of 0.5 – 12,5 Hz. It represents the heart cycle as the cardiac ejection occurs a systolic peak is represented followed by the percussion wave where we can identify for instance the dicrotic notch pointed in figure 2.4 which is caused by closure of the aortic valve.

3. Principles of Quadrature Synchronous Sampling

3.1. Basis

It is possible to demodulate a bandpass signal and obtain its phase and quadrature component using the following interpolation equations:

$$p(t) = TB \sum_{n=-\infty}^{\infty} (-1)^{nl} V(nT) sincB(t - nT) \tag{3.1}$$

$$q(t) = TB \sum_{n=-\infty}^{\infty} (-1)^{nl} V\left(nT + \frac{T_c}{4} + mT\right) sincB(t - nT - \frac{T_c}{4} - mT) \tag{3.2}$$

This means that for any sample taken at t = u the quadrature pair will be at t = u+ $\frac{T_C}{4}$ even there is no need to taking both phase and quadrature components in the same period as m ≠ 0 we can obtain the quadrature pair of that phase sample from a different period only if we maintain the sampling interval of u+ $\frac{T_C}{4}$as it can be seen in the Figure 3.1:

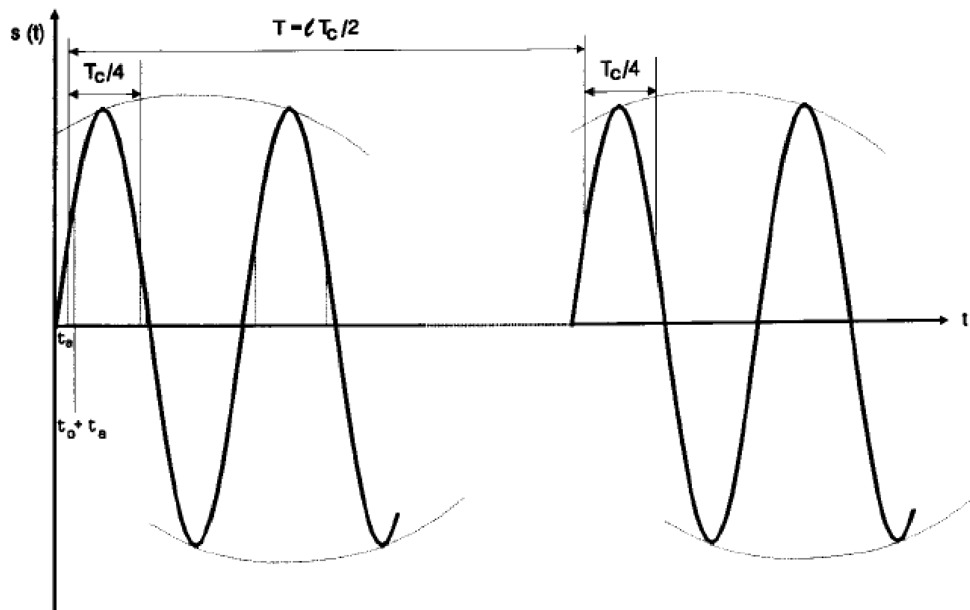


Figure 3.1: Sampling times of in phase and quadrature pairs. [14]

Since the Bioimpedance signal is a bandlimited signal we can obtain its real an imaginary component by sampling at frequency F_C or submultiples of it that fulfil the Nyquist theorem by taking components at t_0 and $t_0 + \frac{T_C}{4}$.

In order to reconstruct de signal we will need a digital low pass filter of Bandwidth B. The SAR analog to digital converter AD7766 does have a digital filter part which in fact acts as a Sync filter plus 2 FIR stages that will help yield a high SNR (109 dB at 128 Ksps. We do not need to take care about the sample/hold circuit nor the digital filter but only for the sampling instants leaded by MSP432.

3.2. Frequency Requirements

The minimal sampling frequency is B and we take 2 pair of samples filling the Nyquist criteria. The available sampling rate at ADC ranges up to 128 Ksps so we can think of sampling the signal at a high frequency and let the zero order sample & hold reconstruct the R (t) and X (t) components.

The frequency response for a zero order hold lasting T_i seconds is:

$$H_0(f) = |T_i sinc\ (fT_i)| \tag{3.3}$$

Therefore $H_0(f)$ should not differ from $H_0(0)$ by more than the resolution 2^{-N} as explained in the Figure 3.2 and Equation 3.4 and 3.5.

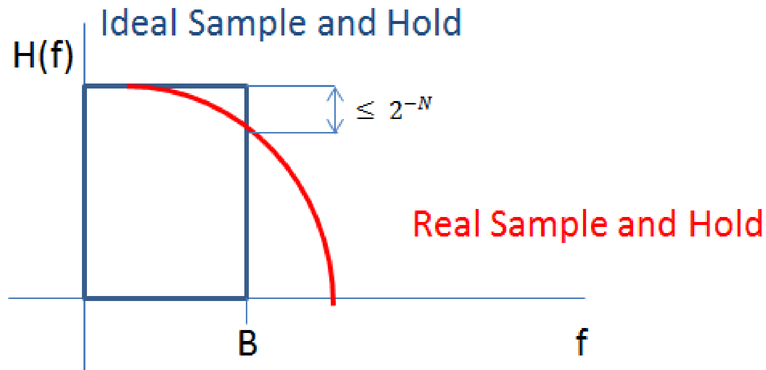


Figure 3.2: Frequency response of a Sample & Hold operation

$$\frac{|H_0(0)| - |H_0(f)|}{|H_0(0)|} \leq 2^{-N} \tag{3.4}$$

Finally given that the frequency responses of the zero order hold is a sinc(x); we obtain that:

$$\frac{\sin(\pi f B_i)}{(\pi f B_i)} \geq 1 - 2^{-N} \tag{3.5}$$

We require fB_i be at least 0.0001903556781 for a 24 bits resolution system. We are interested in the 40 Hz frequency range (BW) so:

$$fB_i = 0.0001903556781 \tag{3.6}$$

$$\boldsymbol{f_i = 210.133\ KHz} \tag{3.7}$$

Since the ADC oversample the signal at 1 MHz rate the criterion is accomplished.

3.3. Timing Requirements

The time domain error is a critical requirement when it comes to demodulate using QSS method.

The uncertainty in the aperture time t_a created by sampling at an instant is a major concern since it propagates as a voltage error hence lowering the SNR. This error depends on the slope of the signal at the sampling point. It will be maximum when the slope of the signal is maximal; therefore the error *E* can be expressed as:

$$E = 2\pi f_p t_a \tag{3.8}$$

Where f_p Is the frequency of the sinusoidal signal (the carrier). The rule of thumb now is to make the *E* less than the quantitation interval so it does not affect the ADC resolution:

$$2^{-(N)} < 2\pi f_p t_a \tag{3.9}$$

$$t_a < \frac{2^{-(N-1)}}{\pi f_p} \tag{3.10}$$

In QSS we take samples always at the same point of the carrier so the major concern is not the maximal slope of the signal but the actual slope where we are taking a sample.

In Bioimpedance measurements the typical reactive component is obtained with θ of 10º (tan θ= 0.176) the formula that describes the impedance is [14]:

$$Z(t) = \cos(2\pi f_p t) - 0.176 \sin(2\pi f_p t) \tag{3.11}$$

However what really matter is to compute the maximal t_a so that the error is less than the quantitation step:

$$|Z(t_0) - Z(t_0 + t_a)| < 2^{-N} \tag{3.12}$$

$$\left|Z\left(t_0+\frac{T_c}{4}\right)-Z(t_0+\frac{T_c}{4}+t_a)\right|<2^{-N} \tag{3.13}$$

Computing the uncertainty times for different resolutions we can build the following table:

Table 3.1: Required uncertainty times for a given frequency and resolution of both resistive and reactive components.

Frequency	Resistive Component			Reactive Component		
-	**16 bits**	**20 bits**	**24 bits**	**16 bits**	**20 bits**	**24 bits**
10 KHz	20.73 ns	90 ps	5.2 ps	3.70 ns	15 ps	900 fs
50 KHz	4.15 ns	18 ps	1.04 ps	740 ps	3 ps	180 fs
100 KHz	2.07 ns	8 ps	520 fs	370 ps	1.5 ps	90 fs
500 KHz	415 ps	1.6 ps	104 fs	74 ps	300 fs	18 fs
1 MHz	207 ps	800 fs	52 fs	37 ps	150 fs	9 fs
2 MHz	104 ps	400 fs	26 fs	18.6 ps	75 fs	4.5 fs
6 MHz	34.6 ps	130 fs	8.67 fs	6.17 ps	25 fs	1.5 fs
12 MHz	17.3 ps	67 fs	4.34 fs	3.09 ps	12.5 fs	750 as

As we can see the Reactive component is the most restrictive one. The ADC sample and hold has an uncertainty jitter in the order of few ps so this accomplishes the time domain requirements for 16 bits of resolution.

4. Analog Front End

Following, we describe the main analog processing blocks designed in this project. A brief circuit description and analysis is provided together with the lab verification results.

4.1. Generating Circuit

4.1.1. Requirements

This circuit should be able to generate a sinusoidal signal of current ideally maintaining the amplitude steady in front of different loads at different frequencies.

The idea is to filter a square wave generated by the MSP432 and convert the controlled voltage signal to a controlled current signal. The criterion of resolution should be maintained at 12 bits (16 bits ideally) with a full scale of ±2.5 Volts. The power consumption should be maintained as low as possible.

Table 4.1: Requirements of generating circuit

Parameter	From	To
Signal wave	Square	Sinusoidal
Controlled Signal Type	Voltage	Current
Total Harmonic Distortion (THD)	-54 dB	
Current peak to peak	200 µA	
Frequency Range	10 kHz - 1 MHz	
Resolution	12 bits (16 ideally)	
Voltage Rails	+5V – 0 V	

4.1.2. Design

The design is composed of an analog active low pass filter and improved howland current pump.

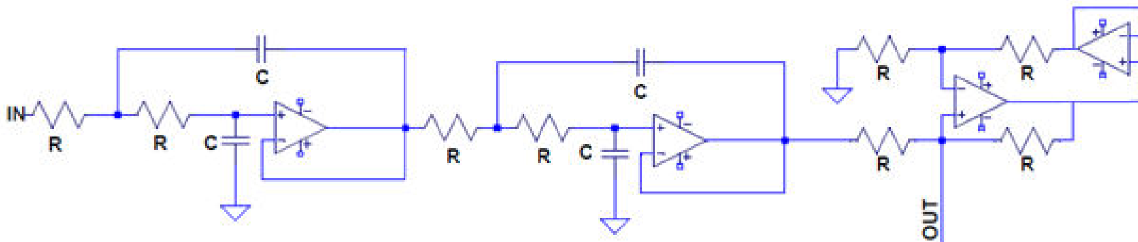


Figure 4.1: Active low pass filter and improved howland current pump design for generating circuit stage

The idea is to get the square voltage signal provided by the MSP432 to be filtered around the desired frequency so that we eliminate the harmonics which compose the square signal leaving a sinusoidal signal with expected total harmonic distortion to be small enough not to interfere in measurements. Finally the improved howland current pump will transform the voltage signal into a controlled current signal.

4.1.3. Implementation

4.1.3.1. Active Filtering

The MSP432 is limited in current output. If we use a passive low pass filter there will be a range of working load impedance. At high frequencies this range is also limited due to the rule of combinations of RC at every stage of the filter in order to avoid loading effect. This causes an output power problem.

Therefore Active filtering will solve that limitation as it will be the operational amplifier which will provide the current needed and also we can improve the THD from 30 dB up to 60 dB (compared with passive filtering) which means we have a natural 10 bits resolution system. Using a little calibration we can aim up to 12 bits of resolution.

The Active low pass filter circuit design is composed by:

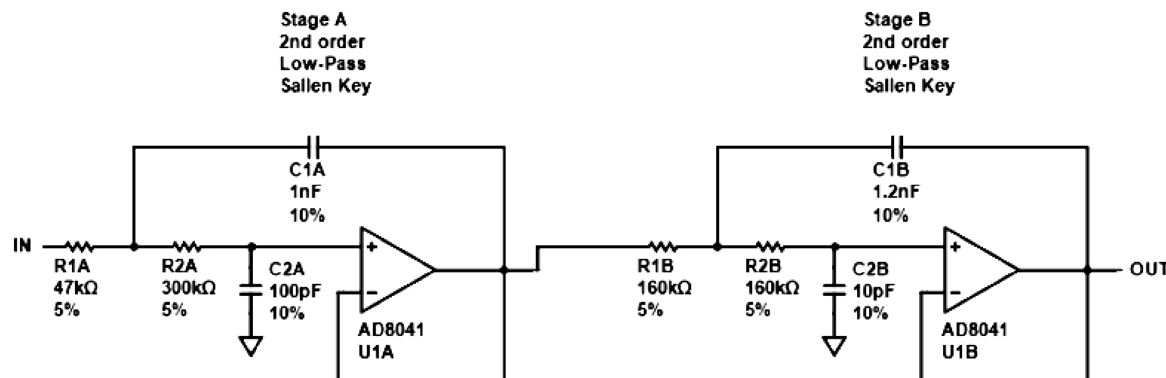


Figure 4.2: Active Low pass Filtering circuit for 10 kHz

It is configured as an active 4th order low pass filter working for a corner frequency of 10 kHz in order to perform the first test. The expected frequency response of the filter should be:

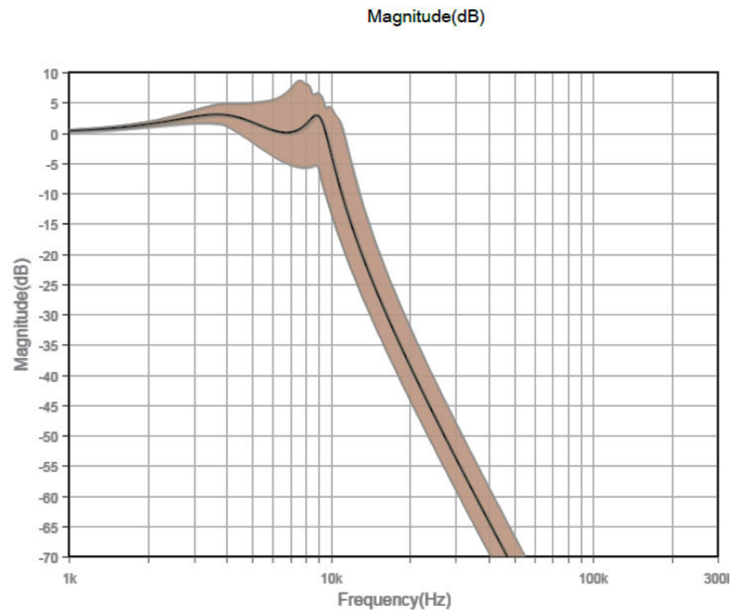


Figure 4.3: Frequency response of active low pass filtering circuit

Yielding -50 dB of attenuation at 30 kHz where it is placed the first harmonic, also the tolerance of the components can make the frequency response to change corresponding to the brown range of the frequency plot.

The measured input wave shows almost no loading effect:

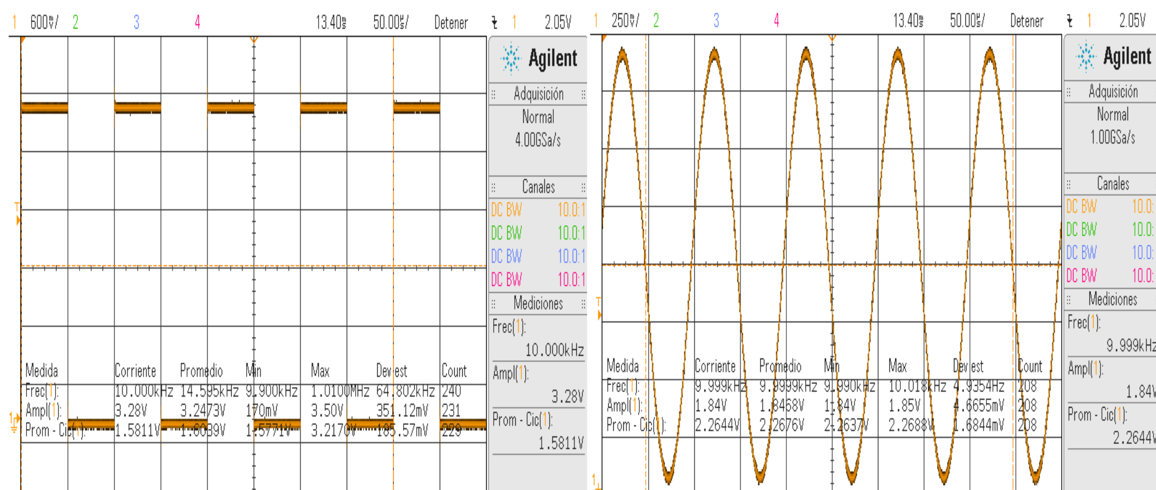


Figure 4.4: Left: MSP432 output wave at the input of the filter. Right: Wave at the output of the filter

The measured output waveform shows amplitude of 1.84 V peak to peak meaning that the attenuation at the fundamental frequency is:

$$Att_{dB} = 20 * Log\left(\frac{V_{OUT}}{V_{IN}}\right) \tag{4.1}$$

$$Att_{dB} \approx -5\ dB \tag{4.2}$$

And the measured THD is as expected -51 dB:

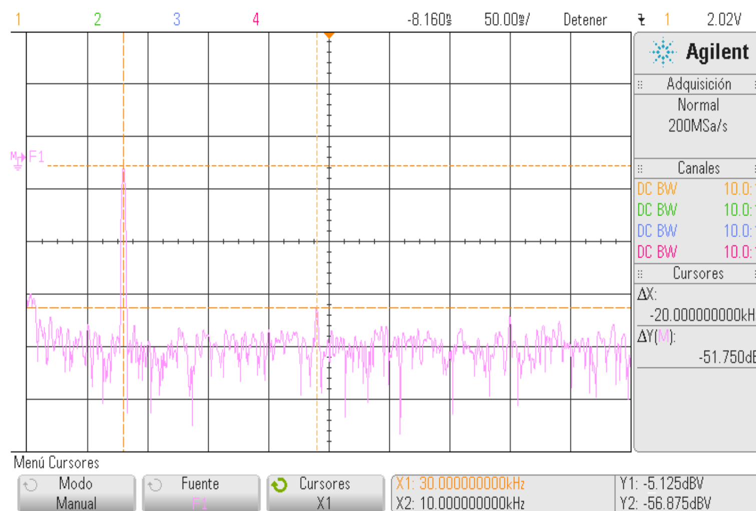


Figure 4.5: Experimental frequency response at the output of the filter

4.1.3.2. Howland Current Pump

The Filtered wave of the previous stage will be converted to a current sine wave. We intend to inject a current through the body with two electrodes and read the voltage developed on a separate pair of electrodes (4-wire measurement scheme).

We will use a Non Inverting Negative Feedback improved Dual Howland current source for this purpose which is a better version of the commonly used howland current source.

The ideal model of current pump circuits is far from the practical model there are many drawbacks.

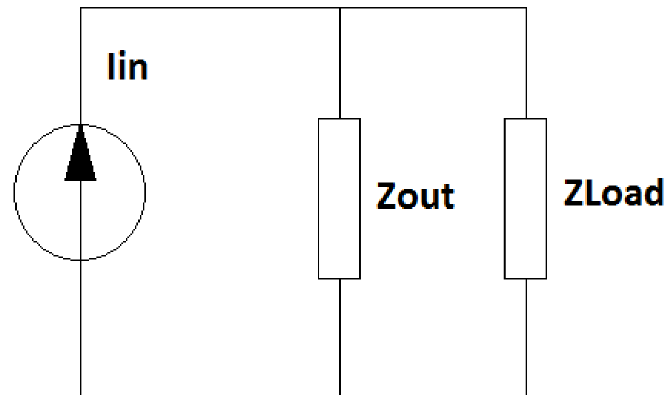


Figure 4.6: Non-ideal current pump model

The main characteristics of a Current pump circuit are:

1- Output Impedance should be as higher as possible (ideally infinite) to avoid loading effects.
2- Bipolar Current Output.
3- The Gain should remain the same at the range of working frequencies.

As mentioned in [15] the best configuration that fulfills the mentioned requirements with special emphasis in output impedance and gain flatness is the howland current pump in dual configuration at negative feedback.

The main circuit of this type of Howland current pump is composed by an operational amplifier in closed loop configuration in both branches and a second operational amplifier configured as a voltage buffer in the negative branch. (Fig 4.7)

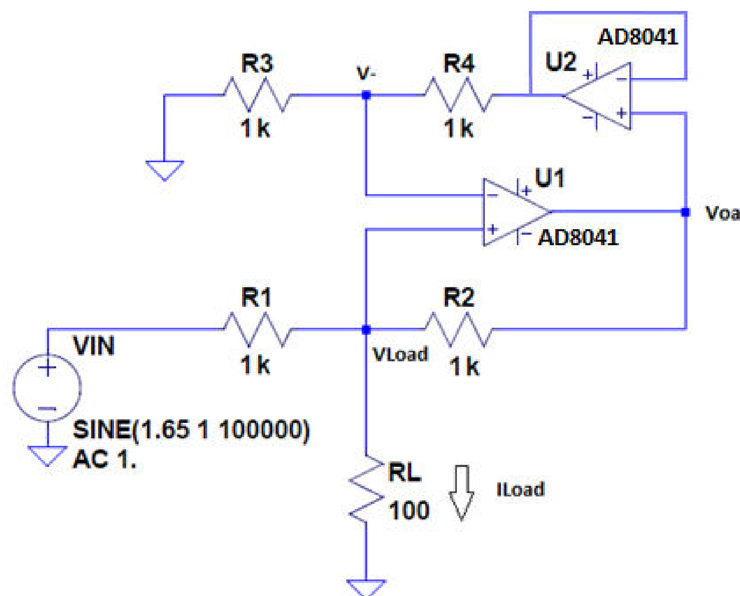


Figure 4.7: Dual configuration at negative feedback Howland current pump circuit.

Analysing the circuit in order to obtain its transconductance g_m lead us to:

$$\mathrm{ILoad} = \frac{\mathrm{Vin}}{\mathrm{R}} \tag{4.3}$$

$$\mathrm{gm} = \frac{1}{\mathrm{R}} \tag{4.4}$$

The current going through the load only depends in the value of V_{in} and R meaning there is no loading effect but this is the case only if all resistors are the same value which is difficult to accomplish due to tolerance in resistors. In Practice the Howland current pump must be trimmed to balance the values of all resistors.

