UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

COMPACT ACQUISITION SYSTEM FOR HIGH RESOLUTION RADARS

A Master's Thesis Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya

by

Rodrigo Snider Fiñana

In partial fulfilment

of the requirements of the

MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Antoni Broquetas Ibars

Barcelona, October 2016

“Only those who will risk going too far
can possibly find out how far one can go.”

Preface to “Transit of Venus”
Poems by Harry Crosby

- T. S. Eliot (1888-1965) -

Acknowledgements (Agradecimientos/Agraïments)

M'agradaria agrair en primer lloc al professor Antoni Broquetas, que m'ha donat la possibilitat de poder realitzar aquest projecte. A més m'ha assessorat durant els 7 mesos d'investigació, responent-me els dubtes i guiant-me en tots els reptes que aquest projecte ha suposat.

També m'agradaria agrair al Departament de Teoria de Senyal i Telecomunicacions, a la Universitat Politècnica de Catalunya i a tot el professorat que ha format part de la meva formació al llarg d'aquests anys.

Agradezco a Pedro Yuste, Simón Rendón y Diego Huérfano, la compañía, los buenos momentos y la ayuda que me han brindado a lo largo del proyecto.

Me gustaría agradecer a mi familia, que me han apoyado en todo lo que he hecho, y que sin ellos no hubiese sido posible llegar hasta aquí. En especial a mis padres que me han dado la oportunidad de estudiar una carrera universitaria, y su apoyo a lo largo de mi vida.

Agradezco a mi hermano Daniel por ayudarme y estar a mi lado todos estos años.

Por ultimo agradezco a todas las personas y compañeros que me han dado su apoyo a lo largo de esta carrera y a lo largo de mi vida.

This work has been supported by the Spanish Ministry of Economy and Innovation (MINECO) in the framework of an I+D Project with code TIN2014-55413-C2-1-P

Table of contents

List of figures

Abstract

Current analog data acquisition systems are complex systems based on high speed FPGA and microprocessors. Digital technologies are growing faster; microcontrollers are implementing more powerful peripherals, like Analog-to-Digital Converters (ADC). This is the case of LPC4370, which is a microcontroller based on ARM cortex architecture that includes an 80 Mega-Samples per second ADC.

Nowadays, in radar applications, expensive digitizer boards are used for data acquisition. While an external ADC is sampling the received analog signal, an FPGA dumps the information from ADC to a fully equipped computer. The price for the whole solution is over thousand US Dollars.

This thesis includes the research carried out in order to design, test and analyze a low cost acquisition device. As LPC4370 is a very powerful microcontroller with respect to the market price, designing a system with these objectives is a very good choice. LPC link2 is a development board based on LPC4370 microcontroller for a price of 20 USD approximately. It also includes two different application approaches; the first one oriented to ground radar systems, and the second to radar systems mounted on unmanned aerial vehicles (UAV).

1. Introduction

This project is carried out at the Signal Theory and Communications (TSC), Department of Escola Tècnica Superior d'Enginyeria de Telecomunicacions de Barcelona (ETSETB), which belongs to the Universitat Politècnica de Catalunya (UPC).

Electronic technology advances every day, devices become obsolete in a shorter time and low cost devices with the same or similar performances appear. More and more devices are needed for obtaining information about the world, either through photographs or images obtained by radar for example. In a radar design, the digitizer is one of the most critical points at the system receiver side. Important parameters of digitizers as sample rate, memory depth, bit resolution and effective number of bits must be taken into account to fit the application requirements.

Contemporary commercial solutions for low speed digitizers (order of kilo-samples per second) have a cost of hundreds of USD, but in case of more than one Mega-sample per second, solution prices are greater than one thousand USD in most cases. These solutions are based on high performance microprocessors and FPGAs, which are usually expensive components. Microcontrollers are reduced computers that implement all peripherals within the same integrated circuit, these components are very cheap in comparison with other type of data processors. Nowadays, new microcontrollers include in their design better peripherals with better performances than years before. This is the case of NXP's LPC4370, which includes a very high performance analog-to-digital converter integrated on chip. This converter has a good resolution (12 bit) and can be configured for high speed sample rate (80 Mega-samples per second). The improvement of these microcontrollers offer the opportunity to reduce substantially the cost of high speed signal digitizer boards.

The main objective of this project is to research and make an analysis of the maximum performance of this microcontroller from a point of view of analog data acquisition. This evaluation will focus on radar applications, designs will focus in low cost acquisition devices, which will allow reducing costs of the overall system.

The secondary objectives are three: first, research about limitations of the microcontroller in acquisition speed terms; second, to perform a deep analysis of LPC4370 supported communication protocols, as USB and Quad SPI, and its

capabilities; and finally to design a system able to acquire digital samples from an analog signal and store the digitized samples in an external device.

Research is divided into three topics. The first topic is the data acquisition by means of the high speed ADC and its speed improvement. The second topic is the design of an acquisition system able to acquire data and send it to an external computer by using USB protocol. The third, is the design of a digitizer able to store acquired data in the same board than the microcontroller, using Quad SPI or parallel bus communications.

1.1. State of the art

Signal acquisition and digitizer techniques have evolved throughout years. Signal acquisition devices can be found in oscilloscopes for analyzing analog signals in terms of voltage and time. Digitizers are in many other systems like radio receivers, radar systems, among other applications. But it is important to know that the beginnings and the evolution of systems in order to improve current ones.

The analog-to-digital converter is the main component of a digitizer; it is in charge of performing the conversion from an analog signal to a binary array of data that describes the input signal. The first design of Flash ADCs was developed by Paul M. Rainey in 1921, this design was used in telephonic communications and it is recognized as the first design of PCM [1].

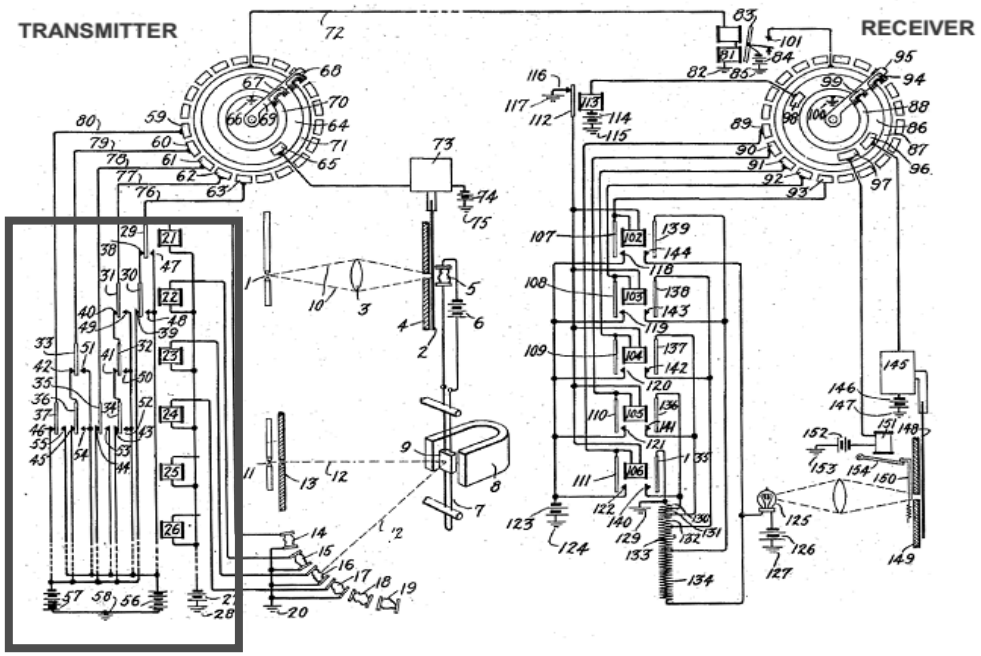


Fig. 1.1: First PCM, including a 5-bit flash ADC. *[1]*

Fig. 1.1 shows the design of the system that allows transmitting codified data using 5-bit PCM. The design on the left side corresponds to the transmitter. The squared part of the image shows the 5-bit ADC, the signal coming from the input is applied on a light-based system that transforms the analog input to digital output. This light system is based on focusing a light beam on a different photocell depending on the voltage of the input signal (32 photocells in total for 5 bits). Each photocell is connected to a set of relays which drive voltage at the 5 digital outputs depending on the codification of the voltage level.

ADCs technology was evolving from rudimentary techniques to the usage of vacuum tubes. In 1950's systems began to migrate their components to transistors and this becomes a great advance in the technology. Between 1950's and 1960's the field of communications and signal processing was widened as a consequence of the cold war and the increase in military research. First modular digitizer boards appeared in 1960's, pioneered by companies as Analogic and Pastoriza Electronics. These boards were compatible with computers of that time and became the first commercial approach to nowadays digitizer modules, with 8-bit resolution and 1MSps devices [2].

From 1960's, systems have been constantly evolving. More powerful acquisition devices were available, so more powerful systems were designed. Today, digitizer boards which operate using more than one ADC in parallel can be found in the market, multiple parallel ADC circuits allow sampling several Giga-samples per second. But the essence of the design of the full digitizer has been remained more or less constant from the beginning.

