_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Ultra low-power S_pO_2 measurement system based on photoplethysmography

Claudio Barriuso Medrano

Tutor empresa: Carlos Millán Navarro

Tutor universidad: Germán Ramos Peinado

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

ANALOG DEVICES

Curso 2015-16

Valencia, 31 de Julio de 2016

Resumen

Hoy en día técnicas de fotopletismografía son empleadas en la monitorización del cuerpo humano para obtener constantes vitales. La principal ventaja es la posibilidad de realizar medidas de manera no invasiva en tiempo real. La obtención del ritmo cardiaco y de la saturación de oxígeno en sangre son algunas de las principales aplicaciones.

Estos sistemas de medida necesitan funcionar durante un largo periodo de tiempo para guardar un registro de las constantes vitales. Existen dispositivos portables que permiten al usuario adquirir estas constantes y almacenarlas. Sin embargo estos dispositivos tienen un tiempo de funcionamiento limitado por el consumo de corriente inherente al empleo de fotopletismografía. Por esta razón es vital encontrar métodos y algoritmos que permitan reducir al máximo el consumo de corriente.

A lo largo de este trabajo se proponen tres técnicas para optimizar el consumo a la hora de obtener la saturación de oxígeno en sangre, sin embargo también pueden aplicarse a la obtención del ritmo cardiaco. Estas son implementadas en un sistema de medida en tiempo real desarrollado con motivo de este trabajo y, posteriormente, evaluadas con un circuito de medida de corriente consumida.

Resum

A dia d'avui les tècniques de fotopletismografia s'utilitzen per a la monitorització del cos huma per a obtindre constants vitals. El principal avantatge és la possibilitat de realitzar mides de manera no invasiva a temps real. L'obtenció del ritme cardíac i de la saturació d'oxygen en sang són algunes de les principals aplicacions.

Aquests sistemes de mida normalment necessiten funcionar durant un llarg període de temps per a guardar un registre de les constants vitals. Existeixen dispositius portables que permeten a l'usuari adquirir aquestes constants i almacenar-les. Tot i així aquests dispositius tenen un temps de funcionament limitat pel consum de corrent inherent a la utilització de fotopletismografia. Per aquesta raó és vital trobar mètodes i algoritmes que puguen reduir al màxim el consum de corrent.

Al llarg d'aquest treball es proposen tres tècniques per a optimitzar el consum a l'hora d'obtindre la saturació d'oxygen en sang, encara que també poden aplicar-se a la obtenció del ritme cardíac. Aquestes són implementades en un sistema de mida en temps real desenvolupat amb motiu d'aquest treball i posteriorment avaluades amb un circuit de mida de corrent consumida.

Abstract

Nowadays photoplethysmography techniques are used to monitor the human body to obtain vital signs. Their main advantage is the ability to perform noninvasive measurements in real time. Peripheral capillary oxygen saturation as well as heart rate extraction are just some of the main applications.

These measuring systems need to be working for a long period of time in order to store a vital signs log. There are portable devices that allow the user to acquire these signs and store them. However these devices have a limited operational time due to the current consumption of photoplethysmography techniques. For this reason it is of great importance to find methods and algorithms that reduce the current consumption as much as possible.

Along this project, three techniques to optimize the current consumption when obtaining the blood oxygen saturation are proposed, however they can also be applied to obtaining the heart rate. They will be implemented in a real time measuring system developed for this project and, then, its current consumption will be evaluated with a custom current measuring circuit.

Index

Chapter 1. Introduction

Body monitoring provides valuable information that properly transformed into manageable signals can be used for different purposes. The most common application is to quickly spot health problems in patients. However over time end consumer products have implemented different technologies to obtain these signals in order to help the user to track information about his body.

There is a wide variety of vital signs that can be gathered with nowadays technology. Some of them are heart beat rate, breathing rate, body temperature, blood pressure and S_pO_2[1]. Only with this information you can build up a sound idea of someone's health, and this can be done within minutes.

However some time ago it was not so easy to obtain some of these signals. For example, to get the S_pO_2 value, you had to extract blood so that it could be analyzed in a laboratory[3]. These methods have changed and now there are faster, non-invasive, solutions. When it comes to obtaining the S_pO_2 the most commonly used method is photoplethysmography(PPG).

PPG is a method that consists in measuring changes in the volume of a certain area of the body's limbs using light. This volume will change with the heart beat so that when there is no blood flowing through the arteries the volume will be small compared to when the is blood flowing, in which the volume will be maximum. There are two possible implementations: reflected and transmitted PPG. The difference resides in how the measuring system is designed. Both implementations work by measuring the diffused light that has travelled through the skin tissues. The received light will vary depending on the volume of the flood flowing through the measured area.

However different light wavelengths will have different absorption for different blood components. As we will explain later this enables us to measure the S_pO_2 by using just two different wavelengths.

S_pO_2 calculation formula needs the amplitude of the reflected light signal in the local maximum and minimum peaks. However, most of the reflected light signal does not give us useful information. This means that for each heart beat period there is a small time period corresponding to the peaks we care about and a large time period that does not give us information.

The aim of this project is to determine if the current consumption of the process for measuring the S_pO_2 can be optimized with some algorithms that sample the received light more precisely in the peak time periods while in the rest of the signal less current is used as precision is not a priority.

We will go through three different algorithm approaches that will vary some sampling configurable variables such as the sampling frequency or LED pulses depending on the current position of the heart beat.

The entire project will be developed from the ground up, taking just the needed hardware and creating a complete software environment to measure the S_pO_2 implementing the new algorithms. Two development boards will be mainly used for this purpose: An Analog Devices photometric development board and a multi-purpose Cortex-M4 processor development board.

This Analog Devices sensor will be placed in one of the fingertips. This is because the signal quality in this zone is high as there are many capillaries inside. The better the

[1] Arterial Oxygen Saturation measured by a pulse oximeter

signal quality the better S_pO_2 value we will obtain.

A computer connected to the processor board will be used to plot the signal, system information, vital sign values, and, in the early state of the project, to verify the viability of the algorithms before implementing them in the processor.

Custom made algorithms will be developed to obtain the maximum and minimum signal peaks as well. A special filter will be designed for this purpose, in order to get rid of the noise and unwanted information.

Current consumption will be verified creating an evaluation system to determine which algorithm is the best when it comes to optimizing current drainage. Then we will discuss in which situation each algorithm could be used.

Chapter 2. Objectives

The main objective of this project is to propose and evaluate ways to reduce power consumption when measuring S_pO_2 using PPG techniques. Three ideas were proposed to achieve this before starting with the project. All of these approaches take advantage of the fact that the information needed to obtain the S_pO_2 value is localized on the values of the peaks of the PPG signal to reduce the power consumption. This means that most of the signal does not give us information, and therefore we can sample with less precision these zones. Each approach will be implemented as an algorithm that will run in real time. In order to reduce the precision (and therefore the consumption) we will change the sampling configuration of the sensor on the fly. Changing the sampling frequency or the number of pulses send to the LEDs in each sample are two ways to achieve this. The aim of this project is to check if these algorithms can be implemented on the microcontroller so that they can run on their own and if they can reduce the current consumed by the measuring system.

Three main elements will be essential in our project. The first one is the sensor we will use to obtain the PPG signal, the ADPD144. It features two LEDs and a photodiode that will obtain the light coming from the LEDs and diffused by the human tissue. The second one is the microcontroller multi-purpose development board. This element will establish the communication with both the sensor and the computer. The last element is the computer, it will plot the PPG signal and show the information of the sensor among other tasks.

To test these algorithms a Matlab application will be developed[6] along with the microcontroller software to obtain the PPG signal from the sensor and send it to the computer. Once the signal is received, the Matlab application will run the signal processing.

Once we have tested with this PC setup the algorithms, we will create another microcontroller software along with another Matlab GUI. This time all the processing tasks will be carried out by the microcontroller, and the Matlab application will just plot graphics and information. The peak detection algorithm will also be implemented in Matlab in the first setup and in the microcontroller in the second one. This peak algorithm will obtain the peaks of the PPG signal in real time thanks to a custom designed filter[7].

In both projects we will have to stablish a communication between the sensor and the microcontroller using an I2C interface and between the microcontroller and the computer using a USB cable.

In order to cope with the ADPD144 sensor, we will have to fully understand its datasheet and its registers. We will also have to understand the driver that the sensor development team has created and we will have to include it in the code of our microcontroller. We will also have to learn how to program the microcontroller, the STM32F405 using the IAR development environment as well as understanding how the board the processor is embedded in works[13]. Matlab programming will also be learnt in order to develop the GUI.

A current consumption measuring system will have to be developed to validate that the algorithms can reduce the current consumption. It will be able to measure the current drained by the sensor and the microcontroller. The consumption of the sensor will be reduced, however the microcontroller consumption may increase, due to the extra processing involved.

Chapter 3. Operational theory

3.1 Photoplethysmogramphy theory

PPG is a optical technique to measure surface tissue capillaries volume. The volume of the capillaries will vary along a heart beat period. When the blood flows through an artery its volume rises to its maximum, following the opening of the aortic valve, and then it lowers until its minimum when no blood flows.

To measure this volume, light is pulsed towards the skin. This beam of light travels through the tissues and it will eventually get out of the skin attenuated. This emerging beam is measured with a photodiode. The received light will vary with the arterial volume. When we have the maximum volume the light will be absorbed more compared to the time period in which the volume is low. With this simple setup we could measure the heart beat rate either by obtaining the mean peak-to-peak period or looking at the FFT of the PPG signal.This is possible thanks to the light absorption of the blood.

There are two types of possible measurement designs depending on the configuration of the sensor:

- **Transmissive PPG** In these type of solutions the light emitting source is opposed to the photodiode in such a way that the light beams travel through the human tissues and they reach the photodiode at the opposite side. This approach is commonly used to measure the PPG signal in the fingers or in the ear lobes.

- **Reflective PPG** Is the one used in the project. In this type of sensors the LED and the photodiode are located in the same side. They work by measuring the light after travelling through the tissues and returning back to the photodiode. This approach is commonly used in zones of the body where it is not possible to put a sensor at the other side, such as in the wrist or in the forehead. The signal however looks almost the same, the only difference is that it looks inverted compared to the transmissive PPG signal.

3.2 S_pO_2 acquisition

As commented before S_pO_2 can be obtained through PPG signals, however the setup is more complex. S_pO_2 can be measured thanks to the different light absorption coefficients of the components in the blood. Lets first have a look at the different blood components[8]:

- **Erythrocytes** Also known as red cells. It is mainly composed by different hemoglobin types, which are the most important light absorption elements. It is responsible for the oxygen and carbon dioxide transport. It is divided into two different groups:

  - **Functional hemoglobin** These are the erythrocytes that are able to carry oxygen. They can be found in two different states depending on the number of oxygen molecules that are binded to them. We call them Oxyhemoglobin(HbO_2) when they carry four oxygen molecules and reduced hemoglobin(Hb) when they carry less oxygen molecules.
  - **Dysfunctional hemoglobin** These are erythrocytes that cannot bind oxygen. This group is mainly made up by Methemoglobin (Oxidized hemoglobin) and CarboxyHemoglobin (hemoglobin combined with carbon monoxide). Methemoglobin can be transformed into reduced hemoglobin thanks to

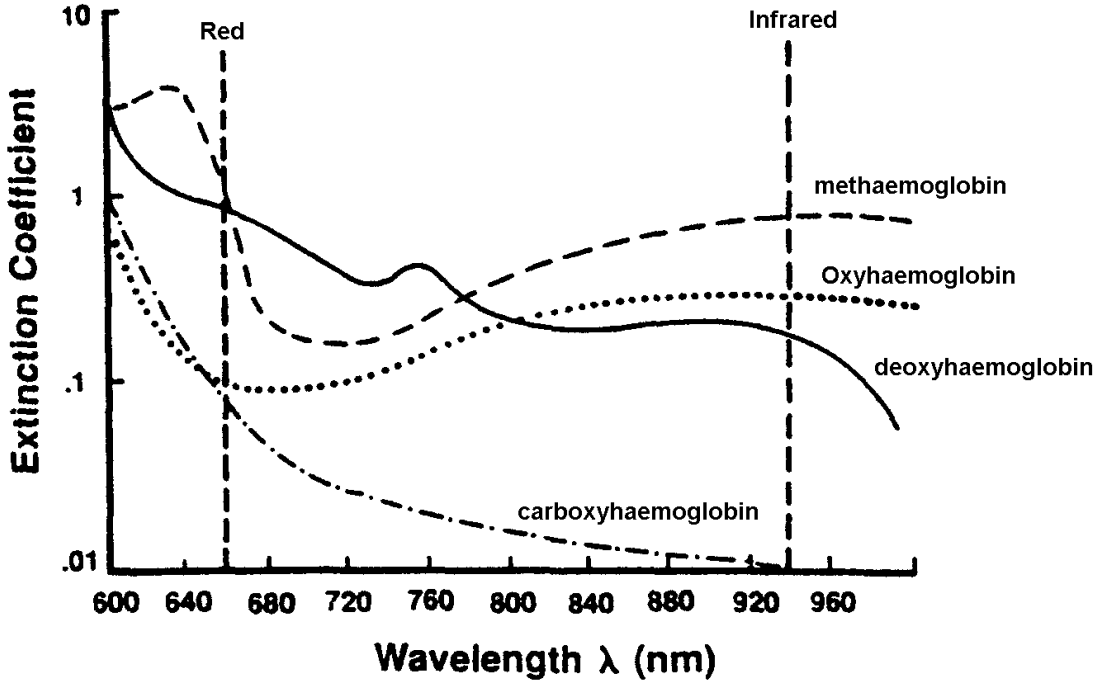


Figure 1: Extinction coefficients of the different blood components

natural reducing systems. CarboxyHemoglobin shows up when there is presence of carbon monoxide in the blood. The affinity of hemoglobin binding with carbon monoxide is 210 larger than that of oxygen[5], which means that they are more likely to bind carbon monoxide than oxygen.

- **Leukocytes** Their main task is to protect the body against external organisms and fight diseases. They are colourless and do not affect light absorption significantly.

- **Platelets** These are not cells strictly speaking. They serve different tasks such as stopping bleeding and destroying bacteria.

Thanks to the different absorption coefficients for the oxyhemoglobin and the hemoglobin we can obtain the S_pO_2 value. This is the conclusion that can be obtained with the Beer-Lambert's law. These extinction coefficients are shown in Figure 1. As we can see for each blood component we have a light absorption graph for different wavelengths values. The fact that these absorption coefficients are not equal for all the components for certain wavelengths will enable us to obtain the S_pO_2 value.

Beer-Lambert's law describes the attenuation of light travelling through a uniform medium containing an absorbing substance. The intensity of light inside this medium decreases exponentially with the distance[5]:

$$I = I_0 e^{-\epsilon(\lambda) c d} \tag{1}$$

Where c stands for the concentration of the substance in $mmol \cdot L^{-1}$, $\epsilon(\lambda)$ the extinction coefficient for a certain wavelength in $L \cdot mmol^{-1} \cdot cm^{-1}$, d the length of the medium in cm and I_0 the intensity of the light before it reaches the medium in W/m^2.

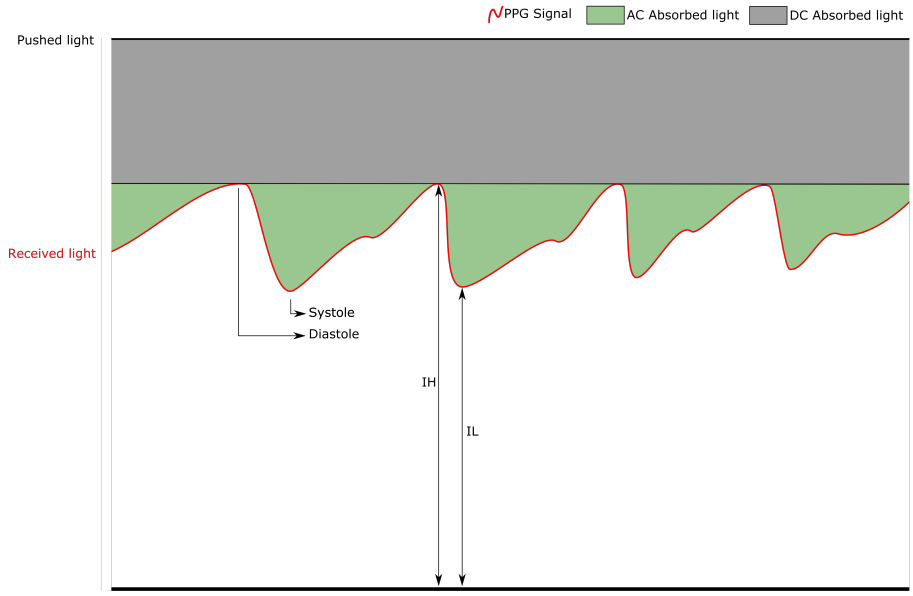


Figure 2: Reflective PPG Signal shape

If the light goes through several mediums, each one applies the exponential attenuation to the incident light. Therefore we can write a generic formula for light going through certain N mediums, each one with a specific concentration:

$$I = I_0 e^{\sum_{i=0}^{N-1} -\epsilon_i(\lambda) c_i d_i} \tag{2}$$

As we know, the PPG signal is made up of different components. The main part of the signal is reflected light that has no information. This is the light that has been reflected on the surface of the skin. Another component is the light that has travelled through the skin tissue but that does not give us information because it has not reached the capillaries. Finally, we have the diffused light that has travelled through the capillaries and that therefore has the information we care about. The amount of light detected will depend on the amount of the blood flowing through the capillaries. This flowing blood will change the diameter of the capillary and therefore the light reflected back to the photodiode.

For this reason we can model the reflected light during systole[2](I_L) and diastole[3](I_H) as shown in Figure 2. The variables d_{min} and d_{max} stand for the capillaries diameter when there is no blood flowing and when there is blood flowing respectively. The variables with the DC subscript are characteristics of the non-variable medium that light goes through. This medium is made up by the skin tissue that does not contain capillaries. It will attenuate the input light intensity by a fixed and constant value. With this formula we can distinguish the constant and the variable attenuation.

$$I_H = I_0 e^{\epsilon_{DC}(\lambda) c_{DC} d_{DC}} e^{-[\epsilon_{Hb}(\lambda) c_{Hb} + \epsilon_{HbO_2}(\lambda) c_{HbO_2}] d_{min}} \tag{3}$$

$$I_L = I_0 e^{\epsilon_{DC}(\lambda) c_{DC} d_{DC}} e^{-[\epsilon_{Hb}(\lambda) c_{Hb} + \epsilon_{HbO_2}(\lambda) c_{HbO_2}] d_{max}} \tag{4}$$

On the other hand, we have the formula for the S_aO_2[4]. It measures the proportion of

[2]Contraction of the heart during the normal heart rhythm
[3]Dilatation of the heart during the normal heart rhythm
[4]**Saturation level of oxygen in hemoglobin**, **obtained from arterial puncture**.

HbO_2 in a solution with HbO_2 and Hb:

$$S_aO_2 = 100 \cdot \frac{c_{HbO_2}}{c_{HbO_2} + c_{Hb}} \tag{5}$$

However here we have two incognitas: c_{HbO_2} and c_{Hb}. We will have to obtain a second equation with the Beer's-Lambert law.

By their own, I_H and I_L have many incognitas. However, if we divide I_H by I_L we will reduce the number of incognitas to three(HbO_2, Hb and Δd):

$$I_L/I_H = e^{-[\epsilon_{Hb}(\lambda)c_{Hb} + \epsilon_{HbO_2}(\lambda)c_{HbO_2}]\Delta d} \tag{6}$$

This formula is valid for a generic wavelength. If we use two wavelengths and we apply an approximation we can obtain a formula with just two incognitas, c_{HbO_2} and c_{Hb} again. The two wavelengths we will use will be λ_R and λ_{IR}. We will take for granted that $\Delta d_R = \Delta d_{IR}$, which is an approximation. This way we will be able to obtain a formula with the two incognitas:

$$Q = \frac{I_{L,R}/I_{H,R}}{I_{L,IR}/I_{H,IR}} = \frac{e^{[\epsilon_{Hb}(\lambda_R)c_{Hb} + \epsilon_{HbO_2}(\lambda_R)c_{HbO_2}]}}{e^{[\epsilon_{Hb}(\lambda_{IR})c_{Hb} + \epsilon_{HbO_2}(\lambda_{IR})c_{HbO_2}]}} \tag{7}$$

We will obtain the natural logarithm of the quotient of the ratios of the red and infra red signals so that it is easier to work with the formula:

$$R = \frac{\ln(I_{L,R}/I_{H,R})}{\ln(I_{L,IR}/I_{H,IR})} = \frac{[\epsilon_{Hb}(\lambda_R)c_{Hb} + \epsilon_{HbO_2}(\lambda_R)c_{HbO_2}]}{[\epsilon_{Hb}(\lambda_{IR})c_{Hb} + \epsilon_{HbO_2}(\lambda_{IR})c_{HbO_2}]} \tag{8}$$

Given S_aO_2, defined in Equation 5, we can rewrite Equation 8 as shown in Equation 9

$$R = \frac{\epsilon_{Hb}(\lambda_R)(1 - S_aO_2) + \epsilon_{HbO_2}(\lambda_R)S_aO_2}{\epsilon_{Hb}(\lambda_{IR})(1 - S_aO_2) + \epsilon_{HbO_2}(\lambda_{IR})S_aO_2} \tag{9}$$

And finally we can obtain the S_aO_2 equation:

$$S_aO_2 = \frac{\epsilon_{Hb}(\lambda_R) - \epsilon_{Hb}(\lambda_{IR})R}{\epsilon_{Hb}(\lambda_R) - \epsilon_{HbO_2}(\lambda_R) + [\epsilon_{HbO_2}(\lambda_{IR}) - \epsilon_{Hb}(\lambda_{IR})]R} \cdot 100 \tag{10}$$

This is a brief summary on how to obtain the S_aO_2 value[5]. With this formula, if we obtain the natural logarithm of the peaks ratios of both wavelengths we can obtain the S_aO_2 value.

The only unknown variable in this formula is S_aO_2. R can be calculated once we obtain the PPG signal peaks. Therefore this is the theoretical formula to obtain S_pO_2 given the reflected light from red and infrared LEDs.

However when it comes to the actual implementation these formulas are not used. Instead, a calibration formula is obtained that relates R and S_pO_2. There are several reasons of why the theoretical model does not give accurate results. The main one is that this model does not take into account light scattering in the skin and the tissues. Moreover blood is not a homogeneous liquid, which may lead to a non-linear absorbency of light. Another reason is the fact that the light paths are not equal for both wavelengths.

The calibration curve is obtained acquiring blood samples of a subject that breaths a controlled mixture of oxygen and nitrogen in a hospital. Each blood sample has a PPG signal ratio R associated. Once several samples have been acquired for different

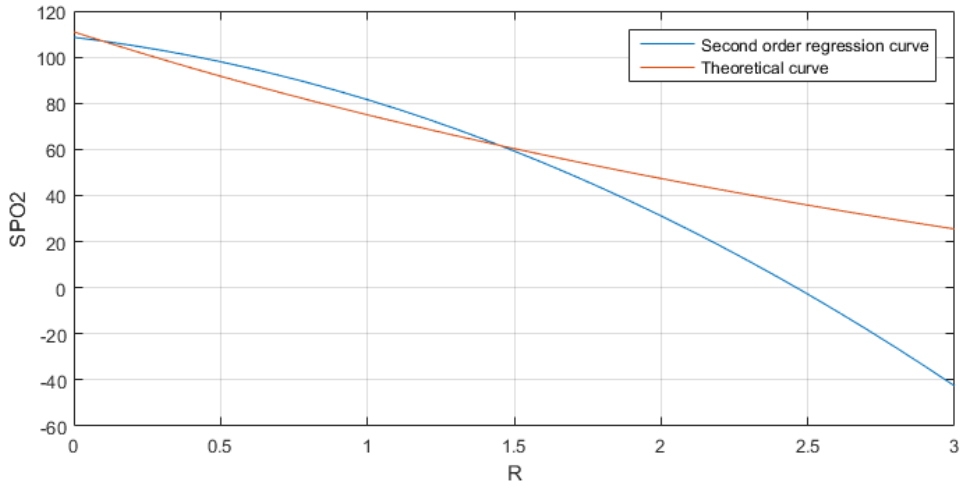


Figure 3: Theoretical and second order curves

concentrations of oxygen a regression curve is obtained for the data. The formula that we will use in this project has been obtained in a hospital of Boston for the ADPD144 sensor, using least squares to adjust the curve to the obtained data. This curve along with the theoretical one calculated with the data from Zijlstra[15] can be seen in Figure 3. The regression curve formula can be seen in Equation 11

$$S_pO_2 = -11.6768R^2 - 15.3598R + 108.5978 \tag{11}$$

3.3 ADPD144 Sensor

This is the sensor we will use to obtain the PPG signal in order to get the S_pO_2 value. It is a photometric system to measure optical signals from synchronous reflected LED pulses. This synchronous measurement technique enables us to reject ambient light interference. It features a photometric front end, two LEDs and a four channel photodiode. To sample the light the sensor has a 14-bit ADC with 2 independently configurable LED drivers. The communication is carried out with a I2C interface. Lets review in-depth these parts of the sensor:

3.3.1 Photometric front end

This is the part of the sensor that obtains the PPG signal. It controls the LEDs and the sampling periods to measure the reflected light. In order to capture the received light the signal goes through different blocks before being sampled as shown in the ANALOG BLOCK of the ADPD144 block diagram shown in Figure 4. This photometric front end has four channels, which means that it can sample for each sampling period four different sources. This is why the four channels of the photodiode are connected inside the sensor to the ANALOG BLOCK. Moreover the sensor can be configured to have two sampling slots for each sample. This means that for each sample we can have two different sampling configurations, which will enable us to obtain the reflected light coming from the red LED and from the IR LED independently.

The current coming from the 4-channel photodiode is converted into voltage thanks to a transimpedance configured operational amplifier. The gain of the amplifier can be changed

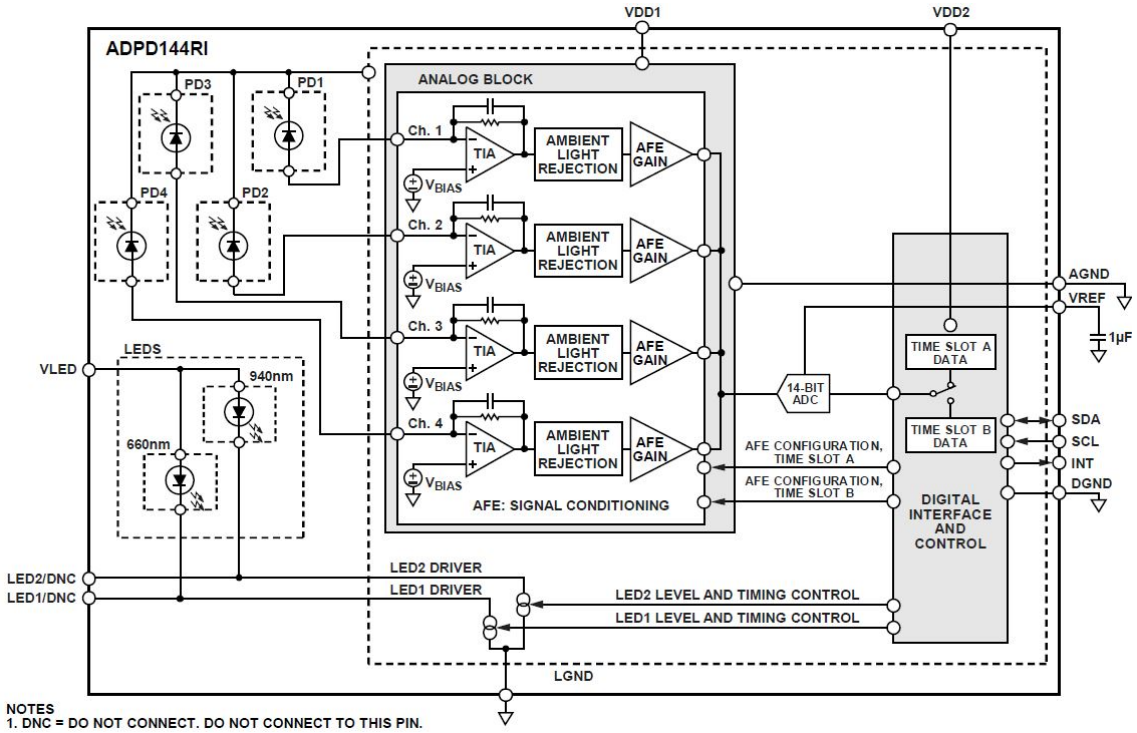


Figure 4: ADPD144 Block Diagram

through registers as we will see later. Then this signal is filtered with an analog filter that gets rid of the ambient light. This is possible thanks to the synchronization with the LED pulses. Then the signal is amplified within the *AFE*[5] *Gain* block. Finally the signals are converted with an ADC and put into a FIFO.