The expression of the output impedance, Z_{out}, is:

$$\mathrm{Zout} = \frac{\mathrm{R1R2R3}}{\mathrm{R2R3} - \mathrm{R4R1}} \tag{4.5}$$

$$\text{if } \mathrm{R1} = \mathrm{R2} = \mathrm{R3} = \mathrm{R4} = R \tag{4.6}$$

$$Zout = \frac{R^3}{0} = \infty \tag{4.7}$$

But this is only the ideal case, in our case the tolerance unbalance the branches meaning that in the worst case is when the positive branch ratio goes minimum while the negative branch ratio goes maximum:

$$\frac{R1 \ \downarrow\downarrow}{R2 \uparrow\uparrow} = \frac{R3 \ \uparrow\uparrow}{R4 \ \downarrow\downarrow} \tag{4.8}$$

$$Zout = \frac{R(1-t)R(1+t)R(1+t)}{[R(1+t)R(1-t)] - [R(1-t)R(1-t)]} \tag{4.9}$$

$$\mathrm{Zout} \approx \frac{\mathrm{R}}{4\mathrm{t}} \tag{4.10}$$

From [6] says that the needed output impedance in order to not affect the resolution for a Bioimpedance measurement is:

$$Z_{Howland} = 2^b(\,ZL_{maz} - ZL_{min}) \tag{4.11}$$

The change in load impedance depending on the position of the electrodes can go from 1 Ω to 200 Ω and we use a system of 24 bits but only 16 effective bits so:

$$Z_{Howland} = 2^{16}(200\ \Omega) \tag{4.12}$$

$$Z_{Howland} = 13\ M\Omega \tag{4.13}$$

A table with the output impedance requirements for a precision of N bits is made:

Table 4.2: Output Impedance required for N bits of resolution

Number of Bits	Output Impedance
6	12.8 KΩ
8	51.2 KΩ
10	204.8 KΩ
12	819.2 KΩ
14	3.3 MΩ
16	13.2 MΩ
18	52.5 MΩ
20	210 MΩ
22	839 MΩ
24	3.36 GΩ

Z_{out}, however, is not constant and will decay with frequency. It has a resistive part (R_{OT}) and a reactive part (X_{OT}) the later one becomes predominant when the frequency increases as it mainly Capacitive. Therefore Z_{out} will decrease by a factor of $\frac{1}{jwCo}$.

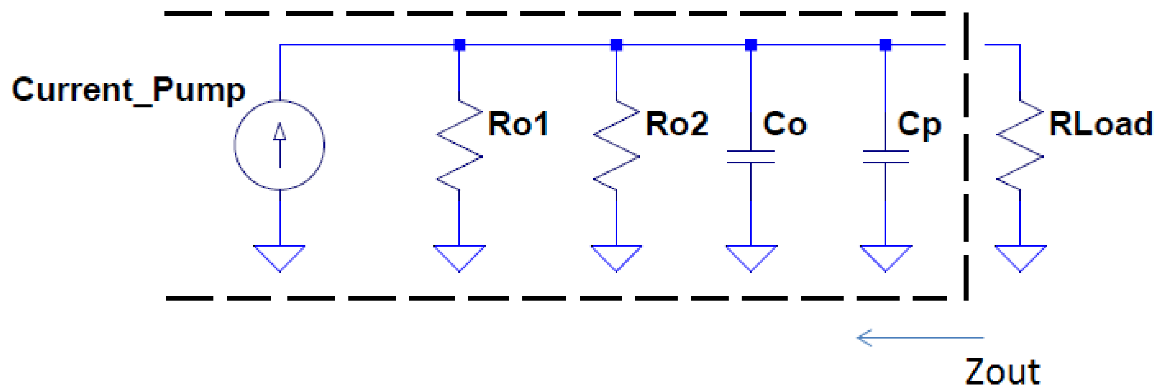


Figure 4.8: Howland current pump output impedance characterization

R_{OT} Will depend on R_{O1} and R_{O2} which will depend on how accurate is the matching of the resistors and the amplifier open loop gain:

$$R_{O1} = \frac{R}{4t} \tag{4.14}$$

$$R_{O2} = \left(1 + \frac{A_{OL(lineal)}}{2}\right) * \left(\frac{R1 * R3}{R1 + R3}\right) \tag{4.15}$$

We will use then a 10 kΩ resistors with 0.01 % tolerance so that Z_{out} ranges to R_{O1} = 25 MΩ at DC. However at higher frequencies Z_{out} will decay because R_{O2} decreases following the open loop gain.

Using AD8041 the Open Loop Gain at DC is 99 dB and at 1 MHz is 45 dB so the R_{O2} will decay from 222 MΩ to 450 kΩ. This means that at 1 MHz the resistive part of Z_{out} of the Howland current pump will be affected as R_{O2} decreased and now predominates over R_{O1} making the parallel R_{OT} be in the order of 440 kΩ.

On the other hand, the equivalent capacitance C_o is defined by:

There is another component which affects the overall Z_{out} which is the Reactive component that will depend on (X_{OT}) depending decaying at $\frac{1}{jwCo}$

$$C_o = \frac{R3 + R4}{2\pi f_H R3\ (R1 \mathbin{/\!/} R2)} \tag{4.16}$$

Being f_H the Op Amp Bandwidth and $\omega_0$the pole at which Open Loop Gain decays:

$$A_{OL}(s) = \frac{A_{OL\ (lineal)}}{1 + \frac{s}{\omega_0}} \tag{4.17}$$

$$\omega_0 = \frac{f_H\ (unity\ Gain\ Bandwith)}{A_{OL\ (Lineal)}} \tag{4.18}$$

From AD8041 Datasheet [16] we obtain:

Table 4.3: AD8041 Bandwidth and pole parameters

Parameter	Value
$\boldsymbol{f_H}$	160 MHz
$\boldsymbol{\omega_0}$	2.846 KHz
$\boldsymbol{C_o}$	0.4 pF
$\boldsymbol{A_{OL_DC(dB)}}$	95 dB
$\boldsymbol{A_{OL_AC(1\ MHz_dB)}}$	45 dB

Z_{out} will range from 25 MΩ at DC to 300 kΩ at 1 MHz so we can achieve a natural 10 bits resolution at 1 MHz in the best case.

Moreover according to [15] the enhanced howland current pump using dual configuration with a feedback at negative branch will improve Z_{out} as now the resistor network is more stable than without it thus increasing base R_{O1} and R_{O2}.

4.2. Acquiring Circuit

4.2.1. Requirements

The Acquiring circuit should be able to obtain the differential signal from electrodes and suit it up for the Analog to digital converter (ADC). This process means that the differential signal should be high-pass filtered and amplified while inserting the common mode voltage of the differential signal match with half the voltage range of analog to digital converter. As the input signal is differential the acquiring circuit must ensure a high CMRR.

Table 4.4: Requirements of acquiring circuit

Parameter	From	To
CMRR	60 dB	80 dB
Voltage Range peak to peak	~200 mV	~1 V
DC Voltage (ADC)	1.25 V	
Frequency Range	10 kHz-1 MHz	
Resolution	12 bits (16 ideally)	
Voltage Rails	+5 V – 0 V	

4.2.2. Design

Acquiring circuit overview is shown in Fig. 4.9. The voltage detection electrodes are buffered to avoid loading effects and the differential signal is high-pass filtered to eliminate electrode offset. Because the ADC needs an offset of 1,25 V, it is provided by the biasing resistor connected to a suitable voltage. Following, a differential amplifier –The gain will need to be adapted at max but ensuring the amplifier outputs are not saturated and ADC range is matched–. It will be added a low pass filter around our carrier frequency in order to reduce the noise at the ADC input.

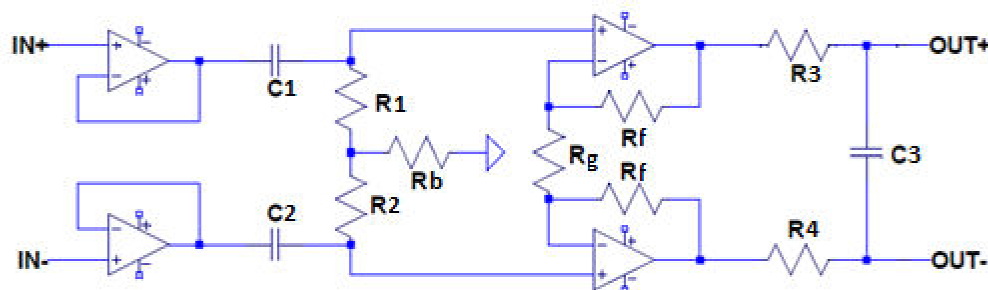


Figure 4.9: Acquiring circuit design for a differential input / output

4.2.3. Implementation

4.2.3.1. Differential Electrode Buffers and High Pass Filter

Since the electrodes exhibit high contact impedance, we will need to buffer their signals in order to increase the ratio of input impedance vs electrode impedance before entering the high-pass filter. A fully differential topology aids in preserving a high CMRR in the signal chain. However, one requirement of the ADC is to have both inputs biased at 1,25 V. The circuit shown in Fig. 4.10 has a cut-off frequency of 88,4 Hz and allows the introduction of an offset equal to $2*I_{bias}*R_b$.

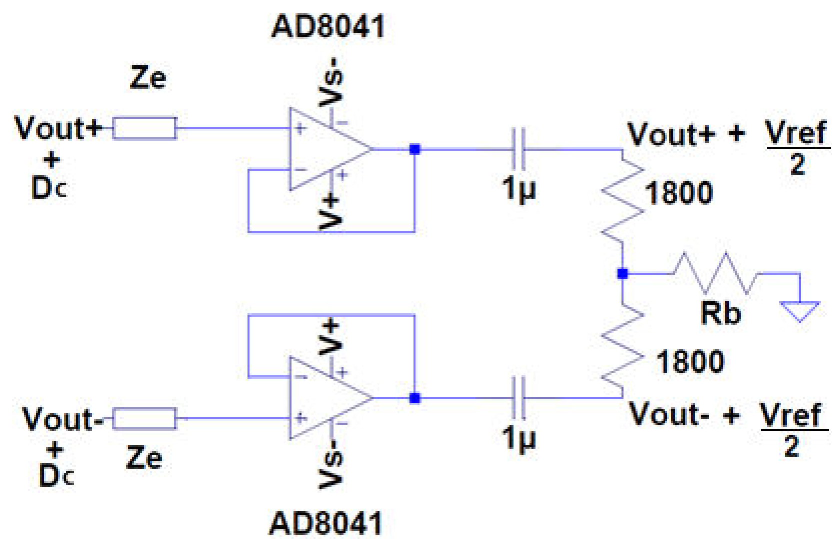


Figure 4.10: Differential buffer and high pass filter

The size of R_b needs to be high enough to maintain the CMRR but low enough to avoid creating undesired added potential due to bias current generated by the two AD8041s. With a 1µA bias current, placing a 1 MΩ resistance yields an added potential of 2 V. In order to suit the ADC input, our differential signals needs to be placed at $\frac{V_{REF}}{2}$ as V_{REF}= 2.5 V the common mode voltage needs to be placed at 1.25 V.

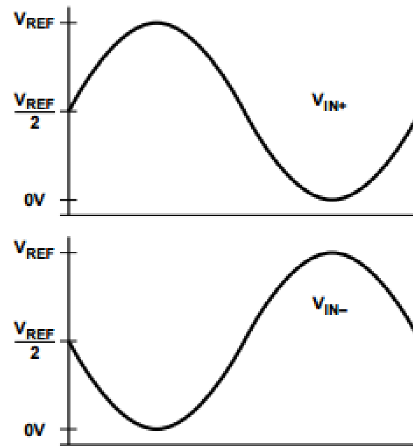


Figure 4.11: ADC differential input voltage range

Knowing that the bias currents are of 1.14 µA (measured) it was selected R_b = 507 kΩ and the expected common mode voltage was 1.25 V.

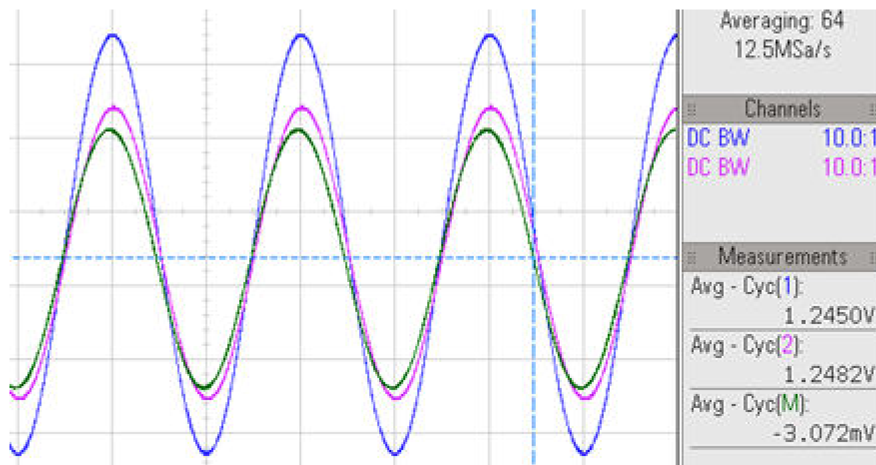


Figure 4.12: Differential signal at the output of differential high pass filter

4.2.3.2. Differential Amplifier and Low Pass Filter

A fully-differential non-inverting amplifier topology was used for this stage (Fig 4.13), adapting the voltage ranges to the ADC's full-scale.

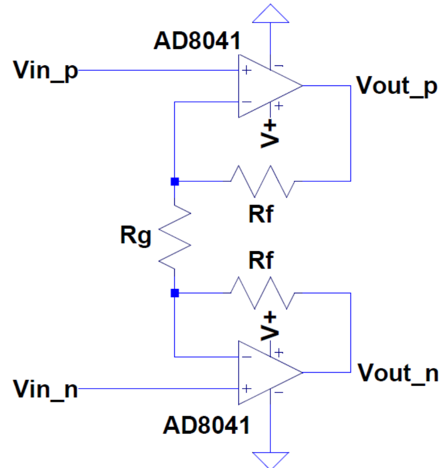


Figure 4.13: Differential Amplifier circuit

The gain design equation is:

$$V_{OUT} = \left(1 + \frac{2R_f}{R_g}\right) * (V_+ - V_-). \quad (4.19)$$

$$G = 37 \; R_{f1} = 17.70 \; K\Omega \;\; R_{f2} = 17.70 \; K\Omega \;\; R_g = 1.03 \; K\Omega \quad (4.20)$$

When using a 10kΩ resistor network in the current source the expected current peak amplitude will be of 100 µA. If it is considered a Basal Load impedance of 200 Ω the voltage read will be around 20 mV. Considering a DC offset of 1.25 V we can amplify the differential signal to at least 1.25 V peak (G = 62.5). Using more amplification will lead to saturation in the ‘negative’ voltage rail.

It is selected an initial G = 37 in order to performs the first experimental test. If it is needed the gain will be maximized always considering the previous restriction.

The op amp used in this stage is also AD8041. It used in early stages because of his CMRR up to 80 dB for a range from dc to 100 kHz enough for 14 bit of resolution. At 1 MHz the CMRR is 68 dB, enough for a 10 bit of resolution.

AD8041 has a high input bias current but it is used as an advantage knowing that the differential voltages need to be placed at 1.25 V (ADC reference).

4.3. Power Circuit

4.3.1. Requirements

The powering circuit should be able to provide the amount of current needed for the entire device and generate 3 different voltage rails at 5 V 3.3V 2.5 V from 9.6 V battery.

Table 4.5: Requirements of power circuit

Power Circuit Requirements	
Parameter	**Value**
Input Voltage	9 V
Output Voltage 1	5 V
Output Voltage 2	3.3 V
Output Voltage 3	2.5 V
PSRR	80 dB

4.3.2. Design and Implementation

The powering circuit will be composed of 3 parallel voltage regulators configured according to datasheets to get the desired voltage rails of 3.3 and 2.5 V (note that the 5 V voltage rail will not need any configuration resistors). Decoupling capacitors of 100 nF and 10 μF will be placed at every input / output IC Pin, close to it. This can be applied for all the IC of the circuit.

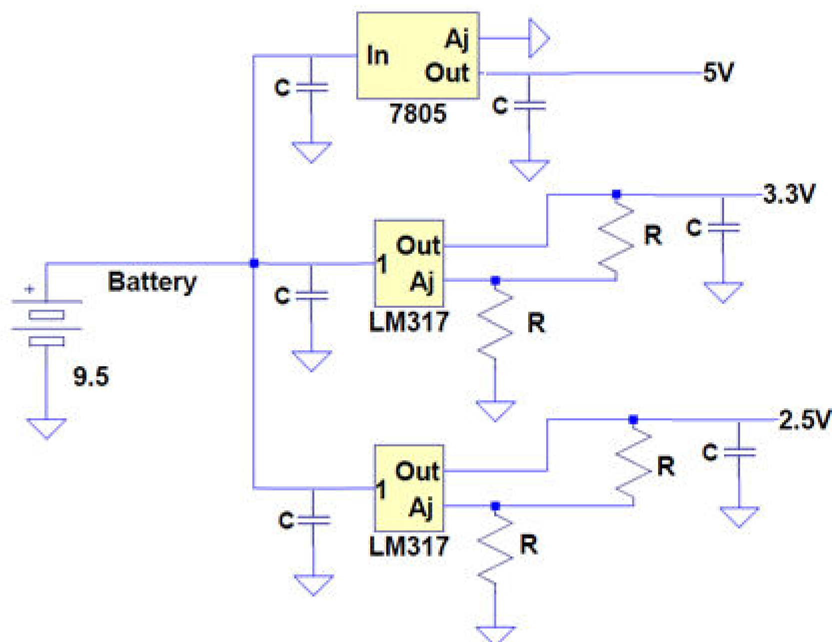


Figure 4.2: Powering circuit design with voltage regulators.

5. Digital Back End

5.1. Generating

5.1.1. Requirements

The digital generating part will be managed by the MCU MSP432 from Texas Instruments and the requirements are quite simple. It should be able to generate a square signal at a frequency selected by the user, therefore the signal should be at the range of the voltage rails of application and it should keep generating the signal until the sampling process is over.

It is decided that in order to be able to witness some cardiac cycles the duration of the generating part will be 10 seconds.

The MCU should also get into the state of sleep during the time there is no action required, as the generation will be independent from MCU processing thanks to the peripherals of the MSP432.

Table 5.1: Requirements of acquiring circuit

Parameter	Value
Signal Wave	Square
Voltage Rails	+5 V – 0 V
Frequency Range	10 KHz – 1 MHz
Duration	10 seconds

5.1.2. Design

The design of the software module that will handle the generation of the input signal wave will be managed by the Timer A module of the MSP432 which relies on a peripheral counter and the crystal oscillator of 48 MHz

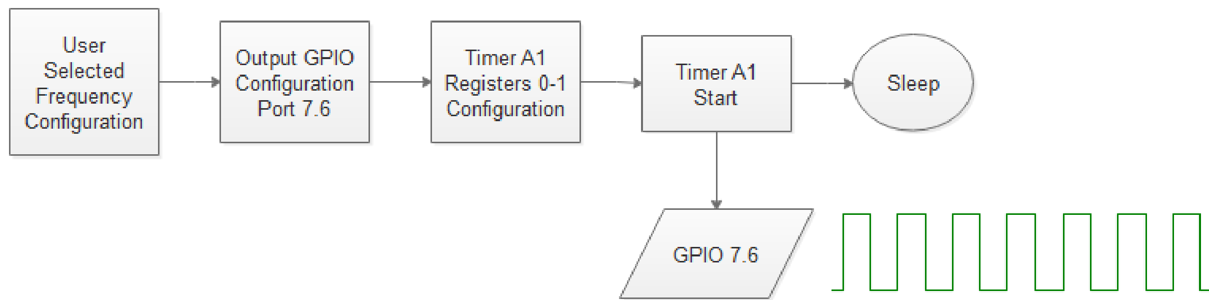


Figure 5.1: Generating Wave digital module design

5.2. Acquiring

5.2.1. Requirements

The digital acquiring part end will be managed by MSP432 ADC7766 and 8 GB micro-SDHC card and the requirements should be generating the sampling instants for the in-phase and the quadrature components and capture the digital information provided by the ADC for a duration of 10 seconds (the time the system is generating the input wave).

After obtaining all the information it should be able to deploy the data on a micro SDHC card.

The critical part of the acquiring software is the error in the sampling instants of the in-phase and quadrature components so it will be used high resolution timers plus a high speed master clock.

Table 5.2: Requirements of acquiring software

Parameter	Value
SDHC communication	SPI-SD 1.1
ADC communication	SPI
Frequency Range	10 KHz – 1 MHz
Duration	10 seconds
N Total Samples	1600

5.2.2. Design

The design of the software module that will manage the acquiring of the digital data will be comprised by 3 peripherals the DMA the SPI and the NVIC. It will be needed 2 DMA and 2 SPI modules which a pair will manage the communication of ADC and the other pair will manage de deploy of the data into the SDHC. The
NVIC module will serve as interrupt handler for waking up the device at sampling instant.

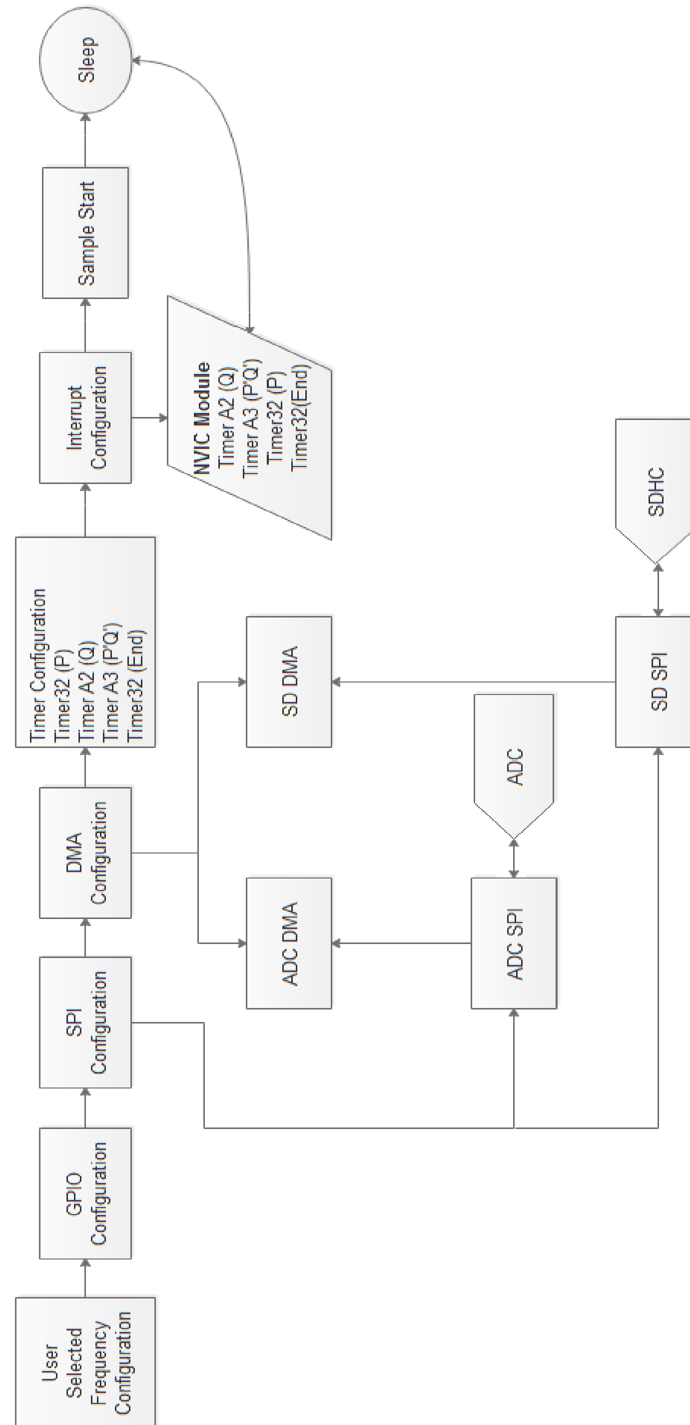


Figure 5.2: Acquiring software module design

5.3. Processing

5.3.1. Requirements

The Processing software should convert the binary data into analog voltage so that it can be displayed the evolution of the voltages of the in-phase and quadrature components over time. It should be able to calculate the frequency spectrum and apply characterizing digital filters (most common case low pass filtering). The software Matlab will handle all this process.