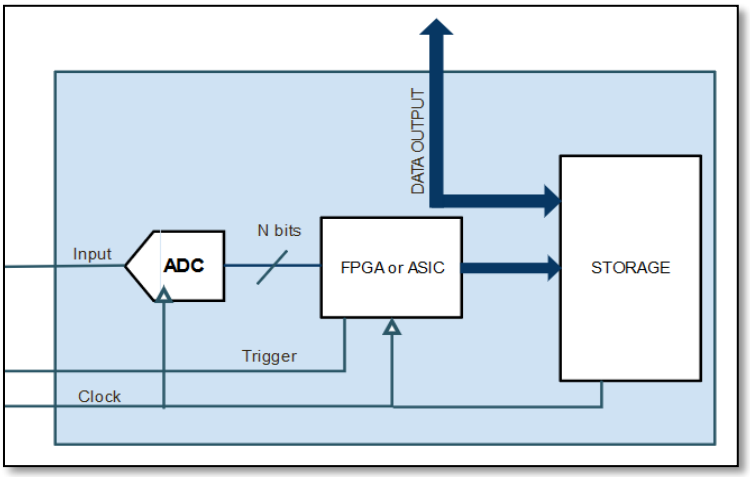


Fig. 1.2: Generic digitizer block diagram.

Nowadays, there are a lot of solutions for radar digitizers, most of them are based on FPGA or specific ASICs, that works in conjunction with a complex computer. The ADC converter samples the signal input and then bulks data on the FPGA or ASIC, depending on the solution. Then this FPGA adapts the data coming from the converter to the protocol used for storing data in the digitizer storage system (*Fig. 1.2*). This storage can be implemented by different types of memory devices, as random access memories (RAM), flash memories or other ones. Usually, this digitizer boards needs to operate all along with an external processor unit. This processor is in charge of moving

the acquired data from the digitizer to an external storage device in order to increase storage capacity.

But an alternative solution has arisen with the evolution of microcontrollers. These devices may include an ADC and memory banks inside the chip where the processor is located. So far this was not a feasible application for microcontrollers due to the lack of technology implemented on them. NXP has recently included an 80 Mega-samples per second ADC in a microcontroller, which is a convenient speed for many radar applications. Trying to adapt this technology to signal acquisition for high performance radar applications is an interesting research task, and it will be the objective of this project. The volume and mass of conventional solutions are critical when the radar receiver has spatial limitations if, for example, the device must be installed on an unmanned aerial vehicle. Then, conventional solutions become sub-optimal. These digitizers also have high power consumption, which is a drawback when the system has limited capabilities for power supply. Designs based on microcontroller can be much reduced in terms of size and power consumption.

According to this technology, a comparison among commercial devices that are currently in the market can be made with devices that are used for radar applications by the UPC, in either ground base or air-borne radars.

1.1.1. Ground base radars

Fig. 1.3: NI 5122 with PXI bus acquisition board.

NI 5122 (*Fig. 1.3*) is a two-channel acquisition board with a maximum sample rate of 100MSps per channel and 14 bits of conversion resolution [3]. The acquired signal is saved into an internal memory up to 512 MB. This board has a bus in order to map the acquired data out in a real time to a computer. Depending on the board version this bus can be PXI, PCI or PCI Express. PXI or PCI allow a maximum transmission data rate of

110MB/s, so it can provide a 1 channel streaming of up to 55MSps or two channels at 25.5MSps for a continuous operation mode. On the other hand, PCI Express admits a data rate flow up to 440MB/s, which can cope with a streaming at 100MSps in both channels at the same time. The price of this board is about 8,000 USD for 8MB of memory depth, and 12.000 USD for the 512MB version.

Fig. 1.4: National Instrument acquisition device, PXI bus.

Fig. 1.4 shows the equipment used in UPC ground station radars. The acquisition board stated before on PXI version, is mounted on a PXI bus, which communicates the board with the data controller (NI PXI-8108). Both, board and the controller are mounted inside a box (NI PXI-1031), that contains a power supply system. The price of this solution is about 15,000 USD (taking into account only one digitizer board).

1.1.2. Air-borne radars

Fig. 1.5: ADLINK PCI-9820 acquisition board.

Used in UAV radar applications at the UPC, PCI-9820 (*Fig. 1.5*) is a two channel acquisition board with a maximum sample rate of 65MSps per channel and 14 bits of

conversion resolution [4]. As in the device commented before, this digitizer devotes 512MB to memory depth and a bus to map out the acquired information to an external computer. The PCI bus allows continuously operating at the maximum sample rate on both channels, dumping all the data to the external computer. The power consumption of this digitizer board is about 9.5W, by calculations from the manufacturer information. The price of this board is around 3,000 USD.

Fig. 1.6: ARBRES-X SAR system.

This board is used on ARBRES-X [5] (*Fig. 1.6*) synthetic aperture radar (SAR [6]) based on LFM-CW radar (*Sec. 2.1.2*). ARBRES-X implements both the transmitter and receiver parts of the radar, inside the box. The receiver part, contains a PCI-9820 acquisition board, connected to a computer. The use of a computer in addition of this board, adds an extra weight and power consumption. The receiver consumes about 30W of power, and the total weight of this system is about 2 Kg.

Fig. 1.7: Octocopter Dji S1000.

ARBRES-X radar is mounted on an octocopter (*Fig. 1.7*) for earth terrain observation purposes.

1.2. Work plan

The project work plan is divided into two stages. The first stage is meant to gather all the relative information to the microcontroller general operation mode and to perform tests on software and firmware in order to set a first contact with the technology. The second stage consists in developing the final prototype, which is an optimization of the prototype for the target of application. This involves some peripherals of the microcontroller for the data acquisition full operation. *Fig. 1.8* shows the initially planned Gantt's diagram, where the stages before are defined as "Firmware and software testings" and "final design" respectively. The first stage was thought to last one month and a half, from early March up to mid-April. The second stage was meant to start in mid-April to be finished during the first days in September. Therefore, it was supposed to take three months, not taking into account summer holidays.

Several project setbacks appeared during the first stage of the project, forcing to drastically change all the planning. The ADC stage was delayed so long due to the lack of information about the operation of the converter along with the direct memory access peripheral (*Sec. 5.2*). The second delay was due to the SPIFI peripheral, the lack of information and the lack of support by the manufacturer. *Fig. 1.9* shows the modifications on the project timeline. The most relevant change that can be observed is the duration of firmware testing, which was extended from 35 planned days to 104 days.

The problem of the SPIFI peripheral has not been solved yet as the information provided by the manufacturer is not enough to ensure the correct understanding of the peripheral. Firmware testings took longer due to the tests performed on this peripheral, and so the slotted time for the next task was also reduced.

The initial planning did not foresee the amount of difficulties caused by the lack of detailed technical documentation. The planning was very well organized and structured. A good task division was performed at the beginning and that allowed performing deep research on this area with a good distribution of time and a balanced workload.

Fig. 1.8: Initial Gantt

Fig. 1.9: Final Gantt

2. Technological aspects

In this section, all the involved technologies in this project will be explained: first, an introduction to radar systems and their operation fundamentals; second, ARM cortex architecture and microcontroller characteristics; and finally analog-to-digital converters characteristics and different types.

2.1. Radar systems

Radio detection and ranging (RADAR) is a system that uses reflected electromagnetic waves produced by objects, in order to specify their location. Radar can measure different object parameters, as altitude (interferometry), distance or speed. Radars can be classified by number of antennas and by transmitted signal characteristics.

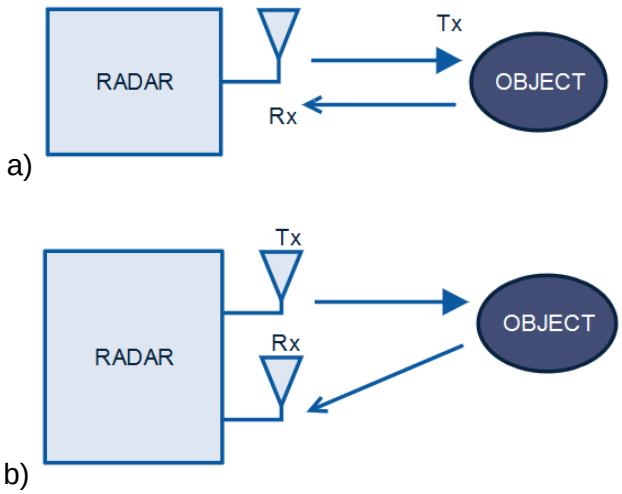


Fig. 2.1: Radar systems distinction by number of antennas: a) monostatic, b) bistatic.

Depending on the number of antennas a radar may be monostatic (*Fig. 2.1.a*), if uses a single antenna for both transmitting and receiving the signal; bistatic (*Fig. 2.1.b*) if uses two antennas, one for transmit and the other for receiving; and multistatic, that uses spatial diversity with combination of multiple monostatic or bistatic radars.

The classification can also be made depending on the transmitted signal. In this way, radars may be pulsed radars or continuous wave radars (CW) [7].

2.1.1. Pulsed radars

Pulsed radars are used to measure Doppler frequencies and, therefore, to measure objects in motion and speeds. These types of radar are based on the transmission of a train of pulses, modulated by the radar operation frequency.