3.3.2 LED Driver

The sensor features two different LEDs. One of them as shown in Figure 4 works at a wavelength of 660nm (in the red spectrum), and the other one at 940nm (in the infrared spectrum). Red and infrared wavelengths have been used historically in S_pO_2 measuring systems due to reasons. The main one is the amplitude difference of the extinction coefficients of the HbO_2 and the Hb at those wavelengths. This enables us to obtain the PPG more precisely. The second reason is the flatness of the extinction coefficients of the functional and dysfunctional hemoglobin at those wavelengths[6].

These LEDs light when the AFE pushes current through the variable current sources connected to them. The current that these sources generate is controlled with different registers of the sensor. For each sampling period the LED lights a certain amount of times, which can also be defined through registers. This way if we average the pulses we can reduce the measuring noise. The pulses width and their period can also be configured. As we can see we have many different variables that enable us to adjust the parameters for each different situation, however there are certain values for some of these registers that produce an optimum result. Each LED can be associated to a sampling slot, this way we will be able to obtain the PPG signal of each wavelength separately.

[5] Analog Front End

[6] Flatness of the extinction coefficients is important as LEDs wavelength can vary and they are not exactly centred at a certain one. This way if the wavelength is slightly different it will not affect the measurement

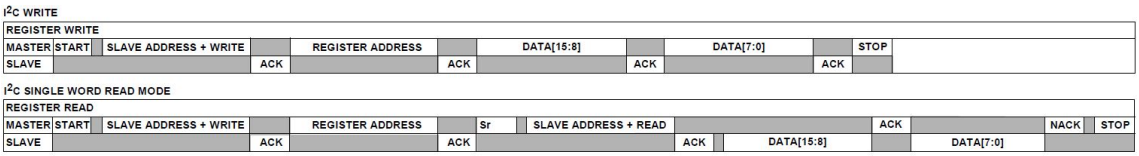


Figure 5: ADPD144 Read/Write procedure

3.3.3 Photodiode

The sensor has a photodiode with four channels. Each channel has a different separation from the LED source. This means that each channel will receive a different amount of reflected light. The aim of these four channels is to process them through software in order to obtain a better signal. However in this project we have added the signal from the four channels to obtain a single PPG signal. The photodiode as well as the LEDs are embedded in the silicon chassis.

3.3.4 Sampling Block

The sensor sampling block can result rather complex. It has to sample all four photodiode channels at two different time slots. Moreover it has to take into account the variable sampling frequency, the number of LED pulses for each sample, the number of averaged samples, etc. It features a 14-bit ADC, however a better resolution can be achieved thanks to the various averaging options available.

After converting the analog signal to the digital domain, a register configurable DC offset is subtracted. Then all the pulses of the sample period are added added up. However, the output will be clipped, it will only contain the 20 least significant bits. This output value is one sensor sample.

Once the sample is ready it is stored in the FIFO. When a new sample arrives the FIFO scrolls like a shift register. The size of this FIFO is 128 bytes. The number of samples that can fit into the FIFO depends on the configuration of the sample slots. When the FIFO is filled and a new sample arrives, the oldest sample is discarded.

As soon as the number of available samples in the FIFO reaches a value defined in a register, the new sample in FIFO interruption rises. The FIFO can be read at any time.

3.3.5 I2C Communication

The ADPD144 features an I2C interface to communicate with a master device. It supports I2C fast mode (400 kbps) data transfer. It has to be configured as a slave with the 7-bit address 0x64. The interface is used to read and write internal sensor's registers. The communication flow diagram is shown in Figure 5. This flow diagram will be implemented in the hardware abstraction layer provided by the microcontroller manufacturer, therefore we will not have to implement it. We will just call the necessary functions to write and read registers.

3.3.6 Sensor registers

The ADPD144 has many different configurable variables that can be set through these registers. All the registers have a size of 2 bytes. The information of the registers is

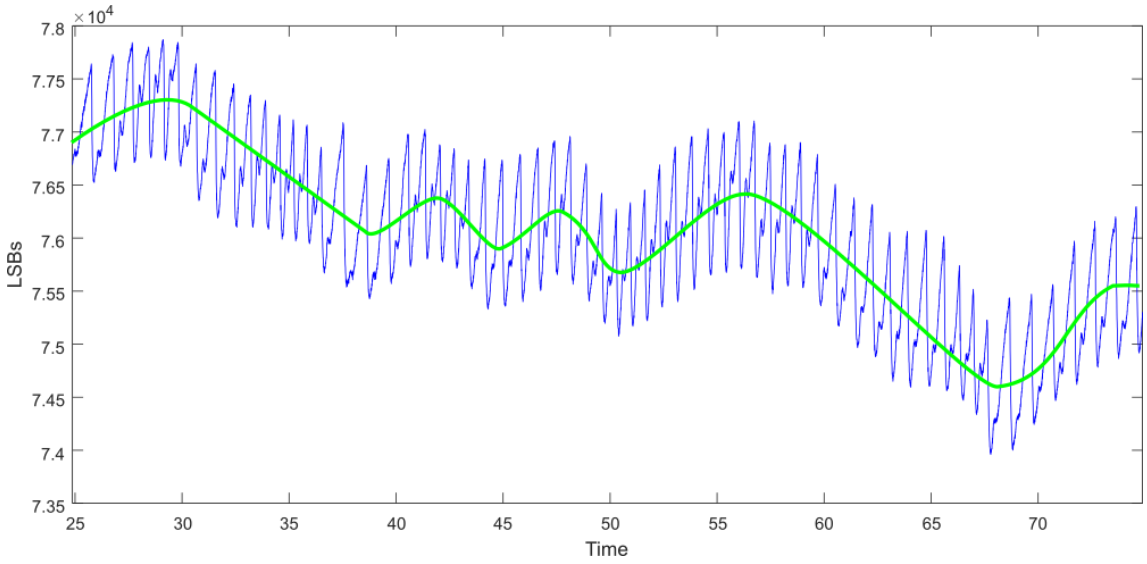


Figure 6: PPG Signal(Blue) and breathing signal(green)

specified in the sensor's datasheet.

3.4 PPG Signal Filtering

PPG signal has a big amount of information. For the purpose of this project, only part of this information is needed. For this reason a filter has to be applied to the signal in order to obtain the valuable data.

The PPG signal is basically made up of these elements:

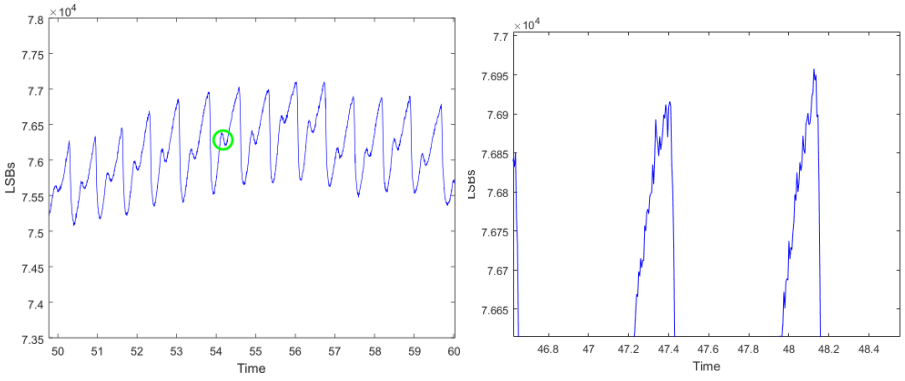
- **DC Signal** Most part of the received signal turns out to be a DC level. When it comes to the reflective PPG, DC signal is caused by several factors. The main one is the light being reflected in the skin surface that does not penetrate into the skin. Another factor is the reflected light beams that penetrate into the skin but that do not reach capillaries.

- **Breathing signal** PPG signal does not only contain information about the heart rate and S_pO_2. It can also provide information about the breath pace. This breathing signal is modulated over the PPG signal as shown in Figure 6 and can be obtained filtering it with a bandpass filter around the maximum and minimum breathing frequencies.

- **Dark Offset DC** Even if no light hits the photodiode there is a DC offset at the output due to the input offset of the TIA amplifier and the photodiode dark current among other factors. The good thing about this part of the signal is that it can be measured and corrected.

- **Ambient Light offset** If the sensor is not properly, placed ambient light can affect the measurement. Nevertheless the sensor we will use in this project has the best in class ambient light rejection filter that will reduce this problem.

- **AC signal** This is the part of the signal we are interested in. It is shown in a blue color in Figure 6. It usually represents 1-3% of the received signal. We will have to design a filter to fit this signal needs in order to differentiate it from the other components.



(a) (b)

Figure 7: Signal artefacts (a) Dicrotic notch and (b) Signal noise

We have already spotted the useful signal for our purpose. Our aim is to obtain the peaks of this signal in order to calculate the S_pO_2 Value. However this signal has some artefacts that makes this a difficult task as shown in Figure 7:

- **Dicrotic Notch** This peak in the PPG signal is produced by te closure of the aortic valve in the heart. This could give information about a subject's health but it is not useful to obtain the S_pO_2 and for this reason we will filter it. Sometimes it can be hard to tell the difference between a PPG signal peak produced by the abrupt flow of blood produced by the heart beat and the flow change produced by the aortic valve.

- **Signal Noise** The signal sampled is not as clean as desired and for this reason a low pass filter has to be applied to get rid of these components.

For these reasons we have to filter the signal in order to find the peaks.

The filter design procedure starts by eliminating the noise of the signal. In order to do so we apply a moving average filter. This filter output is shown in Equation 12.

$$y[n] = \frac{1}{M} \sum_{i=0}^{M-1} x[n-i] \tag{12}$$

Where M stands for the number of samples averaged. In the Z domain the equation of the filter is shown in Equation 13.

$$Y(z) = \frac{1}{M} \sum_{i=0}^{M-1} X(z) z^{-i} = \frac{1}{M} X(z) \sum_{i=0}^{M-1} z^{-i} \tag{13}$$

$$\frac{Y(z)}{X(z)} = H(z) = \frac{1}{M} \sum_{i=0}^{M-1} z^{-i} \tag{14}$$

The summation is a geometric progression. We can apply the property of geometric progressions shown in Equation 15 in order to get a more compact formula. This way we obtain Equation 16.

$$\sum_{i=m}^{n} ar^i = \frac{a(r^m - r^{n+1})}{1-r} \tag{15}$$

$$H(z) = \frac{1}{M} \frac{1-z^{-M}}{1-z^{-1}} \tag{16}$$

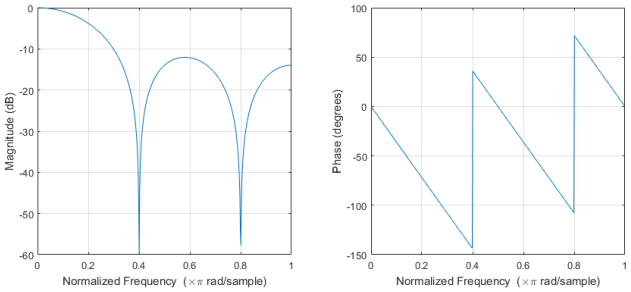


Figure 8: Avg filter example for M = 5 and Fs = 25Hz

If we step to the frequency domain we will obtain the formula shown in Equation 17.

$$H(w) = \frac{1}{M}\frac{1 - e^{-jwM}}{1 - e^{-jw}} \tag{17}$$

Lets see where do we have the notch of this filter. For this purpose we only need the magnitude of the filter $H(w)$. Lets multiply and divide Equation 17 by:

$$\frac{e^{jwM/2}}{e^{jw/2}}$$

With this, we obtain:

$$H(w) = \frac{1}{M} * \frac{e^{-jwM/2}}{e^{-jw/2}}\frac{e^{jwM/2} - e^{-jwM/2}}{e^{jw/2} - e^{-jw/2}} \tag{18}$$

Lets recall this identity:

$$sin(w) = \frac{e^{jw} - e^{-jw}}{2j} \tag{19}$$

This way we can rewrite Equation 18 like this:

$$H(w) = \frac{1}{M}\frac{e^{-jwM/2}}{e^{-jw/2}}\frac{sin(wM/2)}{sin(w/2)} \tag{20}$$

As we only care about the magnitude, we obtain:

$$|(H(w)| = \frac{1}{M}\left|\frac{sin(wM/2)}{sin(w/2)}\right| \tag{21}$$

Therefore the notches will be located at:

$$\begin{aligned} sin(wM/2) &= 0 \\ wM/2 &= k\pi \\ w &= 2\pi\frac{f}{fs} = k\frac{2\pi}{M} \\ f &= k\frac{fs}{M} \end{aligned} \tag{22}$$

In Figure 8 we can see the filter response for $M = 5$

As for now, we have averaged the signal to remove the noise and we have the notches as specified in Equation 22. Now that we have reduced the noise we can look for the peaks. At first a status machine was thought to do this job. This would go through the signal and detect changes in the amplitude in order to determine the peaks. However this approach

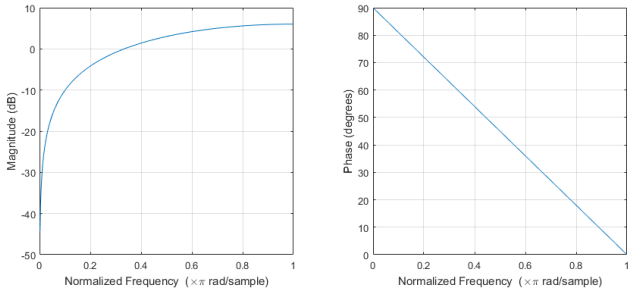


Figure 9: First derivative filter approach

did not work properly and another one was researched. As we know, peaks are signal points whose slope is zero. Therefore one solution could be to get the signal derivative and look for the zero crosses. At first we thought this derivative could have this form:

$$y[n] = x[n] - x[n-1] \tag{23}$$

Which is the general derivative definition. However this did not work as expected. The reason is that the remaining noise produces a lot of slope changes. This can be seen if we obtain the frequency response of this filter as shown in Figure 9. If we take a look at high frequencies we can see in the magnitude that they get amplified. For this reason we looked of another way of achieving this. The solution was to use some kind of averaged derivative. We can do it as shown in Equation 24.

$$y[n] = x[n] - x[n-N] \tag{24}$$

As we can see instead of subtracting the previous arrived sample to the actual one, we subtract the sample arrived N samples ago. This way the high frequencies will not affect as much as it did with the first approach. However we have to be careful and see where are the notches of the filter. For this purpose lets obtain the filter frequency response and obtain the value where the magnitude is zero:

$$Y(z) = X(z) - X(z)z^{-N} \tag{25}$$

$$H(z) = \frac{Y(z)}{X(z)} = 1 - z^{-N} \tag{26}$$

$$H(w) = 1 - e^{-jwN} \tag{27}$$

$$\begin{aligned} |H(w)| &= \left|1 - e^{-jwN}\right| \\ |H(w)| &= |1 - cos(wN) + jsin(wN)| \\ |H(w)| &= \sqrt{(1 - cos(wN))^2 + sin^2(wN)} \\ |H(w)| &= \sqrt{1 + cos^2(wN) - 2cos(wN) + sin^2(wN)} \\ |H(w)| &= \sqrt{2(1 - cos(wN))} \end{aligned} \tag{28}$$

Now we obtain the zero of the frequency response at:

$$\begin{aligned} 1 - cos(wN) &= 0 \\ wN &= 2k\pi \\ 2\pi\frac{f}{fs}N &= 2k\pi \\ f &= k\frac{fs}{N} \end{aligned} \tag{29}$$

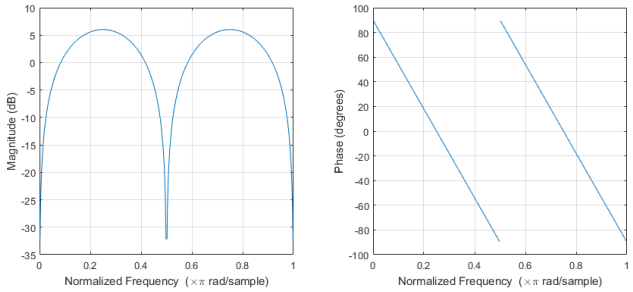


Figure 10: Comb filter example for N=4 and Fs = 25 Hz

This last filter turns out to be known as a comb filter due to its magnitude shape whose notches recall a comb as shown in Figure 10

In summary, the notch of both filters are:

$$\begin{aligned} f_{coAverage} &= k\frac{fs}{M} \quad \forall k \subset \mathbb{N} - \{0\} \\ f_{coComb} &= k\frac{fs}{N} \quad \forall k \subset \mathbb{N} \end{aligned} \tag{30}$$

Right now we have the notches of two filters that applied to the original signal will return a signal in which zero crosses represent peaks in the original signals delayed by the group delay of the filter. However we still have to decide the location of these notches. These notches will be determined by the maximum and minimum heart beat rates. We will design our application to work properly between 40 and 220 beats per minute.

$$\begin{aligned} f_{min} &= \frac{1}{T} = \frac{1}{\frac{60}{40}} = 0.66Hz \\ f_{max} &= \frac{1}{T} = \frac{1}{\frac{60}{220}} = 3.7Hz \end{aligned} \tag{31}$$

At the end f_{min} and f_{max} will determine notch location. We will focus on choosing the notch to fit f_{max} requirements, as f_{min} will be accomplished by the comb filter DC rejection as shown in Figure 10. This way let's set the first notch for $k = 1$ of both filters. We will add a margin to the originally designed 3.7 Hz notch frequency. This is because our filter will have a low number of coefficients, and with these margins excessive attenuation close to the originally designed notches will be avoided. Instead of locating the notch at 3.7Hz, we will locate it at 5Hz:

$$\begin{aligned} f_{coAverage} = 5 = \frac{fs}{M} \rightarrow M = \frac{fs}{5} \quad \forall k \subset \mathbb{N} - \{0\} \\ f_{coComb} = 5 = k\frac{fs}{N} \rightarrow N = \frac{fs}{5} \quad \forall k \subset \mathbb{N} \end{aligned} \tag{32}$$

In Figure 11 we can see the resulting filter for a sampling frequency of 50 Hz. It can be obtained convolving the coefficients of both filters. As we can see the phase is linear, which is essential as this way the group delay will be a constant. This way we will be able to locate the peaks in the original signal simply by subtracting the group delay.

The filter we have designed has to meet some other specifications needed in order to work in real time[7]. The most important one is the filter length. This has to be kept as low as possible in order to detect the peaks as soon as possible. However lowering the number of coefficients could worsen the frequency response.

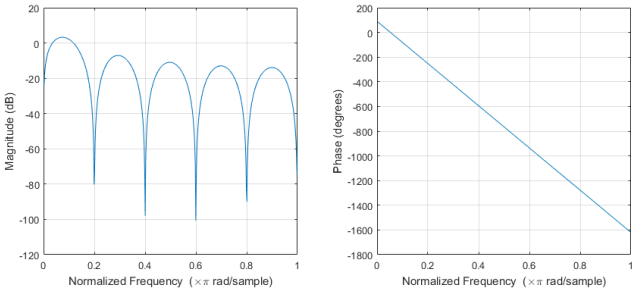


Figure 11: Total filter shape for fs = 50 Hz

Lets have a look at the design filter length. We currently have one filter obtained convolving the averaging and the comb filter. The original filter sizes are:

$$\begin{aligned} Size_{Avg} &= M \\ Size_{Comb} &= N + 1 \end{aligned} \tag{33}$$

Then, the convolution will have a size of:

$$Size_{conv} = M + (N + 1) - 1 = M + N = \frac{2}{5} fs \tag{34}$$

Then the group delay can be calculated this way:

$$GrpDelay = \frac{Size_{conv} - 1}{2} = \frac{2fs - 5}{10} \tag{35}$$

This means that the filtered signal will be delayed $GrpDelay$ samples. As we have commented before it is of high importance to have a low delay group. This is because we will be analyzing the signal in real time and delays must be avoided.

Both the group delay and the heart rate will determine at which percentage of the heart period we can start the low precision acquisition period. Lets say that we want to go from the high precision to low precision acquisition period at the percentage $R\%$. Then there is a maximum heart rate we can face. This can be calculated as shown:

$$\begin{aligned} \frac{2fs - 5}{10} &= \frac{R}{100} T fs \\ T &= \frac{60}{bpm} \\ R &= \frac{(2fs - 5)bpm}{6fs} \end{aligned} \tag{36}$$

In Equation 36, T stands for the heart rate period in seconds and bpm stands for the heart rate pace in beats per minute. This formula can be seen plotted in Figure 12 for different sampling frequencies.

3.5 Low-power algorithms

In many situations PPG systems are interesting to track the evolution of the measured signals over a long period of time. However sometimes this requires the user to wear the

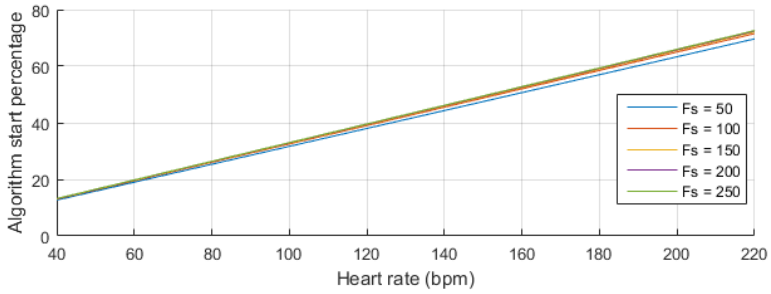


Figure 12: Relation between Heart Rate and LPAP Start percentage

measuring device wherever he goes. For this reason large batteries are needed in order to provide the system the enough current to work properly. Nevertheless it is possible to lower the current consumption using some techniques. As we have already said S_pO_2 calculation only needs the maximum and minimum peaks of the PPG signal. The rest of the signal is useless for this purpose.

Given the shape of a finger PPG signal, we can distinguish different time periods in which we need to sample the received light in the photodiode precisely and time periods in which we can lower the acquisition accuracy in order to save current.

Theoretically in each heart period there are two zones in which we must sample the signal precisely (the peaks) and two other zones that do not give us useful information. However due to the filtering delay previously commented we cannot guarantee that the maximum peak will be detected before this time period starts, therefore we will only consider one high sampling precision zone and one low power consumption zone for each heart period. We will explain this problem in detail later. We will call HPAP to the High Precision Acquisition Period and LPAP to the Low Precision Acquisition Period from now on.

Three algorithms were proposed to lower the current consumption, each of which has a different way to reduce the activity of the sensor in the LPAP.

3.5.1 Pulse variation algorithm

As explained in Section 3.3 for each sample multiple pulses are sent to the LED. Then, the AFE adds these pulses in order to lower the noise. The main idea of this algorithm is to reduce the number of pulses in the LPAP and increase them in the HPAP. The disadvantage of this method is that in the LPAP the noise will be increased noticeably. This noise will not affect the peak detection. This is because of the designed peak detection procedure. First of all, signal filtering will get rid of the high frequencies produced by this noise. Then, we will locate the peaks by looking at the zeros of the derivative. With this information we will look for the local peaks in the original signal. As the noise will not be located close to the peaks of the original signal, it will not be a problem.

In the Figure 13 a representation of the mechanism of this algorithm can be seen.

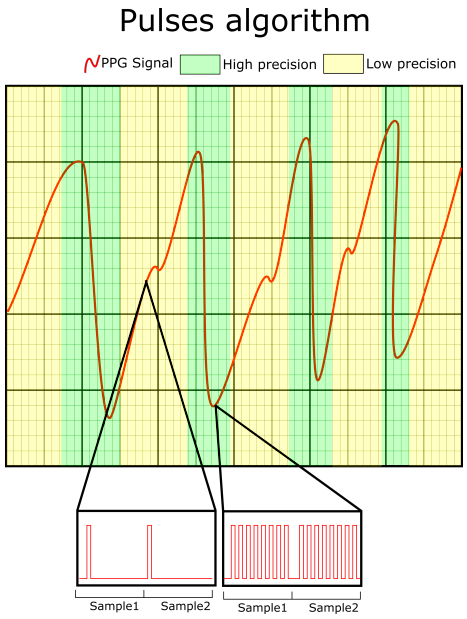


Figure 13: Pulses Algorithm mechanism representation

3.5.2 START / STOP algorithm

This approach is the most aggressive. The main point is to send the ADPD sensor to standby mode in the LPAP and sample again in the HPAP. The main advantage is that this would be the best algorithm when it comes to reducing current consumption, however it has some drawbacks: In case the PPG signals gets distorted, peak estimation may fail making the algorithm turn off in a HPAP, which would lead to a peak loss, which will again lead to a peak loss and so on. For this reason at first sight this algorithm seems to be quite risky as we will corroborate with the tests. Moreover as we will stop sampling we will have to interpolate samples so that there are not abrupt changes in the signal shape that could affect the filter. In the Figure 14 a representation of the mechanism of this algorithm can be seen.

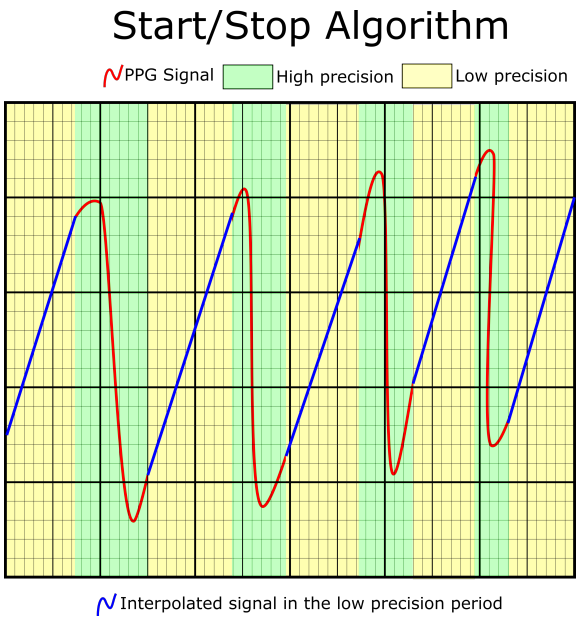


Figure 14: Start/Stop Algorithm mechanism representation

3.5.3 Sampling frequency algorithm

The proposal of this algorithm is to lower the sampling frequency in the LPAP and increase it in the HPAP. The advantage compared to the START/STOP algorithm is that this way a peak will never be lost as we are continuously sampling in the heart rate period. On the other hand the current consumption savings will not be that high. Moreover with this algorithm we will not have noise as we have with the pulse variation algorithm. The disadvantage is that as we will lower the sampling frequency, we will have to interpolate programmatically so that the filter works properly. In Figure 15 a representation of the mechanism of this algorithm can be seen.