Table 5.3: Requirements of processing software

Parameter	Value
Conversion	Digital to Analog
Displaying	Time, Frequency
Digital Filter	Low Pass, High Pass, Noise Reduction

5.3.2. Design

The design of the processing software consists of reading the binary data of each samples and calculating the voltage vectors. Then it will be corrected with the calibration parameters and it will be calculated the impedance evolution over time.

At the end the processing software should show graph of the impedance over time. Optionally it can also be done frequency analysis of the impedance.

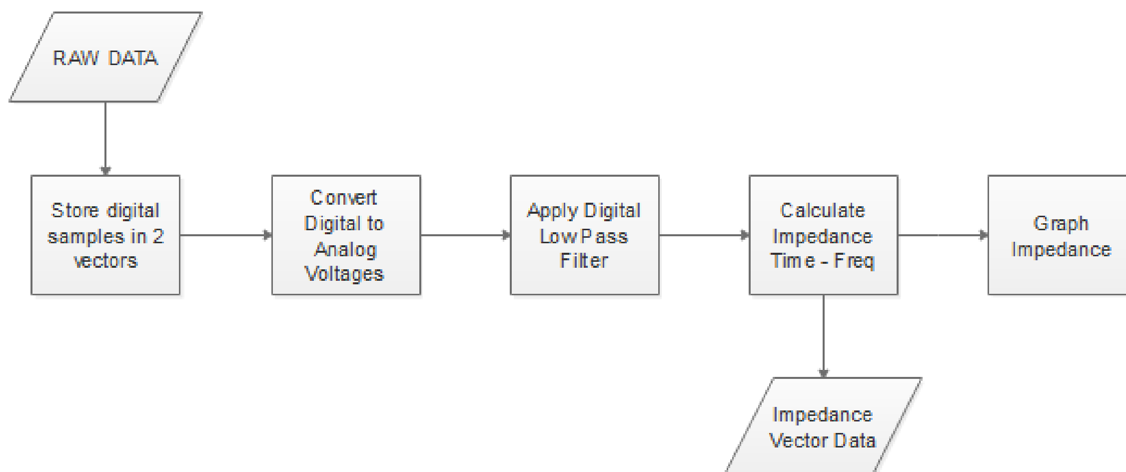


Figure 5.3: Processing software module design

6. Software Implementation

According to the design of the Digital Back End the software implementation of every need is implemented in the MCU MSP432. Below it is declared all the configuration and algorithm developed in order to perform such tasks.

6.1. MCU algorithm

The algorithm of the MCU will be composed of 3 main phases; Configuration, Start-Sleep-End and Storing plus an Active phase; the Sample Process.

6.2. The Configuration

The Configuration phase will set different modules: System's Clock, GPIO, NVIC, SPI, DMA, Timer A and Timer 32.

6.2.1. System's Clock

At the start, the system clock will run at a rate of 3 MHz by default but we will use the high frequency crystal oscillator to set up a new rate of 48 MHz. We will assign the Master System Clock and the Sub System Master Clock to this crystal. [7]

6.2.2. GPIO

The GPIO will be configured as need searching into register tables the secondary functions of the PIN involved into SPI, and Timer operations. [8]

```
  MSP432P401
 ------------------
|                  |
|                  |
|P7.6 (TA1.1)      |--> (O) Square Input Wave Generator
|P8.4              |--> (O) OA Shutdown Pin
|P8.6              |--> (O) ADC7766 Power Up / Down Pin
|P8.7 (DRDY)       |--> (I) DRDY detect
|-------ADC----------------------------------------------
|P1.4 (CS)         |--> (O) Chip Select (Active Low) (ADC CS)
|P2.4 (MCLK)       |--> (O) MCLK (ADC)
|P1.5 (SCLK)       |--> (O) SCLK (ADC)
|P1.7 (S0MI)       |--> (I) SOMI (RX Digital Out ADC)
|P1.6 (SIMO)       |--> (O) SIMO - (TX)
|-------SD----------------------------------------------
|P5.5 (CS)         |--> (O) Manual Chip Select (Active Low) (SD CS)
|P3.5 (SCLK)       |--> (O) SMCLK (SD)
|P3.6 (SIMO)       |--> (O) SIMO (TX Digital In SD)
|P3.7 (S0MI)       |--> (I) SOMI (RX Digital Out SD)
|--------------------------------------------------------
|P1.0              |--> (O) Red LED -> Error
|P2.1              |--> (O) Green LED -> Sampling
|P2.2              |--> (O) Blue LED -> Data transfer
|                  |
|                  |
```

Figure 6.1: GPIO Input and Output

The mark of I or O implies and Input operation or an Output operation for each pin.

The group of P1 and P3 belongs to SPI A and SPI B modules also P7 belongs to Timer A1 module. The P8, P5, P2 and Led Pins will be played manually.

6.2.3. NVIC

The NVIC module controls the Interrupt operation of the MSP432, this module is a newly created one with respect older version like MSP430. The required modules (DMA1, DMA2, TimerA2, TimerA3, Timer 32A and Timer 32B) will enable their interrupt capabilities and mentioned interrupts will be stored in the interrupt vector. [9]

```
SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;              // Enable sleep on exit from ISR
__enable_irq();                                   // Enable Master Interrupts Interrupts
NVIC->ISER[0] = 1 << ((INT_T32_INT1 - 16) & 31);  // Push P1 interrupt in NVIC module
NVIC->ISER[0] = 1 << ((INT_TA2_0 - 16) & 31);     // Push Q1 interrupt in NVIC module
NVIC->ISER[0] = 1 << ((INT_TA3_0 - 16) & 31);     // Push P1' Q1' interrupt in NVIC module
NVIC->ISER[0] = 1 << ((INT_T32_INT2 - 16) & 31);  // Push Tempo interrupt in NVIC module
MAP_Interrupt_enableInterrupt(INT_DMA_INT1);
MAP_Interrupt_enableInterrupt(INT_DMA_INT2);
```

Figure 6.2: Storing interrupt into the NVIC

The priority is set at default knowing that the interrupts will fire in chain style and any interrupt can't impose another one.

6.2.4. SPI

The SPI modules EUSCIB0 and EUSCIB2 will be configured in order to serve ADC and SD communication.

The correct configuration is shown in the following table:

Table 6.1: SPI Configuration

	Communication Type	
Parameter	**SPI – ADC**	**SPI - SD**
Module	EUSCIB0	EUSCIB2
Mode	Master	Master
Source Clock	SMCLK	SMCLK
Bit Clock Rate	4 MHz	200 KHz – 1 MHz
Polarity	1	1
Phase	0	0
Wiring	4 Wire*	3 Wire

The SPI ADC is configured in 4 wires although the Chip Select gate will not be used to feed the ADC because the ADC will run in 3 wire modes as it was detected that it performs better than in 4 wires mode due to gate-clock derives.

6.2.5. DMA

The DMA configuration requires activating certain channels which the DMA will be aware in order to complete the memory transfer operations needed. According to specification [7] the configuration is shown in the following table:

Table 6.2: DMA Configuration

	Communication Type	
Parameter	**DMA SPI ADC**	**DMA SPI SD**
TX Channel	0	4
RX Channel	1	5
Mode	Basic	Basic
Channel Control	Primary + Arbitrary mode	Primary + Arbitrary mode
Transfer	SPI Rx buffer to data buffer	SPI Rx buffer to data buffer
Length	24 bits	As needed

6.2.6. Timer A & Timer 32

The Timer A and Timer 32 will be configured in order to achieve wake up of the device at the correct instants and to generate clock signals. The following table summarizes the uses and configuration of the Timers A needed for this project:

Table 6.3: Timers A Configuration

Type	Timer A0	Timer A1	Timer A2	Timer A3
Purpose	ADC Sampling Clock	Square Wave Generator	Q Sampler	P' and Q' Sampler
Rate	1 MHz	10 KHz – 1 MHz	-	-
Output	Yes	Yes	No	No
Interrupt	No	No	Yes	Yes

The following table summarizes the uses and configuration of the Timer 32 needed for this project

Table 6.4: Timers 32 Configuration

Type	Timer 32 - 0	Timer 32 - 1
Purpose	Main End Timer	P Sampler
Rate	0,1 Hz	40 Hz
Output	No	No
Interrupt	Yes	Yes

6.3. The Start-Sleep-End

The Start-Sleep-End phase should be able to start the Sampler timer and the main end program timer and then put system into sleep mode to save energy. After the main timer is over the system should stop all clocks operation and proceed to storing phase.

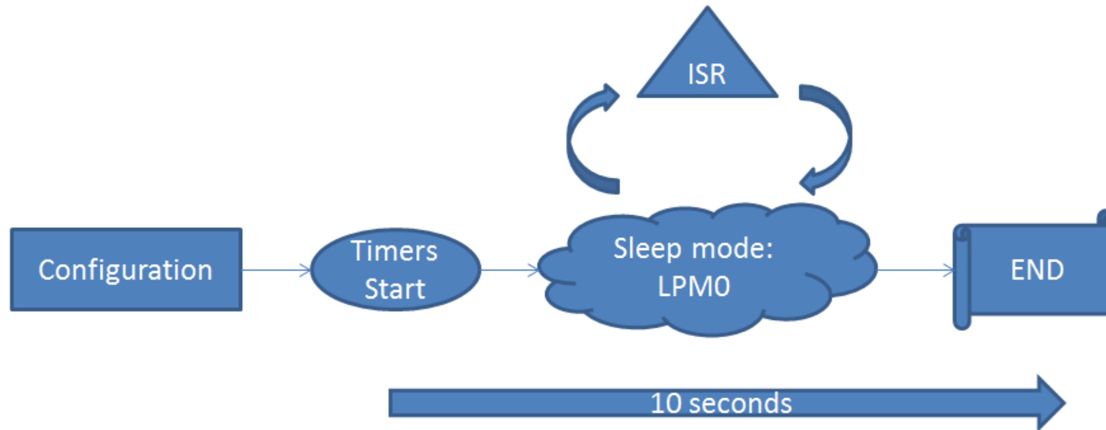


Figure 6.3: Start Sleep End Phase procedure

The system is put into LPM0 sleep mode. This is a mode of operation where CPU turns off but peripherals are still on. The consumption of this power mode is around 60 µA / MHz.

The LPM3 mode enables deep sleep and 650 nA / MHz consumption but the peripherals are turned off so we cannot operate in this power mode because NVIC would not work. [10]

6.4. The Storing

The Storing phase will create a digital Hex vector from the digital received sample vector and proceed to initialize the Secure Digital High Capability (SDHC) Card through SPI and DMA modules.

Finally the storing phase should communicate with the SDHC Card and according to FAT32 standard so that the Digital Sample data is deployed within and archive stored into Root folder.

The following graph shows the communication routine for SDHC card for initialization:

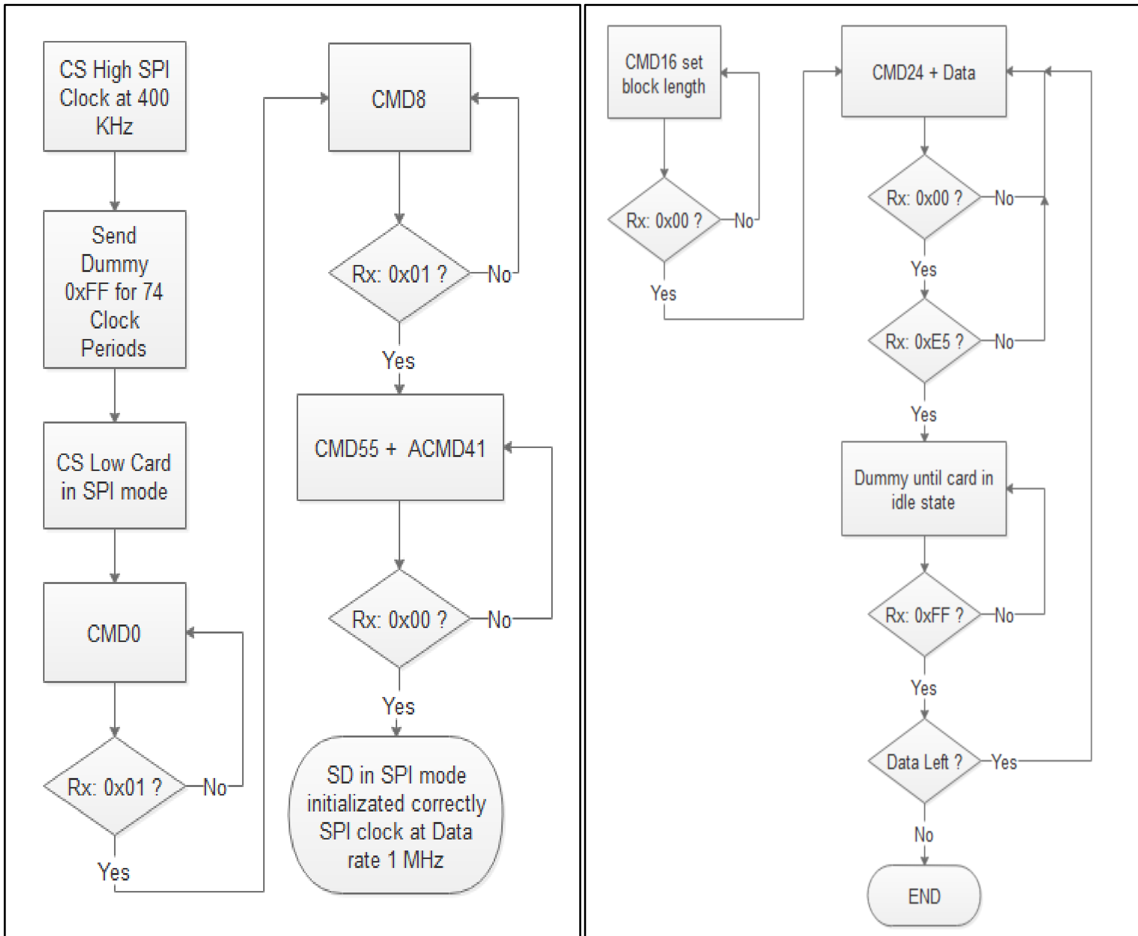


Figure 6.4: Left: SDHC Card initialization of SPI procedure. Right: SDHC Card data transfer protocol

Once the SDHC Card has been put in SPI mode, reset and initialized the data transfer can proceed following the next flow chart:

6.5. The Active phase: Sample

The sample process phase takes place when system is sleeping at a rate of 40 Hz. This phase is called by the Sample timer Interruption and proceeds to activate the SPI operation of sampling plus the DMA operation of storing the 24 bits into a large buffer and then it proceeds to activate the following sampler timers.
After it is complete the system turns back to sleep mode waiting for the next call of general sample timer interruption to start again.

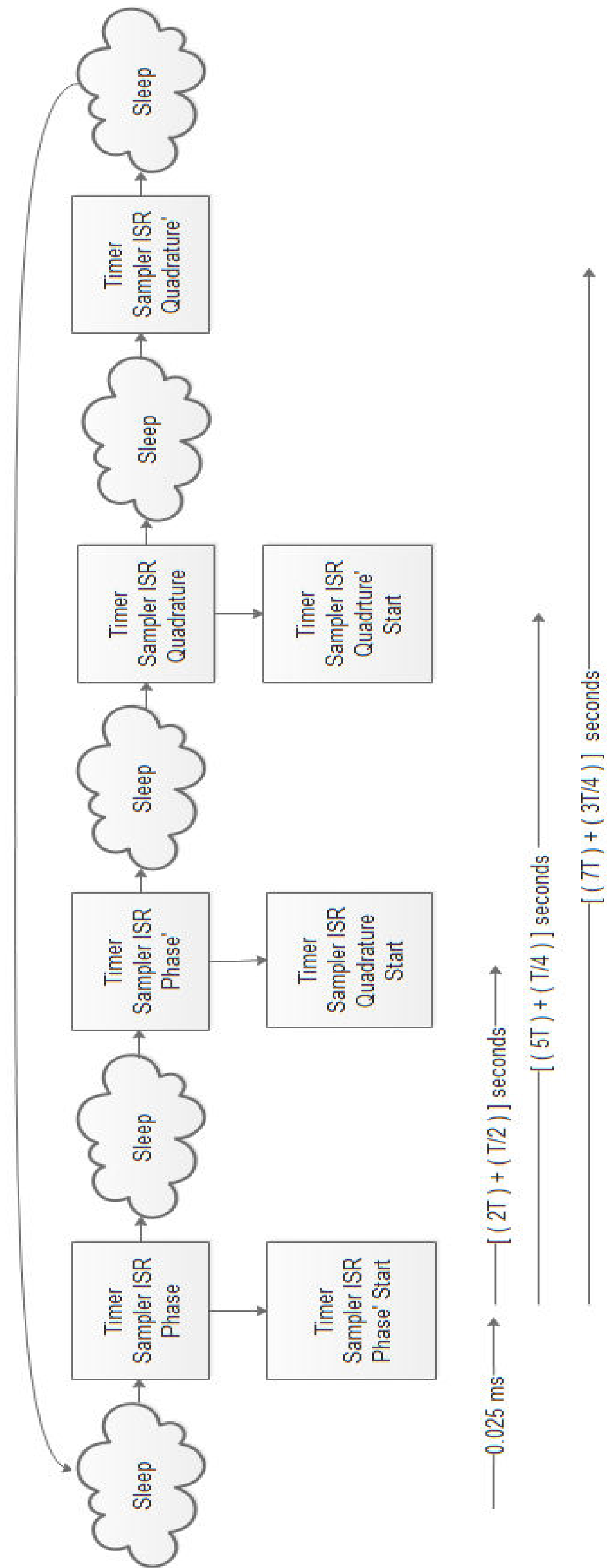


Figure 6.5: Active phase: Sampler program flowchart

The sampling process consist of resetting the ADC thought SPI and take the 1st sample that it is available, otherwise the ADC would not sample correctly due to taking samples at non periodic times multiples of the sampling clock (MCLK) provided to the ADC (AD7766), in this case 1 MHz.

The following capture shows the active phase including the reset of ADC for each sample taken:

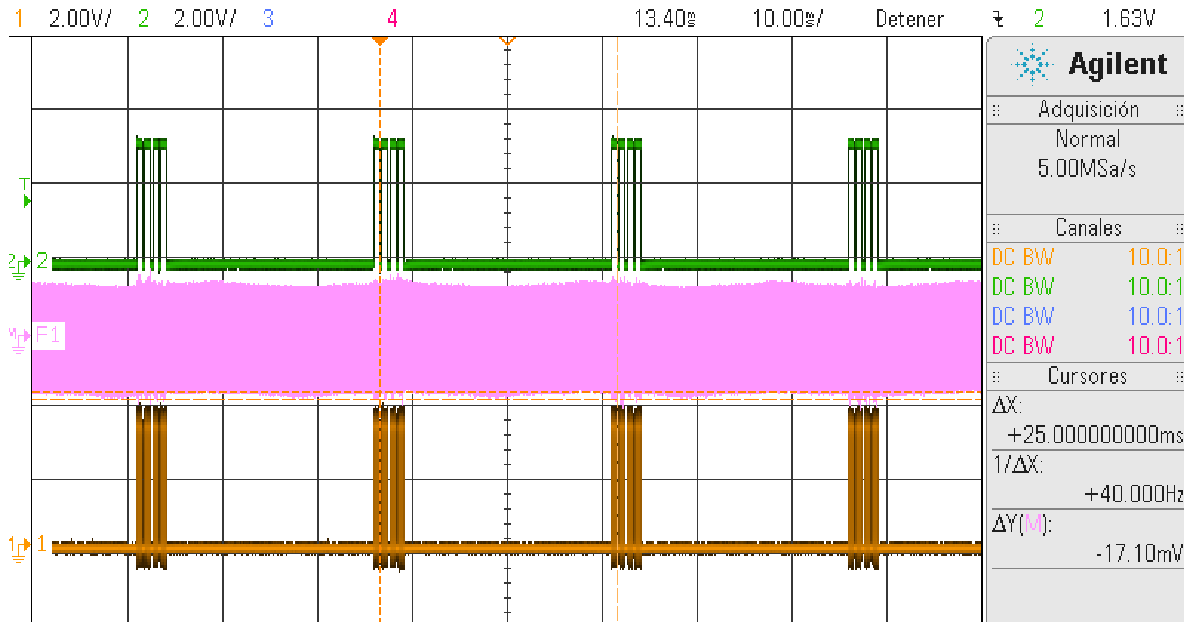


Figure 6.6: Active Phase, green - Data Ready; Purple - Differential signal; yellow - ADC SAR Sample Clock

It can be seen how the active phase takes place every 25 ms (40 Hz). During the Active phase 4 samples are taken obtaining the In-phase, the In-phase' the quadrature and the quadrature' components respectively.

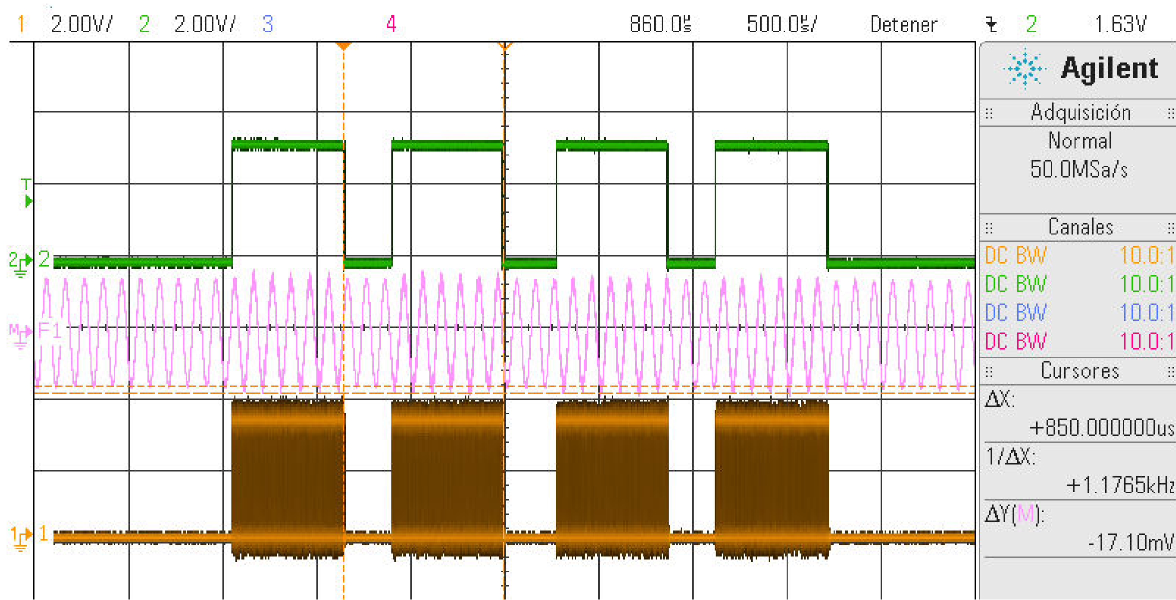


Figure 6.7: 4 Mini active phases, Green - Data Ready; Purple - Differential signal; Yellow - ADC SAR Sample Clock

Now it is seen that the active phase in fact is the sum of 4 mini active phases. Each one takes 1 sample being the first one the in-phase component, the next sample is the in-phase' component is being taken at other period +T/2 of the signal.