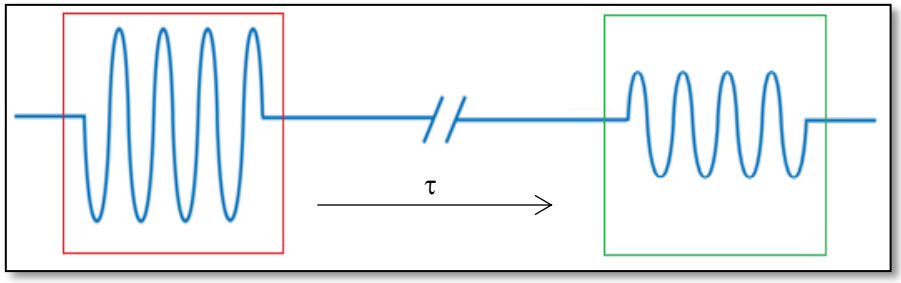


Fig. 2.2: Pulsed radar waveform. Transmission in red and reception in green.

Fig. 2.2 shows a typical waveform for this type of radars. In this example, the transmitted signal is reflected by the object when the wave arrives to it, the reflected wave has the inverse phase than the impinged wave.

$$R = c \cdot \frac{\tau}{2} \tag{2.1}$$

The time difference between the transmitted and received pulse (τ) depends on the range of the object (*Eq. 2.1*). Doppler frequency is extracted by the variation of the range of the object, due to the time between two pulses. If the range cannot be extracted directly by time difference due to the lack of precision in calculations, object speed can be extracted from spectrum variations of the received signal. These variations are due to the Doppler effect of the moving object. The movement modifies the reflected wave frequency, so speed can be calculated from this variation (Doppler frequency).

2.1.2. Continuous wave radars

Continuous wave (CW) radars transmits a continuous signal for object detection, so the radar must transmit and receive the signal at the same time to maintain wave continuity. There are two types of CW radars according to the frequency components: multiple frequency (MF) radars, based on transmission of a multiple tone signal, and linear modulated in frequency (LFM) continuous wave radars.

MF-CW radars:

Range detection of an object can be performed with no need to modulate and demodulate the transmitted signal. To perform this detection technique, the radar must compare the received signal phase with the transmitted one. The range calculation (*Eq. 2.2*) depends on the radar operating wavelength (λ) and the phase difference (φ).

$$R = \frac{c \cdot \varphi}{4\pi f_o} = \frac{\lambda \cdot \varphi}{4\pi} \tag{2.2}$$

As deduced, if the radar operates with a shorter wavelength the maximum range which can be detected with this technique is reduced. When the phase difference is greater than 2π, longer ranges can be confused with lower ones. In order to reduce this effect MF-CW radars were created. As the phase difference due to distance depends on the frequency of the signal, by using two different sinusoids in the same signal the phase difference between the transmitted and received waves will be different in each frequency component. Therefore, the range can be determined by the variation in phase between both frequencies (*Eq. 2.3*).

$$R = \frac{(\varphi_2 - \varphi_1) \cdot c}{4\pi \cdot (f_2 - f_1)} \tag{2.3}$$

Now the frequency difference can be tuned in order to be a low value for maximizing the maximum detectable range, independently of the radar operating frequency.

LFM-CW radars:

These radars are based on the modulation of a shape in time for the transmitting signal. Then the frequency of this signal will be linearly modified in time.

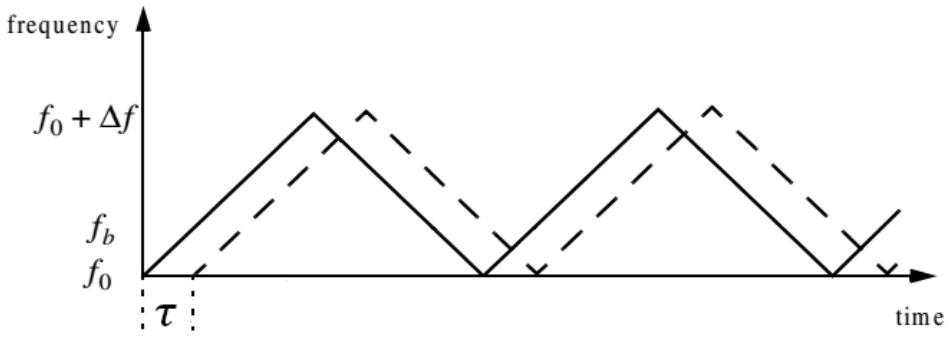


Fig. 2.3: Linear frequency modulation of a triangular shape. *[7] modified.*

A relation of the frequency of the transmitted (solid line) and received (dashed line) signal of LFM radar is shown in *Fig. 2.3*. As previously commented, the frequency of this

signal is changing along time, describing a shape (in this case triangular). The period of the frequency-modulated shape is called pulse repetition frequency (PRF). The received signal corresponds to an object reflection. The difference of frequencies at a given time instant is known as beat frequency (f_b) and its value depends of the object range and the Doppler Effect. If the object is stationary, the range can be obtained directly by the time between transmission and reception of the same frequency value (*Eq. 2.1.*).

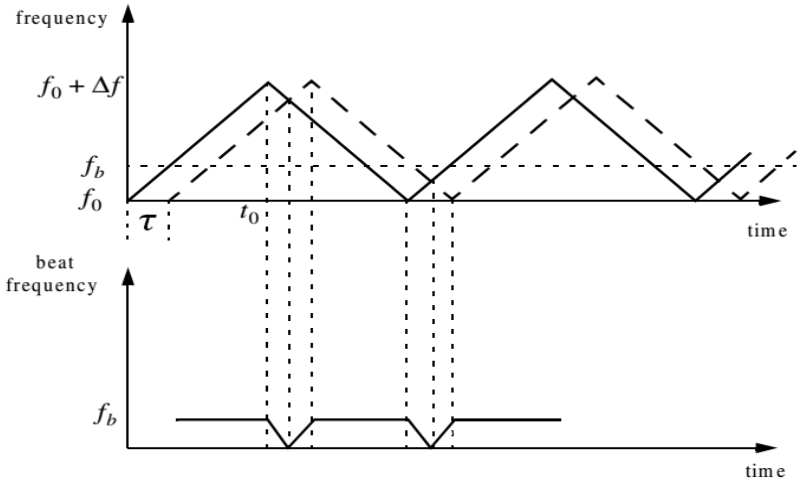


Fig. 2.4: Static object reflection characteristics. *[7] modified.*

When a static object is impinged by the transmitted signal, the beat frequency at the receiver is as in *Fig. 2.4*. If the object is not stationary, the beat frequency changes depending on the movement direction of that respect to the radar and its speed (Doppler Effect).

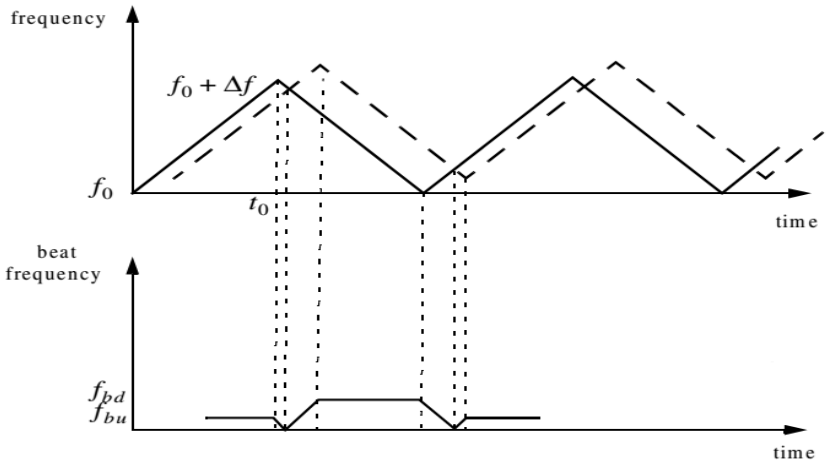


Fig. 2.5: dynamic object reflection characteristics. *[7]*

Fig. 2.5 shows the beat frequency result for a non-stationary object. Now two different beat frequencies are defined depending on the slope of the signal, they are (f_{bu})

for ascending and (f_{bd}) for descending slope. The differences are due to Doppler Effect, and its effects in range and range rate can be calculated by (*Eq. 2.4*) and (*Eq. 2.5*) respectively.

$$R = \frac{c}{8 f_m \Delta f}(f_{bu} + f_{bd}) \tag{2.4}$$

$$R_r = \frac{\lambda}{4}(f_{bd} - f_{bu}) \tag{2.5}$$

Range rate describes the object speed from the point of view of the radar position, as a factor of distance.

2.2. ARM cortex-M

ARM cortex is a microcontroller family based on ARM-architecture microprocessors. A microcontroller (also called MCU) is a small computer that integrates a processor unit, memory and peripherals inside the same integrated circuit. MCU are specifically designed for embedded applications.

ARM [8] is a Reduced Instruction Set Computer (RISC) architecture, based on 32 bit instructions. The instruction list comprises a few basic instructions; it makes the microprocessor structure simpler and allows less power consumption than using complex instructions. For this reason, these processors are better than Complex Instruction Set Computer (CISC) for small and low power devices.