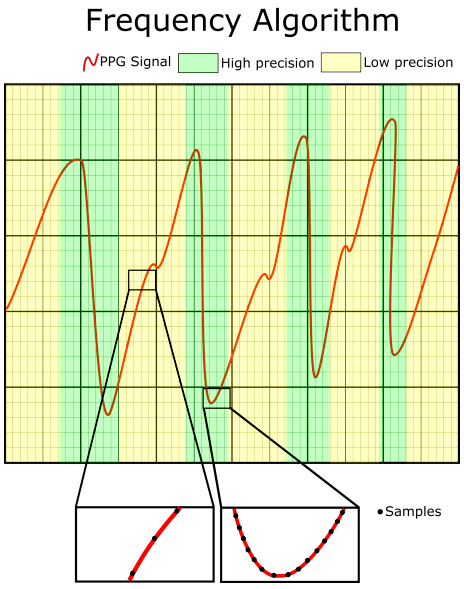


Figure 15: Fs Algorithm mechanism representation

3.6 State of the art

PPG is a mature technique to measure differences in the variations of the volume of the capillaries due to the blood flow. The average current that the devices that implement this technology drain is rather high, however these devices are usually plugged to the mains and the consumption is not a critical factor.

However over the last years new PPG applications have emerged. Most of the times they require to run from a battery, and therefore current consumption has to be taken into account.

Some techniques to reduce power consumption are well known. One of the most important advises is to use efficient LEDs and photodiodes with high responsivity. Moreover instead of pushing continuous light, we can just push current to the LEDs when sampling and integrate the reflected light in the AFE, also known as synchronized sampling. We can also configure the sensor to raise the new sample interruption once it has sampled several of them, this way the communication is more efficient.

Entire PPG systems on chip are being develop to reduce the consumption. The one used in this project is a good example of this approach as we can see in Figure 18. The drawback of these devices is that they reduce the freedom to choose the LEDs and the photodiodes.

The tendency of PPG technology is to integrate all the processing on the chip as well. This way all the filtering and processing tasks are carried out by the hardware, which is always faster and more efficient than software implementations. Moreover it is easier for the developer to work with them as most of the job is already done.

Chapter 4. Methodology

In this section we will review how the project was organized and managed. This is one of the most important part of the project as it helps to set deadlines for objectives and tasks. This schedule forces you to hurry up when a task is not as developed as it should be.

4.1 Project management

This project has taken 6 months of work to be finished. Along these 6 months the author of this project has carried out the objectives set before starting. The development of the project started the 1^{st} of February and finished the 31^{st} of July. During the months of February, March, April and May the author has worked for 6 hours a day, whereas during June, July and August he has worked for 8 hours a day. The number of weeks worked along these months is 26. Taking into account the vacation days, the author has worked 870 hours.

This project has been developed within the company Analog Devices S.L.U as a part of an internship. The author has been supervised by his supervisor along these 6 months. The design centre manager of Valencia's headquarters of Analog Devices has also given some advice about how to carry out the project. The university tutor has also helped giving advice at some points of the project.

Once a week the author of the project met with his supervisor in order to catch up with the new tasks developed.

The author of the project had a diary in which every day he used to write the new work carried out as well as the problems faced. This was really useful when writing this report.

4.2 Tasks temporal distribution

Before starting with the project several tasks were arranged given the objectives that had been set. These divide the objectives into smaller specific tasks easier to focus on. Then, for each task a number of weeks were assigned. This is how we set the deadlines for each task. Along the project some tasks took longer than expected. This may be because the implementation approach faced some problems that required researching sort outs. Nevertheless enough time was set for each task and most of them could meet the deadline. We will go through now the different task we divided this project in:

1. **Gathering information** The very first task was, as in every project, to look for information about the state of the art and to get to know the hardware we were going to be working with. We had already been working in the past with the ADPD144 sensor as well as the STM32F4 microcontroller, and for this reason it was easier to catch up. This task took the first half of February.

2. **Develop the research measuring system** Before implementing the algorithms directly on the microcontroller we thought that we should try to implement them in Matlab. This is because it is much easier to work with signal filtering within Matlab. For this reason we started developing the research measuring system in first place. It was necessary to develop both the microcontroller c and the Matlab code almost at the same time. This is because the microcontroller communicates with the computer, and it is easier to implement the communication at the same time. This task was thought to take one month and a half to be finished.

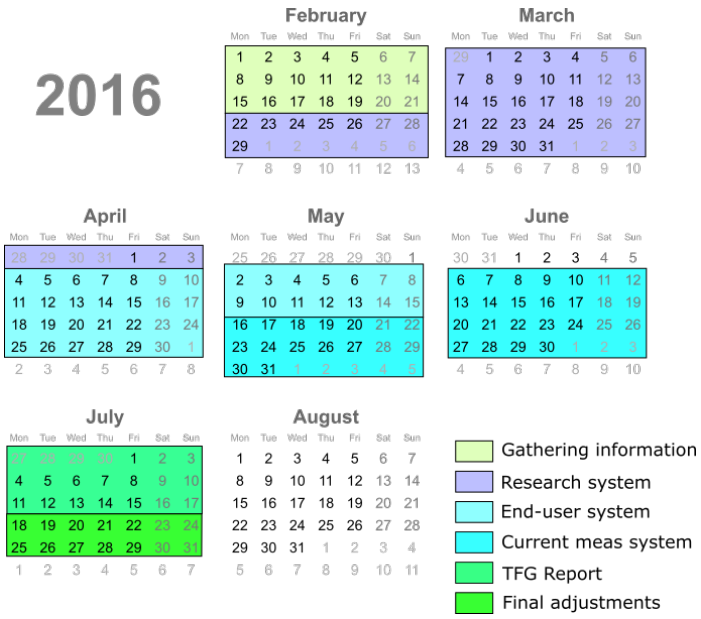


Figure 16: Task distribution

3. **Develop the end-user measuring system** After checking that the algorithms could work, we implemented the code on the microcontroller. This task is harder than the previous one, however as we knew how to implement it, we thought it was going to take one month and a half as well.

4. **Create the evaluation system** The evaluation system was created to verify the current reduction thanks to the use of the algorithms. We thought that it was going to take just a month to complete, however it ended up taking one month and a half due to some problems we faced as we will go through later in Chapter 6.

5. **TFG report** This task took half of the month of July.

6. **Final tuning** The last half of July was reserved to troubleshoot the small problems that could appear in the software as well as to clean the code.

In Figure 16 we can appreciate the task distribution along time.

Chapter 5. Implementation

The implementation of the project is the part that took most time to complete. As commented in the Objectives section, two different projects were carried out to ease the process of implementation. The main objective is to implement the power reduction algorithms directly on the microcontroller. However coding it directly on the microcontroller without even testing if it could be done was not a good idea. For this reason we started with a system in which the PPG signal was gathered with the sensor and the microcontroller and then processed in Matlab. In this first approach Matlab was also responsible for sending the necessary commands to the sensor through the microcontroller in order to change its parameters as the algorithms dictate. As Matlab is a user friendly environment to process signals it is easier to implement the peak detection algorithms as well as the power reduction algorithms.This setup will be called *Research measuring system* as it could not be used in and end-user product.

Once we checked that the algorithms could work, another system was developed. In this system we also have the sensor connected to the microcontroller, and this microcontroller to another Matlab GUI. This time all the processing is carried out by the microcontroller. The computer application will just plot the PPG signal and other important information. This means that the microcontroller could run without needing any device attached, however to check that the algorithm is working properly we will watch the output of the sensor and the algorithm on the computer. This setup will be called *End-user measuring system* as it could be used in and end-user product.

In the following sections the development process of these two systems will be covered. As the *End-user measuring system* has many things in common with the *Research measuring system*, some redundant information may be omitted.

5.1 Research measuring system

The research measuring system was developed to test if the algorithms could be implemented in a real scenario. The measuring systems have two main different parts: the microcontroller programming and the Matlab GUI. We developed both parts in parallel, because this way it is easier to develop the communication protocol between the computer and the microcontroller board. However before starting coding we had to understand the hardware we were going to deal with.

5.1.1 Getting to know the hardware

The sensor we are using is embedded on an evaluation board that helps us to setup the measuring process. It features a connector that carries the lines of the I2C bus. We will connect this board to the microcontroller with this socket. The sensor itself is on the right side of the board, as can be seen in Figure 17. The ADPD is an IC chip, however in the photo it is covered with a transparent window with a black frame that is meant to improve the measurement quality. Nevertheless the sensor is behind this window and can be seen in Figure 18. We have highlighted the photodiodes and the LEDs locations on the image.

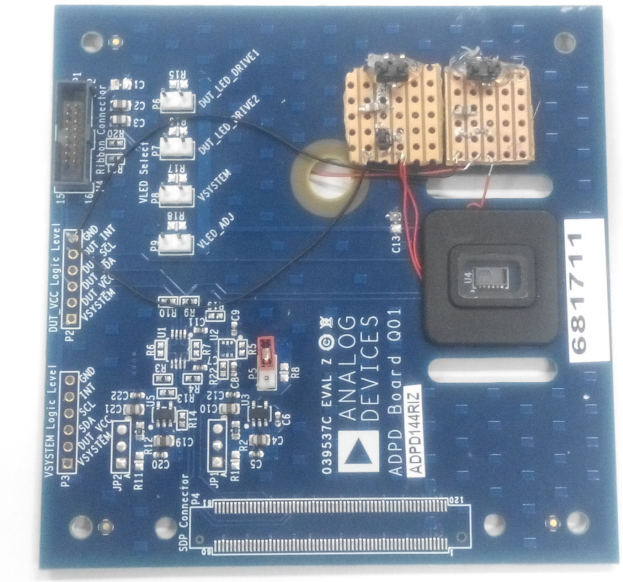


Figure 17: Sensor board photo

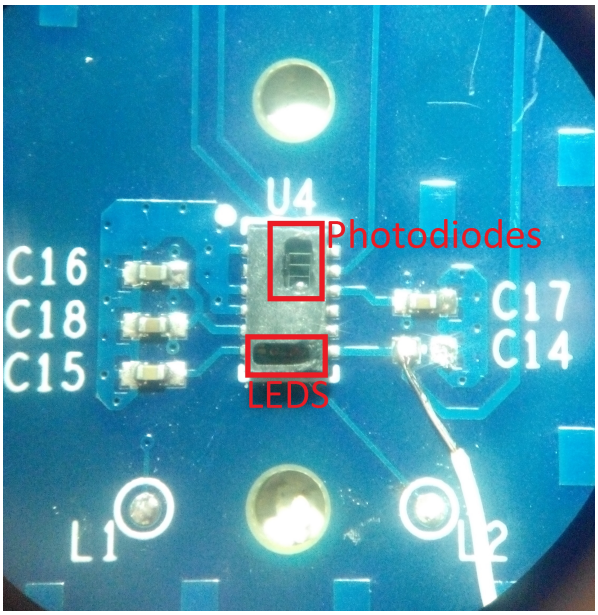


Figure 18: ADPD sensor detail

In order to establish the communication with the sensor, an I2C capable microcontroller is needed. For this reason we looked for a microcontroller development board. We chose an Analog Devices development board with a Cortex-M4 microcontroller. These are the most important characteristics of the board:

- **Cortex-M4 microcontroller** This is the board's main component. It is a ST32F405 microcontroller. It features 32bits, 168MHz clock, 140 I/O ports, 15 communication interfaces (3xI2C, 4xUSART, 2xUART, 3xSPI,2xCAN,2xSDIO), USB OTG HS/FS, 3 DAC, 2 ADC, Ethernet, TRNG.

- **FTDI** This is a USB to serial UART interface that will allow us to communicate the microcontroller with the computer.

- **Bluetooth** This is a fully integrated Bluetooth 2.1 + EDR class 2 module. It is connected to the microcontroller through a UART interface.

- **Breakout socket** This is the socket we will use to connect the sensor with the microcontroller. It has 14 pins with the necessary pins to establish the I2C communication. It also has the supply pins to feed the elements on the sensor board. Apart from these pins there are some other routed to the microcontroller. Analog Devices has different sensor boards and some of these have components hooked to a SPI interface. The spare PINs are thought to hold the SPI lines.

- **Power System** The board may run powered by a USB or by a battery. This system feeds all the integrated circuits. It has an Analog Devices AD5061 IC that handles the charging process and the current source for the rest of the board. It is also connected to the microcontroller through an I2C interface that enables us to change the output current or to start/stop charging the battery. We will not use this feature as we will focus on obtaining the S_pO_2.

- **Level shifters** The output voltage of the lines of the interfaces are fixed to 3.3V by the microcontroller. However the ADPD sensor works with a 1.8V signaling. This is why we need level-shifters to lower the voltage in the communication between the sensor and the ST chip.

There are some other features available on the board we will not use, such as the micro SD Card reader or some other breakout sockets. A photo of the board can be seen in Figure 19

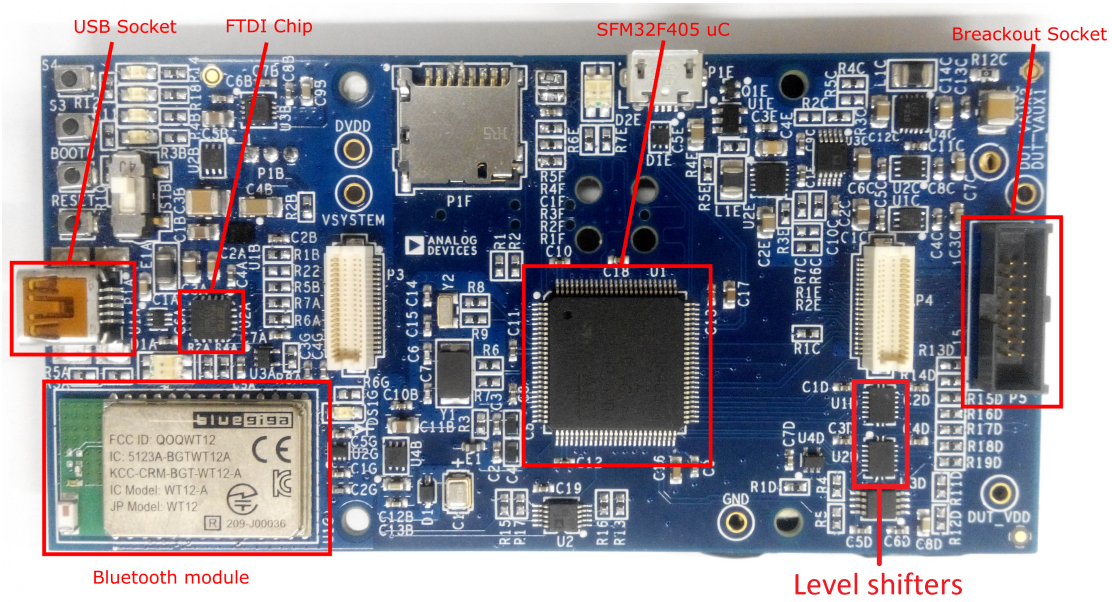


Figure 19: Development board and its components

Apart from this board and the sensor board, in some parts of this project we have also used a breakout board of the cable that communicates the microcontroller with the sensor. This board can be seen in Figure 20, where we have highlighted with the red rectangle the part of the board we will use. The aim of this circuit is to help us to debug the signals of the I2C interface. It makes it easy to hook an oscilloscope probe to any line and watch its response. It has been used when we have faced problems when reading the sensors registers and samples. The most common problem that this board helps to detect is the

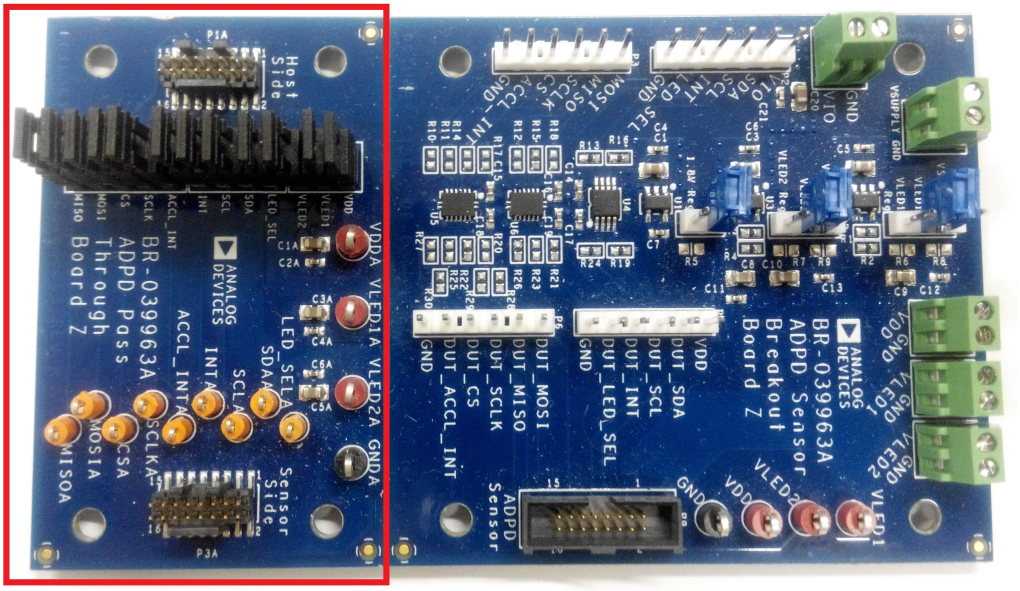


Figure 20: Development board and its components

overrun of the FIFO of the sensor, that happens when the samples are not read on time and they get lost.

We will be working with these three boards along with the computer to build the S_pO_2 power reduction measuring system. The way we will connect each other is shown in Figure 21 .The tasks of the different components are:

- **Computer** We will run in the computer a Matlab GUI. This Matlab GUI will ask the microcontroller board to receive the PPG signal as well as the sensor registers. It will also send register values to write them to the sensor. Apart from controlling the sensor through the microcontroller, it will process the incoming PPG samples. It will first filter them to obtain the peaks, then it will estimate where the next peak should take place, and with this information it will decide when to send the necessary commands in order to put the sensor in the LPAP and when to return to the HPAP.

- **Microcontroller Board** It has two main tasks. The first one is to listen to the UART interface for incoming commands from the computer and attend them. Some commands may require the microcontroller to send read or write requests to the sensor using I2C transactions. Its second main task is to receive incoming data from the ADPD and send it to the computer where it will be processed.

- **Breakout Board** The aim of this board is to help debugging errors related to the I2C communication or unusual sensor behaviour. It is useful to detect ADPD FIFO overruns due to excessive delays reading samples.

- **Sensor Board** This is the component that holds the ADPD sensor, which pulses light and samples the received light as specified with the Matlab GUI. Its main task is to attend the microcontroller requests, whether they are register read or register write commands.

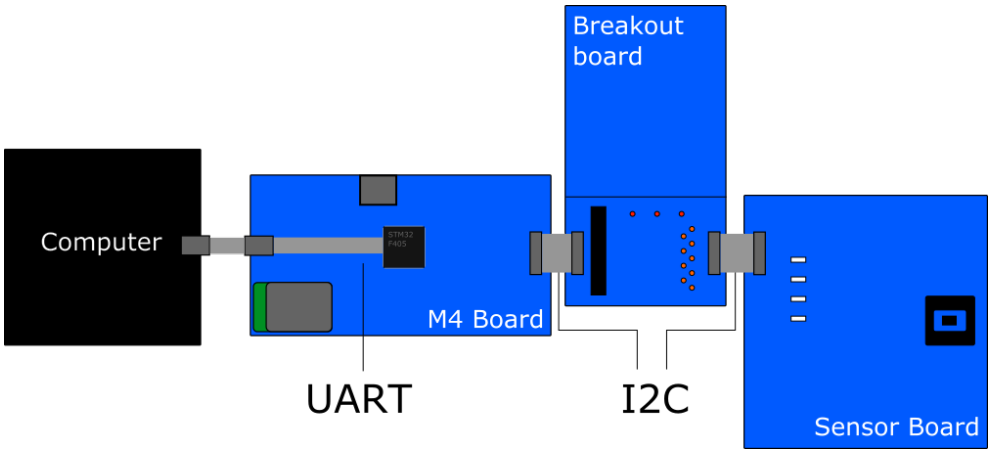


Figure 21: Measurement setup and connection among the boards

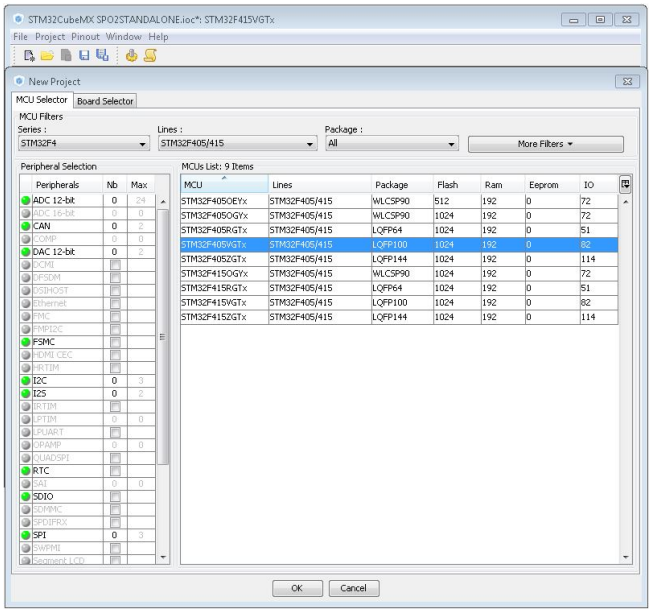


Figure 22: CubeMX project setup

5.1.2 Initial project development

Once we have the hardware setup we can start developing the software. We will start going through the process of creating the microcontroller software and then we will move on to the Matlab GUI. When programming these types of chips it is rather useful to start with a template or an example. In this case the manufacturer of the chip (ST) provides an application to ease developers work in the early stage. It is called STM32CubeMX. It is a graphical software that generates the basic code as specified by the user using wizards. This way we can easily configure the pinout, the interfaces and the clock configuration. To start with, we open the program and create a new project choosing *File-New Project...* from the top menu. Then we select the processor of our board *STM32F405VGTx* as shown in Figure 22 .Then we click *OK*.

A screen with the microcontroller icon will appear. In this view we can configure the pinout of the microcontroller. In order to configure the pin routing we have to look at the microcontroller board schematic summarized in Figure 23. We will tell the ST program to configure the pins of each module with the corresponding ones from the microcontroller.

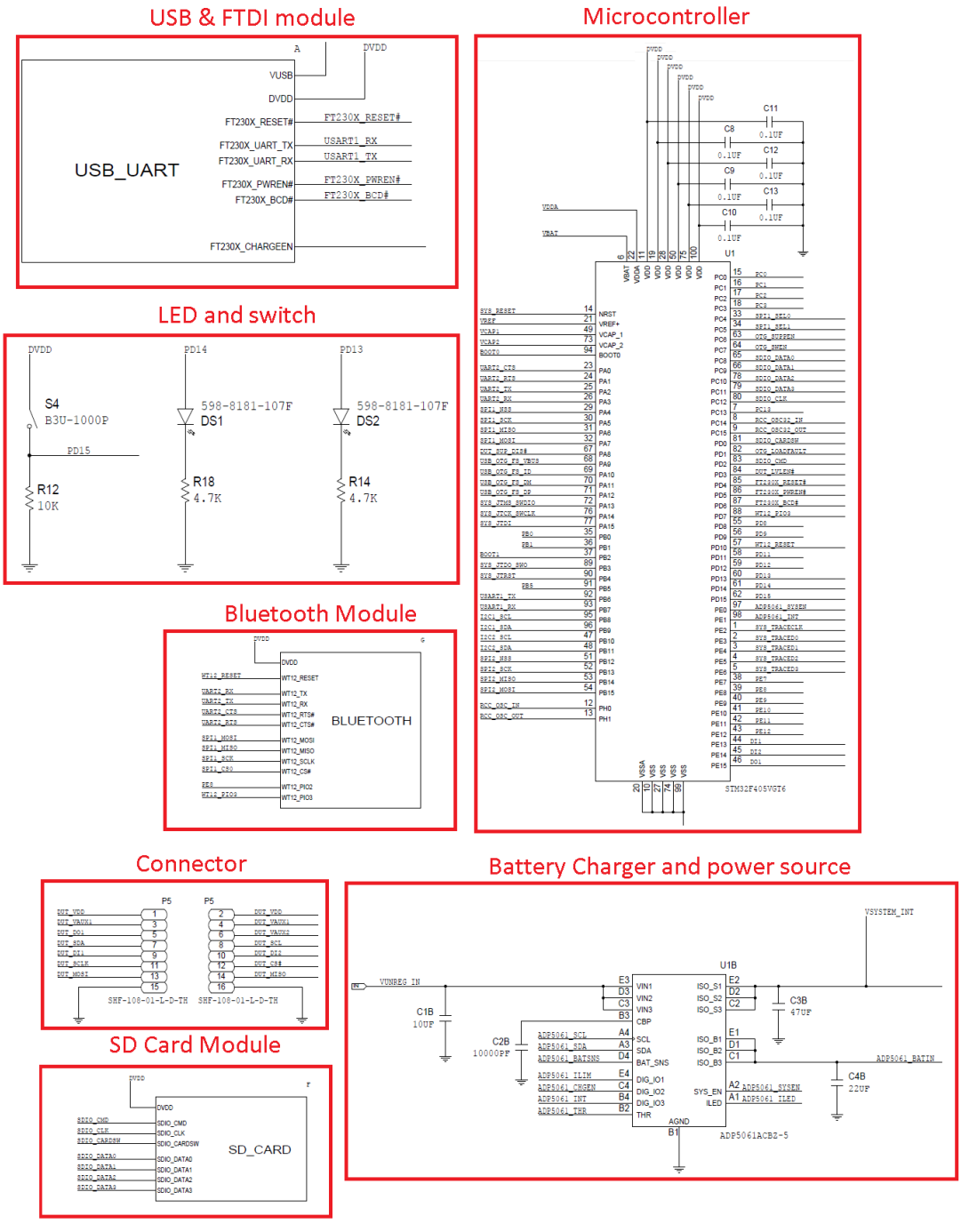


Figure 23: Most important sections of the microcontroller board schematic

Let's enumerate the devices and pins we have to connect so that the program can work properly:

1. **FTDI** For the UART communication with the computer. It uses two pins, UART1TX and UART1RX, named PB6 and PB7 on the microcontroller.

2. **ADPD** For the I2C communication with the sensor. It uses two pins for the I2C interface and 1 interruption pin. SDA and SCL are connected to PB11 and PB10 of the microcontroller, whereas the ADPD interruption pin will be connected to pin PE13.

3. **GPIOs** Some GPIOS will also be needed. We will enable two of them to light 2 LEDs to use them as debug flags. These LEDs are routed to pins R18 and R14. We will also configure a push button just in case we want to use it at some point. It is physically connected to pin PD15

4. **Bluetooth** Bluetooth is used in one section of the project. For this reason we will enable it. It connects to the microcontroller using a UART interface with hardware flow control. This means that it requires 4 pins: UART2TX, UART2RX, UART2CTS and UART2RTS. These two last pins stand for *Clear To Send* and *Ready To Send* and are meant to implement the hardware control flow. They are physically connected to PA2, PA3, PA0, and PA1 respectively.