The quadrature and quadrature' components are taken at any other period +T/4 so that the synchronous sampling is effective.

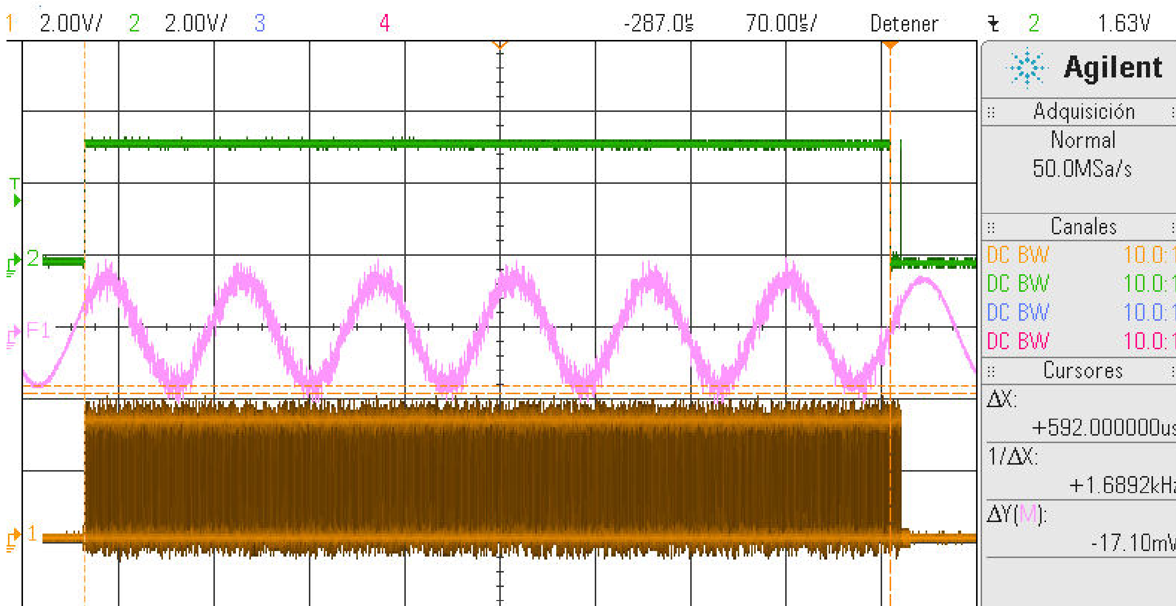


Figure 6.8: Active Phase, ADC sample process, Green - Data Ready; Purple - Differential signal; Yellow - ADC SAR Sample Clock

In order for a sample to be valid the ADC must be feed with the Sampler clock. The SAR ADC needs a sampler clock to be able to approximate the conversion of the voltage successively at every period of the clock. If the clock is taken away from the SAR ADC, the output stream becomes undetermined unless you reset the ADC.

The problem comes when taking a sample at; (x + T/4) or (x + T/2) it must have a sampler clock with a period resolution of at least T/4 so that each T/4 a new sample is available. This is not scalable as for high frequencies (>=1MHz) it will be need a sampler clock at least 4 times higher than the maximum frequency the ADC7766 allows (1 MHz).

So the solution comes whenever we want to take a sample we must reset the ADC and feed it with the sampler clock at its maximum rate. After the initialization of the ADC is completed it comes that the 1st sample is a valid one.

We must repeat this process for every sample we want to take ensuring that the next sample is taken at a time after the last initialization process of the ADC ended while being sure that we respect the +T/4 or +T/2 rule.

The final note is that if the DMA is not used and the SPI is run manually the bit clock of SPI will only transfer 8 cycles (8 bits is the SPI buffer length) and then stop until it performs the operations needed to send another 8 bits.

This derive makes critical the process of reading the 24 bits that ADC is sending to us because the bit clock will stop and some bits will be lost.

Below it is shown the correct use of SPI and DMA in order to read the 24 bits the ADC is sending.

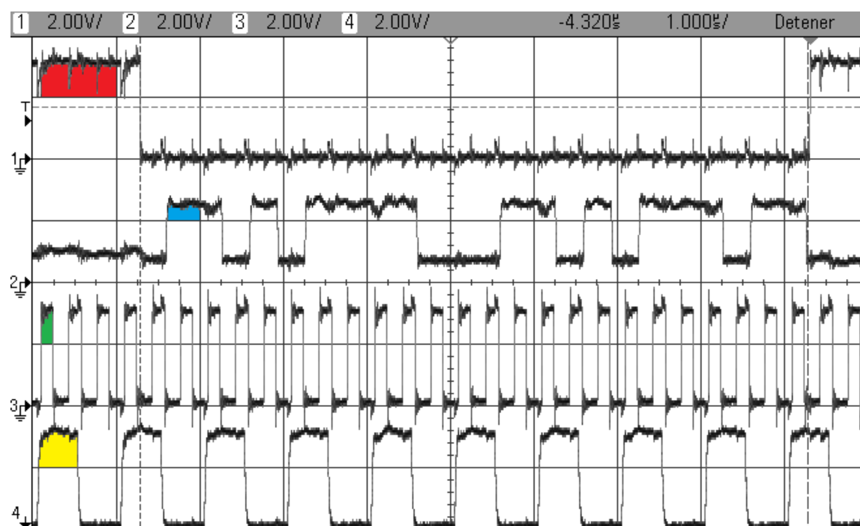


Figure 6.9: Single sample transmission: Red: Chip Select. Blue: Received 24 bits. Green: Bit Clock. Yellow: Sampler Clock.

6.6. Matlab Algorithm

The Data is stored into an SD card. The number of samples is 1600 which is 4 samples during at a rate of 40 Hz during 10 seconds. The sequence of the samples is:

1- In-phase component
2- In-phase' component
3- Quadrature component
4- Quadrature' component

The prime components were taken at T/2 from the non-prime ones. This means that if we subtract and each one and divide by 2 the offset should be erased. This is in fact a digital low pass filter by doubling the number of samples with the criterion aforementioned:

$$final\ P = [(P + offset) - (P' + offset)]/2 \quad (6.1)$$
$$final\ Q = [(Q + offset) - (Q' + offset)]/2 \quad (6.2)$$

After the offset is erased for each pair of in-phase and quadrature components it will be calculated the Module of the voltage using:

$$|V| = \sqrt{final\ P^2 + final\ Q^2} \quad (6.3)$$

And then it will be calculated the module of the impedance using the calibration parameter *K*:

$$|Z| = \frac{|V|}{|K|} \quad (6.4)$$

It will also be calculated the equivalent parallel resistance and parallel Capacitance by using the angle calibration parameter Alf:

$$Z_p = |Z| * \cos(\emptyset - Alf) \quad (6.5)$$
$$Z_q = |Z| * \sin(\emptyset - Alf) \quad (6.6)$$

$$R = (1 + \tan(\emptyset - Alf) * \tan(\emptyset - Alf)) * Zp \quad (6.7)$$
$$C = -(\tan(\emptyset - Alf)/(2 * \pi * f * R) \quad (6.8)$$

Moreover it will be computed the normalized Impedance module by subtracting the Mean of all the samples in the impedance module.

Finally digitals filters (low, high) will be applied in the normalized impedance module in order to reduce some undesired components. Then a FFT will be computed before and after the filters.

The evolution over the 10 seconds of time of the impedance module along with the parallel equivalent resistance and capacitance and the FFT will be shown in a graph.

7. Experimental Results

7.1. Calibration

The calibration will be done using known Impedance as loads. This impedance will be composed of a parallel R and C components plus a series of parallel and C components placed in simulating the electrode set.

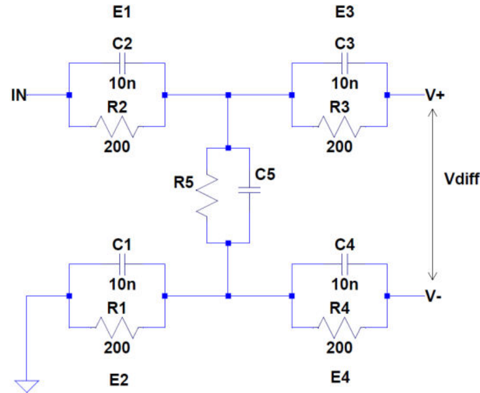


Figure 7.1: Experimental simulation electrode scenario plus load using capacitors and resistors.

The simulated electrode impedance at 10 kHz will be of 185 Ω and at 1 MHz will be of 7 Ω. This only recreates the case where it was placed ECG Gel between the electrode and the skin, reducing the electrode contact impedance.

Using the known impedance (98.655 Ω) and the calculated experimental voltage module we will be able to get the calibration parameter that fill:

$$|K| = \frac{|V|}{|known\ Z|} \tag{6.5}$$

Moreover using a resistive impedance the same value of a capacitance impedance (at 10 kHz for instance) will mean that a known α=90 º will be expected.

Computing the experimental 'Alf' and subtracting to 90 º will lead to the calibration angle:

$$Alf = \propto - 90\,º \tag{6.6}$$

At the end a known load impedance of (70.42 Ω at 10 kHz, 1.0769 Ω at 1 MHz) composed by a parallel resistance of 201.61 Ω and a parallel capacitance of 147.53 nF is used at 10 kHz and 1 MHz but this time the calibration parameters

will be used to calculate the impedance and its components mentioned in the previous chapter.

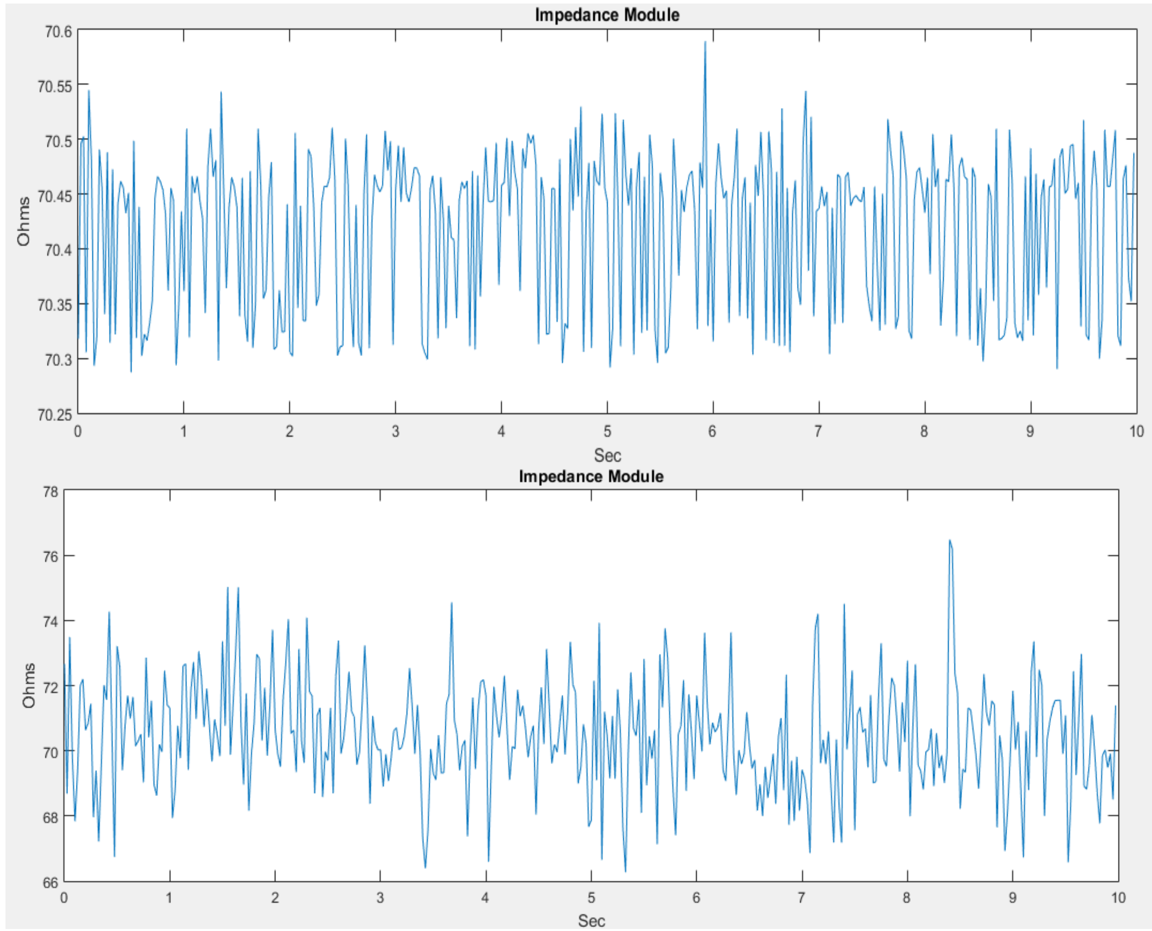
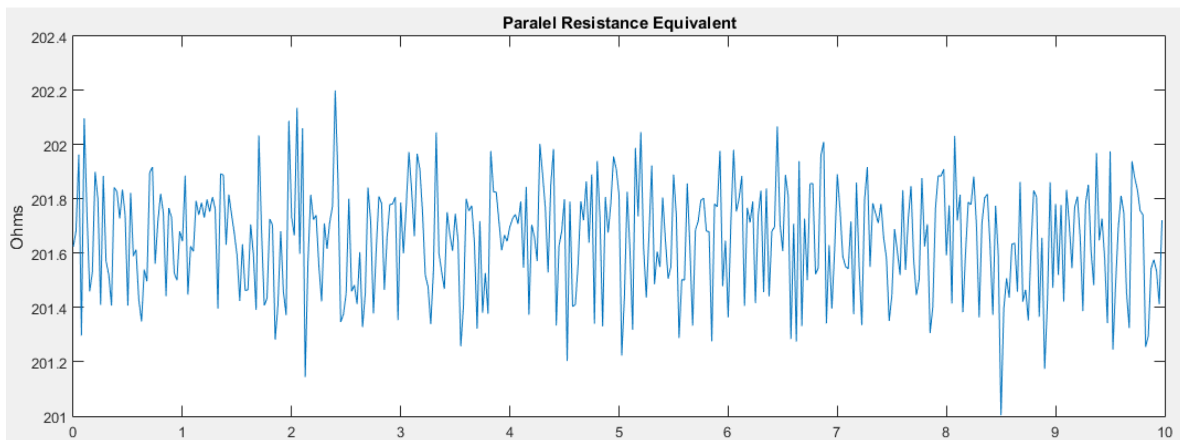


Figure 7.2: Impedance Module of a known load. Up 10: kHz. Down: 1 MHz.



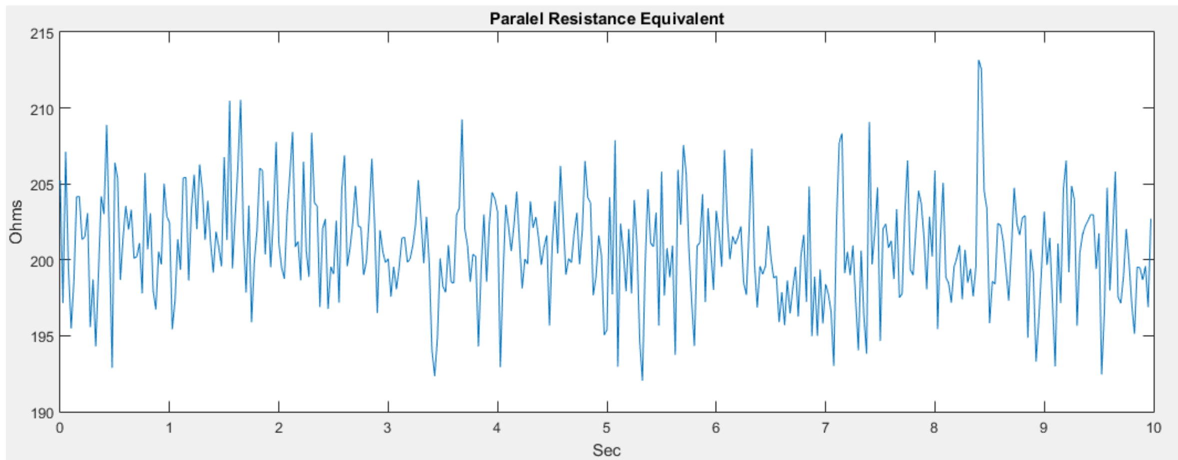


Figure 7.3: Parallel Resistance of a known load. Up 10: kHz. Down: 1 MHz.

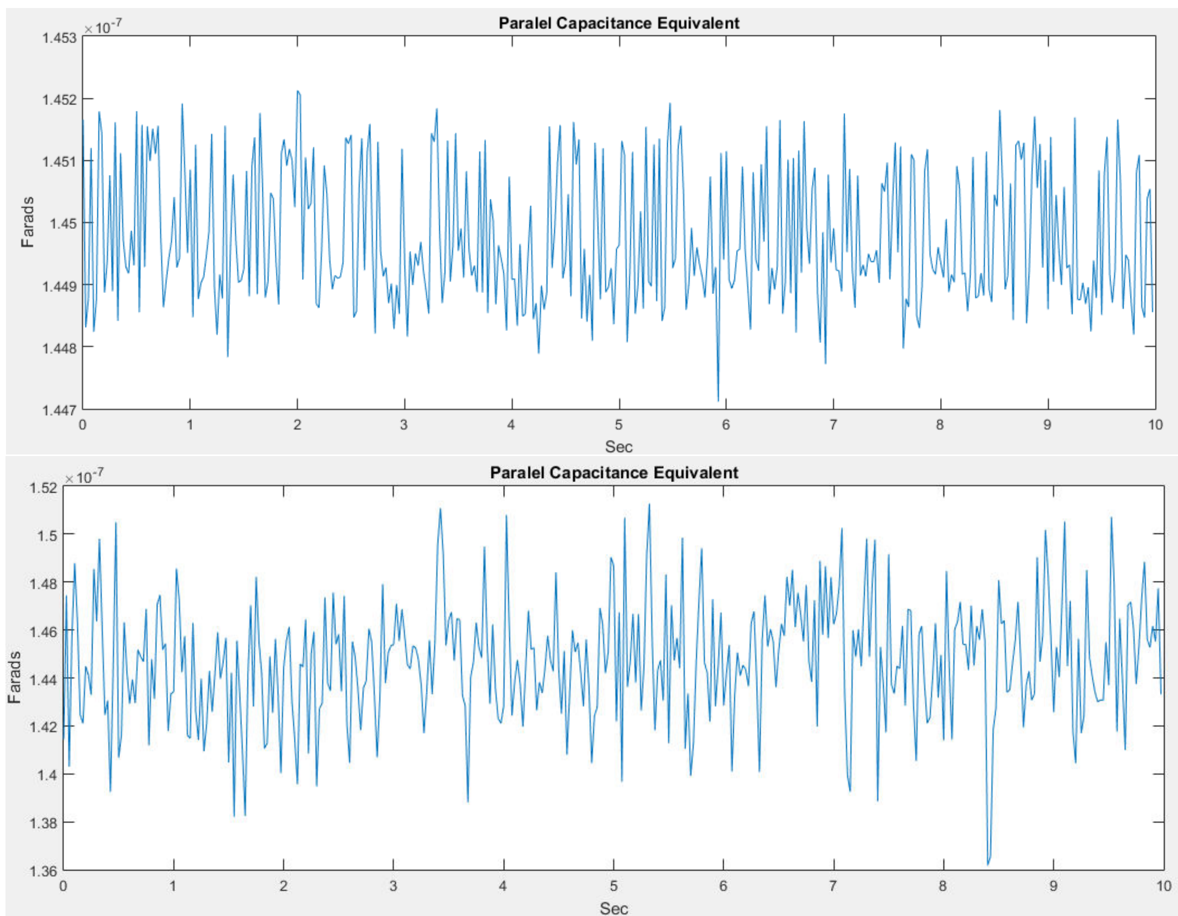


Figure 7.4: Parallel Capacitance of a known load. Up 10: kHz. Down: 1 MHz.

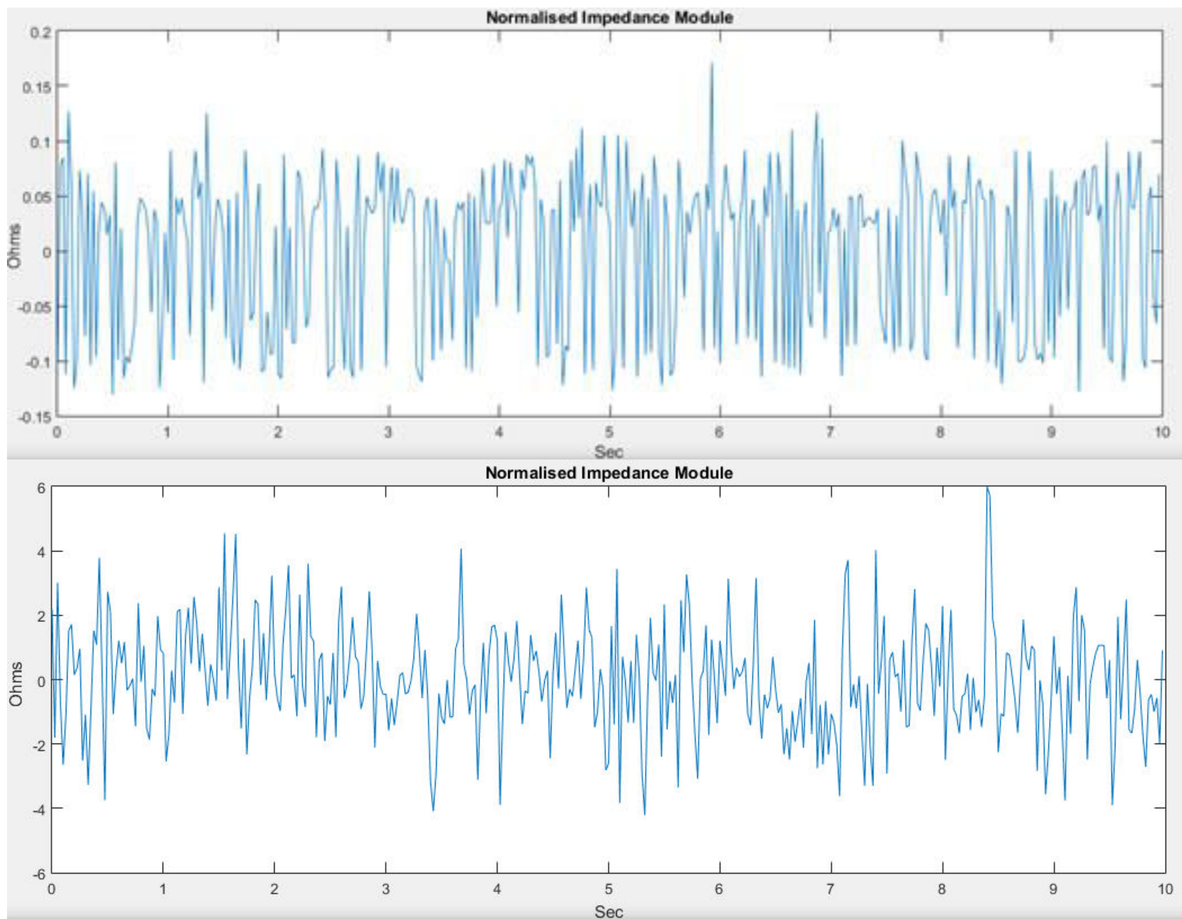
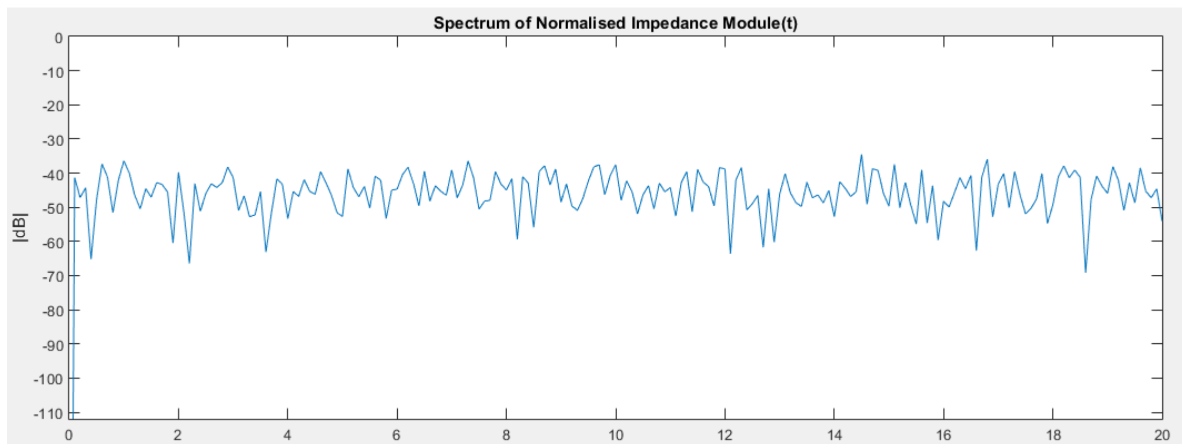


Figure 7.5: Noise Impedance. Up 10: kHz. Down: 1 MHz.