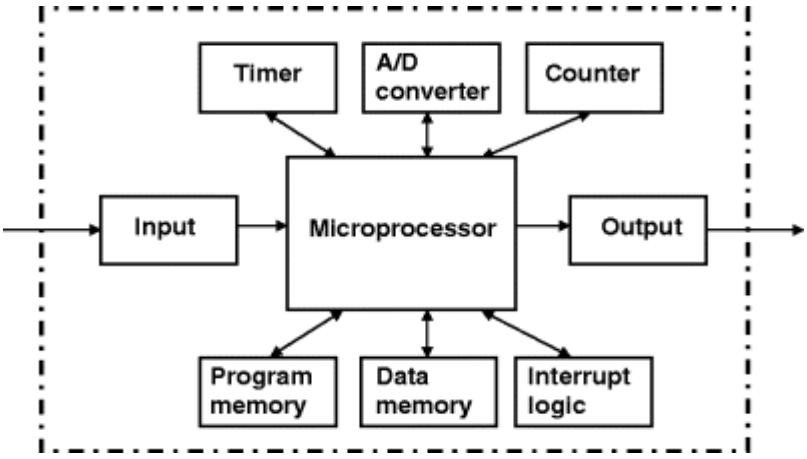


Fig. 2.6: Microcontroller internal structure. *[9]*

As *Fig.* 2.6 shows, microcontroller comprises a microprocessor and a set of peripherals and memory banks. All peripherals and memory banks are mapped to the

microprocessor by data busses, known as AMBA high-performance bus (AHB). Each peripheral have associated registers mapped to addresses, which can be accessed by the microprocessor by reading or writing a specific address using AHB; registers allows the interaction between the microcontroller core and the available peripherals. Microcontroller AHB bus also connects the core processor to input or output peripherals, allowing the external interaction with the environment, and with memory banks.

Program memory (that is the memory in charge of programming code allocation) can be located at the same bus as the rest of peripherals or data memory, or have a separated bus specifically meant for that purpose. This depends on the organization of the memory architecture. Microcontroller implements a set of memory banks within the same integrated circuit for data allocation purpose; random access memories (RAM), read only memories (ROM) and it usually includes electrically erasable and programmable read only memory (EEPROM). Flash memories are used for program code allocation. Depending on the design flash memories can be included inside the integrated circuit (Built-in flash), or externally allocated in the board (flashless devices).

Some characteristics, which must be taken into account for the proper selection of the microcontroller, are explained at next sections.

2.2.1. Clock frequency and instruction pipelining

Pipelining is a technique that uses multiple instruction execution at the same clock period. Instead of sequentially executing a single instruction per cycle, each instruction is split into stages, so different stages from different instructions can be executed at the same time, improving the total throughput. To perform this division, the microprocessor is also physically divided into stages. Therefore, increasing the number of stages implies reducing the number of electronic components that each stage has, as the processor has been divided into more stages. Using less electronic components in every stage allows increasing clock frequency, so at the end the number of instructions per second will increase.

The instruction execution line comprises several functional stages, which commonly are: instruction fetching, that is, reading the instruction from program memory, and moving this instruction to the execution register; and execution, when the instruction is executed. But this process can also be divided into many more stages, like instruction

decoding, memory accessing, among others. Modern processors with high performance includes ten, twenty, or more stages [10].

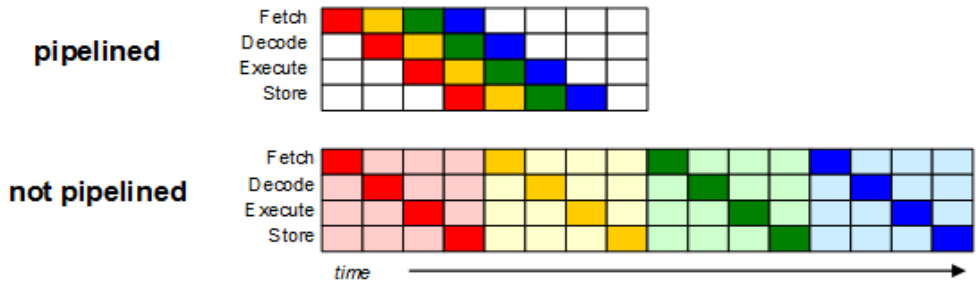


Fig. 2.7: Instruction with/without pipelining comparison. *[11]*

Fig. 2.7 shows a 4 stage instruction comparison between pipelining and non-pipelining methods. The vertical axis corresponds to instruction stages, and the horizontal axis to the clock period. Red, yellow, green and blue boxes are the first, second third and fourth instructions respectively. As shown, using pipelining the duration of these four instructions is reduced; different stages from different instructions can be performed at the same time. Hence, the effective time, in mean terms, is reduced by a ¾ factor by a 4-stage pipeline. It can be observed that the latency (instruction time between fetch and store) is not reduced with this technique, so the total instruction time will not depend on the level of pipelining, but the execution throughput does.

The instructions that break the normal program flow (branch instructions) must be taken into account in order to optimize the instruction execution. This type of instructions causes inefficient pipelining as further instructions, which are being executed when the break occurs, are discarded, reducing the total throughput. A method called branch prediction is used to increment throughput when the branch is executed. This method allows modifying the concurrent instructions when branch is fetched or decoded (depends on the architecture), losing only 1 or 2 stage time in comparison with losing all the instruction time.

2.2.2. Memory organization

There are two types of memory architectures, Harvard architecture and Von Neumann architecture [12]. Both are used in microcontrollers, and it is important to know which one is being used by each microcontroller in order to optimize its code execution. The type of architecture defines how the memory and peripherals are mapped to the central

processor unit (CPU). Von Neumann architecture is mostly used in microprocessors. It is also used in microcontroller structures but it is not that important. On the other hand, Harvard architecture is used in embedded solutions, and it is usually implemented on microcontrollers.

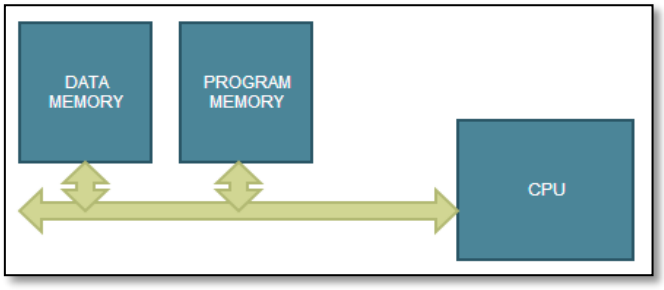


Fig. 2.8: Von Neumann architecture.

Von Neumann is a memory architecture that shares the same data bus for data memory and program memory (*Fig.* 2.8). In this architecture, data memory and program memory can be the same physical structure (usually RAM). Using this architecture, the microprocessor cannot fetch instructions from memory and perform memory operations at the same clock cycle; so instructions with memory access must wait for finishing the operation before the next instruction is executed. This architecture is useful when the program code needs to be allocated in RAM, as data is normally composed by variables that are stored in RAM, the processor does not need to use 2 separated memories.

The processor is slowed down due to the instructions meant to memory operations. If complex instructions (more bits than memory bus has) are executed, it is forced to perform more than one fetching operation per instruction with this architecture, and so it is also slowed down. On the other hand, this structure easily allows to self-modify the program code.

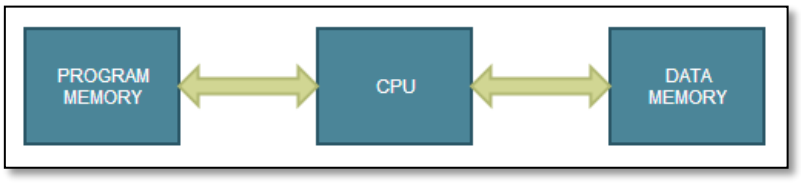


Fig. 2.9: Harvard architecture.

Harvard is a memory architecture based on separating program memory from data memory. Both memories are not sharing the same bus (*Fig.* 2.9), what means that the processor can execute program code at the same time that is writing or reading data from memory. By using this architecture, program memory is usually a ROM memory.

The width of data bus and program bus can be different as well. This architecture does not allow self-modifications of the code due to the program memory read-only characteristics.

In reference to previous section (*Sec. 2.2.1*), it can be deduced that if Von Neumann architecture is used, instructions with memory access will reduce throughput in the same way as break instructions does.

Cortex-M4 family has Harvard memory architecture and Cortex-M0 implements Von Neumann architecture.

2.2.3. Peripherals

Peripherals are the main characteristic of microcontrollers and define the structure of the MCU itself, making each one different from the others. The number of available peripherals increase design complexity, and therefore high performance general purpose microcontrollers have a big number of peripherals in order to fulfil a great number of possible applications.

A peripheral is a circuit or block that complements the microprocessor functionalities, allowing it to get external information out of the CPU. It provides communication, external interaction and storage capabilities to the integrated circuit. There are four types of peripherals: those which implements communication protocols, timing and program interruption, digital input/output and analog peripherals.