When we started with the project we did not know which features of the board we were going to use. At first we thought that it could be a good idea to configure the SDCard just in case we would want to log data. We also thought that it could be interesting to setup the battery controller chip. Moreover Analog Devices has developed some other sensor boards that have different features. Some of them have an SPI accelerometer. We thought that in a future we may use a different sensor board, and for this reason we configured the accelerometer so that in a future it could be used effortlessly. For these reasons apart from the pins commented before we configured these ones:

1. **ADP5061** This is the chip that controls the voltage source and the one that charges the battery. It is connected to the microcontroller through an I2C interface. It uses two pins for this bus and two other pins configured as GPIOs for the interruption and the supply enable. They are physically connected to pins PB9, PB8, PE0, and PE1.

2. **SDCARD** The interface that connects the SDCard with the microcontroller is the SDIO. It uses 4 pins for data (Data[0:3]) one pin for the clock signal and another pin to detect Card presence. We will connect them to pins PC8-11 for the Data pins, PC12 for the clock and PD0 for the card presence detection.

3. **SPI interface** We also setup the SPI Interface. It uses 4 lines. We will connect MISO to PB14, MOSI to PB15, SCK to pin PB13 and NSS(Chip Select) to pin 12. As this is thought to work with the ADXL362 accelerometer present in some sensor boards we will also need an additional pin for its interruption. This line is routed from the connector socket to pin PE14 of the microcontroller.

Now that we know which pins need to be connected we can set them up within the application. Some of them will be connected as GPIOs and some other as interface lines. Finally, we configured the clock input so that the microcontroller runs with the external clock provided by the board. Once all the pins have been configured as shown in Figure

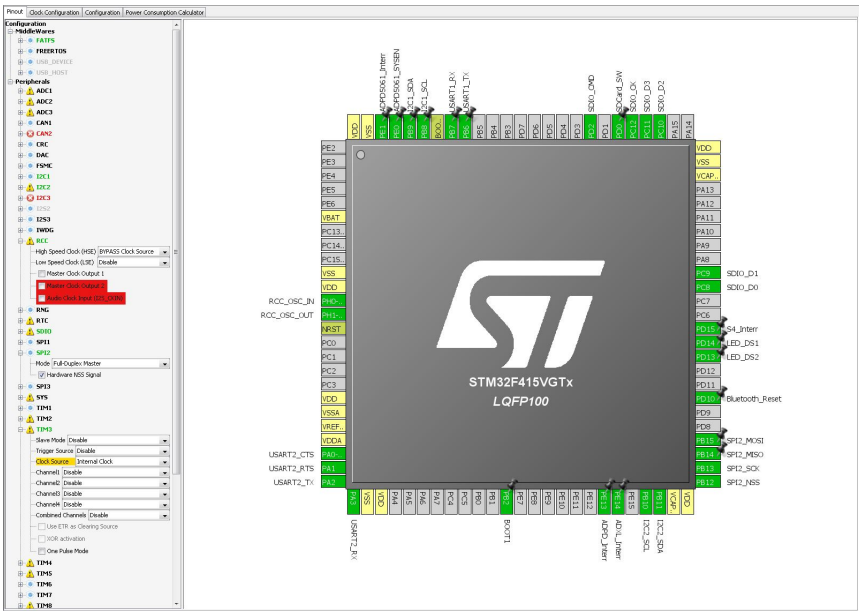


Figure 24: Microcontroller pinout selection

24 we need to enable the hardware chip select of the configured SPI, bypass the internal clock of the microprocessor, and enable the timer TIM3, as can be seen in the left column of Figure 24. At first we thought that having a timer could be very handy to measure times between peaks, but it is finally used in the *End-user measuring system*.

The next setting we have to configure is the microcontroller clock. To do so we click on the *Clock configuration* tab at the top of the program. As we have seen before, the main clock source of the microcontroller is the external oscillator located on the board. This clock signal will go through different PLL prescalers that will feed the different clocks of the modules of the microcontroller.

To assure the best performance we will set the main clock frequency (HCLK) to its maximum, 168 MHz. Before setting this value we will have to set the PPL Source to HSE (External Clock). Then we will set the external clock frequency that according to the board schematic is 8MHz as shown in Figure 25.

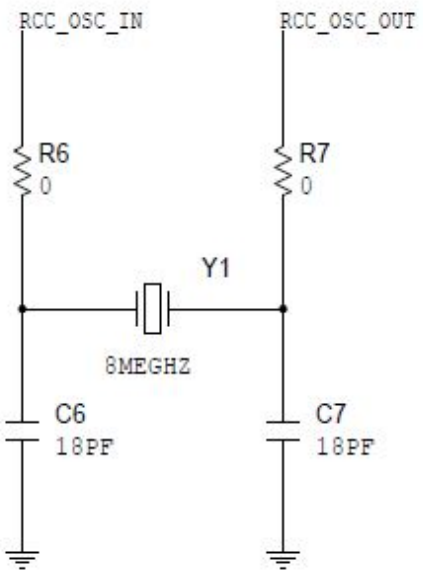


Figure 25: Microcontroller external clock value

Then we will set the peripherals clock to its maximum value. The battery charger (Interface I2C1), the Bluetooth module (Interface UART2), the ADPD Sensor (Interface I2C2), the accelerometer (Interface SPI2) and the SDCard (Interface SDIO) work with the APB1 [7] clock, whereas the FTDI (Interface UART1) works with the APB2 Clock. These clock sources have a maximum values of 42MHz for the APB1 and 84MHz for the APB2 as we can see in Figure 26. We will setup these clocks to its maximum frequencies. When setting up these values the program automatically calculates the value of the PPL's so that we obtain the desired frequencies. The resulting clock configuration is shown in Figure 27

We have almost finished with the project setup. However we still have to configure the interfaces, the timers, and the interruption vector. To do so we will switch to the *Configuration* tab at the top of the program. There a table with the enabled interfaces, the timer, the GPIOs, the NVIC[8] among other configurable elements will turn up. We will click on the NVIC button and we will enable:

1. **System Tick Timer** We will need this interruption for the timer. It will be called when the timer internal counter reaches its maximum value.
2. **EXTI line1 interrupt** We need this interruption for the AD5061 battery charger chip. Although we configured it was not used in this project.
3. **USART1 global interrupt** This interruption will be called when there is a new event regarding USART1 communication. This is the interface that communicates the microcontroller with the computer through the USB cable.
4. **USART2 global interrupt** This interruption will be called when there is a new event regarding USART1 communication. This is the interface that communicates the microcontroller with the computer through Bluetooth.
5. **EXTI line[15:10] interrupts** Used for the ADPD, push button and SPI interrupts.

[7] Advanced Peripheral Bus 1
[8] Nested Vectored Interruption Controller

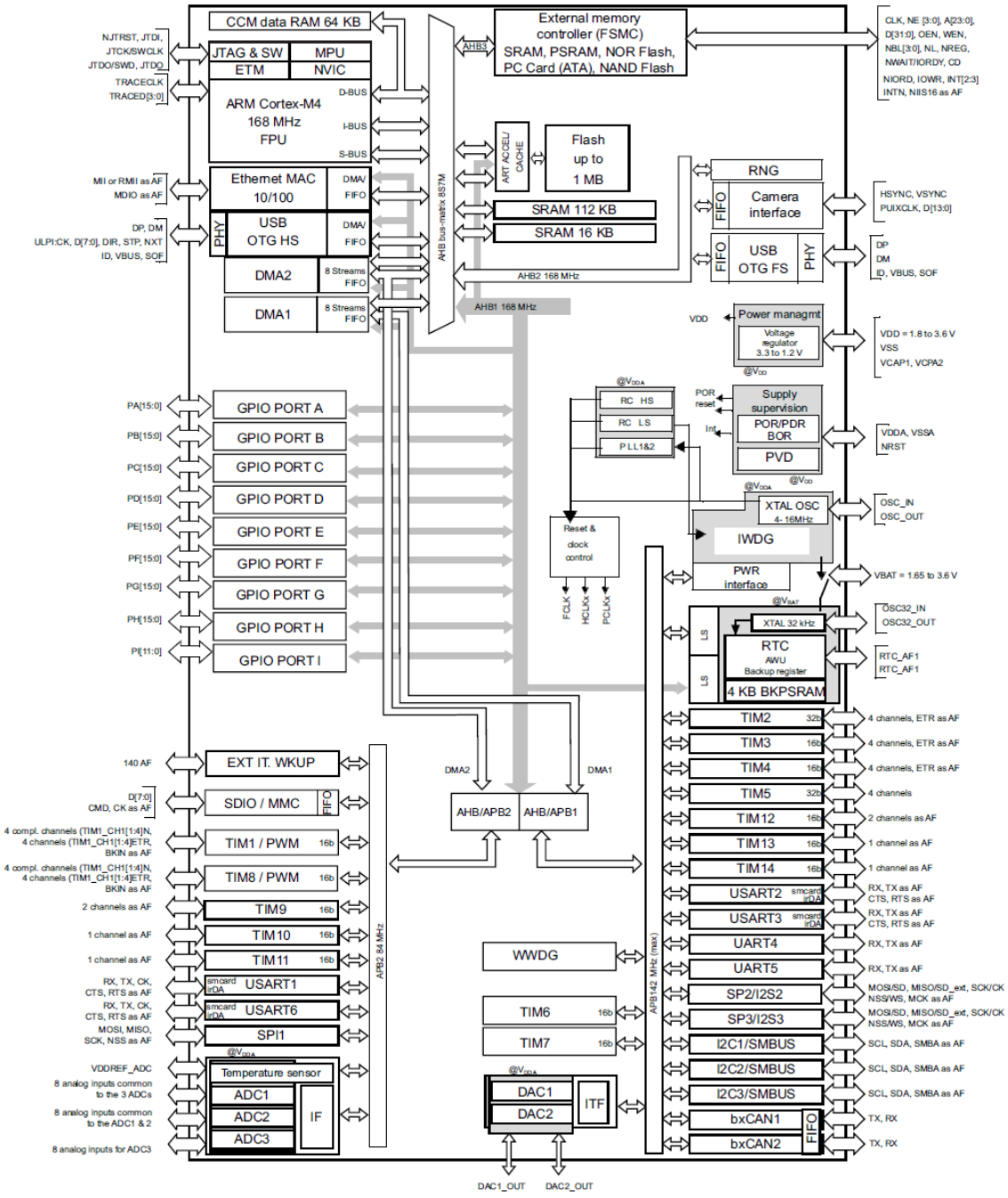


Figure 26: Microcontroller block diagram

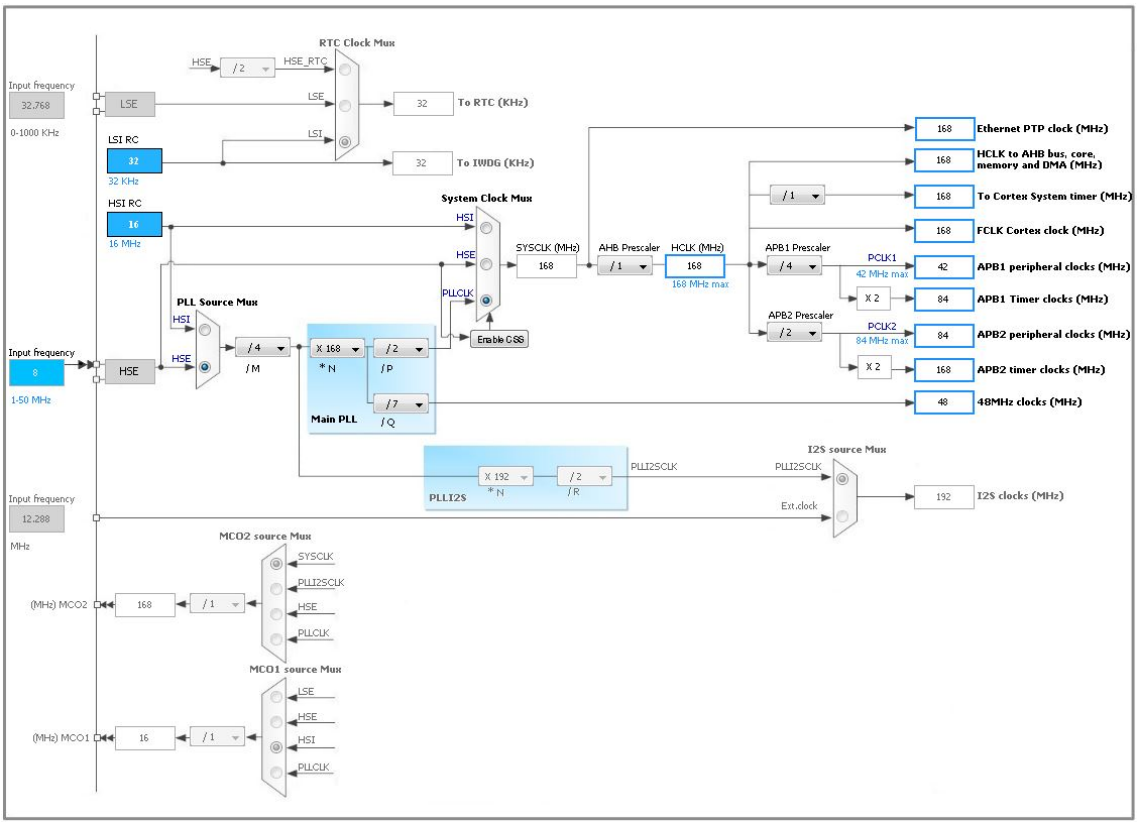


Figure 27: Microcontroller clock configuration

Once we have configured the NVIC, the will move to the I2C interfaces. We will set the clock speed of their communications to 400kHz (the maximum allowed by the protocol) because both the ADP5061 and the ADPD sensor can run at this frequency as shown on their datasheets. The rest of the parameters will be left as they are set by default.

Then we will configure the UART interfaces. We will set the baud rate of both to 921600 Bits/s, which is the maximum value allowed by the computer.

We will leave the configuration of the SPI as it is by default. The timer will not be used for this project setup, however it will be used in the *End-user measuring system*.

The next thing to do next is to generate the code. Before generating it we have to adjust the generation parameters clicking on *Project-Settings*. In the *Project* tab we will set where to store the code. Then in the *Code Generator* Tab we will enable these options:

1. **Copy all used libraries into the project folder** This will copy all the necessary files of the processor hardware abstraction layer to the project. This way we will be able to compile the project in another computer without installing the drivers of the processor.

2. **Generate peripheral initialization as a pair of '.c/.h' files per IP** This way the code will be more organized as it will generate a pair of files per interface/module initialization.

3. **Keep user code when re-generated** This is a feature we will not use very often. It is useful when you want to modify the project configuration and you want to keep you code untouched. The problem is that it restricts the zones of the code you can write, this is why we did not use this function. Whenever we wanted to modify

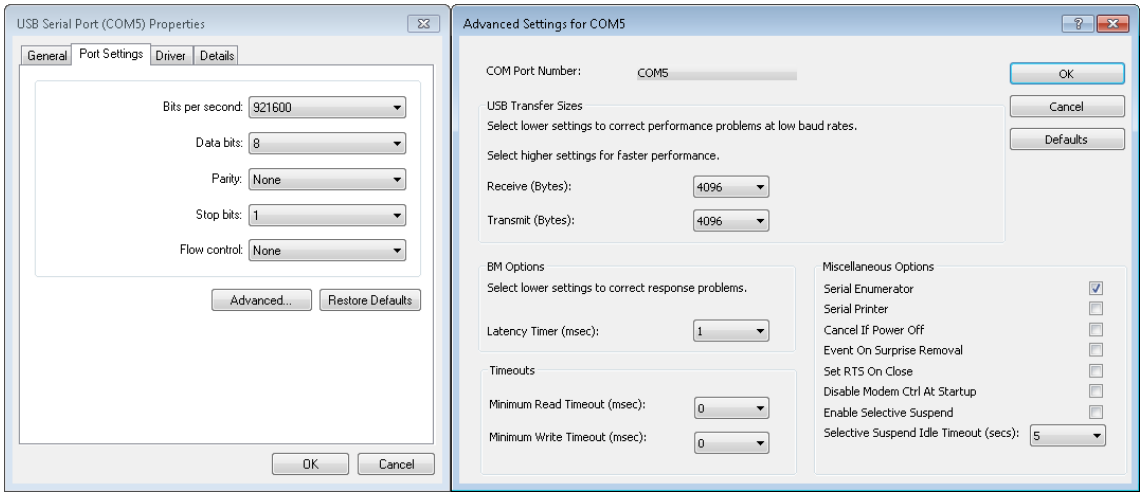


Figure 28: Serial port configuration

something we changed it directly from the code.

Once we have finished we click *OK* and we can finally generate the code clicking on *Project-Generate Code*.

5.1.3 UART Communication development

Let's review how the program has been developed. We will divide the microcontroller project into three main parts: Computer communication, Sensor communication, and Main loop.

The computer communication is maybe one of the trickiest parts. There are some requirements that have to be accomplished to make it work properly. One of the main characteristics is that the communication is bidirectional, which means that both the computer and the microcontroller will be able to send data to the other device.

The computer side is easier to implement for several reasons. The first one is that we will code it on Matlab, and with a couple of functions we will achieve it. The second one is that a computer has more memory flexibility, its memory is bigger and there is no need to optimize every single array. The third one is that receiver and transmitter buffer sizes can be set up to 4096 bytes in the computer driver as shown in Figure 28, which means that the odds of loosing data are low as we have enough time to read arrived bytes before it gets filled and a FIFO overrun occurs. However the microcontroller side is much complex[4]. The memory available is limited and the FIFO is a small shift register, which means that we have to be careful in order to avoid FIFO overruns and therefore data loss.

At first we thought to send ADPD samples as raw data through the interface. However we realized that this approach had some drawbacks. The main one is that if several commands are sent and the microcontroller answers to these commands, we cannot know which response corresponds to each command sent. This problem can be clearly seen if we imagine a situation in which we want to read a register while gathering the PPG Signal. As the PPG Signal would be sent in raw data format, we could not distinguish the response among all the other bytes. For this reason we decided that a packet structure should be implemented. This way each request and response would have a header defining the type of packet and its content length. With this approach we could distinguish different types of responses. This means that the PPG signal cannot be sent as raw data. At least we

Byte	0	1	2	3	4	5	6	7-255
Content	Packet detection			Packet number		Packet length	Packet type	Data
Value	0x53	0xF0	0x0C	0-65536		0-121	0-255	...

Table 1: Packet header and Data

have to introduce the raw data in packets before sending them.

The designed packet structure is rather simple. It has a header identification to detect a new packet, two packet identification bytes, a packet length byte and two bytes to identify the number of packet of PPG signal. In Table 1 we can see the packet structure. Lets have a look at each header element:

1. **Packet detection sequence** These are three bytes that define the beginning of a packet. We will use them to detect new packet arrivals. We chose three bytes with a big difference among its values. This way it would be rather difficult to misunderstand PPG samples with the same three bytes because PPG signal does not have such abrupt amplitude changes.

2. **Packet number** This field is used when sending PPG samples. It identifies the packet with a packet counter number. This way we can know if a packet has got lost, which is especially useful when working at high sampling frequencies, as the odds of loosing packets increase.

3. **Packet length** This field is made up by a single byte that specifies the size of the data carried by the packet. This value can be any number between 0 and 121. The maximum value is 121 because the maximum size of the complete packet is 128. Therefore if we add the packet header size to 121 we will obtain 128

4. **Packet type** This is the one byte packet identification field. There are different packet types as we will explain later. Some of them may be a "read request" packet or a "PPG Data" packet. It is used when attending the command.

5. **Data** Array of bytes with the data carried by the packet. The maximum array size is 121.

The maximum packet size is 128. At first we set this value to 16 bytes. It was enough to store one sample. However we realized that for some commands we were going to need more space. The command Multi-Write is a good example. This command sends several register numbers along with their value to write to the sensor.

Now that we have defined the packet structure we can start designing the code that will receive the bytes from the computer and that will attend the commands. We will define two code pieces that will handle the UART communication. In the first place we have the UART interruption handler. This piece of code will receive data and put it in a matrix buffer. Then the *attendCommand* function will read this buffer and process the commands sent by the computer. We have divided the code this way for a reason. The amount of time that we are inside the interruption should be as small as possible. Otherwise we could run into interruption overruns. For this reason the UART interruption handler function will just receive the data and put it into the matrix. the *attendCommand* function however

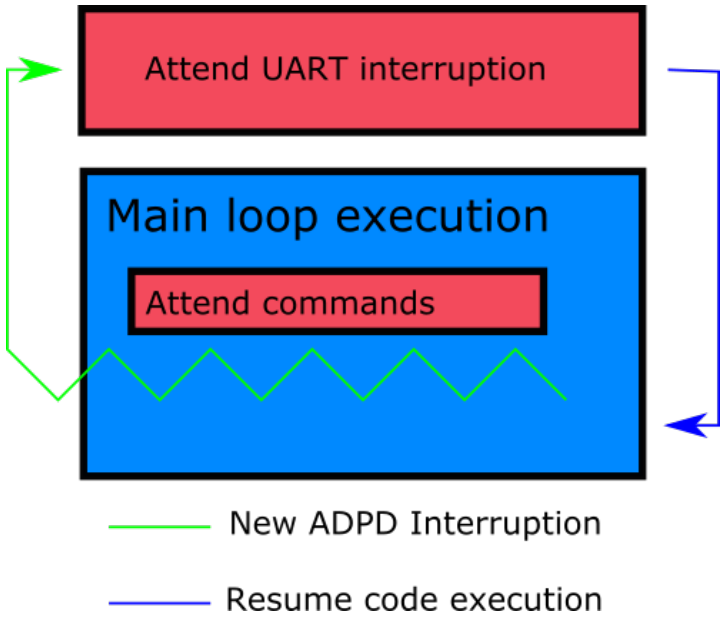


Figure 29: Main execution loop and UART interruption

can take more time to execute as it will run in the main loop and it can be interrupted at any time. This working procedure can be seen in Figure 29.

The buffer is a 16x128 bytes matrix called *commandStack* that stores the commands until they are read. This matrix can store up to 16 commands, because each row is reserved for a single command. This approach is not optimized, because in case the command's size is smaller than 128 bytes we will be wasting memory. However we wanted a quick and safe way to store the commands because this project was not the final one, and the optimization was not that important, in the *End-user measuring system* instead of a matrix we use a circular buffer as we will explain in that section.

Let's have a look at how the main UART pieces of code are designed

UART interruption Handler function *HAL_UART_RxCpltCallback* in the code. The task of this function is to receive the incoming data from the computer and store it on the commandStack matrix. The microcontroller calls this interruption handler as soon as there is a certain number of bytes available in the FIFO of the UART interface. The number of bytes necessary to be called has to be configured, otherwise it will never be called. Moreover every time it is called it has to be reconfigured so that it can be called the next time. At first we configured this value to 1, which means that the interruption would be called each time a byte from the computer arrived. However this turned out to be a bad idea due to the high baud rate we set. The main problem we had is that the processor was not fast enough to call the interruption and set again the interruption for the next byte, and eventually a sample could be lost. The code of this approach can be seen summarized in Code 1. As we can see in line 7 we start the reception of data asking the UART HAL[9] to raise the interruption when a byte arrives with the function *HAL_UART_Receive_IT*. The two first arguments of this function are the HAL UART handler and the array where the incoming bytes will be stored. Then, inside the function once the byte is stored in the commandStack we configure again the interruption(line 18)

```
1 uint8_t response[128];  //Temp array to store data
2 UART_HandleTypeDef * huartOut;  //UART Handler
3
```

[9]Hardware Abstraction Layer

```
4  int main(void)
5  {
6    HAL_UART_Init(&huartout);
7    HAL_UART_Receive_IT(huartOut,response,1);
8    //Main for loop
9    while(1)
10   {
11     attendCommandStack();
12   }
13 }
14
15 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle)
16 {
17   addArrivedByteToCommandStack();
18   HAL_UART_Receive_IT(huartOut,response,1);
19 }
```

Code 1: First UART handling attempt

However as this approach did not work properly, we thought in another way of reading the data. The best solution was to read the entire header first and then the rest of the data. This is possible thanks to the packet length field we created. Therefore we will start asking the HAL to raise the interrupt once 7 bytes of data have arrived. Then, once inside the interruption we will read the number of expected bytes and then we will configure the interruption to raise when these bytes have been read. This code is summarized in Code 2. Here we can see in line 8 that we ask the HAL to wait for 7 bytes, and once in the interruption function depending on whether we are waiting for another packet or for data we set it to wait for 7 bytes or the number of bytes of the packet (*response[5]*)

```
1  uint8_t response[128];  //Temp array to store data
2  UART_HandleTypeDef * huartOut;  //UART Handler
3  uint8_t NOW_READ_DATA;
4  int main(void)
5  {
6    NOW_READ_DATA=0:
7    HAL_UART_Init(&huartout);
8    HAL_UART_Receive_IT(huartOut,response,7);
9    //Main for loop
10   while(1)
11   {
12     attendCommandStack();
13   }
14 }
15
16 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle)
17 {
18   addArrivedByteToCommandStack();
19
20   if(NOW_READ_DATA){
21     HAL_UART_Receive_IT(huartOut,response,response[5]);
22     NOW_READ_DATA=0;
23   }
24   else{
25     HAL_UART_Receive_IT(huartOut,response,7);
26     NOW_READ_DATA=1;
27   }
28 }
```

Code 2: Second UART handling attempt

The code shown before is similar to the real one, however due to the size of the original it has been summarized. For this reason the part of the code that copies the bytes from the response array to the commandStack has been omitted. However we can give a brief idea of how it is managed. As specified before commands are stored in the rows of the command stack. The program knows where to store the next packet thanks to a pointer that indicates the next available row in the commandStack. This pointer is incremented each time a new packet is introduced. However as the commandStack is a *circular matrix*, once the pointer reaches the end of the command stack its value is set to zero. The same thing happens when reading from the commandStack. There is a pointer that indicates which is the next packet that should be read, and is also a circular pointer. These pointers are automatically managed by two functions, named *lastCmdAReceived* and *nextCmdPtr* respectively. These functions have to be called when putting a new command and when reading a new one, so that the pointers get updated. These functions also detect overruns of the FIFO. This happens when too many packets need to be attended but there is not enough time to attend them. In other words, this happens when the $nextAttendPtr - lastReceivedPtr > 16$. An illustration explaining the mechanism of the commandStack is shown in Figure 30

The command attention function The main objective of this function is to attend the commands sent by the computer. It is located in the main loop, so that it is executed in every iteration. It reads the command stack until there are no new commands to process. It has a switch structure in which depending on the command that needs to be attended it does one task or another. We can see this in Code 3. We go through all the new commands with the while loop, then depending on the type of the current command, we go into a certain task. The types of commands are declared as enumerations in order to maintain a clean program.