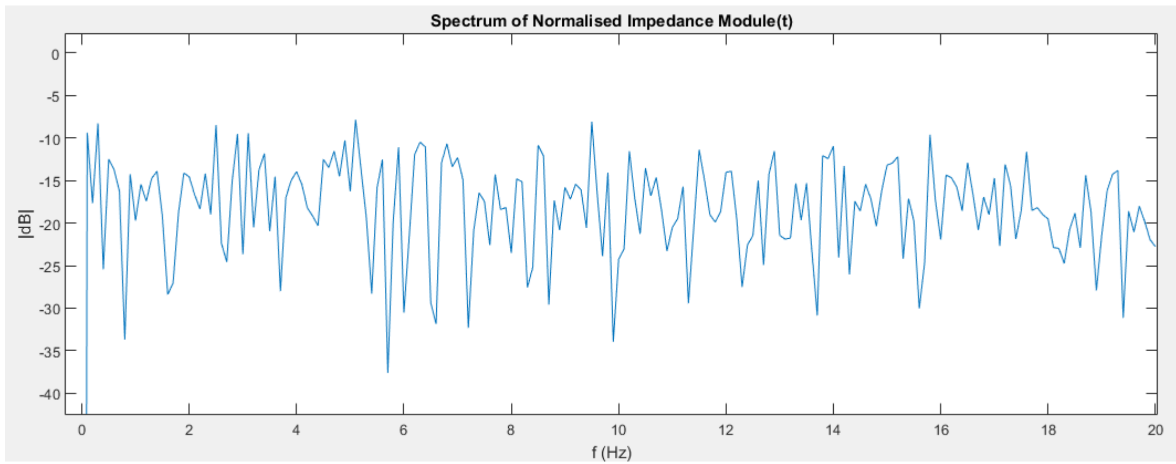


Figure 7.6: FFT of Noise Impedance of known load. Up 10: kHz. Down: 1 MHz.

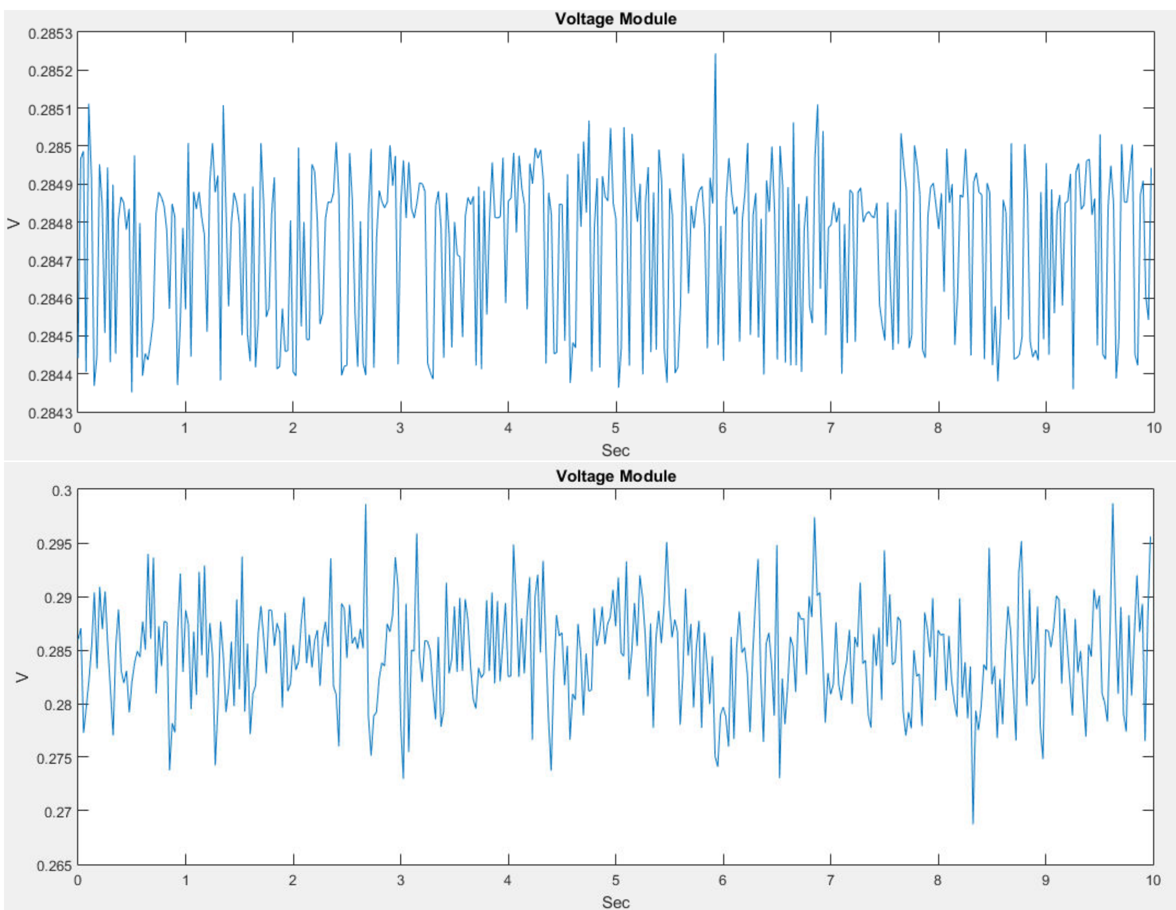


Figure 7.7: Captured Voltage Module (for Noise calculation). Up 10: kHz. Down: 1 MHz.

The Following table shows the precision of the entire system:

Table 7.1: System precision

Type	10 kHz	1 MHz
Standard Deviation Voltage Module Noise	212 µV	5 mV
Standard Deviation Impedance Module Noise	0.03 Ω	1.6 Ω
Impedance Noise Power	-34 dB	-10 dB
Impedance Module Error %	0.05 %	2 %
Experimental (Voltage Full range = 5 V) Resolution	14 bits	10 bits

7.2. IPG

First it was configured a function generator with a modulated voltage change of 1 % simulating the IPG of human body.

The system was able to obtain perfectly the modulated signal as it is observed:

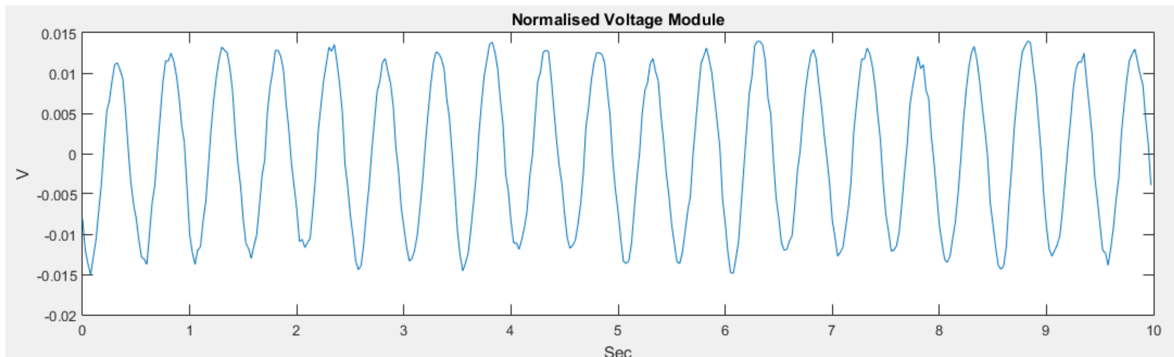


Figure 7.8: Normalized Voltage module of a modulated 2Hz Sine at 10 kHz using 1 % modulation index

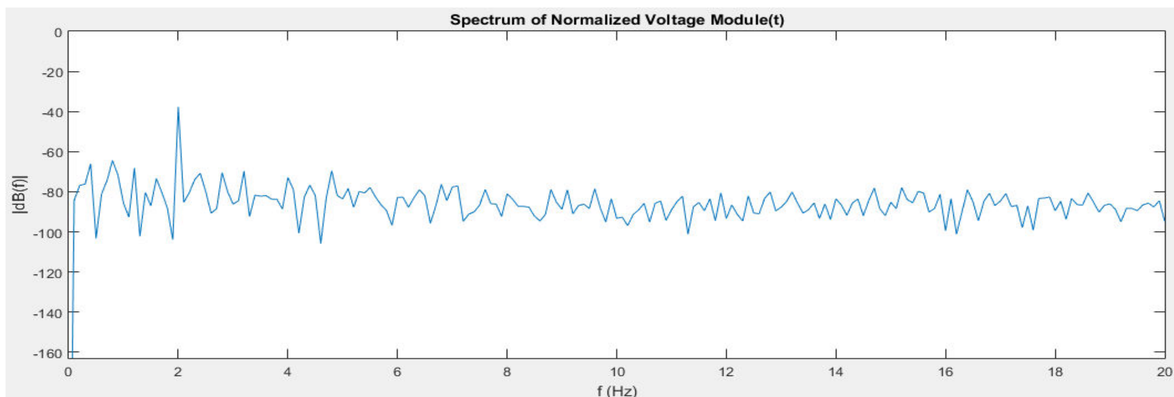


Figure 7.9: FFT captured of a modulated 2Hz Sine at 10 kHz using 1 % modulation index

Then it was configured to work with a modulation of 0.1 % because IPG changes can be from 0.1 to 1 % depending on factors such the subject or electrode's position:

Again the system is able to obtain really well the modulated signal:

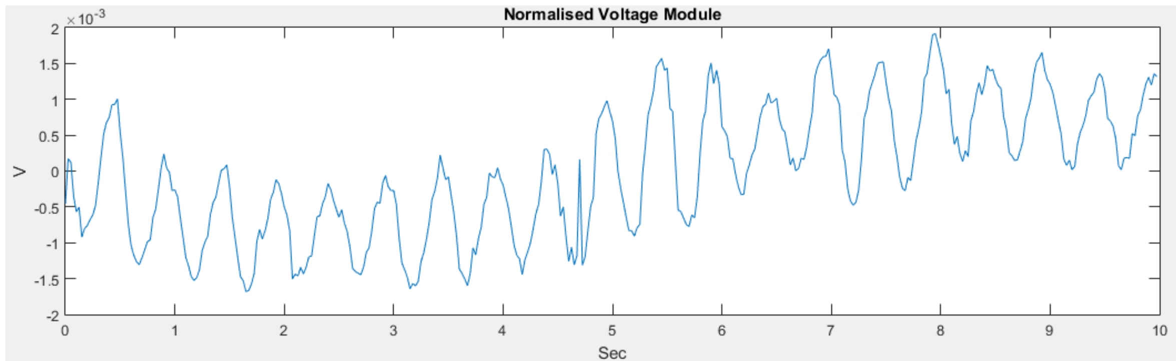


Figure 7.10: Normalized Voltage module of a modulated 2Hz Sine at 10 kHz using 0.1 % modulation index

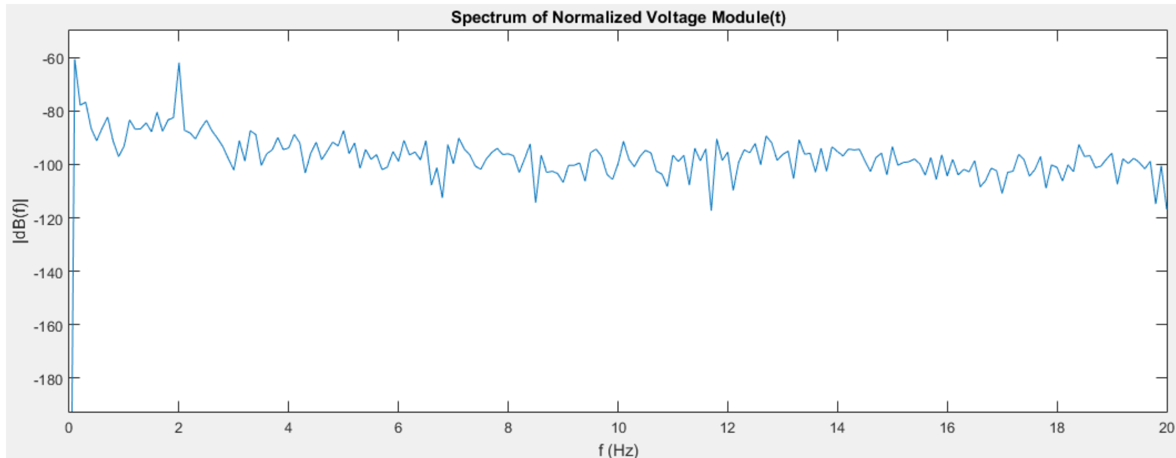


Figure 7.11: FFT captured of a modulated 2Hz Sine at 10 kHz using 0.1 % modulation index

Finally using electrode set RT34 SKINTACT [17] [18] it was performed the measure of IPG on the human body several times, below is the best result obtained:

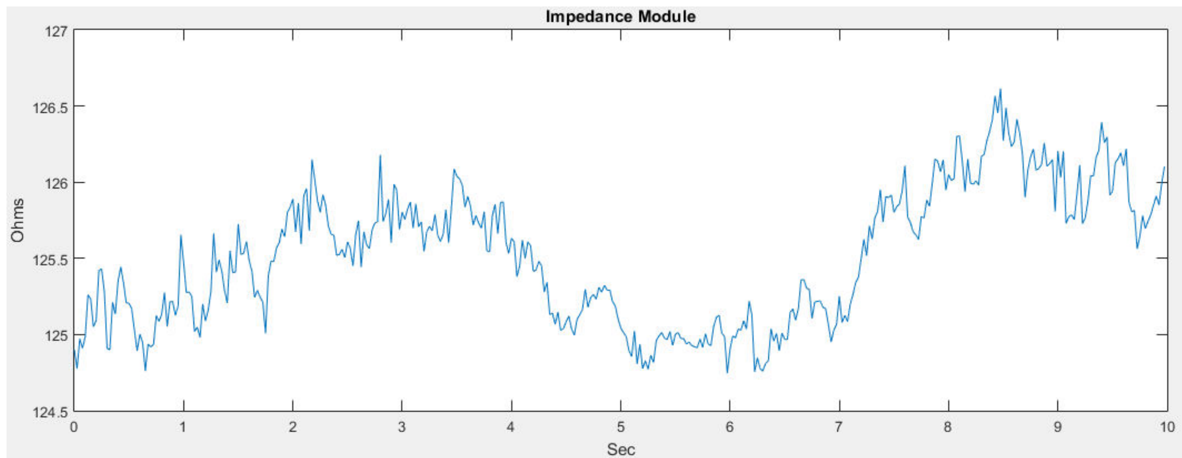


Figure 7.12 IPG at 10 kHz

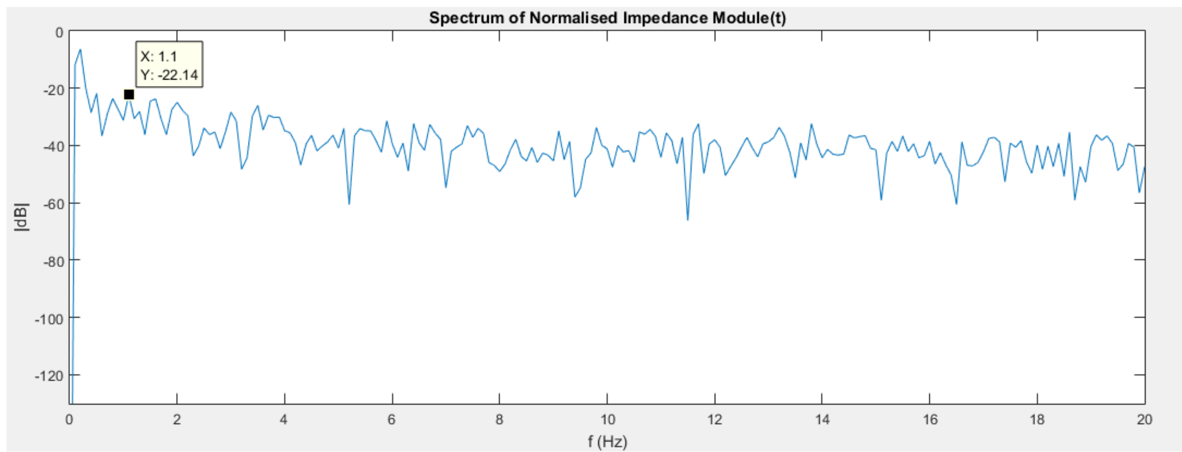


Figure 7.13: FFT of IPG at 10 kHz

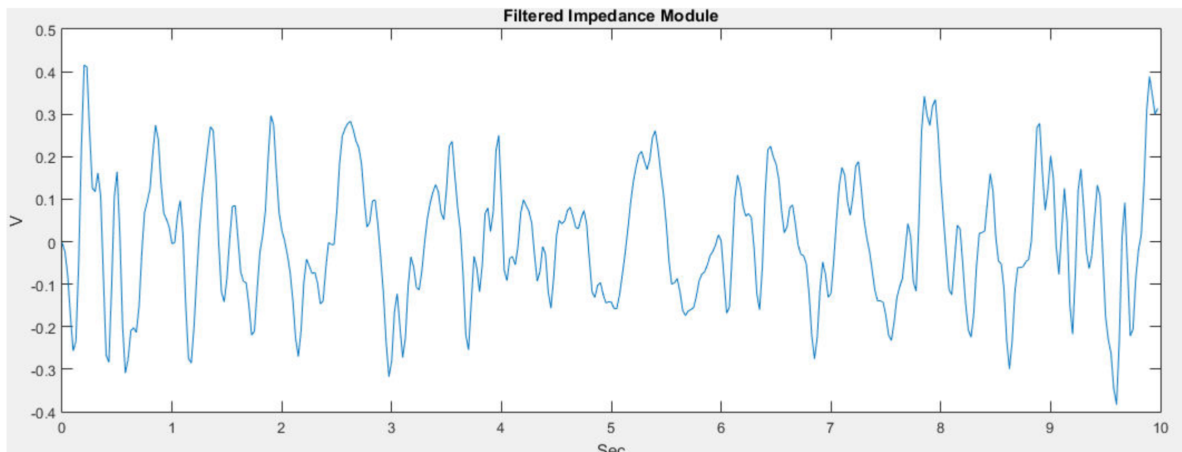


Figure 7.14: Normalized and Filtered IPG at 10 kHz

The system IPG harmonic has not enough power to be clean enough to see it in the results above. The breathing can be well appreciated in Fig 7.11.

However the system was not tested at 1 MHz as the 10 kHz test was worse than expected it has no sense to test at 1MHz where the limitations will be bigger.

8. Conclusions and future work

8.1. Conclusion

It has been build a system from scratch using MSP432 MCU able to obtain through QSS; the load impedance and calculate the parallel equivalent resistance and capacitance with 14 bits of resolution at 10 kHz and 10 bits of resolution at 1 MHz. The system can also demodulate a 2 Hz signal at 10 kHz with an index modulation of 1 % and 0.1 % quite well.

However the results of the IPG at 10 kHz indicate that there is so much noise coupled into the measurements therefore the captured IPG does not appear as well as it was expected.

The system is able to deploy the data into the SD Card so that it can be used without a PC connection. Although the power consumption is not optimized the system can be run by a battery of 9.4 Volts.

The final conclusion is that once the system is optimized in noise and power consumption the entire build system is perfectly viable to perform multifrequency (10-kHz – 1 MHz) IPG analysis at low power consumption.

8.2. Future Work

There is future work to perform following this project. The key aspects to perfection this system should be:

- Use another type of operational amplifiers at the Active Low pass filter so that the generated noise in this stage is lowered.
- Change the voltage rails of the system to +3.3 – 0 V.
- Program multifrequency user selectable output in the interface of MSP432
- Build logic gates so that the circuit performs the filters at the desired working frequency
- Build a new module with another MSP432 plus a LED display to show the IPG results live.

9. Bibliography

[1] M. Kristine Huseby, *FPGA Based development platform for biomedical measurements*. Master Thesis, Universitas Osloensis, Spring 2013, 1,2.

[2] O. Puig Mas, *Electronic design comparison for obtaining the IPG at the wrist*. Master Thesis, Universistat Politecnica de Catalunya, September 2015 1,2.

[3] G. Xercavins Torregrosa, *Multichannel Bioimpedance meter for cardiovascular time interval analysis*, Universitat Politecnica de Catalunya, October 2016, 1,2

[4] Kubicek WG, Patterson RP, Witsoe DA (1970): *Impedance cardiography as a non-invasive method for monitoring cardiac function and other parameters of the cardiovascular system*. Ann. N.Y. Acad. Sci. 170: 724-32.

[5] Durrer D. *Electrical aspects of human cardiac activity: a clinical-physiological approach to excitation and stimulation*. Cardiovascular Res. 1968; 2:1–18

[6] Aaron S.Tucker, Robert M. Fox, Rosalind J. Sadleir: *Biocompatible High Precision Wideband Improved Howland Current Source With Lead-Lag Compensation.* IEE Transactions on biomedical circuits and systems Vol 7 February 2013

[7] Texas Instruments*. MSP432P4xx Family Technical Reference Manual*, 302

[8] Texas Instruments*. MSP432P4xx Family Technical Reference Manual*, 504

[9] Texas Instruments*. MSP432P4xx Family Technical Reference Manual*, 105

[10] Texas Instruments*. Designing an Ultra-Low-Power (ULP) Application with MSP432™ Microcontroller.* March 2015

[12] Antoni Ivorra, Meritxell Genesca, Anna Sola, Luis Palacios, Rosa Villa, Georgina Hotter and Jordi Aguilo. *Bioimpedance dispersion width as a parameter to monitor living tissues*. Institute of physics publishing physiological measurements. 26. 1-9. November 2014

[13]Lee Stoner, Joanna M. Young, Simon Fryer. *Assessments of Arterial Stiffness and Endothelial Function Using Pulse Wave Analysis.* International Journal of Vascular Medicine, 2012.