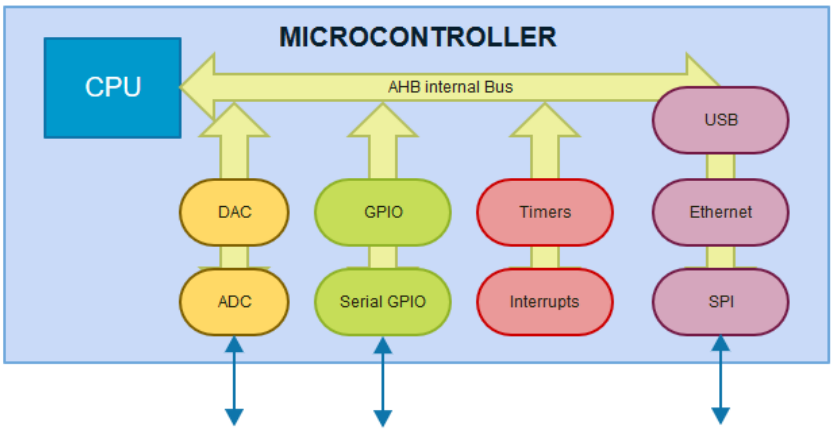


Fig. 2.10: peripheral organization in microcontroller structure. *(Yellow, analog IO; green, digital IO; red, timers and interrupts; purple, communication peripherals)*

Communication peripherals allow using communication protocol standards to exchange information with other devices, like other microcontrollers or other systems. There are many protocols that can be included in a microcontroller: Ethernet, SPI, CAN, USB, and many others.

Digital input/output ports are used for exchange digital information with the environment. Usually they are used to create a user interface with LEDs or LCDs for output information and switches to receive user information.

Analog input/output peripherals are used to digitalize/create analog signals. These signals are obtained or created as a combination of discrete values in time, with discrete amplitude levels.

Timing peripherals are used for clock edge counting. Since clock frequency is known, then time intervals can be defined by counting clock periods. Interruption service is a peripheral able to stop the normal instruction flow in order to execute special instructions when an event occurs. This event can be internally generated, for example, due to an internal CPU error or exception, or externally generated by other peripheral. Depending on the complexity of the microcontroller, the interrupt peripheral may have a greater number of interruptions types available, grouped into a vector known as Nested Vector Interrupt Controller (NVIC), which can handle a variable number of interrupts depending on the microcontroller design.

2.2.4. Integrated circuit package

Packaging is an important parameter for the hardware design stage of the application. The type of used IC package on hardware designs will define the complexity of the board in design and fabrication terms, it also will define the prototyping cost. Microcontroller packages options are limited by the manufacturer, usually high performance MCUs are sold only on high density packages. Simpler MCU are sold in more maneuverable ones, due to the lower pinout number and to allow homemade prototyping for amateur designers and developers.

Dual in-line package (DIP) is oriented to prototyping designs; the developer can easily mount this package in a protoboard or soldering it on a breadboard and testing the application with no need of designing a PCB. But the low pin density of this package precludes the fabrication of high number of pins (2.54mm pin pitch usually).

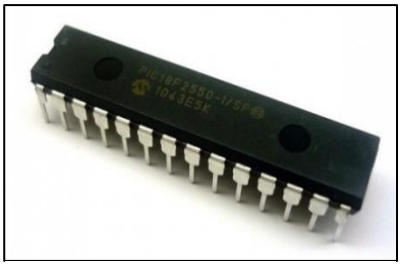


Fig. 2.11: DIP package

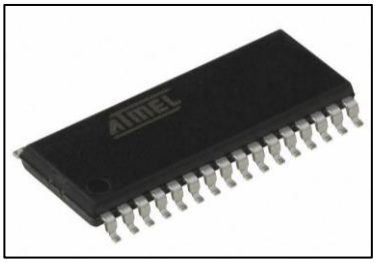


Fig. 2.12: SOP package

Small outline package (SOP) is similar to DIP packages, but oriented to surface mount. SMD techniques allow incrementing pin density (1.27mm pin pitch usually), but the developer is forced to design and solder a PCB for the development. Manufacturers usually build SOP and DIP packages for the same chip in order to allow prototyping and integration on smaller designs.

Quad flat package (QFP) is the result of increasing pin density of SOP. In order to increase this number and reduce the IC size, the package design uses the four sides of the chip to include pins. This type of package is used in most of the cases in more complex microcontrollers due to the large number of pins that allow mapping.

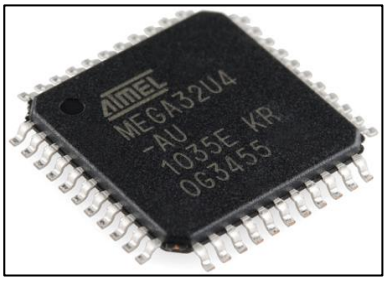


Fig. 2.13: QFP package

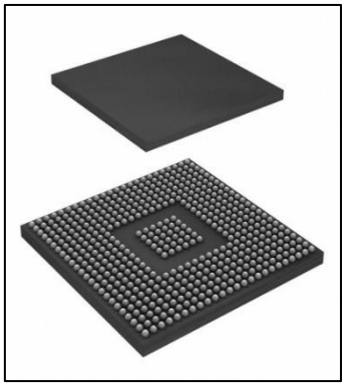


Fig. 2.14: BGA package

Ball grid array (BGA) is used in the most advanced microcontroller designs. This packaging implement an extreme pin density, based on the total pin occupation of the integrated circuit area. This package forces the user to design a very complex PCB with more than two layers most of the times, and to follow a complex soldering process. Pins are placed on the bottom side of the component with little solder balls, and the soldering process must be performed by using special ovens. The component must be correctly located on the PCB before melting the solder balls in the oven, adjusting the component on the right position.

2.3. Analog-to-Digital Converter

Data storage and data processing always belong to digital domain. When a digital system needs to process analog signals, these must be previously transformed into digital form. Digital representation of analog domain is performed by discretizing the signal into a set of samples; each sample contains information about the amplitude of the signal at a given moment.

An analog-to-digital Converter (ADC), is a device that converts an analog voltage value to a digital value expressed in bits (sample), its value gives an approximation to the analog input voltage [13]. Sample value is the result of the comparison between the input and a reference voltage, extended to all the possible ADC output levels (2^n, where n is the number of bits). The result of this relation is rounded to the nearest integer; bits cannot take non integer values. Note that in order to obtain an accurate representation of the input, is very important to keep constant the reference voltage. Noise at the reference implies noise at the representation, even if the input is not noisy.

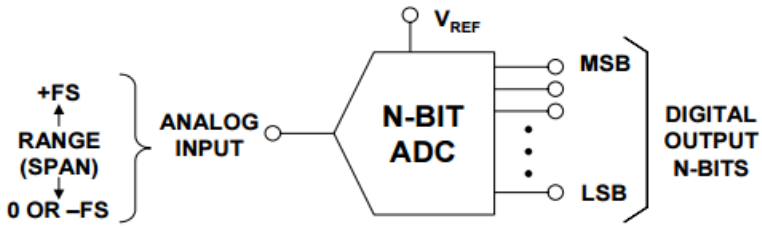


Fig. 2.15: ADC symbol. *[13]*

Symbol of *Fig. 2.15* is used to represent ADC component in block diagrams and circuit schematics. Analog input pin is drawn at the left side of the symbol, and the corresponding N binary outputs at the right. Output pins are arranged from most significant bit (MSB) to the least significant bit (LSB).

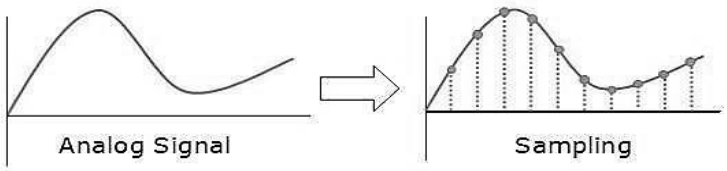


Fig. 2.16: Analog-to-digital signal conversion. *[13]*

Signal conversion process is divided into three steps; sampling, quantization and coding. At the end of the process, the signal is discretized and digitalized as in *Fig. 2.16*. In digital domain, signal is only determined at discrete time instants (samples), the time

lapse between two consecutive samples is called sampling period (inverse of sampling frequency). Division of the time line in discrete points is performed at the sampling step. In order to maintain synchronism and signal coherence, sampling frequency must be constant; if sampling frequency is not constant, or the capturing process has been paused, then samples must include additional information about the sampling instant.

Once the sampling time is defined, next step is to quantize the input voltage on output levels. Each sample can take a limited output values, depending on the input voltage (quantization), this value is expressed at the output in a unique way (coding), which depends of ADC bit-resolution. At the end, the output will express, in bits, the relation between the analog input voltage and the reference voltage defined for the converter. It can be mathematically expressed as in *Eq. 2.6*.

$$S = round\left(\frac{2^n \cdot V_{in}}{V_{ref}}\right) \tag{2.6}$$

This equation can be graphically expressed in next example: it is considered a 3 bit resolution ADC, with a reference voltage of 7 volts, next output relation due to an input voltage is obtained.

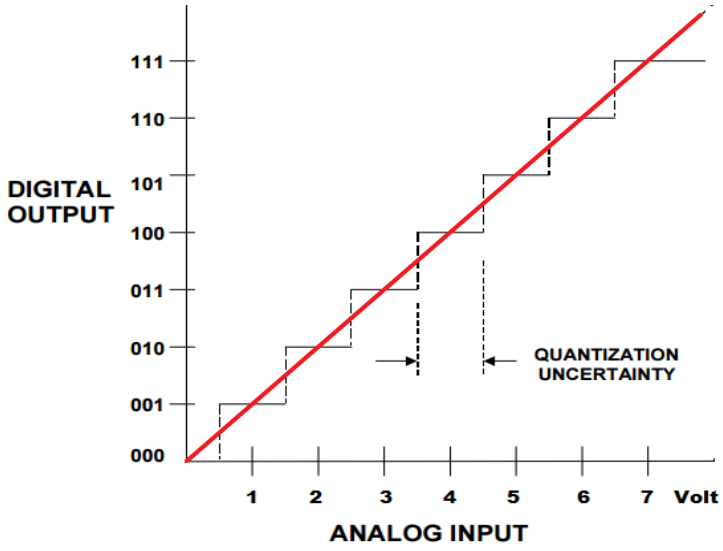


Fig. 2.17: Output response example.