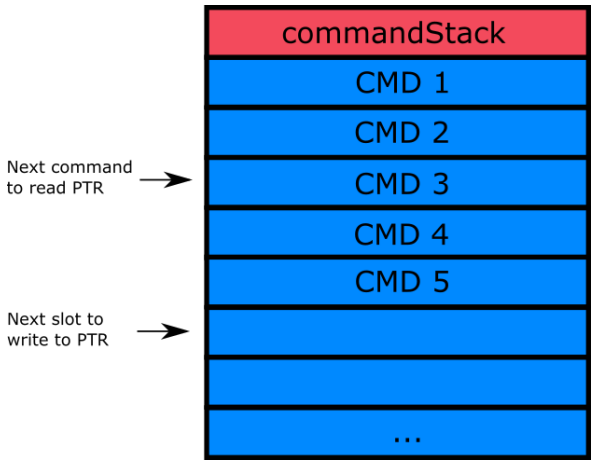


Figure 30: CommandStack working mechanism

```
1  void attendCommandStack()
2  {
3    while(nextAttendPtr<(lastReceivedPtr+16*samepage) &&
        nAdpdFifoLevelSize<100) //<100 to prevent fifo overrun
4    {
5      switch(commandStack[nextAttendPtr][3])
6      {
7        case WRITEPACKET:
8          ...
9          break;
10       case READREGISTER:
11         ...
12         break;
13     }
14     nextCmdPtr();
15   }
16 }
```

Code 3: Command stack attention summarized function

Now that we know what happens with the packets since they are sent until they are processed, lets have a look at some of the most important packet types we have been using in this project:

1. **STARTMEASUREMENT (0xC0)** This packet is used to start sampling the data with the ADPD sensor. As soon as this command arrives, the main loop goes into a state in which it writes the default register values to the ADPD and prepares the necessary variables to start gathering data.

2. **INIT (0xE2)** This packet is used as a flag so that the microcontroller knows when the computer wants to engage with it.

3. **STOP (0xE0)** This packet is used to stop the measurement and set the variables to their default value.

4. **RESTART (0xE2)** This packet puts the main loop in a state in which the program looks for flag packets to engage with the computer. It is used to "Restart" the program.

5. **WRITEPACKET (0xD0)** This packet is used to request to write a register of the ADPD. It contains the register and the value to be written.

6. **READREGISTER (0xD1)** This packet contains an ADPD register read request. It specifies the register number to be read.

7. **ADPDDATA (0xD2)** This packet contains samples of the PPG Signal sampled by the sensor.

8. **MULTIWRITEWITHONOFF (0xD4)** This packet contains several registers that have to be written in the ADPD sensor as well as its values. Prior to writing them the program has to set the ADPD sensor operation mode to Standby. Then, after writing the registers it has to return to the sampling operation mode. This command is useful when entering the LPAP and the HPAP as with one only command you can change all the settings you need, saving time.

9. **MULTIWRITEWITHOUTONOFF (0xD5)** This packet is similar to the previous one. The difference is that with this command the operation mode of the ADPD sensor is not changed.

This is basically the working flow that bytes sent from the computer go through until they are processed as packets. We will now go through the process of communicating with the ADPD sensor.

5.1.4 I2C Sensor Communication development

The interaction with the ADPD sensor is easier compared to the PC communication. One of the main reasons is that the I2C transactions are automatically handled by the HAL. We do not have to care about packet headings and number of bytes to read because this task is carried out by the HAL and the sensor's library.

To start with, we took the ADPD library and we had a look at its code in order to determine how to implement it in our project. Inside the main header file of the library we located the functions used to interact with the device. We have included some of the most important ones in Code 4.

```
1  /* Adpd control functions */
2  void Init(void);
3  int16_t AdpdDrvSoftReset(void);
4  int16_t AdpdDrvOpenDriver(void);
5  int16_t AdpdDrvCloseDriver(void);
6
7  /* Adpd register read and write functions */
8  int16_t AdpdDrvRegRead(uint16_t nAddr, uint16_t *pnData);
9  int16_t AdpdDrvRegWrite(uint16_t nAddr, uint16_t nRegValue);
10
11 /* Adpd slot selection and operation mode setting */
12 int16_t AdpdDrvSetOperationMode(uint8_t nOpMode);
13
14 /* Adpd read data from hardware fifo */
15 int16_t AdpdDrvReadFifoData(uint8_t *pnData, uint16_t
       nDataSetSize);
16 int16_t AdpdDrvGetParameter(AdpdCommandStruct eCommand, uint16_t
        *pnValue);
17
18 /* Adpd register interrupt callback */
19 void AdpdDrvDataReadyCallback(void (*pfAdpdDataReady)());
20 /* Adpd interruption handler */
21 void AdpdISR();
```

Code 4: List of functions declared in the ADPD Library

Here we can see that there are different functions to control the sensor's parameters without having to modify specific registers. This eases our task because we do not have to worry about which registers we should write in order to carry out a certain task, as there is a function that does this for us. There are also other functions such as the *AdpdDrvOpenDriver* or the *AdpdDrvCloseDriver* that handle the library status.

Apart from these functions there is a particular one rather important: the *AdpdDrvDataReadyCallback*. We will use this function to set the pointer to the function of our code that should be called whenever the ADPD interruption rises. This will be called within the *AdpdISR* function, as this is the interruption handler of the library.

ADPD Library
Wrapping functions
Our code
HAL functions

Table 2: Code hierarchy (Bottom = closest to hardware)

Now we know how to interact with the library. However we have not clarified how the library communicates with the ADPD sensor. As we can see in the functions *AdpdDrvRegWrite* and *AdpdDrvRegRead*, the library uses two external functions that we have to create in order to send and receive I2C commands from the sensor. These functions are called *ADPD_I2C_Transmit* and *ADPD_I2C_TxRx*. It is quite easy to create this two new functions, as we only have to call the I2C read and write functions of the HAL linking the arguments. Sometimes this is called in the programming argot library wrapping, i.e. defining functions that link the code with the bottom code layers. This wrapping functions can be seen in Code 5. In the Table 2 we can see the code hierarchy in our project once we add the library.

```
1  ADI_HAL_STATUS_t ADPD_I2C_Transmit(uint8_t * I2CData, uint16_t
       size)
2  {
3    return (ADI_HAL_STATUS_t) HAL_I2C_Master_Transmit(hi2c,
4      (uint16_t) ADPD_I2C_ADDRESS,
5      I2CData,
6      size,
7      (uint32_t) 100);
8  }
9
10 ADI_HAL_STATUS_t ADPD_I2C_TxRx(uint8_t *pTxData, uint8_t *
       pRxData, uint16_t RxSize)
11 {
12   return (ADI_HAL_STATUS_t ) HAL_I2C_Master_TxRx(hi2c,
13     (uint16_t) ADPD_I2C_ADDRESS,
14     pTxData,
15     pRxData,
16     RxSize,
17     (uint32_t)100);
18 }
```

Code 5: Wrapping functions

Once we put all this code together we can access the library functions and start configuring the sensor. Before going into the main loop we engage with the device and we perform a first-init configuration as shown in Code 6. In this code we can appreciate the work flow of this initialization. In first place we tell the wrapping functions which is the HAL I2C interface handler, so that when they are called by the library they can tell the HAL which interface it has to interact with. Then we have the function that opens the library. This function basically sets up the interruption configuration. Next we have the set callback function. As we can see in the code we have created a function called *ADPDInterruptionRise*. This will be the function that will handle the interruption in our code environment. The aim of the last function of this code is to configure the

sensors registers. To do so we ensure that the ADPD is in idle mode and then we call proceed with the register configuration. This function is made up by a for loop in which in each iteration a register is written by calling the function of the library.

```
1  /*Set I2C handle pointer to the wraping functions*/
2  ADPD_I2C_Interface_Init(&hi2c2);
3
4  /* Initialize ADPD Driver*/
5  AdpdDrvOpenDriver();
6
7  //Configure ADPD Interruption function
8  AdpdDrvDataReadyCallback(ADPDInterruptionRise);
9  //Write the sensor's default values
10 configure144BoardInitialRegisters();
```

Code 6: Sensor nitialization process

When it comes to the interruption, its work flow is shown in Figure 31. As soon as the interruption line rises, the uC notices it and calls a callback defined in our code. In this callback we call the Library's interruption function handler. This, among other things calls back the function of our code we defined with the *AdpdDrvDataReadyCallback*. We could actually call the *AdpdISR()* and the *ADPDInterruptionRise()* functions within out code's interruption callback, however this approach is mode adequate.

As we saw in Section 3.3 this interruption rises when there is a certain number of samples in the sensor's FIFO. The number of necessary samples to rise the line is defined in one of its registers. In our project, as operating in real time is a priority, we decided to configure the sensor so that it raises the interruption with each new sample.

This information gives us a brief idea of how the library has been configured and included in our code. Now that we are able to detect available samples when sampling lets have a look at how these samples are managed.

5.1.5 Main program function

Up to this point we have gone through the process of communicating with the ADPD sensor and the computer. However we still have to link somehow this communication so that the computer can access the sensor's information. This is done within the main function.

The main function has two different sections of code: Initialization and the loop.

Initialization This fragment of code boots up the HAL, the library, and sets the variables to their default values. To start with we call the *HAL_Init* function, that sets up the

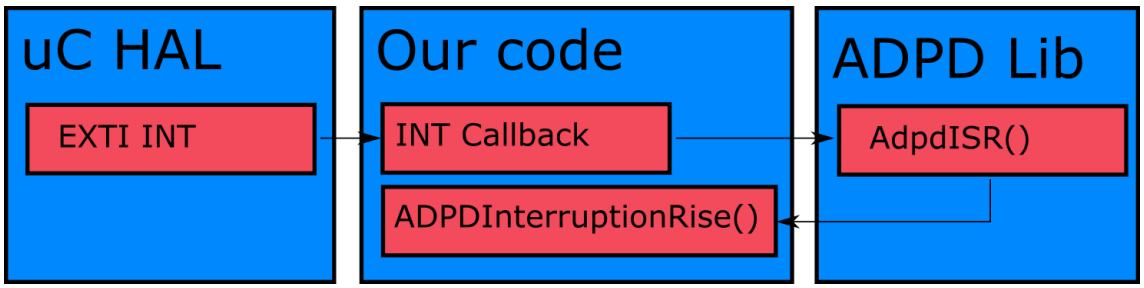


Figure 31: Interruption procedure

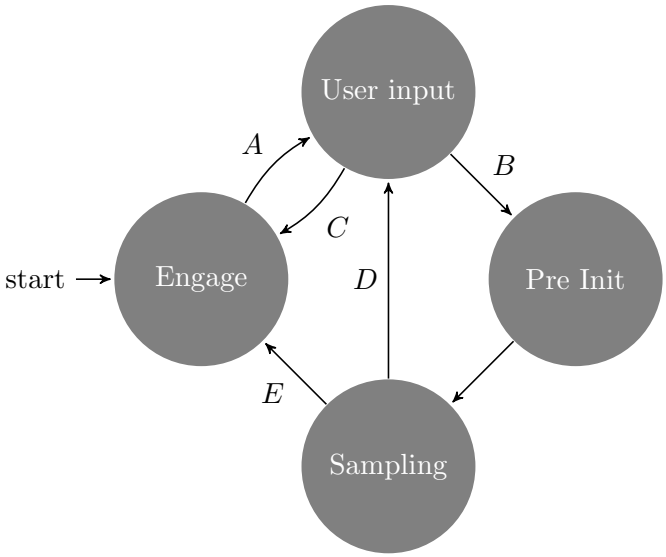


Figure 32: Main Loop Status machine

NVIC. Then we configure the clock with the *SystemClock_Config* function generated in our main file by the STCubeMX program. This one sets up the clock source as well as the PLL values. Once the bus clocks are set we can boot the interfaces. Most of them are initialized with this function prototype: *MX_[interface]_INIT();*. Once we have configured the interfaces we init the ADPD Sensor as specified in Code 6. Now that all the hardware is ready we set the necessary variables to its default value. In this section the *commandStack* memory allocation takes place.

Main Loop The main loop is the code that will run until the board is powered off and it implements a status machine that handles incoming events. This machine has four states:

1. **Engage** In this state we will wait until a packet of type INIT is received. Any other packet will be discarded
2. **User Input** In this state we will attend the commands sent by the computer. We will leave this state when a STARTMEASUREMENT packet arrives or then a RESTART packet arrives.
3. **Pre Init** In this state we prepare the ADPD registers to start measurement. We also call the ADPD Library function that starts the measurement process. As soon as this is done we move to the Sampling state.
4. **Sampling** This is the main state. In this one we will obtain samples from the ADPD sensor and send them to the computer. At the same time we will attend incoming packets from the computer.

As we can see in all the states we attend the user input. This status machine has been represented in Figure 32. This is achieved with the code summarized in Code 7. Here we can see the structure of the status machine. There is a while loop and in each iteration the status of the machine is checked. We can also appreciate in line 27 that whichever state we are in, we will always attend the user input.

```
1 //Status machine enum
2 typedef enum {
3 ENGAGE,
4 USERINPUT,
```

```
 5 PREINIT,
 6 SAMPLING,
 7 } programSTATUS;
 8
 9 while (1)
10 {
11  /* STATUS MACHINE */
12  switch(MSMStatus)
13  {
14    case ENGAGE:
15    ...
16    break;
17    case USERINPUT:
18    ...
19    break;
20    case PREINIT:
21    ...
22    break;
23    case SAMPLING:
24    ...
25    break;
26  }
27  attendCommand();
28 }
```

Code 7: Status machine simplified code

As we have commented before, the *Sampling* state is the most important one. In each iteration of the loop we check if the interruption flag has raised. If so, we ask the ADPD Sensor how many samples are available in its FIFO. Then, we read them all and reset the interruption flag. Before sending this information to the computer we have to normalize the samples. As we will modifying the number of pulses, we need to normalize the signal to a certain number of pulses so that when we change the number of pulses the signal does not have abrupt changes. As the maximum number of pulses of the ADPD is 256 we will normalize the samples so that the amplitude is the same as if we had 256 pulses[10]. To perform the normalization we have to multiply the samples by $256/ActualNumberOfPulses$.

All this process is carried out by the *readFIFOandPreparePacket* function. Once the outgoing buffer is filled with the new normalized samples, the packet heading is added and they are sent to the computer within the Sampling state.

5.1.6 Adding bluetooth connection

At some point of the evaluation procedure we thought that the USB port supply lines were too noisy and that this affected the performance of the ADPD sensor. PSRR of the TIA amplifier is not infinite, and therefore the output signal sampled is influenced by the supply.

As the board can run with a battery we proposed to establish the connection with the computer using a bluetooth dongle. This way the board would be isolated from the noisy computer supply. To do so we took the module driver developed by Analog Devices and we configured the UART interface to work with it.

[10] Actually this is not strictly true, as with more pulses we would have less noise

Integrating this bluetooth communication in our code was not complicated. The fact that both the FTDI and the bluetooth use UART interfaces helped, as the only thing we had to change in order to send or receive data through one or another interface was the handler. This way we can declare a pointer of type *UART_HandleTypeDef* that points to the UART handle we want to use(bluetooth or FTDI) in order to use it whenever we read or write. With this setup, changing the pointer means changing the communication path.

The only thing we have to do is detect whether the computer is connected through the USB cable or linked through bluetooth. To detect it one possible approach is to listen to both UART interfaces until we receive a INIT packet from one of them. We implemented this on the Engage status. Here we first check if there is any device linked to the bluetooth, if so we wait for an INIT command. If there is not a device connected then we use the FTDI UART. In Code 8 we can see a simplified code that shows how it is implemented. In line 2 we check if we have connected devices, and in lines 4 and 9 we set the UART pointer depending on which interface we connect to.

```
1  case ENGAGE:
2    if(checkConnectedDevices()==1)
3    {
4      huartOut = &huart2;
5      HAL_UART_Receive_IT(huartOut,response,7);
6    }
7    else
8    {
9      huartOut = &huart1;
10     HAL_UART_Receive_IT(huartOut,response,7);
11   }
12   HAL_UART_Transmit_IT(huartOut,prueba,sizeof(prueba));
13 break;
```

Code 8: Bluetooth / FTDI engaging state

5.1.7 Matlab application

Matlab is a useful tool when developing graphical applications in research projects. It has some limitations and is not as flexible as other programming languages, however it provides a wide variety of functions that can be easily used and easy function plotting. Moreover it has libraries and functions to communicate with external devices that come in very handy in projects such as this one.

With GUIDE (GUI Development Environment)[14] you can create graphical interfaces that run Matlab code. Different elements can be added to the window, each one can have different callbacks that will run when a certain event related to the element triggers. This way the user can interact with the created environment.

Our goal is to create a graphical environment with different purposes. The application must display the PPG signal in real time, as well as the sensors registers. It also has to display the most representative configuration as text, so that at any point the user can quickly know which is the active configuration. This application should also allow the user to freely change the register values of the sensor.

However the final objective of our GUI is to obtain the S_pO_2 value applying the power reduction techniques explained in Section 3.5. For this reason these algorithms will have to be implemented in this GUI. The user should be able to choose the algorithms as well as their most important parameters.

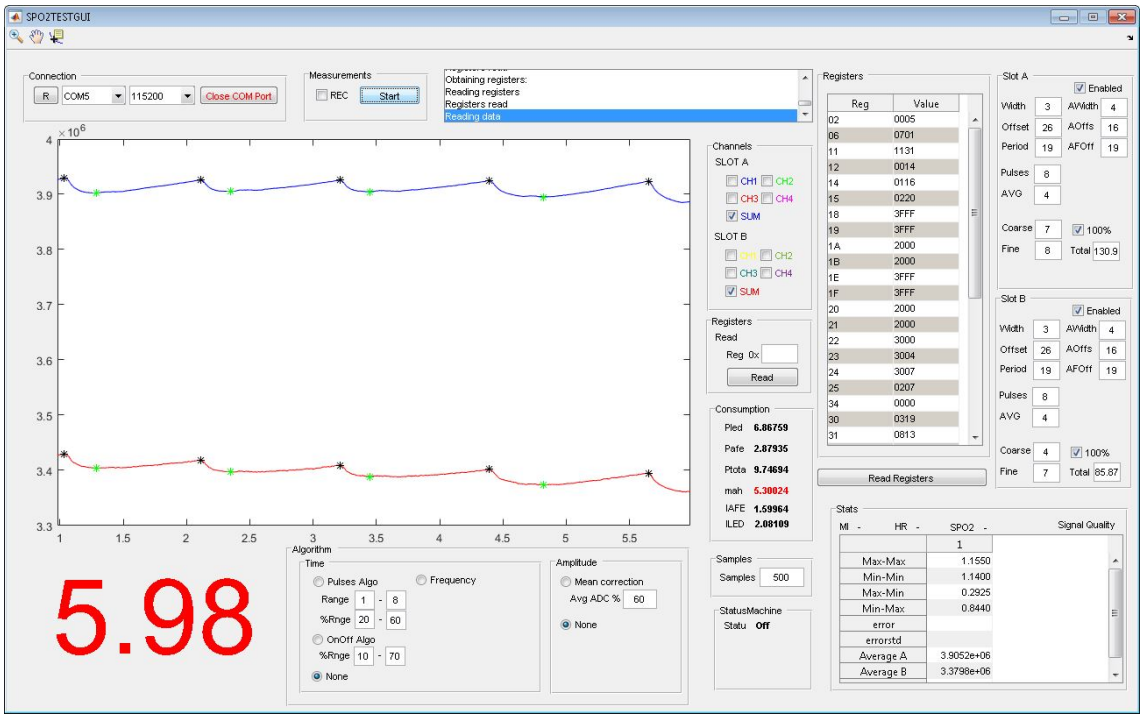


Figure 33: Matlab GUI for the research measuring system

Moreover the application should have a vital constant information area in which the S_pO_2 and the HR values would be represented.

All these features have been implemented in the Matlab GUI as shown in Figure 33. At the top of the figure we can see the connection configuration as well as a debugger information box. On the right hand side we have the register list with their values as well as the most important configuration written in text. In the center of the screen we have the PPG signal plot and the channel display selection. We also have in the middle section the consumption information. At the bottom side we can find the algorithm control boxes and on the bottom right hand we have the statistics and vital constant values. We will explain in this section a brief idea on how it has been done.

The application has a main while loop that runs as soon as the user pushes the Start Button. In the callback of this button we carry out several tasks: First we clean the variables of the Matlab GUI, then we set the ADPD initial configuration for the measurement, then we send the start measurement command, and finally we enter the while loop as we can see in Code 9. This while loop runs until the user hits the stop button. Inside the main loop we call several functions. Lets give a brief explanation of what do they do:

- **readUartIntoBuffer** This function reads the UART and stores the content in a temporal buffer.

- **parseBufferIntoCommandStack** This function takes the UART buffer and divides it into commands that are stored in the Matlab's command stack. This command stack is similar to the microcontroller's command stack. It is a matrix with 16 rows ready to receive 16 commands and 128 columns that will store the content of the commands.

- **processADPDData** This function is the one that processes the raw bytes of the

commands and stores each channel information of each time slot in buffers inside a struct.

- **plotADPDData** This function plots the ADPD data. It reads the previously created struct and draws the PPG Signal. It will show the channels specified by the user with the GUI. It is also possible to plot the summation of the 4 photodiode channels.

- **rangingStatusMachine** This function obtains the peaks and estimates the next peak location. Then it runs the power reduction algorithms thanks to the status machine.

- **plotStatistics** Plots statistical information such as the S_pO_2 value, the HR value, the average peak to peak period, the average min peak to max peak time, or the peak estimation error.

- **plotPeaksData** Plots the obtained peaks as well as the estimation peaks.

```
%Clear the application variables
handles = clearVariables(handles);

%Configure the registers
firstTimeBootRegisterConfig(handles);

%Send command to start measuring:
packet = hex2dec(['53';'F0';'0C';'00';'00';'00';'C0'])';
fwrite(handles.s,packet);

while(strcmp(get(hObject,'String'),'Stop'))
  %Lee datos y los deja en handles.rawUartBuffer
  handles = readUartIntoBuffer(handles);

  %Read handles.rawUartBuffer and store it in the commandStack
  handles = parseBufferIntoCommandStack(handles);

  %Read commandStack and store the data in the struct
  handles = processADPDData(handles);

  %Plot ADPD Data
  plotADPDData(handles);
  hold on

  %Obtain the peaks and store them in handles.peaks
  handles = rangingStatusMachine(handles);

  %Plot graphical statistics
  handles = plotStatistics(handles);

  %Plot the peaks
  plotPeaksData(handles);

  hold off

  %Force to draw
  drawnow;
end
```

Code 9: Matlab main loop

These functions carry out the necessary tasks to accomplish the specifications that the Matlab GUI needs to have. Now we will go through how some of the most interesting task have been implemented:

Communication with the microcontroller In order to communicate with the microcontroller we use *serial*, *fopen*, *fread* and *fwrite* functions. With the first one we will set the com port object settings, and then we will stablish the interface communication with the second command. To do so we will need the number of the COM port associated with the device as well as the configured bit rate. As we know we configured this value to 921600 bauds for the USB. In case we want to use the bluetooth communication the procedure is similar. The only difference is that we will use the COM port of the bluetooth dongle. We use command *fread* to obtain data from the microcontroller. In every main loop iteration, we call function *readUartIntoBuffer*, which reads all the available bytes in the COM port's FIFO. Finally, we use *fwrite* function to send bytes to the device. These bytes are packets that the microcontroller will interpret. A sample code with the basic initialization and usage of these commands can be seen in Code 10

```
1 %Stablish the connection with the device
2 handles.s = serial('COM5','BaudRate',460800,'DataBits',8);
3 fopen(handles.s);
4
5 %Write the initialization packet
6 packet = hex2dec(['53';'F0';'0C';'00';'00';'00';'E1'])';
7 fwrite(handles.s,packet);
8
9 %Read all the bytes in the COM port's FIFO
10 newData = handles.s.BytesAvailable;
11 handles.rawUartBuffer = [handles.rawUartBuffer, fread(handles.s,
      newData)'];
```

Code 10: Communication example with the microcontroller

Signal filtering In order to obtain the peaks we have to filter the signal as specified in section 3.4. To do so in Matlab we will use the *filter* function to filter the signal along with the *conv* function to convolute the filtering coefficients of the Comb filter and the LPF to obtain the total filter as shown in Code 11. After filtering the signal we look for the zeros with another function that takes into account the group delay and finds the peak in the original signal[12].

```
1 notch=5;
2
3 %Create the comb filter coefficients for a notch frequency of 5
      Hz
4 K = round(fs/(notch));
5 BcoefsComb = [1 zeros(1,K-1) -1];
6 AcoefsComb = 1;
7
8 %Create the LPF coefficients for a notch frequency of 5Hz
9 BcoefsLPF = ones(1,floor(fs/notch));
10 AcoefsLPF = length(BcoefsLPF);
11
12 %Lets convolute the coefficients to obtain the total filter
      coefficients
13 BcoefsTotal = conv(BcoefsComb,BcoefsLPF);
14 AcoefsTotal = conv(AcoefsComb,AcoefsLPF);
15
```

```
16 %Number of invalid samples due to filtering
17 notValidData =length(BcoefsTotal);
18
19 %Fitler the signal
20 signalInAFilt = filter(BcoefsTotal,AcoefsTotal,signalInA);
```

Code 11: Matlab signal filtering

Implementing the power reduction state machine Once we detect the peaks in the original signal we estimate where the next peak will take place. To do so we take into account the last 10 peak-to-peak periods and we weigh them in such a way that the newest period is the most representative one and the oldest one does not affect the estimation noticeably. The formula used to estimate the next peak is:

$$Peak_{i+1} = Peak_i + \frac{\sum_{j=0}^{9}(10-j)\,(Peak_{i-j} - Peak_{i-j-1})}{45} \tag{37}$$

We tried other different estimators such as exponential weighting ones or EMA among others, however the average estimation error was greater with these methods. Had it not been for this fact, we could have used 2^n weighting to reduce the computation time.

Once we have this estimation, we can enter the power reduction status machine. This is the portion of code where the algorithms will be implemented. Lets recall that the aim of these algorithms is to sample the PPG signal with more precision in those zones we are interested in, which are the peaks, while the rest of the signal does not give us information to obtain the S_pO_2 value and therefore we can sample with less precision.

At first several approaches were thought, and the most suitable solution was to create some sort of status machine to manage the algorithms. This state machine is executed once per main while loop. This way the normal main loop flow can continue. For this reason it is not strictly a status machine on its own, however it is part of the main loop. In the Figure 34 we can see the state machine diagram.

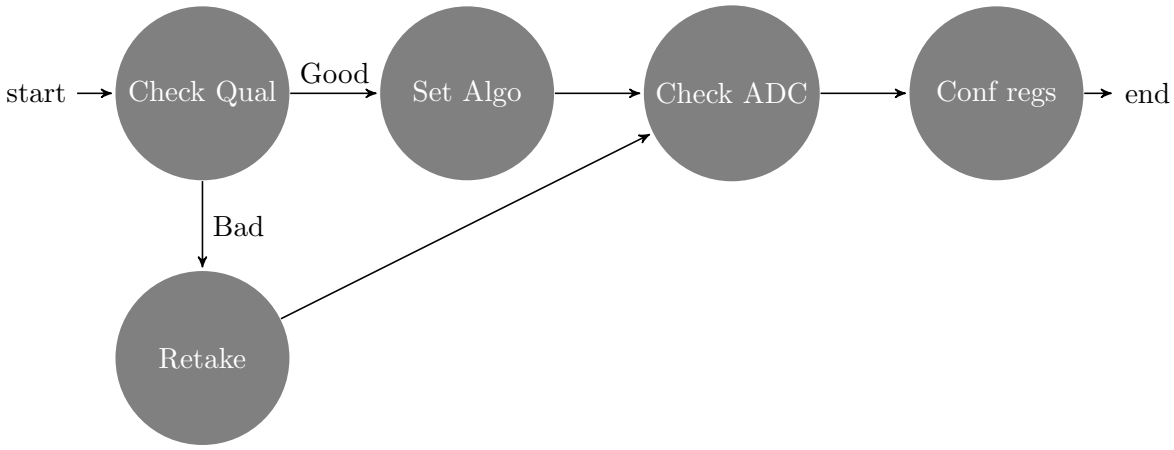


Figure 34: Matlab status machine

These are the five status the machine has:

1. **Check Quality** In this state we check the quality of the received signal. To do so we look at the standard deviation of the peak to peak periods. If the standard

deviation is high compared to a normal value defined for a subject at rest, then the signal quality is bad.