[14] Ramon Pallas-Areny, John G. Webster*. Bioelectric Impedance Measurements using synchronous sampling. IEEE Transactions on biomedical engineering*. Vol 40. Nº 8. August 1993

[15] Dhouha Bouchaala, Qinghai Shi, Olfa Kanoun, Nabil Derbel. *A High Accuracy Voltage Controlled Current Source for Handheld Bioimpedance Measurement*. International Multi-Conference on Systems, Signals & Devices. March 18-21 2013

[16] Analog Devices. *AD8041 Datasheet*. Revision B

[17] SKINTACT Electrodes Catalog PDF. Leonhard Lang Publication October 2015

[18] SKINTACT Electrodes RT34 Technical Information PDF. Revision A. April 2014

Annex A: MCU Code

```
/*****************************************************************************
*
* MSP432 main.c template - Empty main
*
*
*  Created on: 20/5/2016
*      Author: Josh
*
*
*
*           MSP432P401
*         -----------------
*     /|\|                 |
*      | |                 |
*      --|RST              |
*        |                 |
*        |P7.6/TA1.1 InGen |--> Square Input Wave Generator
*       |P8.4              |--> Calibration Pin
*      |P8.6               |--> ADC7766 Power Up / Down Pin
*       |P8.7 (DRDY)       |--> DRDY detect
*       |-------ADC---------------------------------------------
*       |P1.4 (CS)               |--> Chip Select (Active Low) (ADC CS)
*       |P2.4 (MCLK)      |--> MCLK (ADC)
*        |P1.5 (SCLK)      |--> SCLK (ADC)
*        |P1.7 (S0MI)      |--> SOMI (RX Digital Out ADC)
*        |P1.6 (SIMO)      |--> SIMO - (TX)
*       |-------SD---------------------------------------------
*       |P5.5 (CS)               |--> Manual Chip Select (Active Low) (SD CS)
*       |P3.5 (SCLK)      |--> SMCLK (SD)
*        |P3.6 (SIMO)      |--> SIMO (TX Digital In SD)
*        |P3.7 (S0MI)      |--> SOMI (RX Digital Out SD)
*        |---------------------------------------------------------
        |P1.0                  |--> Red LED -> Error
*       |P2.1              |--> Green LED -> Sampling
*       |P2.2              |--> Blue LED -> Data transfer
*
*
* CONFIGURATION NUMBER
*
* 1-   5   KHz
* 2-   10  KHz
* 3-   20  KHz
* 4-   50  KHz
* 5-   100 KHz
* 6-   200 KHz
* 7-   500 KHz
* 8-   1   MHz
* 9-   2   MHz
* 10-  6   MHz
* 11-  12  MHz
*
```

```
*/

/* DriverLib Includes */
#include <driverlib.h>
#include <msp.h>

/* Standard Includes */
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

/* Own Includes */
#include <autoconfig.h>


void main(void){

        WDTCTL = WDTPW | WDTHOLD;                   // Stop WDT

        setup_clock();                              // Set-Up ALL CLOCKS

     Configure_Master_Interrupts();           // Configure Master Interrupts

     Configure_GPIO();                              // Configure GPIO PINS

        SelecT_InputFreq(2);                  // Configure Input Frequency

        Configure_Timers();                         // Configure Timers

     Configure_SPI();                               // Configure SPI Bus

     Configure_DMA();                               // Configure DMA

        InitADC_OA();                         // ADC7766 Power Up Sequence

     Timers_START();                                // Starts Sampling Process


    while(1){
   __no_operation();                                // For debugger
    }
}


/*
* autoconfig.h
*
*  Created on: 20/5/2016
*      Author: Josh
*
*
* CONFIGURATION NUMBER
*
* 1-  5   KHz
```

```
* 2-  10  KHz
* 3-  20  KHz
* 4-  50  KHz
* 5-  100 KHz
* 6-  200 KHz
* 7-  500 KHz
* 8-  1   MHz
* 9-  2   MHz
* 10- 6   MHz
* 11- 12  MHz
*
* PUBLIC FUNCTIONS :
*
*             void setup_clock(void);
*             void SelecT_InputFreq (int number1);
*             void Configure_Master_Interrupts(void);
*             void Configure_GPIO(void);
*             void Configure_Timers(void);
*             void Configure_SPI(void);
*             void Configure_DMA(void);
*             void Init_ADC_OA(void);
*             void Timers_START(void);
*             void Data_Dump(void);
*             void error(void);
*
*             void SD_init(void);
*             void cs_enable(void);
*             void cs_disable(void);
*             void sd_spi_data_dump(void);
*             void send_command_sd(uint8_t []);
*             void send_sincro_command_sd(uint8_t []);
*             void send_esp_command_sd(uint8_t []);
*             void send_data_token_sd(uint8_t []);
*             void configure_DMA_TX(int);
*             int check_response (int);
*             int check_data_response (int);
*             void dec_hex(uint32_t);
*             void cmd_plus_data_creation(uint32_t);
*             void clean_DMA(void);
*             void clean_data_DMA(void);
*
*/





/* SPI Configuration Parameter */
extern const eUSCI_SPI_MasterConfig spiMasterConfig;


/*----------------------------------- SPI DMA Data ---------------------------
------*/
extern uint8_t mstxData[3];
```

```
extern uint8_t msrxData[3];
extern uint8_t bufferedData1[4800];
extern uint8_t mychar_bit[41984];
extern uint32_t umode;
extern uint32_t umode1;
extern uint32_t ii;
extern uint32_t sampleX;
extern int myT_Square;
/*----------------------------------- SPI DMA SD -----------------------------
----*/
extern uint8_t sdtxData[16];
extern uint8_t sdrxData[16];
extern uint8_t sdtx_TOKEN_Data[531];
extern uint8_t sdrx_TOKEN_Data[531];
extern uint8_t sdtx_sincro_Data[10];
extern uint8_t bufferedSD_RX[8];
/*------------------------------------SPI MSG SD PROTOCOL----------------------
------*/
extern uint8_t Dummy[16];
extern uint8_t Dummy1[10];
extern uint8_t CMD0[16];
extern uint8_t CMD8[16];
extern uint8_t CMD55[16];
extern uint8_t ACMD41[16];
extern uint8_t CMD16[16];
extern uint8_t CMD24_PLUS_ROOT_DATA[531];
extern uint8_t CMD24_PLUS_FAT_DATA[531];
extern uint8_t CMD24_PLUS_CPY_ROOT_DATA[531];
extern uint8_t CMD24_PLUS_CPY_FAT_DATA[531];
extern uint8_t CMD24_PLUS_ARCHIVE_DATA[531];

/*--------------------------------MAIN FUNCTIONS-------------------------------
----------------*/
void setup_clock(void);                                                 // Set
up all Clocks
void SelecT_InputFreq (int);                                      // Configure
Square Wave Frequency
void Configure_Master_Interrupts(void);                    // Configure Master
Interrupts
void Configure_GPIO(void);                                        // Configure
GPIO PINS
void Configure_Timers(void);                                      // Configure
Timers
void Configure_SPI(void);                                         // Configure
SPI Bus
void Configure_DMA(void);                                         // Configure
DMA
void Init_ADC_OA(void);                                                 // Init
ADC and Operational Amplifiers
void Timers_START(void);                                          // Starts
Sampling Process
void Data_Dump_PC(void);                                          // Dump RX
Data buffer into SD
void Data_Dump_SD(void);                                          // Dump RX
Data buffer into SD
```

```
void error(void);                                                    // Check
error function
/*--------------------------------SD FUNCTIONS---------------------------------
----------------*/
void SD_init(void);                                                  //
Initialization of SD - SPI Protocol
void cs_enable(void);                                                // SD CS
Low State
void cs_disable(void);                                               // SD CS
High State
void sd_spi_data_dump(void);                                   // Dump
Buffered Data to SD
void send_command_sd(uint8_t []);                              // Send
Command to SD
void send_sincro_command_sd(uint8_t []);                 // Send Command to SD
void send_esp_command_sd(uint8_t []);                    // Send Special
Command
void send_data_token_sd(uint8_t []);                     // Send Data Token to
SD
void configure_DMA_TX(int);                                          //
Reconfigure DMA TX Buffer Length
int check_response (int);                                      // Check SD
Response for all Cases
int check_data_response (int);                                 // Check SD
Response for Archive Data writing
void dec_hex(uint32_t);                                              //
Convert decimal offset to hexadecimal offset
void cmd_plus_data_creation(uint32_t);                   // create cmd 24
void clean_DMA(void);                                                // Clean
SD DMA
void clean_data_DMA(void);                                     // Clean SD
DMA for archive data operations


/*
 * autoconfig.c
 *
 *  Created on: 20/5/2016
 *      Author: Josh
 *
 *
 * CONFIGURATION NUMBER
 *
 * 1-   5   KHz
 * 2-   10  KHz
 * 3-   20  KHz
 * 4-   50  KHz
 * 5-   100 KHz
 * 6-   200 KHz
 * 7-   500 KHz
 * 8-   1   MHz
 * 9-   2   MHz
 * 10-  6   MHz
 * 11-  12  MHz
 *
```

```
 *
* PUBLIC FUNCTIONS :
*
*            void setup_clock(void);
*            void SelecT_InputFreq (int number1);
*            void Configure_Master_Interrupts(void);
*            void Configure_GPIO(void);
*            void Configure_Timers(void);
*            void Configure_SPI(void);
*            void Configure_DMA(void);
*            void Init_ADC_OA(void);
*            void Timers_START(void);
*            void Data_Dump(void);
*            void error(void);
*
*            void SD_init(void);
*            void cs_enable(void);
*            void cs_disable(void);
*            void sd_spi_data_dump(void);
*            void send_command_sd(uint8_t []);
*            void send_sincro_command_sd(uint8_t []);
*            void send_esp_command_sd(uint8_t []);
*            void send_data_token_sd(uint8_t []);
*            void configure_DMA_TX(int);
*            int check_response (int);
*            int check_data_response (int);
*            void dec_hex(uint32_t);
*            void cmd_plus_data_creation(uint32_t);
*            void clean_DMA(void);
*            void clean_data_DMA(void);
*
*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
/* DriverLib Includes */
#include <driverlib.h>
#include <msp.h>


/* SPI Configuration Parameter */
const eUSCI_SPI_MasterConfig spiMasterConfig =
{ EUSCI_B_SPI_CLOCKSOURCE_SMCLK, 48000000, 3000000,
        EUSCI_B_SPI_MSB_FIRST,
        EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT,
        EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH, UCMODE_2};


/* DMA Control Table */
#if defined(__TI_COMPILER_VERSION__)
#pragma DATA_ALIGN(MSP_EXP432P401RLP_DMAControlTable, 1024)
#elif defined(__IAR_SYSTEMS_ICC__)
#pragma data_alignment=1024
#elif defined(__GNUC__)
```

```
__attribute__ ((aligned (1024)))
#elif defined(__CC_ARM)
__align(1024)
#endif

/* Statics */
static int myT_Square=0;
static DMA_ControlTable MSP_EXP432P401RLP_DMAControlTable[32];

#define MAP_SPI_MSG_LENGTH    3
#define MAP_SD_SPI_MSG_LENGTH    16
#define MAP_SD_SPI_SINC_LENGTH   10
#define MAP_SD_SPI_TOKEN_LENGTH   531
#define MAP_SD_SPI_RXT_LENGTH   531

uint8_t mstxData[3] = { 0 };
uint8_t msrxData[3] = { 0 };


uint8_t sdtx_TOKEN_Data[531] = { 0 };
uint8_t sdrx_TOKEN_Data[531] = { 0 };
uint8_t sdtx_sincro_Data[10] = { 0 };
uint8_t sdtxData[16] = { 0 };
uint8_t sdrxData[16] = { 0 };

uint8_t bufferedData1[5400] = { 0 };
uint8_t mychar_bit[41984] = { 0 };
uint32_t tarp[1800] = { 0 };

uint32_t umode = 0;
uint32_t umode1 = 0;
uint32_t mitarp =0;
uint32_t pru =0;
uint32_t ii;
uint32_t ie=0;
int8_t e=0;
int8_t p_q_p=0;
uint32_t xc=0;
uint32_t w_sd=0;
uint32_t sampleX = 0;
uint32_t sd_data_loop=0;
uint32_t data_read=0;
uint16_t pounter=0;
uint32_t B_l=0;

uint32_t pos_dec=0;
uint8_t pos_hex[4] = { 0 };

uint32_t drdycount=0;



uint8_t Dummy[16] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
             0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t Dummy1[10] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
```

```
            0xFF,0xFF};
uint8_t CMD0[16] = {0x40,0x00,0x00,0x00,0x00,0x95,0xFF,0xFF,
            0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t CMD8[16] = {0x48,0x00,0x00,0x01,0xAA,0x87,0xFF,0xFF,
            0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t CMD55[16] = {0x77,0x00,0x00,0x00,0x00,0xFF,0xFF,0xFF,
            0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t ACMD41[16] = {0x69,0x40,0x00,0x00,0x00,0xFF,0xFF,0xFF,
            0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t CMD16[16] = {0x50,0x00,0x00,0x20,0x00,0xFF,0xFF,0xFF,
            0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};




uint8_t CMD24_PLUS_ROOT_DATA[531] = {0x58,0x00,0x00,0x40,0x00,0xFF,0xFF,0xFF,
            0xFE,0x42,0x20,0x00,0x49,0x00,0x6E,0x00,
            0x66,0x00,0x6F,0x00,0x0F,0x00,0x72,0x72,
            0x00,0x6D,0x00,0x61,0x00,0x74,0x00,0x69,
            0x00,0x6F,0x00,0x00,0x00,0x6E,0x00,0x00,
            0x00,0x01,0x53,0x00,0x79,0x00,0x73,0x00,
            0x74,0x00,0x65,0x00,0x0F,0x00,0x72,0x6D,
            0x00,0x20,0x00,0x56,0x00,0x6F,0x00,0x6C,
            0x00,0x75,0x00,0x00,0x00,0x6D,0x00,0x65,
            0x00,0x53,0x59,0x53,0x54,0x45,0x4D,0x7E,
            0x31,0x20,0x20,0x20,0x16,0x00,0x64,0xEE,
            0xB4,0x7B,0x49,0x7B,0x49,0x00,0x00,0xEF,
            0xB4,0x7B,0x49,0x03,0x00,0x00,0x00,0x00,
            0x00,0x44,0x41,0x54,0x41,0x20,0x20,0x20,
            0x20,0x54,0x58,0x54,0x20,0x10,0x5E,0xF6,
            0xB4,0x7B,0x49,0x7B,0x49,0x00,0x00,0xC6,
            0xB6,0x53,0x49,0x06,0x00,0x7E,0xA2,0x00,
            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

uint8_t CMD24_PLUS_CPY_ROOT_DATA[531] = {0x58,0x00,0x00,0x40,0x40,0xFF,0xFF,0xFF,
            0xFE,0x2E,0x20,0x20,0x20,0x20,0x20,0x20,0x20,
            0x20,0x20,0x20,0x10,0x00,0x64,0xEE,0xB4,
            0x7B,0x49,0x7B,0x49,0x00,0x00,0xEF,0xB4,
            0x7B,0x49,0x03,0x00,0x00,0x00,0x00,0x00,
            0x2E,0x2E,0x20,0x20,0x20,0x20,0x20,0x20,
            0x20,0x20,0x20,0x10,0x00,0x64,0xEE,0xB4,
            0x7B,0x49,0x7B,0x49,0x00,0x00,0xEF,0xB4,
            0x7B,0x49,0x00,0x00,0x00,0x00,0x00,0x00,
            0x42,0x47,0x00,0x75,0x00,0x69,0x00,0x64,
            0x00,0x00,0x00,0x0F,0x00,0xFF,0xFF,0xFF,
            0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
            0xFF,0xFF,0x00,0x00,0xFF,0xFF,0xFF,0xFF,
            0x01,0x49,0x00,0x6E,0x00,0x64,0x00,0x65,
            0x00,0x78,0x00,0x0F,0x00,0xFF,0x65,0x00,
            0x72,0x00,0x56,0x00,0x6F,0x00,0x6C,0x00,
            0x75,0x00,0x00,0x00,0x6D,0x00,0x65,0x00,
            0x49,0x4E,0x44,0x45,0x58,0x45,0x7E,0x31,
            0x20,0x20,0x20,0x20,0x00,0xB3,0xEE,0xB4,
            0x7B,0x49,0x7B,0x49,0x00,0x00,0xEF,0xB4,
```

```
                0x7B,0x49,0x04,0x00,0x4C,0x00,0x00,0x00,
                0x42,0x74,0x00,0x00,0x00,0xFF,0xFF,0xFF,
                0xFF,0xFF,0xFF,0x0F,0x00,0xCE,0xFF,0xFF,
                0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
                0xFF,0xFF,0x00,0x00,0xFF,0xFF,0xFF,0xFF,
                0x01,0x57,0x00,0x50,0x00,0x53,0x00,0x65,
                0x00,0x74,0x00,0x0F,0x00,0xCE,0x74,0x00,
                0x69,0x00,0x6E,0x00,0x67,0x00,0x73,0x00,
                0x2E,0x00,0x00,0x00,0x64,0x00,0x61,0x00,
                0x57,0x50,0x53,0x45,0x54,0x54,0x7E,0x31,
                0x44,0x41,0x54,0x20,0x00,0xB4,0xEE,0xB4,
                0x7B,0x49,0x7B,0x49,0x00,0x00,0xEF,0xB4,
                0x7B,0x49,0x05,0x00,0x0C,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

uint8_t CMD24_PLUS_FAT_DATA[531] = {0x58,0x00,0x00,0x31,0x84,0xFF,0xFF,0xFF,
                0xFE,0xF8,0xFF,0xFF,0x0F,0xFF,0xFF,0xFF,0xFF,
                0xFF,0xFF,0xFF,0x0F,0xFF,0xFF,0xFF,0x0F,
                0xFF,0xFF,0xFF,0x0F,0xFF,0xFF,0xFF,0x0F,
                0x07,0x00,0x00,0x00,0xFF,0xFF,0xFF,0x0F,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00};

uint8_t CMD24_PLUS_CPY_FAT_DATA[531] = {0x58,0x00,0x00,0x38,0xC2,0xFF,0xFF,0xFF,
                0xFE,0xF8,0xFF,0xFF,0x0F,0xFF,0xFF,0xFF,0xFF,
                0xFF,0xFF,0xFF,0x0F,0xFF,0xFF,0xFF,0x0F,
                0xFF,0xFF,0xFF,0x0F,0xFF,0xFF,0xFF,0x0F,
                0x07,0x00,0x00,0x00,0xFF,0xFF,0xFF,0x0F,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00};

uint8_t CMD24_PLUS_ARCHIVE_DATA[531] = {0x00};


void setup_clock(void){

             /* Configuring pins for peripheral/crystal usage and LED for output
*/

                /* HFXT*/
             MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_PJ,
                     GPIO_PIN3 | GPIO_PIN4, GPIO_PRIMARY_MODULE_FUNCTION);

             /*LFXT*/
             GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_PJ,
                         GPIO_PIN0 | GPIO_PIN1, GPIO_PRIMARY_MODULE_FUNCTION);

             /* Setting the external clock frequency LFXT, HFXT. This API is
optional, but will
              * come in handy if the user ever wants to use the
getMCLK/getACLK/etc
              * functions
              */
             CS_setExternalClockSourceFrequency(32768,48000000);
```

```
            /* Starting HFXT in non-bypass mode without a timeout. Before we
start
             * we have to change VCORE to 1 to support the 48MHz frequency */
            MAP_PCM_setCoreVoltageLevel(PCM_VCORE1);
            MAP_FlashCtl_setWaitState(FLASH_BANK0, 2);
            MAP_FlashCtl_setWaitState(FLASH_BANK1, 2);

            /* Starting HFXT and LFXT in non-bypass mode without a timeout. */
            CS_startHFXT(false);
            CS_startLFXT(false);

            /* Initializing the clock source as follows:
                *       MCLK = HFXT = 48MHz
                *       HSMCLK = HFXT = 48MHz
                *       ACLK = LFXT = 750KHz (Ideally I need 1.024MHz)
                */
            MAP_CS_initClockSignal(CS_MCLK, CS_HFXTCLK_SELECT,
CS_CLOCK_DIVIDER_1);
            MAP_CS_initClockSignal(CS_HSMCLK, CS_HFXTCLK_SELECT,
CS_CLOCK_DIVIDER_1);
            MAP_CS_initClockSignal(CS_ACLK, CS_HFXTCLK_SELECT,
CS_CLOCK_DIVIDER_64);

            /* Enabling FPU for DCO Frequency calculation */
            MAP_FPU_enableModule();

}

void Configure_Master_Interrupts(void){
               /* Enabling MASTER interrupts */
               MAP_Interrupt_enableMaster();
            SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;                // Enable sleep
on exit from ISR
            __enable_irq();
       // Enable Master Interrupts Interrupts
            NVIC->ISER[0] = 1 << ((INT_T32_INT1 - 16) & 31);// Push P1 interrupt
in NVIC module
            NVIC->ISER[0] = 1 << ((INT_TA2_0 - 16) & 31);          // Push Q1
interrupt in NVIC module
            NVIC->ISER[0] = 1 << ((INT_TA3_0 - 16) & 31);          // Push Q1'
interrupt in NVIC module
            NVIC->ISER[0] = 1 << ((INT_T32_INT2 - 16) & 31);// Push Tempo
interrupt in NVIC module
}
void Configure_GPIO(void){

               /* Input Signal  */
               P7DIR |= BIT6;
       // P7.6 set as output
               P7SEL0 |= BIT6;
       // P7.6 set TA1.1

               /* MCLK Signal  */
```

```
          P2DIR |= BIT4;                                                   //
P2.4 set as output
          P2SEL0 |= BIT4;                                                  //
P2.4 set TA0_1

          /* DRDY */
          P8DIR &= ~BIT7;
      // P8.7 set as input  to check data ready of adc

          /* ADC7766 Power Up / Down */
          P8DIR |= BIT6;                                                   //
P8.6 set as output simulate SYNC pin also put it High
            P8OUT &= ~BIT6;
      // P8.6 set Low

            /* Calibration */
            P8DIR |= BIT4;
      // P8.4 set as output auxiliar to calculate Filter Offset Pin
            P8OUT &= ~BIT4;
      // P8.4 set Low

            /* Red LED */
            P1DIR |= BIT0;
            P1OUT &= ~BIT0;
      // Red OFF

            /* Green LED */
            P2DIR |= BIT1;
            P2OUT &= ~BIT1;
      // Green OFF

            /* Blue LED */
            P2DIR |= BIT2;
            P2OUT &= ~BIT2;
      // Blue OFF

            /* SD-SPI CS PIN Controlled Manually */
            P5DIR |= BIT5;
      // 5.5 set as output

            P5DIR |= BIT2;
      // 5.5 set as output
            P5OUT &= ~BIT2;
      // Blue OFF

            /*  Configure SPI MASTER PIN: CLK, MOSI & MISO for SPI0 (EUSCI_B2)
for SD Data Dump */

            MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P3,
            GPIO_PIN5 | GPIO_PIN6, GPIO_PRIMARY_MODULE_FUNCTION);
            MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P3,
            GPIO_PIN7, GPIO_PRIMARY_MODULE_FUNCTION);

            /*  Configure SPI MASTER PIN: CLK, MOSI & MISO for SPI0 (EUSCI_B0)
for ADC  */
```

```
            MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P1,
            GPIO_PIN4 | GPIO_PIN5 | GPIO_PIN6, GPIO_PRIMARY_MODULE_FUNCTION);
            MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
            GPIO_PIN7, GPIO_PRIMARY_MODULE_FUNCTION);


}

void SelecT_InputFreq (int number1){

            switch( number1 )
            {
             case 1 : myT_Square = 9600;
             break;
             case 2 : myT_Square = 4800;
             break;
             case 3 : myT_Square = 2400;
             break;
             case 4 : myT_Square = 960;
             break;
             case 5 : myT_Square = 480;
             break;
             case 6 : myT_Square = 240;
             break;
             case 7 : myT_Square = 96;
             break;
             case 8 : myT_Square = 48;
             break;
             case 9 : myT_Square = 24;
             break;
             case 10 : myT_Square = 8;
             break;
             case 11 : myT_Square = 4;
             break;
          }
}

void Configure_Timers(void){

            /* Timer MCLK Signal PIN */
            TA0CCTL0 &= ~CCIFG;
      // Clear Interrupt Flag
            TA0CCR0 = 48-1;
      // MCLK period
            TA0CCR1 = 24-1;
      // TA0CCR1 MCLK duty cycle
            TA0CCTL1 |= OUTMOD_7;
      // TA1CCR1 output mode = reset/set

            /* Input Signal Timer TA1.2 Set-Up */
            TA1CCTL1 &= ~CCIFG;
      // Clear Interrupt Flag
            TA1CCR0 = myT_Square-1;
      // Signal Period
```

```
			TA1CCR2 = myT_Square/2;
	// TA1CCR1 Signal Duty Cycle
			TA1CCTL2 |= OUTMOD_7;
	// TA1CCR1 output mode = reset/set

			/* Q1 Sampler Timer for ADC7766 TA3.0 Set-Up */
			TA2CCTL0 &= ~CCIFG;
	// Clear Interrupt Flag
			TA2CCR0 = ((2*myT_Square)+(myT_Square/2)+(myT_Square/4))-1+302;
	//1M -240 to correct for software delay Sample period in ticks
(Tq(t)*48e6)
			//TA2CCR0 = ((10*myT_Square)+(myT_Square/4))-1;	//1M -240 to
correct for software delay Sample period in ticks (Tq(t)*48e6)
			TA2CCTL0 |= CCIE;					//
Enable Timer2_A interrupts

			/* P1' and Q1' Sampler Timer for ADC7766 TA3.0 Set-Up */
			TA3CCTL0 &= ~CCIFG;
	// Clear Interrupt Flag
			TA3CCR0 = ((2*myT_Square)+(myT_Square/2))-1+302;//10k -302 to
correct for ADC INIT (CAL FOR 10K)
			TA3CCTL0 |= CCIE;					//
Enable Timer3_A_N interrupts

			/* P Sampler Timer 32 bits 40 Hz for ADC7766 */
		MAP_Timer32_initModule(TIMER32_1_BASE, TIMER32_PRESCALER_1,
TIMER32_32BIT, TIMER32_PERIODIC_MODE);
		MAP_Timer32_setCount(TIMER32_1_BASE,1200000-1);
		MAP_Timer32_enableInterrupt(TIMER32_1_BASE);

			/* Tempo  Timer for 10 seconds Set-Up */
		MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1,
TIMER32_32BIT, TIMER32_PERIODIC_MODE);
		MAP_Timer32_setCount(TIMER32_0_BASE,480600000-1);
		MAP_Timer32_enableInterrupt(TIMER32_0_BASE);

}

void Configure_SPI(void){


			/*------------------------ ADC SPI -------------------------*/

			/* Put SPI state machine in reset */
		UCB0CTLW0 |= UCSWRST;

		/* Configure SPI */
		UCB0CTLW0 |= UCMST|UCCKPH|UCSYNC|UCMSB|UCMODE_0|UCSTEM;// Clock
polarity high, MSB, Master
		UCB0CTLW0 |= UCSSEL__SMCLK;			// SMCLK Clock Source
= 48MHz
		UCB0BRW = 0x000C;				// SPICLK = 3072Khz
// 3000 Khz

		/* Enable the SPI module */
```

```
          MAP_SPI_enableModule(EUSCI_B0_BASE);

            /*------------------------ SD SPI -------------------------*/

            /* Put SPI state machine in reset */
            UCB2CTLW0 |= UCSWRST;

            /* Configure SPI */
            UCB2CTLW0 |= UCMST|UCCKPH|UCSYNC|UCMSB|UCMODE_0;// Clock polarity 
high, MSB, 3 wire ,Master
            UCB2CTLW0 |= UCSSEL__SMCLK;                    // SMCLK Clock 
Source = 48MHz
            UCB2BRW = 0x00F0;                              // SPICLK = 400 Khz
}
void Configure_DMA(void){


            /* Configuring DMA module */
            MAP_DMA_enableModule();
            MAP_DMA_setControlBase(MSP_EXP432P401RLP_DMAControlTable);

            /* Assign DMA channel 0 to EUSCI_B0_TX0, channel 1 to EUSCI_B0_RX0 
*/
            MAP_DMA_assignChannel(DMA_CH0_EUSCIB0TX0);
            MAP_DMA_assignChannel(DMA_CH1_EUSCIB0RX0);

            /* Set-Up Channel Controls */
            MAP_DMA_setChannelControl(DMA_CH0_EUSCIB0TX0 | UDMA_PRI_SELECT,
                         UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE | 
UDMA_ARB_1);

            MAP_DMA_setChannelControl(DMA_CH1_EUSCIB0RX0 | UDMA_PRI_SELECT,
                         UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | 
UDMA_ARB_1);

            /* Set-Up Channel Transfers */
            MAP_DMA_setChannelTransfer(DMA_CH0_EUSCIB0TX0 | UDMA_PRI_SELECT,
                  UDMA_MODE_BASIC, mstxData,
           (void *) MAP_SPI_getTransmitBufferAddressForDMA(EUSCI_B0_BASE),
           MAP_SPI_MSG_LENGTH);

            MAP_DMA_setChannelTransfer(DMA_CH1_EUSCIB0RX0 | UDMA_PRI_SELECT,
            UDMA_MODE_BASIC,
           (void *) MAP_SPI_getReceiveBufferAddressForDMA(EUSCI_B0_BASE),
           msrxData,
           MAP_SPI_MSG_LENGTH);


        /*------------------------ SD DMA -------------------------*/

       /* Assign DMA channel 4 to EUSCI_B2_TX0, channel 5 to EUSCI_B2_RX0 */
       MAP_DMA_assignChannel(DMA_CH4_EUSCIB2TX0);
       MAP_DMA_assignChannel(DMA_CH5_EUSCIB2RX0);

       /* Set-Up Channel Controls SPI - SD */
```

```
        MAP_DMA_setChannelControl(DMA_CH4_EUSCIB2TX0 | UDMA_PRI_SELECT,
        UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE | UDMA_ARB_1);

        MAP_DMA_setChannelControl(DMA_CH5_EUSCIB2RX0 | UDMA_PRI_SELECT,
        UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | UDMA_ARB_1);

        /* Set-Up Channel Transfers SPI - SD */
        MAP_DMA_setChannelTransfer(DMA_CH4_EUSCIB2TX0 | UDMA_PRI_SELECT,
                    UDMA_MODE_BASIC, sdtx_sincro_Data,
                (void *) MAP_SPI_getTransmitBufferAddressForDMA(EUSCI_B2_BASE),
                            MAP_SD_SPI_SINC_LENGTH);

         MAP_DMA_setChannelTransfer(DMA_CH5_EUSCIB2RX0 | UDMA_PRI_SELECT,
                    UDMA_MODE_BASIC,
                (void *) MAP_SPI_getReceiveBufferAddressForDMA(EUSCI_B2_BASE),
                            sdrxData,
                            MAP_SD_SPI_MSG_LENGTH);



         /*------------------------ ADC DMA Interrupt -------------------------*/

         /* Enable DMA interrupt SPI + ADC */
         MAP_DMA_assignInterrupt(INT_DMA_INT1, 1);
         MAP_DMA_clearInterruptFlag(DMA_CH1_EUSCIB0RX0 & 0x0F);

         /* Enable DMA interrupt SPI - SD */
         DMA_assignInterrupt(INT_DMA_INT2, 5);
         MAP_DMA_clearInterruptFlag(DMA_CH5_EUSCIB2RX0 & 0x0F);

         /* Assigning/Enabling Interrupts */
         MAP_Interrupt_enableInterrupt(INT_DMA_INT1);
         MAP_DMA_enableInterrupt(INT_DMA_INT1);

         /* Assigning/Enabling Interrupts SPI - SD */
         MAP_Interrupt_enableInterrupt(INT_DMA_INT2);
         MAP_DMA_enableInterrupt(INT_DMA_INT2);

}

void InitADC_OA(void){

             /* Operational Amplifiers ON */
             P8OUT ^= BIT4;
        // PUT P8.4 HIGH
             /* Input Signal Timer */