Fig. 2.17 illustrates the output response of an ADC. In black, the ADC ideal output response to linear input voltage variation. In red, the analog representation of the input variation. To avoid conversion errors, the signal at the device input must be within a maximum and minimum voltage dynamic range (DR). An input voltage out of these values will not be correctly converted and can cause device damages. In some cases

DR is defined by the difference between the reference voltage value and 0V. Quantization process has a certain uncertainty (*Eq. 2.7*), which corresponds to 1 LSB. This uncertainty comprises the possible real input values that are not covered by the resolution.

$$U(q) = \frac{V_{ref}}{2^n} \tag{2.7}$$

Quantization uncertainty does not severely impact on the acquired signal whether the signal amplitude at the input is much larger than that value. For small signals, quantization error becomes more significant.

There are many types of ADCs [14], [15], depending on the working principle and the design. Most common types are parallel, delta-sigma, successive approximation and pipeline converters. The parallel converters are the fastest (in sample rate terms), but design is limited in bit resolution. In the other hand Delta-Sigma are the slowest converters, but with the greater bit resolution.

2.3.1. Parallel or Flash ADC

These types of devices are based on the direct comparison of the input voltage with all the possible voltage levels, which depends on its number of bits. At the input pin, signal is divided in $2^n - 1$ branches, each branch is compared with a fraction of the reference voltage and the result of the comparison of every voltage level is codified to binary and outputted.

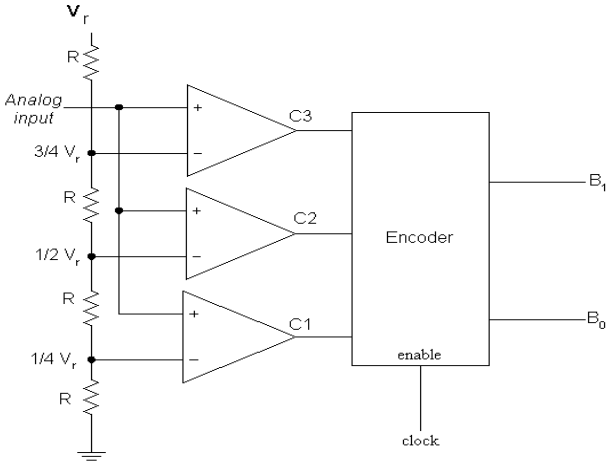


Fig. 2.18: Flash ADC configuration.

Internal circuit schematic is shown at *Fig. 2.18.* It is composed by 2^n resistors of same value (where n is the number of bits of the ADC), $2^n - 1$ comparators and a binary

encoder. Resistors are acting as voltage dividers (Kirchhoff's Law), dividing the reference voltage to different $2^n - 1$ values. In the example of the figure, n is 2 bits. V_i is the voltage at the i-th node of the voltage divider, which are 1/4V_r, 1/2V_r and 3/4V_r. The relationship between the reference voltage and the I-th node voltage is shown at (*Eq. 2.8)*.

$$V_i = \frac{V_r}{2^n} \cdot i \tag{2.8}$$

Each V_i is compared with the analog voltage input, if the input is lower than V_i, converter's output will be a logic '0', otherwise will be '1'. Finally, encoder transforms the $2^n - 1$ comparator outputs, to n-bits binary codification; depending on the ADC manufacturer design criteria it can be BCD, two's complement, gray, or other codifications.

Parallel ADC design suffers from dynamic errors if no sample and hold (S&H) structure is used. Dynamic error appears when high frequency signals are injected at the input pin of the converter, non-linear capacitances due to semiconductor junctions can induce signal distortion. Also if the comparator outputs are not driven at the same time, values can be confused with the next sample ones, so it is important the input signal to be equally distributed to all comparators. Impact of these errors increases with the bit resolution of the device and the operating sample rate. Sample and hold circuits in conjunction with previous design, allow eliminating dynamic errors by ensuring that the input signal at the comparator is not changing while clock edge occurs. Also the input capacitance of sample and hold is non-linear, but its value is smaller than the previous case, so the dynamic error due to this effect will be also reduced.

This type of ADC is the fastest one, but also is the most expensive due to the large number of components that it includes (exponential with n-bits). Parallel ADCs designs are then limited on resolution, usually lower or equal than 12 bits. Increasing the resolution by 1 bit means to multiply by 2 the total number of components.

2.3.2. Delta-Sigma ADC

Delta-Sigma ADCs uses the oversampling method for performing the analog-to-digital conversion process [16]. The input signal needs to be relatively slow in comparison with the converter, a block known as delta sigma modulator samples multiple times the signal (Oversampling).

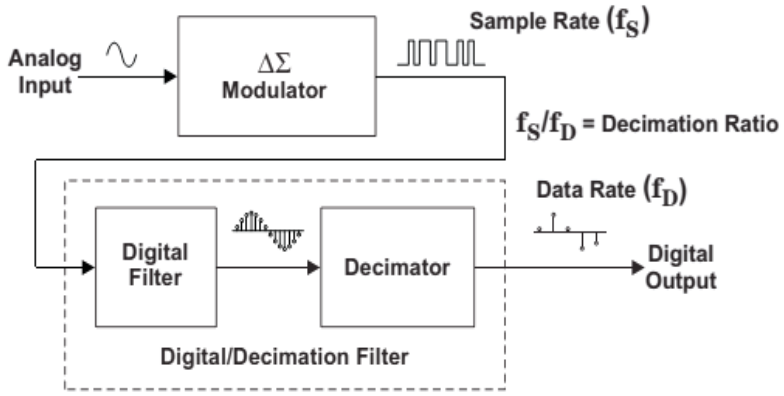


Fig. 2.19: Delta-Sigma ADC block diagram.

As shown in *Fig. 2.19,* these devices are mainly composed by an oversampler (delta-sigma modulator), and a digital filter followed by a decimator.

The modulator objective is to digitize the input signal and reduce noise at the lower frequencies. To reduce the noise, modulator implements a function that is called noise shaping, which consists on moving the noise from the lower frequencies to the higher ones out of the bandwidth of the device. The output of the modulator is a digital Pulse Width Modulated (PWM) signal. Duration of these pulses (duty cycle) are proportional to the input signal voltage value.

The digital filter is a low pass filter, it transforms the PWM signal to samples and suppresses the higher frequencies, where the lower frequency noise components are allocated after the modulator. Finally the decimator reduces the number of samples at the output of the filter, in order to reduce the amount of the output data, but without losing information.

Delta-Sigma ADCs are used for high resolution and low noise measures at low frequencies. These devices can have a large number of bits, but with reduced input bandwidth.

2.3.3. Successive approximation converter

This type of converter is based on giving the better output approximation due to the input signal, using an algorithm based on successive comparison between the output and the input. Each comparison the error is bounded. At the last iteration the error is minimized

to quantification uncertainty, and the comparator outputs the best approach. Next figure (*Fig. 2.20*) shows the block diagram of a successive approximation converter.

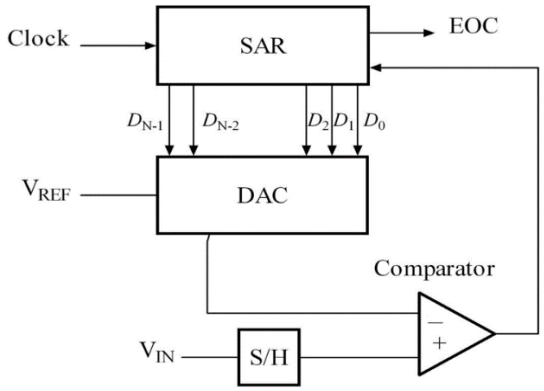


Fig. 2.20: Successive approximation converter block diagram.

Sample and hold discretize the input signal, and then holds voltage value until the approximation process finishes. S&H output voltage enters to a comparator that correlates it with the signal generated at successive approximation register (SAR) block. SAR modifies this value until the difference between the input $V_{S\&H}$ and V_{DAC} becomes minimum.

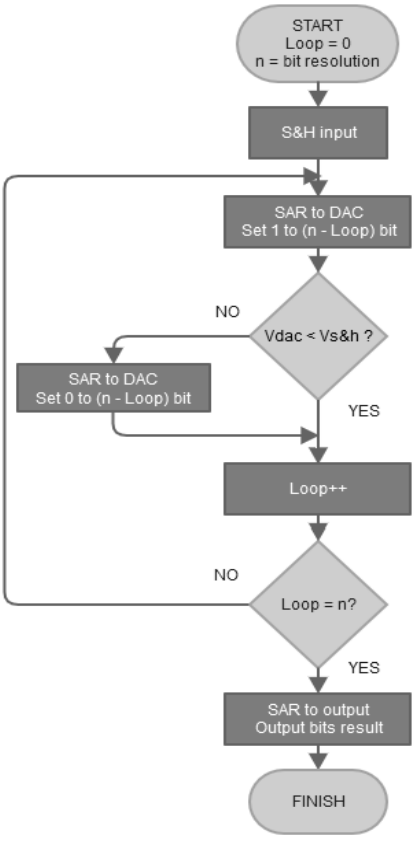


Fig. 2.21: Successive approximation algorithm flow chart.