2. **Check ADC** One of the most important configurations to set is the amount of light that goes through the LEDs. This value is set so that the received amplitude of the PPG signal is in a certain range of the ADC range. This is because the ADC is not linear in all of its range. Therefore we adjust the current that goes through the LEDs so that the amplitude of the PPG signal reaches a value between 60% and 80% of the ADC range. For this reason when the signal goes beyond these limits the registers of the ADPD are reconfigured to return to a normal situation. To reconfigure it we do not write the register in the same state. Instead we wait for the register configuration state to do so. This is achieved with a buffer in which we store the actions we want to be taken once we reach the register configuration state.

3. **Set Algorithm** Within this state we set the registers that will be written at a certain estimated heart rate period percentage to the sensor. This time is calculated based on the chosen algorithm, the next peak estimation and the Low and High current consumption starting percentages. However these registers are not written in this state. Instead we store these write requests in a buffer and later on we write them all in the reconfiguration state.

4. **Configure Regs** This is the state in which all the registers are written. As explained before, we have a buffer in which we tell this state which actions it should take at a certain heart rate period percentage. This way we can for example tell this state to enter the low power mode of the Pulses algorithm at the 30% of the estimated period and go back to the high power mode at the 80%. In order to specify which action to take we have a number that references a case of a switch structure. Along with this number we also have to specify the percentage of the estimated period at which the state should run the code.

5. **Retake** This state sets the necessary configuration to try to obtain a signal with good quality. This state is used when the peak estimation is not accurate and as a result the algorithms do not work properly. Once the signal has been retaken the algorithm can run again.

Variable handling To handle all the variables in the application we will use the *handles* object. This object is created by the *GUIDE* matlab tool. It contains the properties of all the elements of the GUI. It can also be used to store the user variables. The good thing about using this object is that every time a callback is run, it is possible to access to it. However using this object has some drawbacks. The main problem he had to face is that once we are running the main while loop, if the *handles* is updated externally by another callback this object does not get updated within the while loop. For this reason for certain variables we had to use *global variables*, as these can be accessed in real time from any part of the code. It is said that having *global variables* is not a good idea as having the control of where each variable is modified is complicated. However in this situation we had no choice and we used them.

This gives us a brief idea on how the Matlab application for the research measuring system was developed.

5.2 End-user measuring system

The *End-user measuring system* was developed to implement the algorithms in the microcontroller that had already been evaluated in the *Research measuring system*. As

implementing the algorithms directly in the microcontroller is much harder than in Matlab we wanted to make sure that the algorithms worked before porting them on the microcontroller. This is why we created two measuring systems.

This second system has many things in common with the first system. Both the I2C and the UART communication are designed almost the same way. There are some differences we will explain in the following sections.

In this measuring system not only we had to develop the microcontroller code but a Matlab GUI as well. The purpose of this Matlab GUI is just to plot graphs and information, no processing is done within this application. Again, the Matlab GUI design has many things in common.

5.2.1 Microcontroller project setup

The microcontroller project was thought to be end-user code. This means that once finished, a developer should be able to take the code and include it in his code. For this reason a flexible setup was created. Instead of hard-coding the algorithms in a single c IAR project, we created two different projects within the same workspace. One project will be designed as a library, and it will process everything related to S_pO_2 and S_pO_2 algorithms. The other project will wrap the first one and will interact with it. We will call this last one the wrapping project.

With this configuration the wrapping code will just acquire the samples from the sensor and send them to the S_pO_2 Library and will handle the communication with the computer. It will also configure and initialize the HAL libraries as well as the clock so that the code can run properly.

This way anyone will be able to take the S_pO_2 Library and include it in his own wrapping code. The only thing it would have to know to work with this library is the function names, their parameters, and the returning values.

This has some other advantages. In case we wanted to sell this S_pO_2 Library, we could just sell the compiled Library. This way the client would not be able to access the code, preventing them to know how the processing is carried out, and protecting this way the intellectual work behind it.

5.2.2 Wrapping project

The wrapping project is very similar to the *Research measuring system* microcontroller code. This is because the tasks of these projects are almost the same. It has to communicate with the ADPD144 sensor and with the computer. The difference now is that this project has to send the received samples from the ADPD144 sensor to the S_pO_2 library and call the necessary functions to obtain the S_pO_2 value to send it to the computer.

For this reason the project has only a few files apart from the HAL files and the modules and sensor libraries. We will focus on the *main.c* and *communication.c* files.

The *main.c* file implements the main status machine as in the other measuring system. The configuration of this status machine has changed slightly as we can see in Figure 35. Two more states have been added: The Stop and the Reset. The main reason is that this way the machine is more organized, and moreover if we need to reset any variable when stopping/resetting we can do it directly within the status machine.

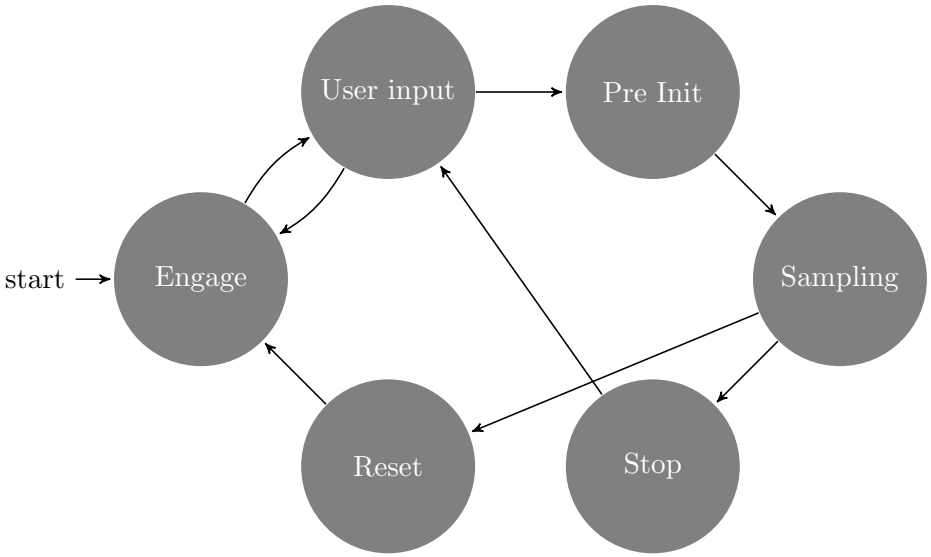


Figure 35: Matlab status machine

One more difference we can appreciate in the status machine is the Sampling state. As we can see in Code 12, apart from reading the ADPD144 FIFO(line 5) and sending those samples to the PC(line 16) we interact with the SPO2Library. First we send the received samples to the Library as shown in line 13, then we call the library update function *SPO2Lib_Tick*. Then we obtain the S_pO_2, HR and HRV values with the Library function *SPO2Lib_getSPO2Value* and finally we send the rest of the information to the PC. This is a summarized code, some bits of code have been omitted.

```
1  case MSM_SAMPLE:
2    if(adpdNewDataAvailable==1)
3    {
4      //Read ADPD Data
5      readAdpdFifoData();
6      if(nAdpdFifoLevelSize==0)
7        errorHandler(11);
8
9      //Convert the raw bytes into uin32 arrays
10     parseBuffer(ADPDProjectFifo,slotAData,slotBData,
           nAdpdFifoLevelSize,&outSize);
11
12     //Send the library the ADPD Values
13     SPO2Lib_updateNewSamples(slotAData,slotBData,sampleTime,
           outSize);
14
15     //Send the ADPD data to the computer
16     sendAdpdData(modifiedSlotAPtr,modifiedSlotBPtr,
           modifiedSlotSize);
17   }
18   //Call the Library update function
19   SPO2Lib_Tick();
20
21   //Obtain the SPO2 Value
22   SPO2Lib_getSPO2Value(&SPO2Val,&MIVal,&HRVal,&HRVVal,&
         signalQuality);
23
24   //Send information to the computer
25   sendPeaksData(minPos,maxPos,minValA,maxValA,minValB,maxValB,
```

```
            newMinPeak,newMaxPeak,minNum,maxNum);
26      sendSPO2MIData(SPO2Val,MIVal,HRVal,HRVVal,signalQuality);
27      sendEstimatedData(estimatedDiference);
28
29      //Attend incoming commands from the computer
30      attendCommand();
31      break;
```

Code 12: Sampling state of the state machine

However the *main.c* file has some other differences. In the initialization routine we setup the S_pO_2 Library we have created. To do so what we have to do is to call the *SPO2Lib_Open* and the *SPO2Lib_setErrorFunction* functions. The first one allocates the necessary memory and the second one sets the error handling function pointer as we will see later. This piece of code is shown in Code 13

```
1 //Open SPO2 Lib
2 SPO2Lib_Open(200);
3
4 //Set the SPO2Library errorHandler function
5 SPO2Lib_setErrorFunction(&errorHandler);
```

Code 13: S_pO_2 Library initialization

The last difference we will comment of the *main.c* is a new function called *parseBuffer*. This function reads the ADPD144 FIFO from the buffer and converts it into 32 bits unsigned int samples. In the *Research measuring system* we used to send the raw bytes to the computer, where these bytes were arranged. In this case the conversion is done within this function in the microcontroller. We are using 32 unsigned integers because we configure the ADPD sensor to return the summation of the four channels instead of returning each channel on its own. This is because in this project we do not take advantage of multiple channels.

The *communication.c* file implements the communication with the PC. It has the UART callback as well as several functions that send different information to the computer. The UART communication is quite similar to the one described in the *Research measuring system*. The main difference is that in this new project we have optimized the command stack and the way memory is managed.

In the previous system we were not worried about optimization because we knew that that system was not the end-user solution. However now optimization is rather important. The command stack of the old system was a matrix of 16 rows and 128 columns where up to 16 commands could fit. Now, the command stack is a 384 byte vector. This means that now the minimum amount of packets that can fit is 3. Moreover in the old solution we used to copy the new incoming bytes from the temporal buffer to the matrix, however this was really inefficient and we solved that by using memory pointers to the new command stack. Lets explain more in detail how it works:

In first place we setup the UART interruption. To do so we have to specify how many bytes should arrive in order to rise the interruption and where to store these bytes. In our case we will wait for 7 bytes and we will store them directly in the location pointed by a variable. This variable will be automatically incremented when the full packet is received, so that when we configure the UART interruption again, the buffer is not overwritten and we can attend the command. The main idea is represented in Figure 36. In this figure, we see in the first state the pointer pointing at the first slot of the buffer. Then, in the second state, a packet has arrived and it has been directly written in the buffer. Then, the buffer

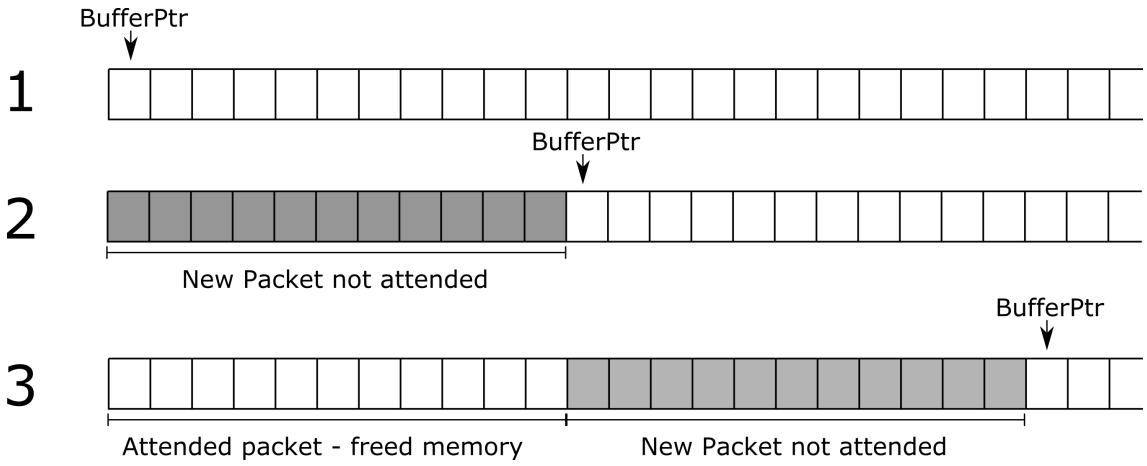


Figure 36: Optimized UART buffer

pointer is moved to the next available buffer slot. Then, a new command arrives, and the previous command is attended. Again the pointer is pointing to the next free slot. This is a circular buffer, and therefore the buffer pointer is also circular. This means that when there is not enough space at the end of the buffer the pointer will return to its beginning. We have to take care in order to prevent commands overwriting. All this job is managed by the functions *getNewPointer*, *putNewPointer* and *getNextPacketPtr*.

As commented before, the *communication.c* file has some other functions. Most of them, such as *sendAdpdData* or *sendSPO2MIData* send information to the computer. They basically prepare a packet and call the HAL function to send them in a non-blocking way.

Last but not least, the have a small c file called ***Common.c***. It has one interesting function that handles errors. Whenever we code something and we think in a situation in which the code can run unexpectedly we add a call to this function. For example, if we detect a UART buffer overrun (when we run out of memory to store new packets) we call it. This function toggles a board LED to notify the event. To do so we configured TIMER3 with a prescaler of 16800 and a count period of 100. This way, as the main clock works at 168MHz, the TIMER3 callback will be called every 100 ms. In the TIMER3 callback we placed a piece of code that toggles the LED. In Code 14 there is an example of the declaration of the error function and the Timer callback.

```
1  /*
2  function: Blinks the LED and stops the execution
3  input: CODE = Error code
4  output: -
5  */
6  void errorHandler(uint8_t CODE)
7  {
8    HAL_TIM_Base_Start_IT(&htim3);
9    printf("CODE: %i\n",CODE);
10   while(1);
11 }
12
13 /* Timer3 Callback */
14 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
15   if(htim->Instance == htim3.Instance) {
16     HAL_GPIO_TogglePin(GPIOD, LED_DS2_Pin);
17   }
18
19 }
```

Code 14: Error function handler and timer callback

5.2.3 S_pO_2 Library

The S_pO_2 Library has been designed to be able to work in any workspace as long as it is configured properly. There are only a few end user functions that can be understood with ease. This library has to meet some requirements in order to work properly:

1. **Fast processing** In most cases the library will be called between sensor samples. This means that the processing has to be performed as fast as possible. Sometimes we do not have as much time as desired. This is due among other things to the fact that reading the ADPD samples and sending them to the computer takes time.
2. **Reduced memory size** Ideally the Library size should be small. This is thought to fit in most of nowadays microcontrollers, and memory size should be optimized. Sometimes we cannot avoid allocating memory but most of the times some techniques can be applied to reduce the size i.e. using circular buffers.
3. **Few functions** We should try to reduce as much as possible the functions a developer has to call in order to make it work properly. This way it will be much easier to include it in any workspace.
4. **Understandable code** The code written should be easily understandable. Commentaries are appreciated in these type of libraries.

It features two c files along with their header files. The main one is called *SPO2Lib.c*. Most of its functions can be accessed from the outside. The developer who includes the library in his code will have to interact with these functions. The other c file is called *signalProcessing.c* and it contains functions that carry out all the processing, filtering and convolving tasks. None of these functions can be accessed externally and they are called from the *SPO2Lib.c* file.

Lets review briefly the necessary functions to interact with in order to include the SP2_Library properly in a workspace:

1. **SPO2Lib_Open** This function takes as the argument the sampling frequency of the high precision sampling zones. With this information it creates the filter coefficients and allocates the necessary memory for the arrays of the library. Moreover it sets the variables to its default values.
2. **SPO2Lib_Open** This function frees the memory allocated during the *SPO2Lib_Open* function.
3. **SPO2Lib_Reset** This function sets the variables to its default values. This function is internal and should only be called within the library code.
4. **SPO2Lib_setErrorFunction** This function takes as argument the pointer to a function that will be called whenever we run into a unexpected state.
5. **SPO2Lib_updateNewSamples** This is one of the most important functions of the library. It takes as arguments the pointers to the buffers where the ADPD144 FIFO have been read, as well as the number of samples. In the first place the

signal is filtered with the filter described in Section 3.4 with function *filterSignal*. Then, we look for the zeros in the filtered signal with function *findZeros*, and finally, thanks to these zeros we look for the peaks in the original signal with function *findMaxAndMinInOriginalSignal*. Once the signal has been processed, we save the last sample for the next iteration. This is because the filter has a certain length, and we need to recall as many samples as the filter's length so that in the next iteration we have enough information to obtain the filtered signal properly.

6. **SPO2Lib_Tick** This is the function that handles the next peak estimation and the state machine.

7. **SPO2Lib_getSPO2Value** This function obtains the S_pO_2, the HR and the HRV values. To obtain the S_pO_2 value the ratio R is obtained and then we apply the calibration curve of Equation 11. In the following sections we will review in depth how the S_pO_2 value is obtained.

8. **SPO2Lib_getSignalQuality** This function returns the quality of the signal. It returns a value between 0 (Bad signal) and 10 (Good signal). It computes the standard deviation of the peak to peak mean value. The higher the value the better the signal quality.

9. **SPO2Lib_setAlgorithm** This function is used to set the active algorithm. The algorithm change will take place in the next heart period.

Apart from these functions there are some others that the user does not interact with. Most of them are called by the main function and their job is to process the information and filter the signal to obtain the S_pO_2 value. We will not get into detail of these functions. However we will comment later on some characteristics that affect the S_pO_2 calculation.

The state machine is a simplified version of the one implemented in Matlab (Figure 34). Some of its states are implemented outside of the machine, however the main states such as the Set Algo and Conf regs are still in the state machine.

There are also some functions the library needs in order to work properly. These are functions that the user has to define. They are not defined within the library because it is thought to work under different circumstances. The most important external functions required are:

1. **AdpdDrvRegWrite** The function that handles the write of a register given the register number and its value.

2. **AdpdDrvRegRead** The function that handles the read of a register given the register number.

3. **AdpdDrvSetOperationMode** This function has to be able to change the operation mode of the ADPD. The argument is a unsigned int that defines the operation mode.

4. **clearFIFOSamples**. This function will flush the content of the FIFO. In order to avoid loosing FIFO samples this is done out of the Library.

5. **getCurrentMs** This function has to return a millisecond time stamp when called.

Some of these functions (the first three ones) are declared in the ADPD144 driver, whereas the last two ones are declared in the wrapping project.

Peak filtering When obtaining the S_pO_2 we realized that finding the peak in the original signal was not the most accurate way of getting a peak. This is because the gathered PPG signal has noise, and therefore most of the times the peak of this signal turns out to be ill defined. For this reason we determined that an extra filtering had to be applied before locating the peak in order to get a more robust peak amplitude and location. We studied which type of filter would fit the best to our signal. We tested several filters with a maximum and a minimum peak of a measured PPG signal and saw the results. This piece of code was written in a Matlab script that generates 8 *subplots* that gives a visual idea of which filter is the most adequate one. This plot can be seen in Figure 37. In this plot, on the left hand side we have the maximum peaks and on the right hand side we have the minimums. In each row we have a different filter type. We studied these four ones:

1. **EMA with fixed length** Exponential Moving average filters have a faster response compared to the simple average filters. This is because newer values are weighted more than the oldest ones. A variable called α determines how aggressive this weighting is. This variable is a function of the filter length. The filter itself has this formula:
$$y[n] = (1-\alpha)\cdot y[n-1] + \alpha \cdot x[n] \tag{38}$$
We can rewrite it this way in the z domain:
$$\begin{aligned} Y(z) &= (1-\alpha)Y(z)z^{-1} + \alpha X(z) \\ Y(z) - (1-\alpha)z^{-1}Y(z) &= \alpha X(z) \\ \frac{Y(z)}{X(z)} = H(z) &= \frac{\alpha}{1-(1-\alpha)z^{-1}} \end{aligned} \tag{39}$$
Once we have the filter coefficients given by the numerator and the denominator of $H(z)$ we can apply the Matlab filter.
We tried different lengths for a fixed alpha in order to see the behaviour of the filter. The drawback of this filtering method is its non-linear phase as we can see using the *freqz* function in Matlab. This means that the group delay will not be constant as we can appreciate with the *grpdelay* function of Matlab.

2. **Normal EMA** Although in the last approach we changed the length independently of the alpha, the alpha value is normally dependent on the length of the filter. The formula is the following:
$$\alpha = \frac{2}{avgNum + 1} \tag{40}$$
Where avgNum is the length of the filter. We tried to filter the signal for different filter filter lengths, now obtaining the adequate α. The results can be seen in the second row of Figure 37

3. **Moving average** This is the most common filter when it comes to signal smoothing. It just averages the last N samples. This filter was applied and can be seen in the third row of Figure 37.

4. **Median filter** Median average is often used to reduce noise. We also applied it to out signal to see the result. It can be appreciated in the forth row of Figure 37.

After having looked at these filters we determined that the one that smooths the signal in a better way is the simple moving average filter with a size of 4. We have to take into account that the signal plotted has a Fs of 200 Hz. This means that when applying this

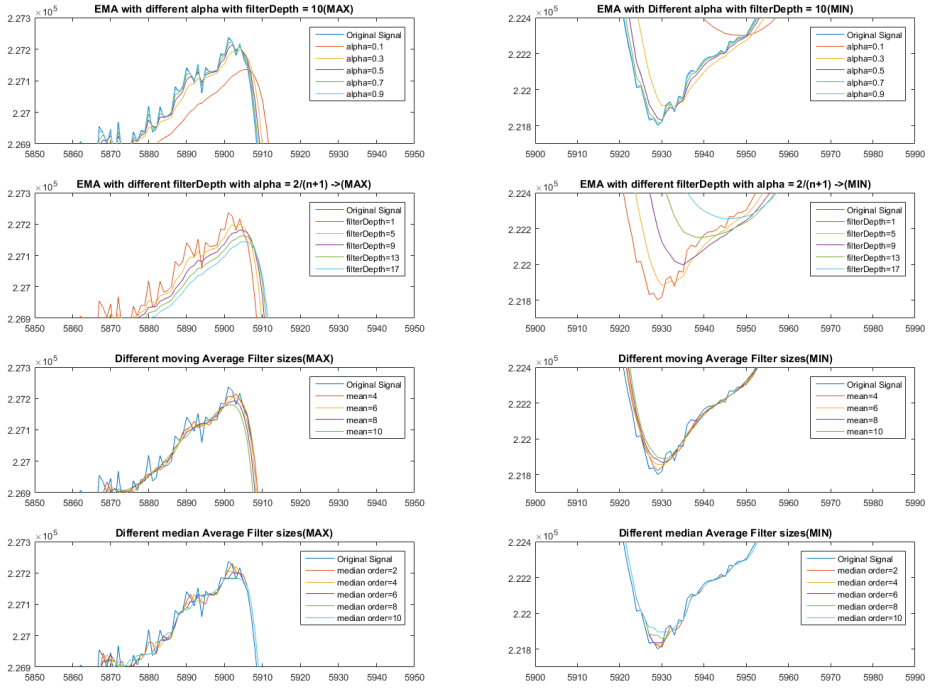


Figure 37: Peak filtering approaches

filter we will have to filter the signal with a moving average filter with a size of $200/50 = 4$. This filter has a notch in 50Hz. This fact made us think that the noise could be debt to the mains electricity as it also runs at 50Hz.

We normally sample the signal at sampling frequencies higher than 50Hz. We have programmed the microcontroller so that it applies this filtering if the Fs we are running at is greater than 50Hz. It is of great importance to notice that if we sample the PPG signal at 50Hz, the mains noise will disappear in our sampled signal, it will be transformed into a DC constant that does not distort the signal. This can be easily understood if we think in a 50Hz sinusoidal wave being sampled at 50Hz. The values of the sampled signal will have the same amplitude, and it will be a constant function. This could be another shortcut to the problem, however we would have less precision in the peaks. If we wanted to take advantage of this phenomenon in USA we would have to sample the signal at 60Hz, as this is the mains frequency.

Anyway, for higher frequencies the advantage of this filter is that it can be implemented easily in the microcontroller. Moreover its phase is linear, which means that the group delay will be a constant. The median filter for example has a non-linear phase and therefore the group delay is not a constant.

The moving average filter is applied just before looking for the peaks. We do not filter all the signal, just a zone of the peaks where we have detected it. This way it does not take too long to calculate it. Moreover optimization techniques have been applied in order to reduce computation time. Theoretically for each sample, as the size of the filter is $fs/50$, we would need to do $fs/50 - 1$ sums. However this can be reduced to a mean of 3 operations. This can be achieved if for each new sample, instead of averaging the signal, we take the last averaged value and we add the new sample and subtract the one that has left the averaging window. This technique is graphically explained in Figure 38 using an

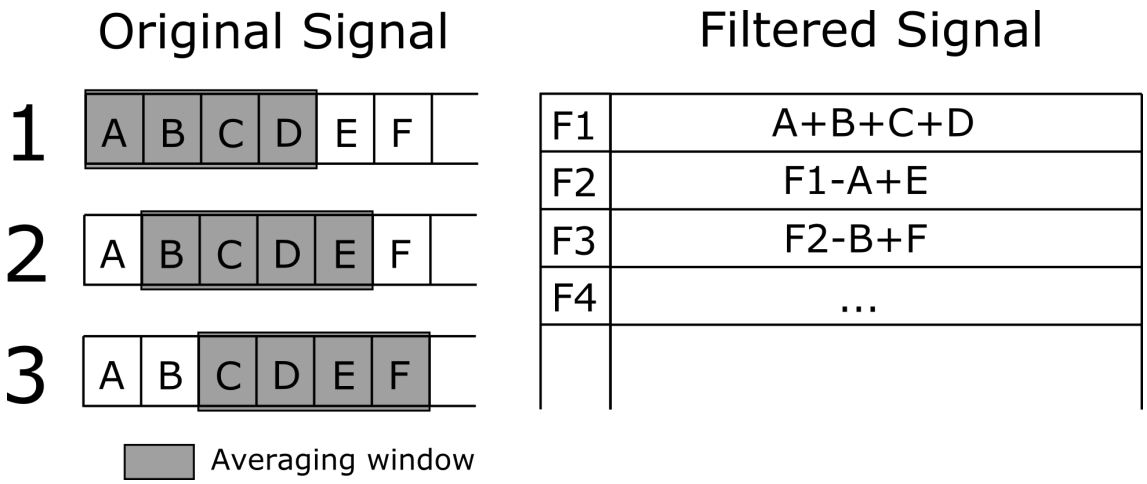


Figure 38: Optimized average filtering techniques

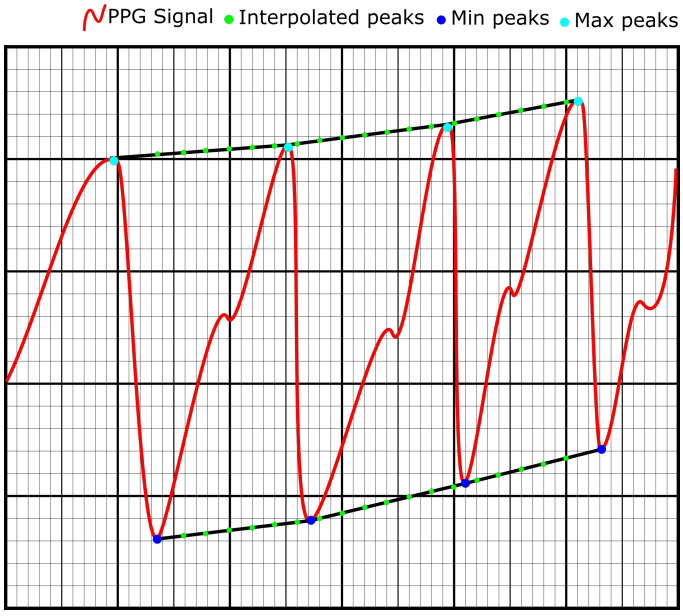


Figure 39: S_pO_2 interpolation

averaging size of 4. As we can see each new filtered signal but the first one is obtained by adding the previous output the new sample and subtracting the oldest one. It is not really necessary to divide by the averaging value since the R value we are looking forward to obtain is a ratio.