             TA1CTL |= TASSEL__SMCLK | MC__UP | TACLR;           // SMCLK, Up
Mode (Counts to TA1CCR0), Clear TAR1, Start Counting from here
             for (pru=0;pru<9600000;pru++){}
}

void Timers_START(void){

             /* Green Led Indicates that Sample Process Start */
             P2OUT ^= BIT1;
```

```
		while (TA1R < 3300){};

		/* Tempo Timer */
	    Timer32_startTimer(TIMER32_0_BASE,true);

		/*P Sample Timer */
	    Timer32_startTimer(TIMER32_1_BASE,false);
	    //PCM_gotoLPM0InterruptSafe();                              // Calls
WFI enter Low Power Mode LPM0 (Clocks ON | Peripherials ON | CPU OFF)

}

void T32_INT2_IRQHandler(void){

		/* P Sampler */
		/* Start MCLK timer at 800 Khz */
		TA0CTL |= TASSEL__SMCLK | MC__UP | TACLR;            // SMCLK, Up
Mode (Counts to TA0CCR0), Clear TAR0, Start Counting from here
		/* ADC7766 ON */
		P8OUT ^= BIT6;                                            // PUT
P8.6 HIGH
		P8OUT &= ~BIT6;                                           // PUT
P8.6 LOW
		while ((P8IN & BIT7)==0){};
		P8OUT ^= BIT6;                                            // PUT
P8.6 HIGH
		/* Wait until DRDY = 0 so ADC is stable */
		while ((P8IN & BIT7)!=0){};
		/* P Sampler UP */
		TA3CTL |= TASSEL__SMCLK | MC__UP | TACLR;            // SMCLK, Up
Mode (Counts to TA2CCR0), Clear TAR2, Start Counting from here
		while ((P8IN & BIT7)==0){};
		TA0CTL = MC__STOP;
		/*Take 1 CHECK SAMPLE*/
	    DMA_Control->ENASET = 1 << (1 & 0x0F);
		DMA_Control->ENASET = 1 << (0 & 0x0F);
		tarp[mitarp]= TA1R;
		mstxData[0] = 0xFF;
		mstxData[1] = 0xFF;
		mstxData[2] = 0xFF;
		P8OUT &= ~BIT6;                                           // PUT
P8.6 LOW
		TIMER32_CMSIS(TIMER32_1_BASE)->INTCLR |= 0x01;
		//Timer32_haltTimer(TIMER32_1_BASE);                      // Stop
mode
		mitarp++;
		p_q_p=0;
}


void  TA3_0_IRQHandler(void){

		/* P' and Q' Sampler */
		/* Start MCLK timer at 800 Khz */
```

```
            TA0CTL |= TASSEL__SMCLK | MC__UP | TACLR;          // SMCLK, Up
Mode (Counts to TA0CCR0), Clear TAR0, Start Counting from here
            /* ADC7766 ON */
            P8OUT ^= BIT6;                                             // PUT
P8.6 HIGH
            P8OUT &= ~BIT6;                                            // PUT
P8.6 LOW
            while ((P8IN & BIT7)==0){};
            P8OUT ^= BIT6;                                             // PUT
P8.6 HIGH
            /* Wait until DRDY = 0 so ADC is stable */
            while ((P8IN & BIT7)!=0){};
            /* Q Sampler UP */
            TA2CTL |= TASSEL__SMCLK | MC__UP | TACLR;          // SMCLK, Up
Mode (Counts to TA2CCR0), Clear TAR2, Start Counting from here
            while ((P8IN & BIT7)==0){};
            TA0CTL = MC__STOP;
            /*Take 1 CHECK SAMPLE*/
            DMA_Control->ENASET = 1 << (1 & 0x0F);
            DMA_Control->ENASET = 1 << (0 & 0x0F);
            tarp[mitarp]= TA1R;
            mstxData[0] = 't';
            mstxData[1] = 'a';
            mstxData[2] = 'e';
            P8OUT &= ~BIT6;
            TA3CTL = MC__STOP;
      // Stop mode
            TA3CCTL0 &= ~CCIFG;
            if(p_q_p==1){
            TA2CTL = MC__STOP;
      // Stop mode
            TA2CCTL0 &= ~CCIFG;
            }
            mitarp++;
}

void  TA2_0_IRQHandler(void){

            /* Q Sampler */
            /* Start MCLK timer at 800 Khz */
            TA0CTL |= TASSEL__SMCLK | MC__UP | TACLR;          // SMCLK, Up
Mode (Counts to TA0CCR0), Clear TAR0, Start Counting from here
            /* ADC7766 ON */
            P8OUT ^= BIT6;                                             // PUT
P8.6 HIGH
            P8OUT &= ~BIT6;                                            // PUT
P8.6 LOW
            while ((P8IN & BIT7)==0){};
            P8OUT ^= BIT6;                                             // PUT
P8.6 HIGH
            /* Wait until DRDY = 0 so ADC is stable */
            while ((P8IN & BIT7)!=0){};
            /* P Sampler UP */
            TA3CTL |= TASSEL__SMCLK | MC__UP | TACLR;          // SMCLK, Up
Mode (Counts to TA2CCR0), Clear TAR2, Start Counting from here
```

```
            while ((P8IN & BIT7)==0){};
            TA0CTL = MC__STOP;
            /*Take 1 CHECK SAMPLE*/
            DMA_Control->ENASET = 1 << (1 & 0x0F);
            DMA_Control->ENASET = 1 << (0 & 0x0F);
            tarp[mitarp]= TA1R;
            mstxData[0] = 'i';
            mstxData[1] = 'k';
            mstxData[2] = 'u';
            P8OUT &= ~BIT6;
            TA2CTL = MC__STOP;
      // Stop mode
            TA2CCTL0 &= ~CCIFG;
            mitarp++;
            p_q_p=1;
}

void T32_INT1_IRQHandler(void)
{
            /* Stop P Sampler*/
            Timer32_haltTimer(TIMER32_1_BASE);                         // Stop
mode

            /* Stop Q Sampler*/
            TA2CTL = MC__STOP;
      // Stop mode
            TA2CTL |= TACLR;
      // Clear TAR2

            /* Stop P' and Q' Sampler*/
            TA3CTL = MC__STOP;
      // Stop mode
            TA3CTL |= TACLR;
      // Clear TAR3

            /* Disable SPI */
            SPI_disableModule(EUSCI_B0_BASE);

            /* Disable DMA */
            MAP_DMA_disableChannel(1);
            MAP_DMA_disableChannel(0);

            /* Input Signal Timer */
            TA1CTL = MC__STOP;
      // Stop mode
            TA1CTL |= TACLR;
      // Clear TAR1

            /* Tempo Timer */
            Timer32_haltTimer(TIMER32_0_BASE);                         // Stop
mode

            /* Clear Interrupt Flag from all timers */
            Timer32_clearInterruptFlag(TIMER32_0_BASE);
            Timer32_clearInterruptFlag(TIMER32_1_BASE);
```

```
            TA0CCTL0 &= ~CCIFG;
            TA1CCTL0 &= ~CCIFG;
            TA2CCTL0 &= ~CCIFG;
            TA3CCTL0 &= ~CCIFG;

            /* Shut Down OAs */
            P8OUT &= ~BIT4;
       // P8.4 set Low
            /* Shut Down ADC */
            P8OUT &= ~BIT6;
       // P8.6 set Low

            /* Shut Down Green Led as Sampling Process is over */
            P2OUT &= ~BIT1;

            /* Blue Led Indicates that DATA transfer to SD card initiated */
            P2OUT ^= BIT2;
            P5OUT &= ~BIT2;
       // Blue OFF

            Data_Dump_SD();
            //Data_Dump_PC();
}

void Data_Dump_SD(void){


            for(ie=48;ie<521;ie++){
            CMD24_PLUS_FAT_DATA[ie] = 0x00;
            CMD24_PLUS_CPY_FAT_DATA[ie] = 0x00;
            };

            for(ie=144;ie<521;ie++){
            CMD24_PLUS_ROOT_DATA[ie] = 0x00;
            };

            for(ie=279;ie<521;ie++){
            CMD24_PLUS_CPY_ROOT_DATA[ie] = 0x00;
            };

            for(ie=521;ie<531;ie++){
            CMD24_PLUS_ROOT_DATA[ie] = 0xFF;
            CMD24_PLUS_FAT_DATA[ie] = 0xFF;
            CMD24_PLUS_CPY_ROOT_DATA[ie] = 0xFF;
            CMD24_PLUS_CPY_FAT_DATA[ie] = 0xFF;
            };

            for(ie=522;ie<531;ie++){
            CMD24_PLUS_ARCHIVE_DATA[ie] = 0xFF;
            CMD24_PLUS_ARCHIVE_DATA[ie] = 0xFF;
            };

          xc = 0;
          for(ie=0;ie<4800;ie++) {
```

```
for (e=7; e>=0; e--){

        switch (e){

        case 7:
                if ((bufferedData1[ie] & 0x80) == 0x80){
                    mychar_bit[xc] = 0x31;
                }
                else{
            mychar_bit[xc] = 0x30;
                }
                break;
        case 6:
                if ((bufferedData1[ie] & 0x40) == 0x40){
                    mychar_bit[xc] = 0x31;
                }
                else{
            mychar_bit[xc] = 0x30;
                }
                break;
        case 5:
                if ((bufferedData1[ie] & 0x20) == 0x20){
                    mychar_bit[xc] = 0x31;
                }
                else{
            mychar_bit[xc] = 0x30;
                }
                break;
        case 4:
                if ((bufferedData1[ie] & 0x10) == 0x10){
                    mychar_bit[xc] = 0x31;
                }
                else{
            mychar_bit[xc] = 0x30;
                }
                break;
        case 3:
                if ((bufferedData1[ie] & 0x08) == 0x08){
                    mychar_bit[xc] = 0x31;
                }
                else{
            mychar_bit[xc] = 0x30;
                }
                break;
        case 2:
                if ((bufferedData1[ie] & 0x04) == 0x04){
                    mychar_bit[xc] = 0x31;
                }
                else{
            mychar_bit[xc] = 0x30;
                }
                break;
        case 1:
                if ((bufferedData1[ie] & 0x02) == 0x02){
```

```
                                        mychar_bit[xc] = 0x31;
                                    }
                                    else{
                                mychar_bit[xc] = 0x30;
                                    }
                                    break;
                            case 0:
                                    if ((bufferedData1[ie] & 0x01) == 0x01){
                                        mychar_bit[xc] = 0x31;
                                    }
                                    else{
                                mychar_bit[xc] = 0x30;
                                    }
                                    break;
                            }
                            xc++;
                    }
                    if ((ie+1)%3==0){
                            mychar_bit[xc] = 0x0D;
                            xc++;
                            mychar_bit[xc] = 0x0A;
                            xc++;
                    }
            }

        /* Start SD Process */
        SD_init();
        /* Blue Led OFF Data transfer is over */
        P2OUT &= ~BIT2;
        /* Call Disable all function? */
}


void SD_init(void){

            /* Initialization of SD_SPI Protocol */

            /* Red Led Indicates that INIT Started */
            P1OUT ^= BIT0;

            /* Enable SD_SPI Module */
            MAP_SPI_enableModule(EUSCI_B2_BASE);

            /* CS HIGH State */
            cs_disable();


            /* Send Dummy Message to generate 400 KHz Clock for 74 clock
Periods */
            send_sincro_comand(Dummy1);

            /* Wait 420 us to let SD stabilize */
            for (w_sd=0;w_sd<20000;w_sd++){}

            /* Wait 210 us to let SD stabilize */
```

```
            for (w_sd=0;w_sd<10000;w_sd++){}

            /* CS LOW State to make SD enter SPI Mode */
            cs_enable();

            /* Send CMD0 */
            send_command_sd(CMD0);

            /* Read Buffered RX Response from SD if not 0x01 Keep sending CMD
and Keep Reading */
            while(!check_response(3)){
            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}
            /* if not detected send again and read again */
            send_command_sd(CMD0);

            }

            /* CS HIGH State */
            cs_disable();

            /* Wait 210 us to let SD stabilize */
            for (w_sd=0;w_sd<10000;w_sd++){}


            /* CS LOW State to make SD enter SPI Mode */
            cs_enable();

            /* Send CMD0 */
            send_command_sd(CMD8);

            /* Read Buffered RX Response from SD if not 0x01 Keep sending CMD
and Keep Reading */
            while(!check_response(3)){
            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}
            /* if not detected send again and read again */
            send_command_sd(CMD8);

            }

            /* CS HIGH State */
            cs_disable();

            /* Wait 210 us to let SD stabilize */
            for (w_sd=0;w_sd<10000;w_sd++){}

            /* CS LOW State 2nd Transmission */
            cs_enable();


            /* Send CMD55 */
            send_command_sd(CMD55);
            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}
```

```
            /* Send ACMD41 */
            send_command_sd(ACMD41);

            /* Read Buffered RX Response from SD if not 0x01 Keep sending Dummy
and Keep Reading */
            while(!check_response(0)){

            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}
            /* Send CMD55 */
            send_command_sd(CMD55);
            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}
            /* Send ACMD41 */
            send_command_sd(ACMD41);
            }

            /* CS HIGH State */
            cs_disable();


            /* Disable SD_SPI Module */
            MAP_SPI_disableModule(EUSCI_B2_BASE);

            /* Configure SD_SPI Module at Max Speed 1 MHz */
            UCB2BRW = 0x0030;                             // SPICLK = 1 MHz

            /* Enable SD_SPI Module */
            MAP_SPI_enableModule(EUSCI_B2_BASE);

            /* INIT END */
            P1OUT &= ~BIT0;

            /* SD in SPI mode, at 1 MHz Speed, Ready for Data Operations */
            sd_data_dump();
}


void sd_data_dump(void){

            /* CS LOW State CMD16 Transmission */
            cs_enable();

            /* Send CMD16 Set Block Length 512 Bytes */
            send_command_sd(CMD16);

            /* Read Buffered RX Response from SD if not 0x00 Keep sending Dummy
and Keep Reading */
            while(!check_response(0)){
            /* if not detected send again and read again */
            send_command_sd(CMD16);
            }

            /* CS HIGH State */
            cs_disable();
```

```
            /* Wait to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_esp_command_sd(CMD24_PLUS_FAT_DATA);

            /* Wait to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}

            /* CS HIGH State */
            cs_disable();

            /* Read Buffered RX Response from SD if not 0x00 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(0)){}

            /* Read Buffered RX Response from SD if not 0xE5 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(1)){}

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_command_sd(Dummy);

            /* Read Buffered RX Response from SD if not 0xFF (Idle) the SD is
still busy Keep sending Dummy and Keep Reading */
            while(!check_response(2)){
            send_command_sd(Dummy);
            }

            /* CS HIGH State */
            cs_disable();

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_esp_command_sd(CMD24_PLUS_CPY_FAT_DATA);

            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}

            /* CS HIGH State */
            cs_disable();

            /* Read Buffered RX Response from SD if not 0x00 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(0)){}
```

```
            /* Read Buffered RX Response from SD if not 0xE5 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(1)){}

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_command_sd(Dummy);

            /* Read Buffered RX Response from SD if not 0xFF (Idle) the SD is
still busy Keep sending Dummy and Keep Reading */
            while(!check_response(2)){
            send_command_sd(Dummy);
            }

            /* CS HIGH State */
            cs_disable();

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            /* CS LOW State Data Token Transmission */
            cs_enable();

            send_esp_command_sd(CMD24_PLUS_ROOT_DATA);

            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}

            /* CS HIGH State */
            cs_disable();

            /* Read Buffered RX Response from SD if not 0x00 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(0)){}

            /* Read Buffered RX Response from SD if not 0xE5 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(1)){}

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_command_sd(Dummy);

            /* Read Buffered RX Response from SD if not 0xFF (Idle) the SD is
still busy Keep sending Dummy and Keep Reading */
            while(!check_response(2)){
            send_command_sd(Dummy);
            }
```

```
            /* CS HIGH State */
            cs_disable();

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_esp_command_sd(CMD24_PLUS_CPY_ROOT_DATA);

            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}

            /* CS HIGH State */
            cs_disable();

            /* Read Buffered RX Response from SD if not 0x00 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(0)){}

            /* Read Buffered RX Response from SD if not 0xE5 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(1)){}

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_command_sd(Dummy);

            /* Read Buffered RX Response from SD if not 0xFF (Idle) the SD is
still busy Keep sending Dummy and Keep Reading */
            while(!check_response(2)){
            send_command_sd(Dummy);
            }

            /* CS HIGH State */
            cs_disable();

            /* Wait 1 ms to let SD stabilize */
            for (w_sd=0;w_sd<48000;w_sd++){}

            for (sd_data_loop=0;sd_data_loop<82;sd_data_loop++)
            {
            cmd_plus_data_creation(sd_data_loop);

            /* CS LOW State Data Token Transmission */
            cs_enable();

            send_esp_command_sd(CMD24_PLUS_ARCHIVE_DATA);

            /* Wait 100 ms to let SD stabilize */
            for (w_sd=0;w_sd<400000;w_sd++){}
```

```
            /* CS HIGH State */
            cs_disable();

            /* Read Buffered RX Response from SD if not 0x00 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(0)){}

            /* Read Buffered RX Response from SD if not 0xE5 Keep Repeating
CMD24 + Data sequence and Keep Reading */
            while(!check_data_response(1)){}

            /* Wait 100 us to let SD stabilize */
            for (w_sd=0;w_sd<4800;w_sd++){}

            cs_enable();

            send_command_sd(Dummy);

            /* Read Buffered RX Response from SD if not 0xFF (Idle) the SD is
still busy Keep sending Dummy and Keep Reading */
            while(!check_response(2)){
                  send_command_sd(Dummy);
            }

            /* CS HIGH State */
            cs_disable();

            for (w_sd=0;w_sd<400000;w_sd++){}

            }

            // DATA DUMP SUCCESSFULL STOP ALL.
}

void error(void){

            static uint32_t k=0;
            P1DIR |= BIT0;
            while (1)
            {
        P1OUT ^= BIT0;
        for(k=0;k<20000;k++);                                    // Blink LED
forever
            }

}


void DMA_INT1_IRQHandler(void){

            /* ADC DMA */

        /* Clear interrupt flags */
       DMA_Channel->INT0_CLRFLG |= (1 << 1);
        DMA_Channel->INT0_CLRFLG |= (1 << 0);
```

```
        /* Disable Channels so they can be set up again */
        DMA_Control->ENACLR = 1 << (1 & 0x0F);
        DMA_Control->ENACLR = 1 << (0 & 0x0F);

        /* Store Data */
        bufferedData1[sampleX] = msrxData[0];
       bufferedData1[sampleX+1] = msrxData[1];
       bufferedData1[sampleX+2] = msrxData[2];

       sampleX++;
       sampleX++;
       sampleX++;

        /* Set Channel DMA Transfer (Configure it with DMA->r.CHannelTransfer =
todos los bits = menos el mode) */
       MAP_DMA_setChannelTransfer(DMA_CH0_EUSCIB0TX0 | UDMA_PRI_SELECT,
                                   UDMA_MODE_BASIC, mstxData,
                           (void *)
MAP_SPI_getTransmitBufferAddressForDMA(EUSCI_B0_BASE),
                           MAP_SPI_MSG_LENGTH);

       MAP_DMA_setChannelTransfer(DMA_CH1_EUSCIB0RX0 | UDMA_PRI_SELECT,
                            UDMA_MODE_BASIC,
                          (void  *)  MAP_SPI_getReceiveBufferAddressForDMA(EUSCI_B0_BASE),
                           msrxData,
                           MAP_SPI_MSG_LENGTH);

        //   DMA_Control->ENASET = 1 << (1 & 0x0F);
        //   DMA_Control->ENASET = 1 << (0 & 0x0F);
}

void DMA_INT2_IRQHandler(void){

              /* SD DMA */
              /* Check if the basic "A" transfer is complete */
       umode = MAP_DMA_getChannelMode( DMA_CH4_EUSCIB2TX0 | UDMA_PRI_SELECT );
       umode1 = MAP_DMA_getChannelMode( DMA_CH5_EUSCIB2RX0 | UDMA_PRI_SELECT );

       if(umode == UDMA_MODE_STOP && umode1 == UDMA_MODE_STOP)
       {
        /* Clear interrupt flags */
        MAP_DMA_clearInterruptFlag(5);
        MAP_DMA_clearInterruptFlag(4);

        /* Disable Channels so they can be set up again */
        MAP_DMA_disableChannel(5);
        MAP_DMA_disableChannel(4);

        /* Set Channel DMA Transfer (Configure it with DMA->r.CHannelTransfer =
todos los bits = menos el mode) */
       MAP_DMA_setChannelTransfer(DMA_CH4_EUSCIB2TX0 | UDMA_PRI_SELECT,
                                   UDMA_MODE_BASIC, sdtxData,
                           (void *)
MAP_SPI_getTransmitBufferAddressForDMA(EUSCI_B2_BASE),
```

```
                              MAP_SD_SPI_MSG_LENGTH);

      MAP_DMA_setChannelTransfer(DMA_CH5_EUSCIB2RX0 | UDMA_PRI_SELECT,
                   UDMA_MODE_BASIC,
                 (void *) MAP_SPI_getReceiveBufferAddressForDMA(EUSCI_B2_BASE),
                  sdrxData,
                              MAP_SD_SPI_MSG_LENGTH);
      }
}

/*-------------------------------- AUXILIAR SD FUNCTIONS -----------------------
----------------------------*/

void cs_enable(void){

            /* Put CS pin manually LOW */
            P5OUT &= ~BIT5;
      // P5.5 set LOW
}
void cs_disable(void){

            /* Put CS pin manually HIGH */
            P5OUT ^= BIT5;
      // P5.5 set HIGH
}
void send_sincro_comand(uint8_t sincro_data[10]){

            /* Charge Command to send into TX Array 512 Bytes */
            for (B_l=0;B_l<10;B_l++){
                  sdtx_sincro_Data[10] = sincro_data[10];
            }
            UCB2IFG = 0x0002;
            MAP_DMA_enableChannel(5);
            MAP_DMA_enableChannel(4);
}
void send_command_sd(uint8_t CMD[16]){

            /* Charge Command to send into TX Array 8 Bytes */
            for (B_l=0;B_l<16;B_l++){
                  sdtxData[B_l] = CMD[B_l];
            }
            clean_DMA();
            configure_DMA_TX(0);
            UCB2IFG = 0x0002;
            MAP_DMA_enableChannel(5);
            MAP_DMA_enableChannel(4);
}
void send_esp_command_sd(uint8_t CMD[531]){

            /* Charge Command to send into TX Array 8 Bytes */
            for (B_l=0;B_l<531;B_l++){
            sdtx_TOKEN_Data[B_l] = CMD[B_l];
            }
            clean_data_DMA();
            configure_DMA_TX(1);
```

```
            UCB2IFG = 0x0002;
            MAP_DMA_enableChannel(5);
            MAP_DMA_enableChannel(4);
}
void cmd_plus_data_creation(uint32_t lopos){

            pos_dec = (lopos)*512 + 8519680;

            pos_dec = pos_dec / 512;
            /* Convert decimal to Hex */
            dec_hex(pos_dec);
            /* Get Hex in parts to store it in pos_hex[0,1,2,3] */

            CMD24_PLUS_ARCHIVE_DATA[0] = 0x58;
            CMD24_PLUS_ARCHIVE_DATA[1] = pos_hex[0];
            CMD24_PLUS_ARCHIVE_DATA[2] = pos_hex[1];
            CMD24_PLUS_ARCHIVE_DATA[3] = pos_hex[2];
            CMD24_PLUS_ARCHIVE_DATA[4] = pos_hex[3];
            CMD24_PLUS_ARCHIVE_DATA[5] = 0xFF;
            CMD24_PLUS_ARCHIVE_DATA[6] = 0xFF;
            CMD24_PLUS_ARCHIVE_DATA[7] = 0xFF;
            CMD24_PLUS_ARCHIVE_DATA[8] = 0xFE;
            CMD24_PLUS_ARCHIVE_DATA[521]=0xFF;
            CMD24_PLUS_ARCHIVE_DATA[522]=0xFF;

            for(data_read=9; data_read<521; data_read++){
                  CMD24_PLUS_ARCHIVE_DATA[data_read] = mychar_bit[(data_read-
9)+(lopos*512)];
            }

}
void dec_hex(uint32_t pos){

    pos_hex[0] = pos >> 24;
    pos_hex[1] = pos >> 16;
    pos_hex[2] = pos >>  8;
    pos_hex[3] = pos;

}
void clean_DMA(void){

        /* Clear interrupt flags */
        MAP_DMA_clearInterruptFlag(5);
        MAP_DMA_clearInterruptFlag(4);

        /* Disable Channels so they can be set up again */
        MAP_DMA_disableChannel(5);
        MAP_DMA_disableChannel(4);

        for (w_sd=0;w_sd<16;w_sd++){
        sdrxData[w_sd] == 0x00;
        }

}
void clean_data_DMA(void){
```

```
        /* Clear interrupt flags */
        MAP_DMA_clearInterruptFlag(5);
        MAP_DMA_clearInterruptFlag(4);

        /* Disable Channels so they can be set up again */
        MAP_DMA_disableChannel(5);
        MAP_DMA_disableChannel(4);

        for (w_sd=0;w_sd<531;w_sd++){
             sdrx_TOKEN_Data[w_sd] == 0x00;
        }

}
void configure_DMA_TX(int def){

             if (def == 0){
        /* Set Channel DMA Transfer now the length of the TX channel data will
be 8 Bytes*/
       MAP_DMA_setChannelTransfer(DMA_CH4_EUSCIB2TX0 | UDMA_PRI_SELECT,
                           UDMA_MODE_BASIC, sdtxData,
                   (void *)
MAP_SPI_getTransmitBufferAddressForDMA(EUSCI_B2_BASE),
                                 MAP_SD_SPI_MSG_LENGTH);

       MAP_DMA_setChannelTransfer(DMA_CH5_EUSCIB2RX0 | UDMA_PRI_SELECT,
                    UDMA_MODE_BASIC,
                  (void  *)  MAP_SPI_getReceiveBufferAddressForDMA(EUSCI_B2_BASE),
                   sdrxData,
                                 MAP_SD_SPI_MSG_LENGTH);
             }
             else{
        /* Set Channel DMA Transfer now the length of the TX channel data will
be 512 Bytes*/
       MAP_DMA_setChannelTransfer(DMA_CH4_EUSCIB2TX0 | UDMA_PRI_SELECT,
                           UDMA_MODE_BASIC, sdtx_TOKEN_Data,
                   (void *)
MAP_SPI_getTransmitBufferAddressForDMA(EUSCI_B2_BASE),
                                 MAP_SD_SPI_TOKEN_LENGTH);

       MAP_DMA_setChannelTransfer(DMA_CH5_EUSCIB2RX0 | UDMA_PRI_SELECT,
                    UDMA_MODE_BASIC,
                  (void  *)  MAP_SPI_getReceiveBufferAddressForDMA(EUSCI_B2_BASE),
                                 sdrx_TOKEN_Data,
                                 MAP_SD_SPI_RXT_LENGTH);
             }

}
int check_response (int type){

             if (type == 0){
                    for (w_sd=0;w_sd<16;w_sd++){
                          if(sdrxData[w_sd] == 0x00){
                          w_sd = 16;
                          return true;
```

```
                        }
                }
                return false;
        }
        else if (type == 1){
        for (w_sd=0;w_sd<16;w_sd++){
                if(sdrxData[w_sd] == 0xE5){
                w_sd = 16;
                return true;
                }
        }
        return false;
        }
        else if (type == 3){
                for (w_sd=0;w_sd<16;w_sd++){
                        if(sdrxData[w_sd] == 0x01){
                        w_sd = 16;
                        return true;
                        }
                }
                return false;
        }
        else{
        for (w_sd=0;w_sd<16;w_sd++){
                if(sdrxData[w_sd] == 0xFF){
                w_sd = 16;
                return true;
                }
        }
        return false;
        }
}
int check_data_response (int type){

        if (type == 0){
                for (w_sd=0;w_sd<531;w_sd++){
                        if(sdrx_TOKEN_Data[w_sd] == 0x00){
                        w_sd = 531;
                        return true;
                        }
                }
                return false;
        }
        else if (type == 1){
        for (w_sd=0;w_sd<531;w_sd++){
                if(sdrx_TOKEN_Data[w_sd] == 0xE5){
                w_sd = 531;
                return true;
                }
        }
        return false;
        }
        else if (type == 3){
                for (w_sd=0;w_sd<531;w_sd++){
                        if(sdrx_TOKEN_Data[w_sd] == 0x01){
```

```
                w_sd = 531;
                return true;
                }
            }
            return false;
        }
        else{
        for (w_sd=0;w_sd<531;w_sd++){
            if(sdrx_TOKEN_Data[w_sd] == 0xFF){
            w_sd = 531;
            return true;
            }
        }
        return false;
        }
}
```