The algorithm used by SAR block for minimum difference searching is shown in *Fig. 2.21*. Algorithm consists on first sample the signal, hold the value, and then perform a loop in order to approximate the output. At first iteration, MSB of the SAR output will be set to logic '1', the remaining less significant bits will stay cleared. DAC will convert this word on analog domain (V_{DAC}), and compared with $V_{S\&H}$. If $V_{S\&H}$ is greater, the loop will continue to the next iteration; if not, previous set bit will be cleared and then continues with the iteration. At next iteration, next lower significant bit is set, and the process is repeated. This iteration is performed for each output bit of the converter, and when the process finishes, the result is outputted.

This method is approaching to the real input value by dividing by 2 the error domain each comparison loop. The working principle is similar to parallel converter, but performed in 'n' steps. The conversion speed is slower than parallel converters but higher than Delta-Sigma converters, and it depends on the bit resolution of the ADC: more bit resolution, lower maximum sample rate. The operation principle allows increasing the bit resolution respect to parallel methods, as the number of components not depends on the bit resolution. On the other hand, the design of these components is complex, and SAR blocks are expensive. They are widely used for applications that need acquisition frequencies greater than 10KSps.

2.3.4. Pipeline converter

This type of ADC is based on the separation of the acquisition process in stages. These stages are composed by low bit-resolution flash analog-to-digital converters. Each stage conversion process consumes a time lapse, the total conversion time is given by the summation times taken by every stage.

The total number of ADC's bits, are distributed in all stages. Each stage has a fraction of the total number of bits, and this number is associated to the parallel ADC that each stage includes. The first stage has the most significant bits, and the last stage the less significant ones.

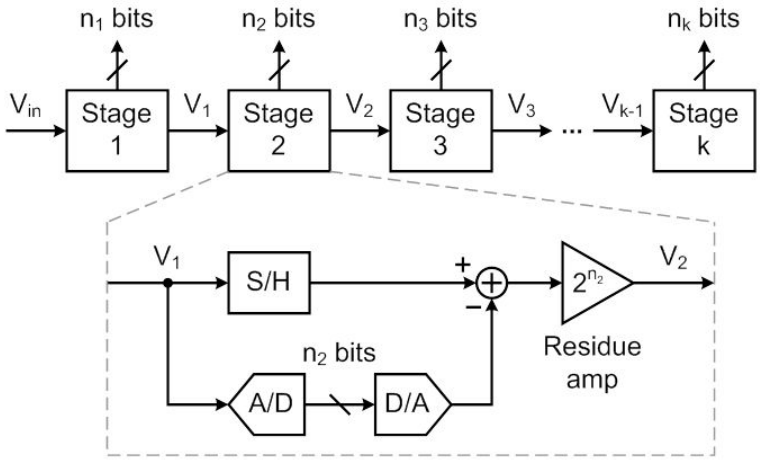


Fig. 2.22: Pipeline converter block diagram.

Fig. 2.22 shows the design block diagram of pipeline converter. Signal enters to the converter and get into stage 1, in this stage, signal is sampled and hold and compared with the converted value of the stage (using parallel ADC + DAC). Output of each stage is n_2 bits. The residue from the error amplification between acquired bits and the signal is amplified by 2^{n_2}, that it comes from the stage output quantization error and next stage dynamic range conditioning, and moved to the next stage. As each stage has a sample and hold block, conversion duration of each stage is 1 clock period.

This type of ADC reduces the number of components in comparison of parallel ADCs due to staging scheme. As the number of components needed for a flash ADC is exponentially related with the bit resolution of this, using smaller parallel converters in each stage, the total number of components are $2^{n_2} * N_s$, where N_s is the number of stages; being $n_2 * N_S$ the bit resolution of the pipeline converter. When the number of stages is increased, the converter can dispose of higher number of bits, but the ADC conversion latency is also increased.

This converter can sample up to 100MSps with 8 bit resolution, 16 bit resolution can be achieved at lower operation frequencies. This type of ADC is recommended to use in high resolution and medium sample rate applications. It is very efficient in complement count and has the lowest power consumption for high resolution converters. This are faster than successive approximation, but with less bit resolution; and slower than flash converters but with greater resolutions.

3. System definition

This project consists on the design and implementation of a digitizer board prototype. The system is based on a LPC4370 microcontroller and it is implemented with a development board (LPC Link2) provided by NXP. This design will need be adjusted within certain limits and requirements to fit specifications.

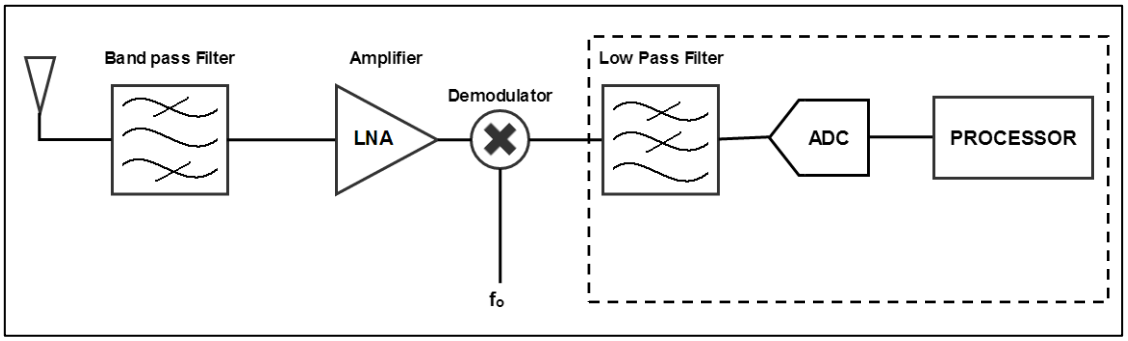


Fig. 3.1: Receiver block diagram.

Fig. 3.1 shows the typical block diagram of a communications system receiver. The signal arrives in the antenna, and then it is band pass filtered keeping the useful bandwidth. Finally, it is amplified with a low noise amplifier (LNA). Once the signal is amplified, the result is mixed with a signal of frequency f_o (which corresponds to radar operation frequency). This mixing process is called demodulation. At the output of this block the signal is a low pass signal. Before entering to the ADC, the signal must be filtered with an antialiasing filter in order to remove higher frequencies noise contributions. Its cutoff frequency, in order to accomplish the Nyquist theorem, is the half of the sample rate of the ADC. In this project, the blocks to be designed are the ones dash-squared in the previous figure.

The first requirement is that the system must be compatible with other receiver elements to be included in practical applications. Hence, it must have an input which allows the injection of an analog low pass signal, which will be digitally converted. After the conversion, the device must save the acquired samples in a storage device. This storage device can be integrated in the previous design, making it in a system for radar applications that needs portability. It may also be external to the system, for example using a computer as an intermediary to store the data for applications that does not need portability, such as ground base radars.

Note that these boards can be used not only for radar applications but also for other types of design and solutions based on data acquisition. Despite this fact, the design performed in this project will be only focused on radar applications.

The second requirement is that the system must have a low noise level and a high sample rate. Commercial digitizers, are linked to “quality parameters”. These parameters are spurious-free dynamic range (SFDR), signal-to-noise ratio (SNR) and signal-to-noise and distortion ratio (SNDR). The prototype should approach the values provided by commercial digitizers. On the other hand, sample rate should be as fast as possible, taking into account technology limitations, even though there are many applications that can use 100KSps digitizers.

The third requirement is that the acquired signal samples must be synchronized in time conserving integrity. In radar applications it is important not to lose the time line continuity in the samples, so losing synchronism or losing samples implies misinterpreting the signal at the processing stage. This means that the acquired signal is mistaken.

Finally the last requirement is that the design must be clock-synchronized with the transmitter wave generator. In radar applications the ADC must be synchronized with the transmitter system in order to avoid mismatches at the sampling periods. This requirement can be fulfilled substituting the clock crystal generator of the link2 board (12MHz) by an external signal of the same frequency shared with transmitter.

In sum, these are all the requirements for design:

- Must be able to fit in a radar receiver circuit.
- Low noise digitizer, similar performance to commercial designs.
- High speed data acquisition system.
- Continuity at the acquisition time line.
- Allow external synchronization with transmitter.

4. Hardware

4.1. LPC link2 development board

In order to test the performance of LPC4370 microcontroller, NXP has introduced into the market a development board named LPC link2 based on this microcontroller. This board allows to develop, debug and test release code in order to explore capabilities, without necessity of designing specific hardware.

Link2 board is based on 100 pin BGA package of LPC4370, and implements all the necessary extra hardware for extend microcontroller connectivity to external devices, to power the microcontroller, boot and extending circuitry to a prototype boards.