When it comes to obtaining the S_pO_2 having the peaks as data, we also apply some techniques to improve the result. As we can see in Figure 39, maximum and minimum peaks are not located in the same time instant. For this reason dividing maximums by the minimums is not a very accurate operation. For this reason we decided to interpolate the maximum and minimum peak points so that the interpolated ones are located in the same time instant as shown in the figure. These interpolated points are obtained every new peak. This means that they are calculated in real time. There is a define in the microcontroller code with which you can specify the number of new points between two peaks. Once we have the interpolated values, we obtain the AC and DC levels of the

signal. They are calculated this way:

$$\begin{aligned} AC &= MAX + MIN \\ DC &= \frac{MAX + MIN}{2} \end{aligned} \tag{41}$$

This is done for both Slots, the one with the red LED and the one with the infrared LED. Then we can calculate the R ratio:

$$R = \frac{\left(\frac{AC_R}{DC_R}\right)}{\left(\frac{AC_{IR}}{DC_{IR}}\right)} \tag{42}$$

All the filtering and processing tasks within the library are carried out using circular buffers. This is of great importance as some of the functions are called once a iteration period. If we did not do it this way, we would end up using too much memory and eventually we would run out of memory. Most of the time using a buffer as a circular buffer has only two implications:

1. **Declaring a size** The size of the circular buffer will be fixed. When writing to it we will only change the location.
2. **Writing pointer** Along with the buffer we will declare a variable that will store the pointer that will indicate where the next byte should be written. Its value should be between zero and the size of the buffer. Once it has reached the maximum value, it should return to zero.

Signal conditioning When working with algorithms, sometimes we have to modify the signal coming from the sensor before processing it. This is because the algorithms distorts the signal in some way. The pulses algorithm for example, changes the number of pulses pushed to the LED. This means that the number of pulses integrated by the AFE will also change. As a result, we will have less PPG signal amplitude, which means that when we change from the high precision sample to the low current consumption using this algorithm the signal will experience an amplitude drop. To fix this problem, we have to normalize the amplitude of the signal. In the code we normalized it to the number of pulses of the high precision sample period. This way the signal will be almost continuous. However, even normalizing the PPG signal there is still an amplitude difference. This may be caused by the parasitic capacitance of the photodiode. To fix this small problem, after normalizing the signal we compute the difference between the last sample of the HPAP and the first one of the LPAP. Then, for the rest of the LPAP period we add this difference. This way we can achieve a continuous signal. It is rather important to reduce this amplitude drop, otherwise the filter will detect it as a peak.

When it comes to the Start/Stop algorithm, we also have to condition the signal. In this algorithm, as in the LPAP period we are not sampling the signal we have to make up the samples in this period so that we maintain the number of samples per second. Otherwise the filter would not work properly. As we have to interpolate samples, we decided that the best idea was to create a linear interpolation between the last sample of the last HPAP period and the first sample of the new HPAP period. This way, as the created samples will be placed along a line the filtered signal will not be affected, because the created samples follow a monotone trend.

Finally, when using the Fs algorithm we also have to modify the incoming signal. As we change the sampling frequency in the LPAP we have to interpolate it so that we have

the same fs as in the HPAP. This way the sampling frequency will be the same and the filtering will work properly. We used a linear interpolation to create the new samples.

This signal conditioning was also done in the *Research measuring system* within the Matlab application.

Chapter 6. Validation

Validation is one of the most important parts of the project. In this section we will go through the setups we created to measure the current consumption of the ADPD144 and the microcontroller. Some of them did not succeed due to some problems. We also studied the accuracy of the output S_pO_2 value using hospital material as explained in the Appendix chapter.

There are mainly two chips we have to monitor in order to obtain the current consumption. One is the ADPD144 sensor and the other the microcontroller. The ADPD144 will reduce its consumption thanks to the algorithms, however the consumption of the microcontroller may increase due to the extra processing tasks it will have to overcome.

When it comes to the microcontroller, we measured the average time per sample that it takes to process the data and obtain the S_pO_2 value. With this value and the mean current consumption of the microcontroller given in its datasheet we determined the mean current consumption increment due to the S_pO_2 Library. We measured it this way because the microcontroller is soldered to a board, and we cannot have access to all of its supply sources. Therefore measuring the current as we have done with the ADPD144 was not possible. We tried to measure the current drainage directly in the USB supply, however the problem is that this line does not only feeds the microcontroller but the ADPD, the bluetooth, the battery charger, and other chips as well.

6.1 ADPD144 unsuccessful validation approaches

We tried three different measuring setups. Two of them did not succeed as we will comment in this section.

6.1.1 Measuring the ADPD144 current consumption with the Tektronix TCPA300 current probe

The first approach was to use a Tektronix current probe. The good thing about this solution is the implementation easiness. The probe itself can be seen in Figure 40. The probe has a hole at the end that will hold the supply line of the ADPD144. The setup we created to measure the current is shown in Figure 41. As we can see we removed the jumper of the breakout board that connects the supply line from the side of the microcontroller and the side of the ADPD sensor. Then, we connected a cable that goes through the current probe hole and then it returns back to the breakout board. Finally, we connected the probe to the oscilloscope and watched the result. As we can see in Figure 42 the signal obtained has low resolution.

Theoretically we can appreciate two big hills that are caused by the two different sampling slots. Then, in each hill we have 8 peaks modulated, that represent the current consumption of the sampling of the 8 pulses that in this example are being pushed to the LEDs. But as previously said, the resolution is mediocre, we should look for another way of measuring the consumption. This is because the resolution of the device as we can read in its datasheet is 1mA, and to obtain the current consumption we need more.

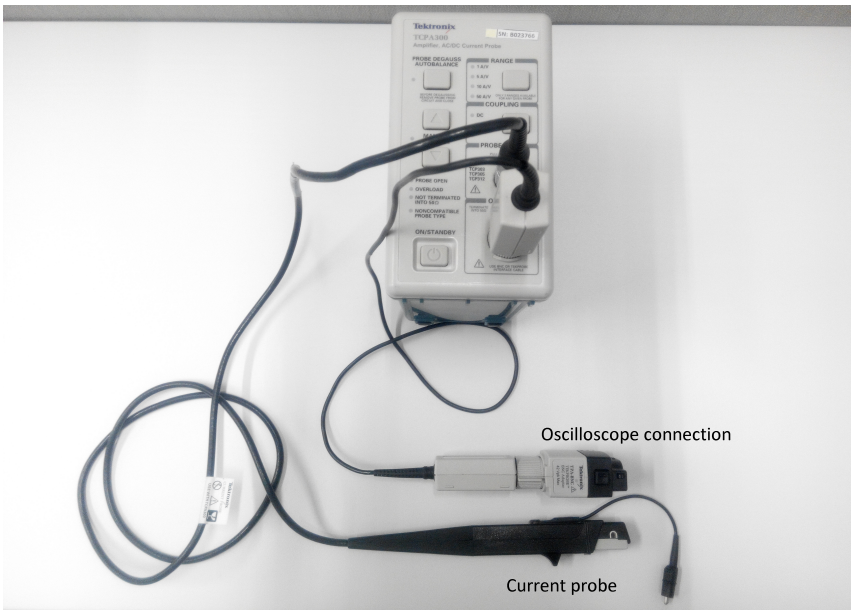


Figure 40: Tektronix current probe

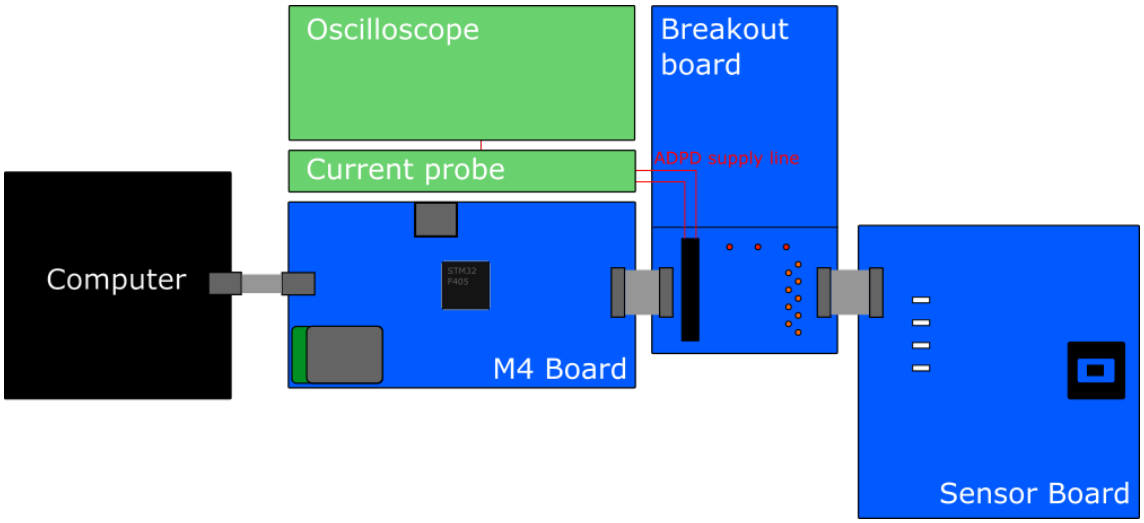


Figure 41: Current probe setup

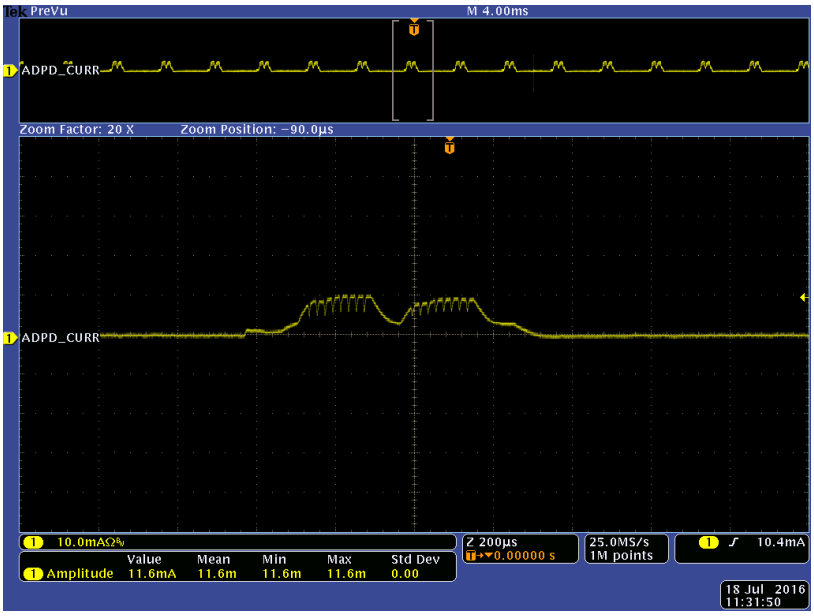


Figure 42: ADPD AFE Current consumption

6.1.2 Measuring the ADPD144 current consumption with the Keithley power supply

The Keithley power supply is a very versatile device. It is a programmable voltage and current supply. It can generate voltage and measure the current drained. This comes in very handy in our project as we can connect its supply to our sensor and measure with the same device the current usage. On its website we can find the VISA drivers. With these drivers we can control the device from the computer. This way we can plot graphics of the voltage or the current over time.

For this purpose we created a visual LabView application[1]. We choose LabView because along the drivers there were simple usage examples that we could use to control laboratory devices[9].

As we want to measure the sensors AFE and LED consumption separately, we had to use two Keithley power supplies. For this reason in the VIs most of the elements are duplicated. The LabView project is mainly made up by a VI in which we can find a state machine. This state machine has four states:

1. **Init** This state initializes the variables for a new measurement
2. **Init Sources** Initialization of the Keithleys. Here we setup the current measurement with the voltage supply. To do so we call some VISA driver VIs.
3. **Acquire** In this state we will remain until the user hits the stop button. Here we call the Read single point VI and we plot it in the graph.
4. **Close** This state was thought to close the connection with the devices.
5. **Wait for user response** We will remain in this state until the user interacts with the graphical interface. Then depending on the action the state machine will change its state.

In Figure 44 we see the developed LabView application. However this measuring system approach was not the one that we finally used. The LabView application worked flawlessly.

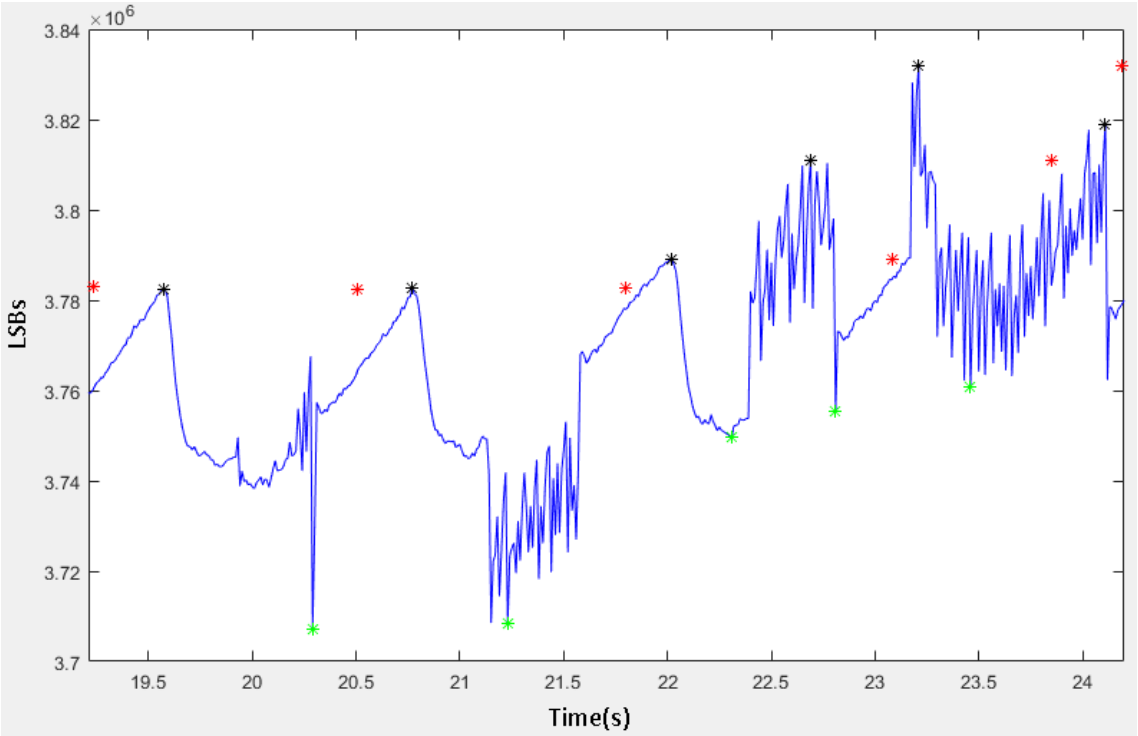


Figure 43: Bad PPG signal due to the sourcemeter supply

However we found a problem related with the sourcemeters. When feeding the sensor with them, the PPG signal eventually showed some kind of strange noise and distorted signal as shown in Figure 43. We thought that the sourcemeters were not able to face the current peaks produced by the AFE and the LEDs, and therefore the signal was distorted.

For this reason we could not use these sourcemeters to measure the current consumption. Before giving up, we tried putting some bulk capacitors from the supply line to the ground, close to the ADPD sensor, so that they could feed the sensor during the peaks. However it did not succeed. The signal had less distortion, but it was not perfect.

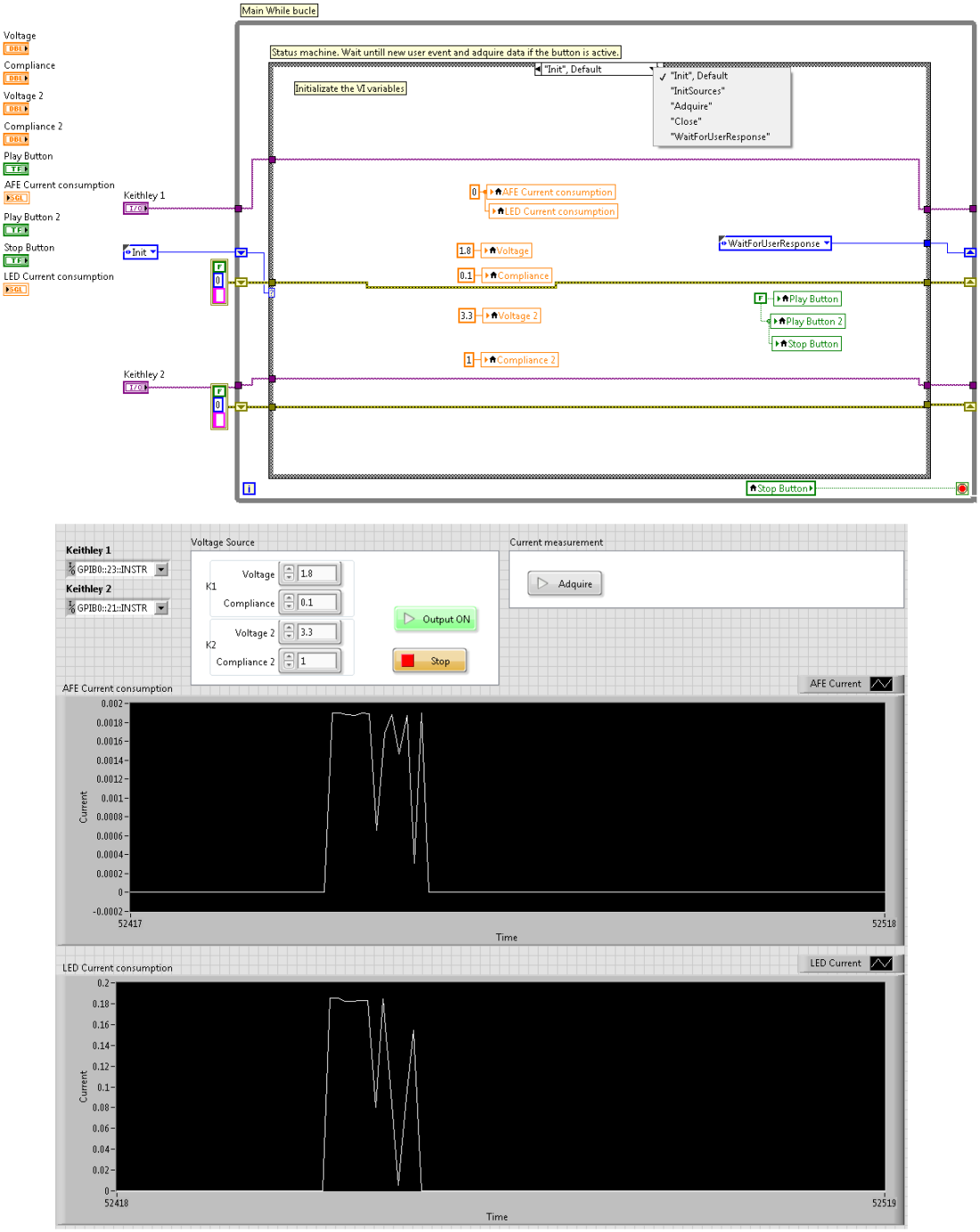


Figure 44: Keithley sourcemeter labview interface

6.2 ADPD144 final consumption validation approach

The previous measuring system approaches were rather easy to implement. However as they did not work we had to struggle a little bit more to create a working measuring system. We decided to measure the current by adding a small resistor to the supply line. This way, by obtaining the voltage across the resistor we can get the current flowing, as we also know the resistor value.

A low resistor value is required so that the sensor keeps working properly. If the resistor is too big, the voltage drop across the resistor could be big enough to prevent the sensor from working. Having a low resistor however means that the voltage across it will be rather small as we are measuring low currents. For this reason we had to use instrumentation amplifiers[2]. These amplifiers have a very high input impedance and a high gain. These specifications fit our needs.

As we had to measure the AFE and the LED lines current consumption we had to use two instrumentation amplifiers, one for each line. We had in the laboratory two different instrumentation amplifiers along its breakout boards, perfect for our project. We used the AD8237 for the AFE line and the AD8421 for the LED one. Lets see now how we designed the gain of the amplifiers

- AD8237 This amplifier is the one that will feed the AFE. The AFE consumption as we can read in the ADPD144 datasheet can reach 9.3mA. The amplifier gain as we can read in its datasheet is given by the formula:

$$G = 1 + \frac{R2}{R1} \tag{43}$$

With R2 and R1 we will set the gain. These resistors are located between the Vout and FB lines, and FB and REF lines respectively as we see in Figure 45. We will configure the gain so that the output peak amplifier voltage is 2.5V. However in first place we will calculate the output voltage in function of the R1,R2, the sensing resistor(Rg) and the flowing current:

$$\begin{aligned} I = \frac{V_{Rs}}{Rs}; V_{Rs} = \frac{V_{out}}{G}; V_{Rs} = \frac{V_{out}}{1 + \frac{R2}{R1}} \\ I \cdot Rs = \frac{V_{out}}{1 + \frac{R2}{R1}} \\ V_{out} = I \cdot Rs \cdot \left(1 + \frac{R2}{R1}\right) \end{aligned} \tag{44}$$

If we set Rg = 1Ω and R2 = 27$k\Omega$, given that $V_{out}|_{I=9.3mA} = 2.5V$, we can calculate R1:

$$R1 = \frac{R2}{\frac{V_{out}Rs}{I} - 1} \tag{45}$$

With this formula we obtain a value of 100.81Ω. Therefore a value of 100Ω will do the job. This means that the gain will be:

$$G_{AD8237} = 1 + \frac{27000}{100} = 271 \tag{46}$$

As we can see in the specification table of the datasheet of the AD8237 the maximum gain is 1000, therefore it will run properly.

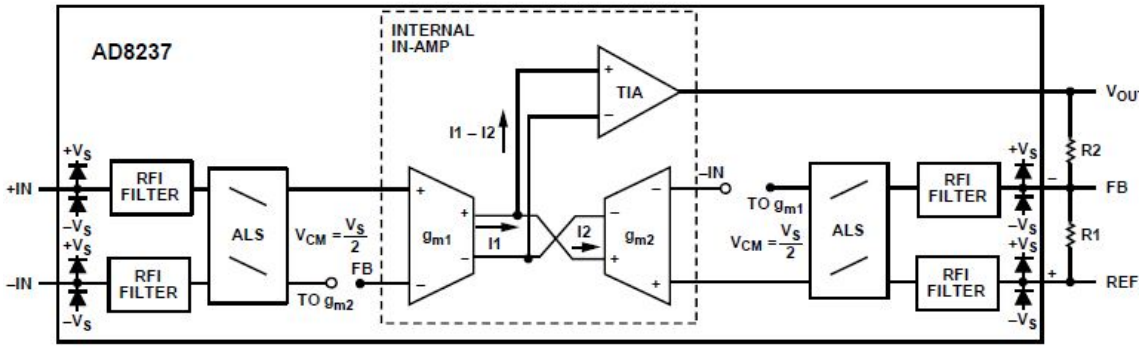


Figure 45: AD8237 Simplified schematic

- AD8421 This is the instrumentation amplifier that will help us to measure the LED current. The LEDs peak current drainage can be up to 150 mA when obtaining the PPG signal on the finger. We will obtain the resistor value so that the peak instrumentation amplifier output voltage to be 2.5V. The gain of this amplifier is set with just a resistor, Rg. The gain is given by Equation 47

$$G = \frac{9900}{Rg} + 1 \tag{47}$$

Then, we can calculate the resistor Rg in function of the sensing resistor Rs, the desired output voltage V_{out} and the input current I as we did with the AD8237. This is shown in Equation 48

$$Rg = \frac{9900}{\frac{V_{out}}{Rs \cdot I} - 1} \tag{48}$$

For the indicated values we obtain an Rg of 631.91Ω. We will take a 620Ω resistor value for Rg. With this Rg, we can obtain the gain using Equation 47, obtaining the value shown in Equation 49.

$$G = \frac{9900}{620} + 1 = 16.96 \tag{49}$$

In the AD8421 datasheet we can see that the maximum gain is 10000, therefore our configuration will run properly.

When testing our setup we faced some problems. The signal was not as sharp as it was supposed to be. All the peaks showed smoothed, and at first we did not know why. Then we thought this could be due to some capacities that can be found on the sensor board. These elements could be feeding the sensor during the abrupt peaks until they discharge. Then, they would eventually charge again to smooth the next peak. This is the aim of bulk capacitors, they are designed to prevent excessive current drop of the supplies.

For this reason we looked for bulk capacitors in the sensor board. We found two of them, one for the LED supply line and the other one for the AFE line as we see in Figure 47.

These capacitors do not really affect the current measurement. Their effect is that they feed the ADPD during the abrupt current drops, and they also drain current when charging. We tried to modify the circuit as shown in Figure 46. We removed the capacitors and located them behind the measuring resistor. This way, the current flowing through the resistor will be the summation of the current proportioned by the supply and the capacitor. To organize everything properly we cut two small bakelite prototyping boards. These boards have a resistor in the supply line, as well as the capacitors and a two-pin connector that

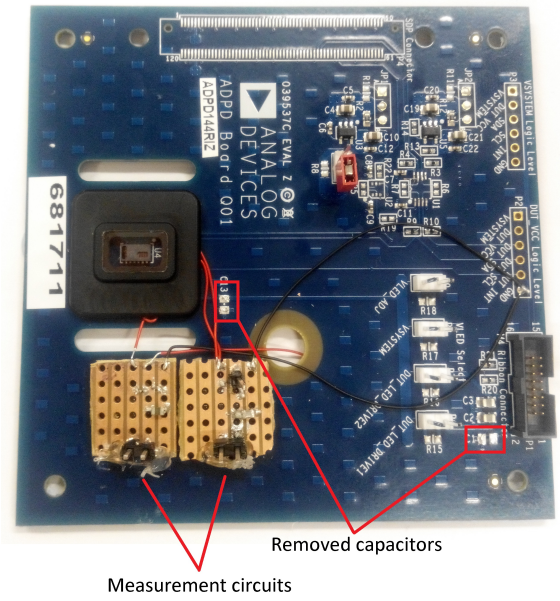


Figure 46: Modified sensor board to measure current consumption

we will connect to the differential inputs of the instrumentation amplifiers. We also have the necessary cables to interface with the ADPD sensor. The two different measuring circuit approaches are shown in Figure 47. As we can see the difference between them is the location of the capacitors. As we will obtain the average value with an oscilloscope, we ended up using the circuit on the right. This is because the capacitors filter the consumption peaks of the LED and the AFE, and as the number of samples the oscilloscope can take for each measurement is limited, we have to avoid short pulses. Otherwise the measured mean consumption could be inaccurate.