Annex B: Matlab Code

```
% Create Auxiliar A Vector
A = cell(1600,1);
PQ1 = zeros(400,2);
PQ2 = zeros(400,2);
PQ = zeros(400,2);
P = zeros(400,1);
Q = zeros(400,1);
V = zeros(400,1);
Vnorm = zeros(400,1);
Z = zeros(400,1);
Zp = zeros(400,1);
Zq = zeros(400,1);
R = zeros(400,1);
C = zeros(400,1);
Znorm = zeros(400,1);
Alf = zeros(400,1);
K = zeros(400,1);
fe = zeros(201,1);


%Create Time Vector
x = zeros(400,1);


% Create Auxiliar Variables
temp =0;
upper_part = 0;
voltage=0;
v1=1;
v2=1;
v3=1;
v4=1;
aa=0;

% Calibrated Values
AlfCal = -0.397916437122059;
KCal = 0.002953152882812;

%Create Mean values
Vmean=0;
meanZ=0;
meanR=0;
meanC=0;

% Create Time vector
for i=1:400
x(i,1)= 0.025*(i-1);
end
```

```
fileID = fopen('D:\josemi\archivos\UPC\5B\Proyecte Final de
Carrera\Blocs\Bloc 8 - Calibration\Data 10K 98_655\DATA (9).txt','r');

% Define format String
formatSpec = '%s';
% Scan the File Line by Line and store the values in vector A
for i=1:1600
mychar = fgetl(fileID);
A{i,1} = cellstr(mychar);
BA = char(A{i,1});

    %Read First Bit Value and start processing
    if BA(1) == '0'
        %Process normal
        for e=2:24
            if BA(e) == '1'
            voltage = voltage + ((2^(23-(e-1)))*298.0232239*10^(-9));
            end
        end
    else BA(1) == '1'
        % 1st bit is a 1 so Process oposite
        for e=2:24
            if BA(e) == '0'
            voltage = voltage + ((2^(23-(e-1)))*298.0232239*10^(-9));
            end
        end
        voltage = 0-voltage;
    end

    switch(aa)
        case 0
            PQ1(v1,1)=voltage;
            v1=v1+1;
            aa=aa+1;
        case 1
            PQ2(v2,1)=voltage;
            v2=v2+1;
            aa=aa+1;
        case 2
            PQ1(v3,2)=voltage;
            v3=v3+1;
            aa=aa+1;
        case 3
            PQ2(v4,2)=voltage;
            v4=v4+1;
            aa=0;
    end
    voltage = 0;
end


% Create PQ(400,2) Vector without offset error
for i=1:400
PQ(i,1) = (PQ1(i,1)-PQ2(i,1))/2;
PQ(i,2) = (PQ1(i,2)-PQ2(i,2))/2;
end
```

```
for i=1:400
P(i,1) = PQ(i,1);
Q(i,1) = PQ(i,2);
end

% Calculate Module of V and Z from PQ vector pairs and store it in vector
for u=1:400
   V(u,1) = (sqrt((PQ(u,1))^2+(PQ(u,2))^2));
   K = V(u,1) / 44.993375369191;
   Alf(u,1) = atan((Q(u,1))/(P(u,1)));
end

% Calculate mean of K vector
Kmean= mean(K);
% Calculate mean of alpha vector
Alfmean = mean(Alf);
Alfus = Alfmean + 1,0972245569318724;

for u=1:400
%     % Calculate Module of Z from PQ vector pairs and store it in Z
vector
     Z(u,1)= (sqrt((P(u,1))^2+(Q(u,1))^2))/(KCal);
%     % Calculate in-Phase vector of Z from PQ vector pairs and store it
in Phase vector
     Zp(u,1)= Z(u,1)*cos(Alf(u,1)-AlfCal);
%     % Calculate Quadrature vector of Z from PQ vector pairs and store
it in Phase vector
     Zq(u,1)= Z(u,1)*sin(Alf(u,1)-AlfCal);
%     % Calculate Quadrature vector of Z from PQ vector pairs and store
it in Phase vector
     R(u,1) = (1+tan(Alf(u,1)-AlfCal)*tan(Alf(u,1)-AlfCal))*Zp(u,1);
     C(u,1) = -(tan(Alf(u,1)-AlfCal))/(2*pi*10000*R(u,1));
end

Vmean = mean(V);
meanZ = mean(Z);
meanR = mean(R);
meanC = mean(C);

for u=1:400
    Vnorm(u,1) = V(u,1) - Vmean;
    Znorm(u,1) = Z(u,1) - meanZ;
end

% Close File
fclose(fileID);

% Store Module of Z vector in a text file
fileID2 = fopen('D:\josemi\archivos\UPC\5B\Proyecte Final de
Carrera\Blocs\Bloc 8 - Calibration\Data 10K 98_655\10K_Z.txt','w');
fileID3 = fopen('D:\josemi\archivos\UPC\5B\Proyecte Final de
Carrera\Blocs\Bloc 8 - Calibration\Data 10K 98_655\10K_P.txt','w');
fileID4 = fopen('D:\josemi\archivos\UPC\5B\Proyecte Final de
Carrera\Blocs\Bloc 8 - Calibration\Data 10K 98_655\10K_Q.txt','w');
```

```
fileID5 = fopen('D:\josemi\archivos\UPC\5B\Proyecte Final de
Carrera\Blocs\Bloc 8 - Calibration\Data 10K 98_655\10K_V.txt','w');

for u=1:400
% Store Module of V vector in a text file 2
fprintf(fileID5,'%.10f\n',V(u,1));
% Store Module of Z vector in a text file 2
fprintf(fileID2,'%.10f\n',Z(u,1));
% Store in-Phase of Z vector in a text file 3
fprintf(fileID3,'%.10f\n',P(u,1));
% Store Quadrature of Z vector in a text file 4
fprintf(fileID4,'%.10f\n',Q(u,1));
end

%Close Files
fclose(fileID2);
fclose(fileID3);
fclose(fileID4);
fclose(fileID5);


figure(1);
plot(x,Vjiji);
title('Voltage Module')
xlabel('Sec')
ylabel('V')
figure(2);
plot(x,P);
title('Phase Voltage')
xlabel('Sec')
ylabel('V')
figure(3);
plot(x,Q);
title('Quadrature Voltage')
xlabel('Sec')
ylabel('V')
figure(4);
plot(x,Z);
title('Impedance Module')
xlabel('Sec')
ylabel('Ohms')
figure(5);
plot(x,Zp);
title('Impedance Real')
xlabel('Sec')
ylabel('Ohms')
figure(6);
plot(x,Zq);
title('Impedance Imaginary')
xlabel('Sec')
ylabel('Ohms')
figure(7);
plot(x,R);
title('Paralel Resistance Equivalent')
xlabel('Sec')
ylabel('Ohms')
```

```
figure(8);
plot(x,C);
title('Paralel Capacitance Equivalent')
xlabel('Sec')
ylabel('Farads')
figure(9);
plot(x,Vnorm);
title('Normalised Voltage Module')
xlabel('Sec')
ylabel('V')
figure(10);
plot(x,Znorm);
title('Normalised Impedance Module')
xlabel('Sec')
ylabel('V')


%Fourier Transform
Fs = 40;            % Sampling frequency
T = 1/Fs;           % Sampling period
L = 400;            % Length of signal
t = (0:L-1)*T;      % Time vector
figure(11);
Y= fft(Znorm);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;
plot(f,20*log10(P1));
title('Spectrum of Normalised Impedance Module(t)')
xlabel('f (Hz)')
ylabel('|dB|')


N     = 40;  % Order
Fstop = 10;   % Stopband Frequency
Astop = 80;  % Stopband Attenuation (dB)

% Construct an FDESIGN object and call its CHEBY2 method.
h  = fdesign.lowpass('N,Fst,Ast', N, Fstop, Astop, Fs);
Hd = design(h, 'cheby2');


Fstop2 = 0.5;   % Stopband Frequency

% Construct an FDESIGN object and call its CHEBY2 method.
h2  = fdesign.highpass('N,Fst,Ast', N, Fstop2, Astop, Fs);
Hd2 = design(h2, 'cheby2');


Zefe = filter(Hd,Znorm);
Zfiltered = filter(Hd2,Zefe);
figure(12);
plot(x,Zfiltered);
title('Filtered Impedance Module')
```

```
xlabel('Sec')
ylabel('V')


Y= fft(Zfiltered);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;
figure(13);
plot(f,20*log10(P1))
title('Spectrum of Filtered Normalised Impedance Module(t)')
xlabel('f (Hz)')
ylabel('|P1(f)|')


Y= fft(V);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;
figure(14);
plot(f,20*log10(P1))
title('Spectrum of Voltage Module(t)')
xlabel('f (Hz)')
ylabel('|P1(f)|')


Y= fft(Vnorm);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;
figure(15);
plot(f,20*log10(P1))
title('Spectrum of Normalized Voltage Module(t)')
xlabel('f (Hz)')
ylabel('|dB(f)|')
```