Once we have the measuring circuit lets review how the mean current was calculated. We used a Tektronix MSO4053B oscilloscope. This device can measure the area under the gathered signals. Knowing the area and the measuring time we can calculate the mean current by dividing the area by the measuring time. Then, we have to divide this value by the amplifier gain. One more thing to take into account when measuring is the probes and the amplifiers offset. This offset is present even if the differential inputs of the amplifiers are connected together. Ideally this would have to be zero. For this reason we will have to subtract this offset to the area obtained by the oscilloscope.

To calculate this offset we just have to connect the amplifier inputs together and see the mean value for a 10 seconds signal sample. Once we have the mean value, in order to obtain the real area we will subtract this offset.

All in all, the formula to obtain the mean current in function of the area given by the oscilloscope is shown in Equation 50:

$$I = \frac{Area_{meas} - OffsetCurrent \cdot Time_{meas}}{G \cdot Time_{meas}} \tag{50}$$

In our measuring protocol we decided that we should run each algorithm for 10 seconds. Then we measured the area of as many heart rate periods as can fit in these 10 seconds. With this area we calculated the mean current consumption. We take an integer number of heart pulses because this way the algorithm comparison is more fair.

This is how we measured the output signal of the instrumentation amplifier. However we faced one more problem: the oscilloscope can take up to 20 megasamples for each

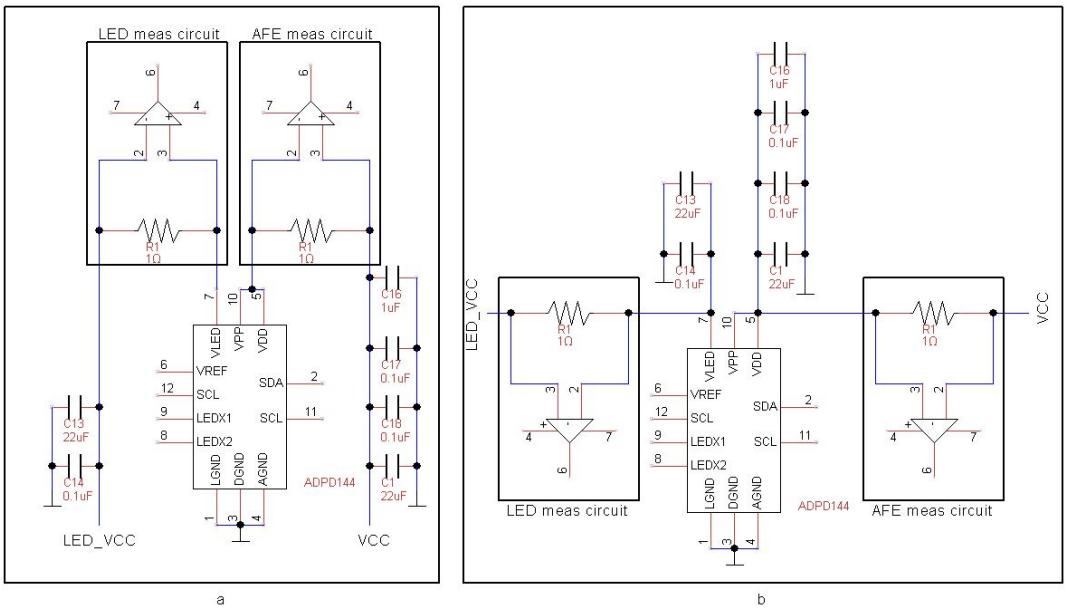


Figure 47: Two possible measuring alternatives. (a) with the measuring circuit between the ADPD144 and the capacitors and (b) the capacitors between the measuring circuit and the ADPD144.

measurement. This means that if we want to gather 10 seconds of the signal, our resolution will be 0.5 us. This is not enough in our case as the LED pulses have a length of 3 us. For this reason we decided to do the measurement with the bulk capacitors between the ADPD and the measuring resistor circuit on the right of Figure 47. They help us to smooth the signal and this way the measured area will be more accurate. Moreover the integrated area will be the same. The difference between putting the capacitors in one location or the other can be appreciated in Figure 48. As the probes label say, the white graphs show the AFE and LED consumption with the capacitors between the sensor and the measuring circuit, whereas the green and the yellow are taken with the measuring circuit between the sensor and the capacitors. As we can appreciate the LED pulses of the first probe (yellow one) are rather narrow compared to the R3 probe. This is the reason why we finally used the circuit on the right of Figure 47.

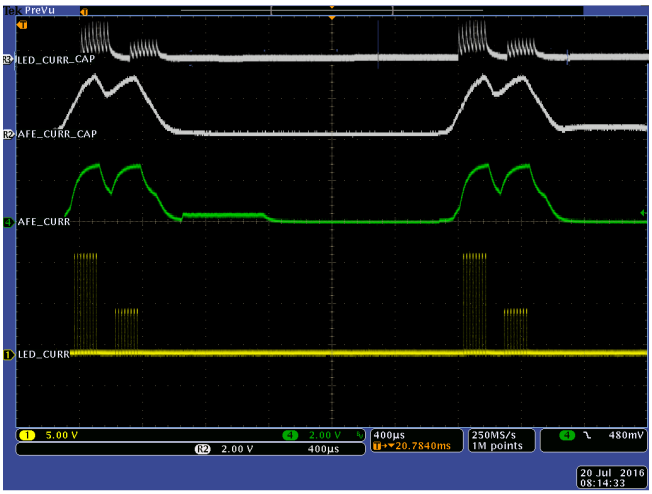


Figure 48: Measurement with the capacitors between the ADPD and the measuring circuit (White graphs) and with the measuring circuit between the ADPD and the capacitors (Yellow and green graphs)

6.3 Microcontroller validation

The microcontroller current consumption measurement is rather different. The microcontroller is embedded in the board, and it has 7 supply lines, 6 digital and one analog. All supplies lines come from the same line that feeds the rest of the components of the board. This makes impossible to measure the current consumption of the microcontroller. We cannot place a resistor as we did with the sensor because the line traces are not accessible, and if we could do so, we would be measuring the current drainage of other components.

As we concluded that this was not possible, we found an alternative way of measuring the consumption. We looked for the figures of current consumption of the STM32F405 microcontroller and we found an application note from the manufacturer that gives us the information about the consumption of the STM32F4 series. In this application note we can find the consumption for the different power modes. We found that for the Dynamic Run mode, it drains 40mA in average, and 310uA in Stop mode.

With this information we can estimate the current consumption of the microcontroller. We just need to know how long the microcontroller is processing the information and how long the microcontroller is in Stop mode. We configured the microcontroller to go to Stop mode when it finishes the processing, and to return to Dynamic mode when the new ADPD sample interrupt rises. In order to measure precisely the timing we created a software subsystem that manages it. It stores the clock cycles that it has taken to the microcontroller to process the library functions and sends these data to the computer, where it is averaged and plot at the bottom of the GUI. To obtain the timing precisely we read a register of the microcontroller that counts the number of clock cycles. This register is located in the 0xE0001004 address and it is an unsigned int.

Then we just add the processing times of the different library functions to obtain the amount of time we have been in the Dynamic mode. The rest of the sampling period the microcontroller will be in Stop mode.

The formula that we will use to obtain the current consumption for the different algorithms is shown in Equation 51. The T stands for the heart rate period in seconds, HP_p stands for the HPAP percentage normalized to 1,LP_p stands for the LPAP percentage normalized to 1, HP_{fs} The sampling frequency of the HPAP, LP_{fs} The sampling frequency of the LPAP, t_{sam} the time that it takes for the microcontroller to read one sample of the sensor, t_{lib} the time consumed by the library, H_c the current consumption of the microcontroller in dynamic mode and L_c the current consumption of the microcontroller in stop mode. For each sampling period, we have one HPAP and one LPAP. Inside the HPAP we will have a certain amount of time in which we will be processing $HPAP_{proc}$, and another one in which we will be in Stop mode $HPAP_{iddle}$. The same thing happens with the LPAP. For this reason in order to write the final formula we have declared in first place these auxiliary variables. When it comes to the Start/Stop algorithm, we will take for granted that $LPAP_{Proc} = 0$ and that $LPAP_{iddle} = T \cdot LP_p$

The main idea of this formula is to weight the processing time by the current consumption of the dynamic mode of the microcontroller and the idle time with the consumption of the microcontroller in stop mode

$$\begin{aligned}
HPAP_{Proc} &= T \cdot HP_p \cdot HP_{fs} \cdot (t_{sam} + t_{lib}) \\
HPAP_{Idle} &= T \cdot HP_p \cdot HP_{fs} \cdot \left(\frac{1}{HP_{fs}} - (t_{sam} + t_{lib}) \right) \\
LPAP_{Proc} &= T \cdot LP_p \cdot LP_{fs} \cdot (t_{sam} + t_{lib}) \\
LPAP_{Idle} &= T \cdot LP_p \cdot LP_{fs} \cdot \left(\frac{1}{LP_{fs}} - (t_{sam} + t_{lib}) \right) \\
Total_{Curr} &= \frac{(HPAP_{Proc} + LPAL_{Proc}) \cdot H_c + (HPAP_{Idle} + LPAL_{Idle}) \cdot L_c}{T}
\end{aligned} \tag{51}$$

Chapter 7. Results

When talking about the consumption of the algorithms we can separate the current consumption of the ADPD and the current consumption of the microcontroller. Our algorithms are thought to reduce the current drained by the ADPD sensor, however executing the algorithms means to process more information, which could increase the microcontroller consumption.

We will study the current consumption of these two devices separately. All the measurements shown in this chapter have been obtained as explained in the Evaluation chapter.

7.1 ADPD144 Current consumption

The following results have been obtained as described in Section 6.2, with the configuration shown in Figure 47 (b). To represent the current consumption of the different implemented algorithms, we will first carry out some measurements with no algorithms running. This way when we measure the current drained with the algorithms running, we can tell how many times we have reduced the consumption compared to a normal situation in which the algorithm is not running. We will measure 10 times the consumption of each algorithm. Then we will obtain the mean consumption and then we will compare them. Our base configuration will be the following: We will use a sampling frequency of 400Hz, with a 2 sample average. This means that the ODR[11] will be 200Hz. We will use 8 pulses and the current of the LEDs will be fixed to 115mA.

- **No algorithm** With no algorithms we concluded that the total average consumption was 2.40 mA, 1.56 mA due to the AFE and 0.84mA due to the LEDs.

- **Pulses Algorithm** After running the 10 tests, the total average current drained was 1.57 mA, 1.08 mA due to the AFE and 0.49 mA due to the LEDs.

- **START/STOP algorithm** This algorithm is the one that saves more current, however it has some drawbacks as we will comment later on. The average current consumption was 1.16mA, 0.70mA drained by the AFE and 0.45 by the LEDs.

- **Fs algorithm** This algorithm drains an average amount of 1.47mA, 0.92mA of which by the AFE and 0.55mA by the LEDs with the most conservative configuration, however it can be configured to consume only 1.18 mA.

In Table 3 we can see the comparison between the different algorithms. We can also see the sensor configuration for the HPAP and LPAP zones. The current reduction is referenced to a situation in which no algorithm is running. We have also included the theoretical current reduction obtained from the formulas given in the ADPD144 sensor datasheet as well as the mean measured current consumption.

As we can see, the most efficient algorithm when it comes to current consumption is the Start/Stop. However this one has some drawbacks when it is applied to a real PPG signal. If the beat to beat period of the heart has too much variation, then the algorithm will end up working poorly. This is because the odds are that a peak will be lost at some point, which will lead to a bad peak estimation and so on. This could be fixed if a more intelligent algorithm was developed that could detect outliers and ignore them or if we

[11] Output Data Rate

Algo	HPAP Config	LPAP Config	**%Reduction**	%Theo.reduction	Mean curr(mA)
None	8 Pulses 200Hz ODR 400Hz Fs		**0**	0	2.40
Pulses	8 Pulses 200Hz ODR 400Hz Fs	1 Pulse 200Hz ODR 400Hz Fs	**35.5**	39.71	1.57
Fs	8 Pulses 200Hz ODR 400Hz Fs	8 Pulses 50Hz ODR 100Hz Fs	**39**	41.12	1.47
Fs	8 Pulses 200Hz ODR 400Hz Fs	8 Pulses 50Hz ODR 50Hz Fs	**47.92**	48.06	1.25
Fs	8 Pulses 200Hz ODR 400Hz Fs	8 Pulses 25Hz ODR 25Hz Fs	**50.76**	51.49	1.18
Start/ Stop	8 Pulses 200Hz ODR 400Hz Fs	-	**51.9**	55	1.16

Table 3: Current consumption comparison among the algorithms

changed dynamically the LPAP percentage depending on the HRV. The Fs algorithm is quite stable, peaks cannot be lost as long as the LPAP Fs is not too low. For this reason this algorithm is a good approach. The pulses algorithm does not save too much current. This is why is considered the worst one. However it will not get lost either because we are always sampling.

7.2 Microcontroller Current consumption

The microcontroller consumption evaluation procedure has been carried out as explained in Section 6.3. As we have commented in the Validation chapter, in order to measure the microcontroller current consumption we will use the mean current consumption given in the microcontroller datasheet and the time that it takes to the microcontroller to process the algorithm. As we can see in its datasheet, the consumption of the microcontroller in Dynamic mode is 40mA, whereas its consumption in Stop mode is 390uA. When processing the microcontroller will be in a Dynamic power mode and then it will go to Stop mode, in which the microcontroller consumes less power. In Table 4 we can see the time that it has taken to process the algorithms in us. As we can see, the *GetSPO2* is the function that takes most of the time. This is because it has to compute the interpolated values and it also has to calculate the HR, HRV as well as the signal quality. We can conclude

Time(us)	**Pulses**	**Fs**	**Start/Stop**
UpdateSamples	55	59	57
Tick	9	9	9
GetSPO2	311	310	309
Acconditionate	1.2	1.3	1.1
Total	376.2	379.3	376.1

Table 4: Time taken to process the library functions in us

that the average time that it takes to the microprocessor to process the library functions is more less 376 us. This is the average consumption time per new sample. Then, with the formula of Equation 51 we can calculate the current consumption for each heart rate pace. We have created a graph in which we can see the microcontroller current consumption if we use the S_pO_2 library with the three algorithms and the consumption if we just obtain the ADPD samples. This graph can be seen in Figure 49. As we can see the pulses algorithm always drains more power, whereas the other two algorithms can even reduce the consumption of the microcontroller for low heart rates. This is because the pulses algorithm keeps the same Fs in the LPAP, and therefore the microcontroller will have to process the same number of samples as in the 'No algo' case, with the difference that the processing time will increase as the library will be running. On the contrary, the Start/Stop and Fs algorithms reduce the number of samples that have to be processed in the LPAP. For this reason they can reduce the current consumption.

However this figure does not give us representative information. We should calculate the current consumption increment due to the algorithms, taking the S_pO_2 gathering as a common factor. This way we would be comparing a device that just measures S_pO_2 and another one that measures S_pO_2 efficiently. As the parts of the code that compute the status machine, the peak estimation, the read/write register of the ADPD and the signal conditioning are the *SPO2Lib_Tick* and the *AcconditionateSamples* functions, we can obtain the time difference between the two situations. This difference is more less 10us for all the algorithms as we can see in Table 4. With this information we can calculate the consumption difference, as shown in Figure 50. As we can see, there is almost no difference between the current consumption of the pulses algorithm and the consumption of 'No algo'. This is because now we are considering that the 'No algo' measurement takes more time as it also has to obtain the S_pO_2. Now the differences between the 'No algo' case and the Start/Stop or Fs algorithms are even greater. Therefore, with the Start/Stop and the Fs algorithms, we not only reduce the current consumption of the sensor but we reduce the current consumption of the microcontroller as well.

For this microcontroller we can see that we can even reduce the current consumption by almost 10mA if we use the Start/Stop or the Fs algorithms. Taking into account that in the sensor side we can save up to 1.24mA, we can say that the Start/Stop and the Fs algorithms can save current. When it comes to the pulses algorithm, as the current savings achieved by the sensor is greater than the current increment due to the microcontroller extra processing, we can also say that it saves current, however the difference in this case is rather small.

However this microcontroller may not be the best choice. Some Cortex-M3 microcontrollers drain less than 2mA in active mode, which means that the total current consumption would be lower.

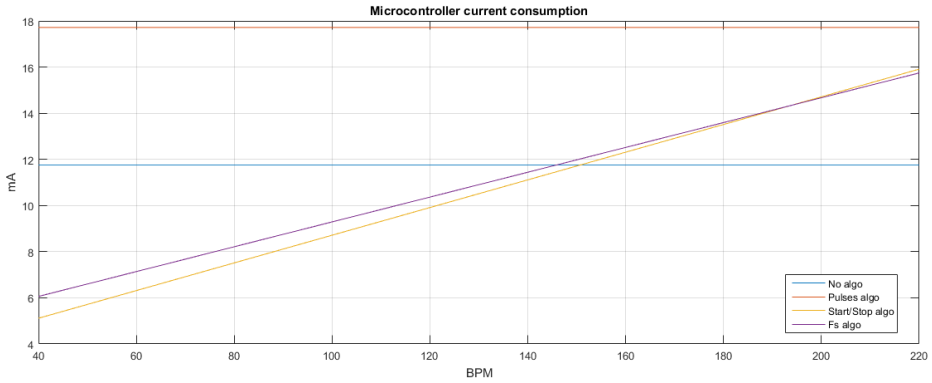


Figure 49: Microprocessor consumption using the library and without the library

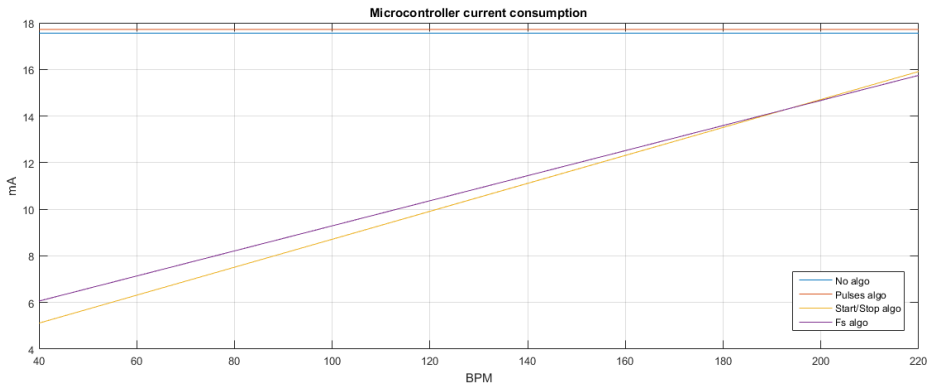


Figure 50: Microprocessor current consumption. Just obtain S_pO_2 vs obtain S_pO_2 efficiently with the different algorithms

Chapter 8. Conclusion and future work

Along this project we have determined that some algorithms can be applied to PPG signal gathering in order to optimize S_pO_2 current consumption. These algorithms as we have seen take advantage of the fact that for the S_pO_2 value we only need the amplitude of the PPG signal peaks. Therefore obtaining the PPG signal with the same accuracy all the time is not smart, as we will only care about the peaks.

The three algorithms tested apply different techniques to test this hypothesis. At first we did not know if these could be implemented, because the sensor we used has some limitations, however we have seen that it is actually possible. We created two measuring systems in which a computer interacts with the sensor through a microcontroller. These measuring systems have helped us to confirm that they can be implemented. To develop them we need to understand how the ADPD144 sensor works, how to program the STM32F4 microcontroller, how the measuring boards used work, etc. We have also created a current measuring system with which we have obtained the current reduction that can be achieved using the algorithms.

We have also proved that all of the algorithms reduce the current consumption of the sensor when measuring the S_pO_2 compared to a setup in which the algorithms are not working. The most efficient algorithm turns out to be the Start/Stop one. This algorithm stops sampling when the signal is not useful and starts sampling again when a peak is approaching. However this algorithm should only be used in situations in which the heart rate does not have too much variability. If the variability is too high and we do not configure the LPAP and HPAP start periods properly, the algorithm could run poorly. The Pulses algorithm is the worst one when it comes to reducing current consumption. It changes the number of pulses in real time depending on the current position of the PPG signal. However it will work better than the Start/Stop one, as it is continuously sampling. The Fs algorithm is the second one when it comes to efficiency. It samples the signal with different frequencies depending on whether we are close to a peak or not. However it is the best algorithm in the ratio $currentReduction/reliability$. This one will work fine most of the times, and the current reduction that can be achieved is almost the same as the Start/Stop one.

Processing these algorithms carry a processing load. This means that the microcontroller managing them will have to process a little bit more time for each sample. However we have determined that for the Start/Stop and the Fs algorithms the consumption of the microcontroller will be reduced for low heart rate paces. This is because it will have to process less samples in the LPAP. The pulses algorithm will slightly increase the consumption of the microcontroller, however the current reduction of the sensor is greater, which means that it still reduces the overall current consumption.

All in all, these algorithms could be implemented in devices that must work for long periods of time fed by a battery. They will reduce the current consumption and the device will last longer with the same battery. Patient monitoring 24/7 could be a possible implementation as long as the patient uses a finger S_pO_2 probe. Moreover these algorithms do not only obtain the S_pO_2 signal but the HR and HRV as well. Therefore they could also be used in smart watches to track information about the user.

There are some tasks that could be carried out in the future to continue with this project. The developed code in the microcontroller has been optimized up to a point. However further optimization could be done. Some signal processing techniques may be applied to obtaining the S_pO_2 value faster. Some code functions that are called several times per iteration could be implemented in assembly code.

Moreover the algorithms could be implemented in a future version of the ADPD144 silicon. This would be rather easy, as the algorithm does not use floating point in any part of the code but to obtain the final S_pO_2 value. This means that all the processing filters could be implemented in hardware. This way the microprocessor would not have to deal with the processing charge.

The designed S_pO_2 library can be ported to any project with the conditions commented in the implementation chapter. However it has some drawbacks. The ADPD144 registers are modified within the Library, which is not recommended, this should be done within the main project so that everything is properly arranged. This could be fixed in a future. One solution could be to define a couple of callbacks in the Library that warn the main program about these register modifications.

S_pO_2 quality should be tested in other parts of the body such as the wrist or the forehead. Some medical applications could benefit of the sensor if they can place it in a more comfortable place than the finger.

Last but not least, in a future the current drained by the microcontroller could be measured precisely if we isolated it in a breakout board. A breakout board would have to be designed along with the sensing resistor to measure the current drained.

Chapter 9. Appendix

9.1 S_pO_2 value validation

Once we had the S_pO_2 peak algorithm running, we wonder if the output value was accurate. We already knew it was, because the calibration formula had been obtained in a hospital in Boston for the sensor we were using, however we just wanted to verify it.

For this reason we decided to compare the output of our system with a reliable medical device. We asked for a pulse oximeter in a hospital to compare the results. This device can be seen in Figure 51. This device is able to measure the S_pO_2 using a finger probe.

In order to compare the value with our system we run both at the same time. Then, we hold our breath for a minute in order to reduce the oxygen saturation of the blood. Our purpose was to sweep some S_pO_2 values between 90 and 100 to check the precision. However we faced some problems along these tests. It turns out that the blood oxygenation does not vary instantaneously with the breath. This means that even if we stop breathing, the S_pO_2 will remain stable, and eventually it will fall. The delay shown here depends on the averaging applied to the signal and the delay of the oxygenation of the blood. This phenomenon can be seen in Figure 52. In this figure we can see the S_pO_2 over time. The first and the third red lines represent the points in which we stopped breathing, and the second and fourth ones the points in which we started breathing again. As we can see the S_pO_2 peak drop is out of these bounds, which means that there is a great delay. This delay in our system was around 30-40 seconds depending on the subject. For this reason we could not compare the S_pO_2 value, because it was impossible for us to achieve stable S_pO_2 values along the desirable range. Nevertheless we could check that the peak value obtained by both systems were almost the same.

If we had wanted to do the test properly, we would have had to achieve an slow sweep between the desired S_pO_2 values. This can be achieved breathing controlled variable mixture of oxygen and nitrogen in a hospital. This is how the calibration formula was originally obtained, and the proper way to validate it.

Figure 51: Medical oximeter used to compare the S_pO_2 Value

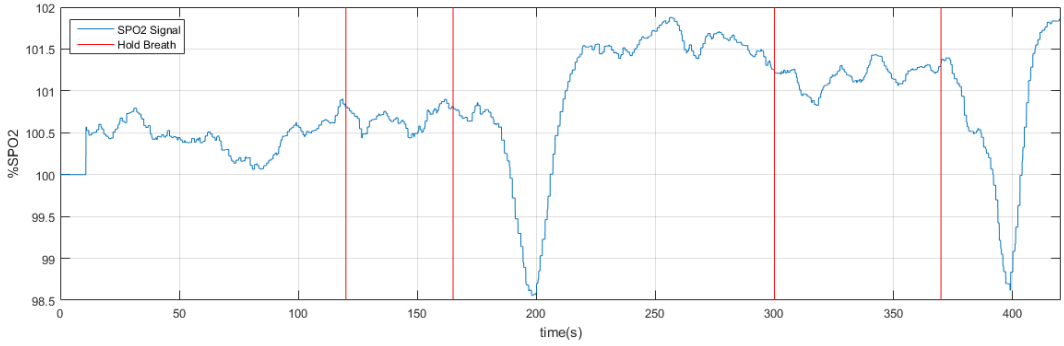


Figure 52: S_pO_2 holding breath

Chapter 10. References

[1] Michael Barr. *LabVIEW 7.1 : programación gráfica para el control de instrumentación*. Madrid etc. : Thomson, D.L., 2005.

[2] Enrique Berjano Zanón. *Amplificadores diferenciales y de instrumentación*. Valencia : Universidad Politécnica de Valencia, 2003.

[3] Leslie A. Geddes. *Handbook of blood pressure measurement*. Clifton New Jersey : Humana Press, cop. 1991., 1991.

[4] Steve Heath. *Embedded systems design*. Oxford : Newnes, 2003.

[5] WEBSTER J.G. *Design of pulse oximeters*. Medical Science Series. New York, NY 10016: Taylor and Francis Group, 1997.

[6] David C. Kuncicky. *Matlab programming*. Upper Saddle River : Pearson Education, cop., 2004.

[7] Sen M. Kuo. *Real-time digital signal processing : implementations and applications*. Chichester, England etc. : John Wiley and Sons, 2013.

[8] Norman Maclean. *Hemoglobina*. Cuadernos de biología. Barcelona : Omega, cop., 1979.

[9] Antonio Mánuel Lázaro. *LabVIEW : programación gráfica para el control de instrumentación*. Madrid : Paraninfo, 1997.

[10] Antonio Mánuel Lázaro and Joaquín del Río Fernández. *Programming embedded systems in C and C++*. Sebastopol : O'Reilly, 1999.

[11] Michael J. Pont. *Embedded C*. London : Addison-Wesley, 2002.

[12] Robert J. Schilling and Sandra L. Harris. *Digital signal processing using MATLAB*. United States etc. : Cengage Learning, cop., 2016.

[13] David Seal. *ARM architecture reference manual*. Harlow : Addison-Wesley, 2001.

[14] Scott T Smith. *Matlab : advanced GUI development*. Indianapolis, IN : Dog Ear, cop., 2006.

[15] Zijlstra W.G., Buursma A., and Meeuwsen van der Roest W.P. *Absorption Spectra of Human Fetal and Adult Oxyhemoglobin, De-Oxyhemoblobin, Carboxyhemoglobin, and Methemoglobin*. CLIN CHEM.37/9, 1633-1638, 1991.