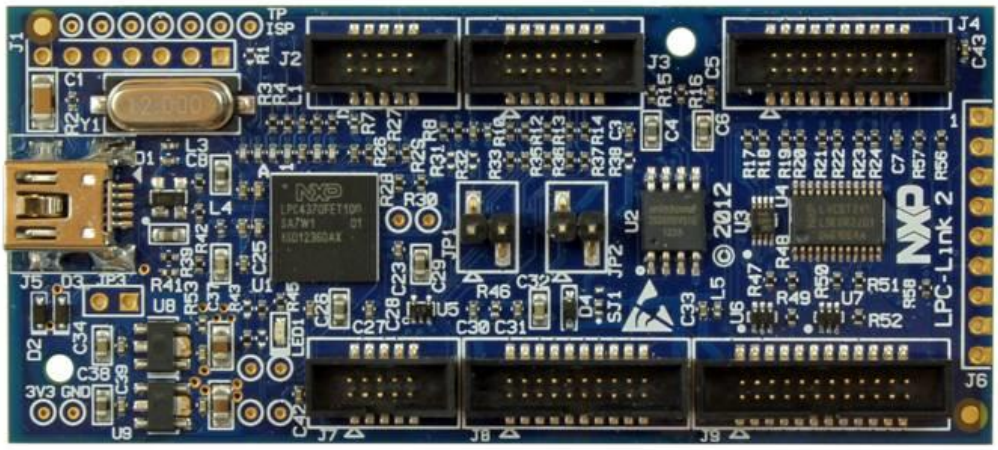


Fig. 4.1: LPC link2 development board. *[17]*

This board allows connecting additional boards by using the male 1.27 mm pitch connectors and the connection by USB port using the micro-b connector provided by the board. This board also includes a 1MB flash memory, where the microcontroller program is stored.

The USB connection provides power supply to the board, but it can also be provided externally if needed. By this connection, the board may act as a Host or as a Device, so its function can change dynamically due to the on-the-go characteristics. It can also be used for programming a boot image provided by NXP. This image is meant to reset the board to manufacturer defaults. The board also incorporates some test-points to map-out internal microcontroller signals, such as general clock or peripheral clocks. Link2

provides a 12MHz clock signal generated by a crystal oscillator which can be replaced by an external clock signal if needed.

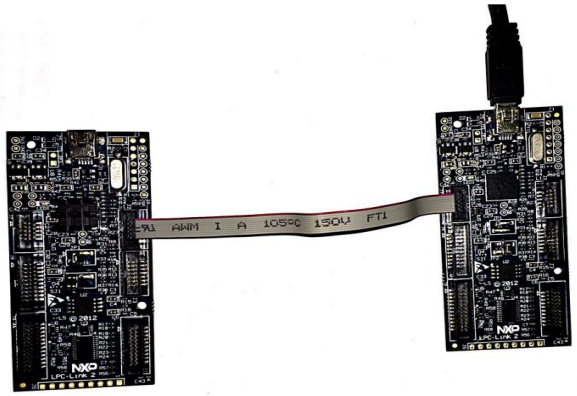


Fig. 4.2: Debugger board configuration

These boards include debugger functionality, what means that the same board can take the role of a debugger for another Link2 board. The Link2 has a JTAG module which can be configured as a master (debugger) or as a slave (target). Then, in order to develop, test and program a Link2, two Link2 boards can be used instead of expensive JTAG devices. The debugging connection is shown at *Fig. 4.2*. The right board is acting as a debugger and the left one as a target, so the debugger must be connected to the computer by USB in order to interact with the IDE for instruction flow control.

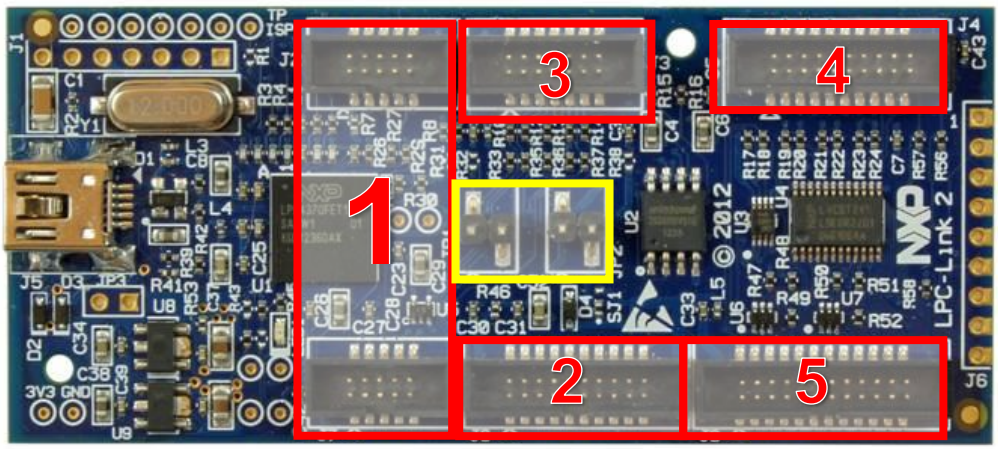


Fig. 4.3: Link2 board connector enumeration.

The connectors numbered as 1 in *Fig. 4.3* are, as shown before, the JTAG connectors for Link2 to Link2 program debug; connector 2 is a universal JTAG connector which can be used with any commercial JTAG debugger. Connector 3 is mapped to a

serial GPIO peripheral pinout, is meant to expand the board to serial digital input/output circuits. Connector 4 has analog ADC inputs, and several GPIO port pins On the other hand, connector 5 is only mapped to GPIO pins for digital data input or output. Two header connectors are marked in yellow that can be jumpered in order to activate special functionalities, such as providing power supply to the target board or selecting if booting from the flash memory in the board. For further information about this board, like schematics or component descriptions, see reference [17].

It is important to know that in some of the tests performed with this board, it has been reported that the flash memory becomes unreachable after being programmed with the application code. This problem had not been solved before, so the way to fix it was given to the manufacturers and their clients [18]. If the flash memory becomes bricked and IDE returns the error code “EF(49): flash driver operation error”, this procedure must be applied to unblock it.

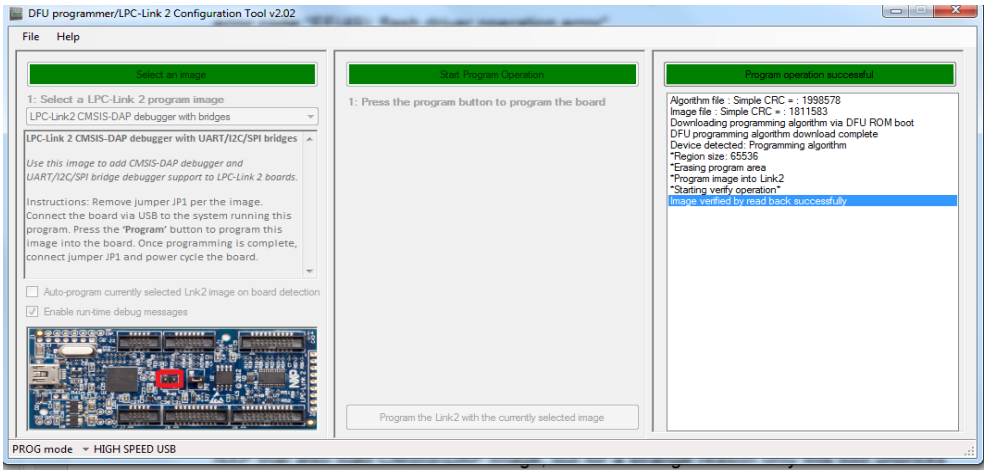


Fig. 4.4: CMSIS-DAP image loading with LPC-link2 Configuration tools.

The solution is to reload the booting image “CMSIS-DAP” to the board using an NXP software named “LPC-Link2 configuration tools” (*Fig. 4.4*). There are other programs provided by NXP that can also load CMSIS-DAP images, but for a strange reason only this tool unbricks the flash memory.

LPC4370 is a microcontroller with an ARM cortex based tri-core processor, which includes high performance peripherals as an 80MSps ADC, external memory parallel protocol controller, quad SPI interface and USB peripheral among other characteristics. This microcontroller can operate with a maximum clock frequency of 204MHz, with 3-stage pipelining. More information about this component can be found in the datasheet [19] and the user manual [20].

4.2. Connector adapter board

In order to allow the developer to acquire analog data using the LPC link2 board in an easy way, an extension board is needed as LPC link2 is not prepared for prototype wiring. The first developed board is a simple connector adapter board (*Fig. 4.5*).This board allows experimenting at first software development stages.

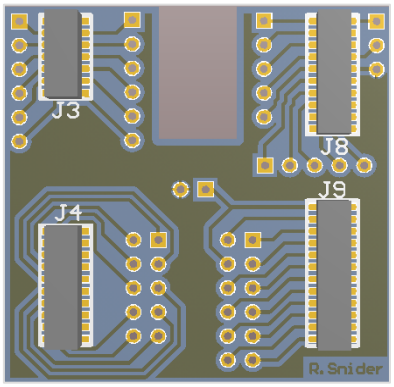


Fig. 4.5: Connector adapter board (Bottom layer).

This board is comprises four 50mil (1.27 millimeters) pitch connectors (J3, J4, J8 and J9), for the connection with link2; and eight 100mil (2.54 millimeters) pitch header connectors. 2.54 pitch connectors are the most common pitch, and allow the developer to easily plug wires for connecting a protoboard or testing signals with external devices.

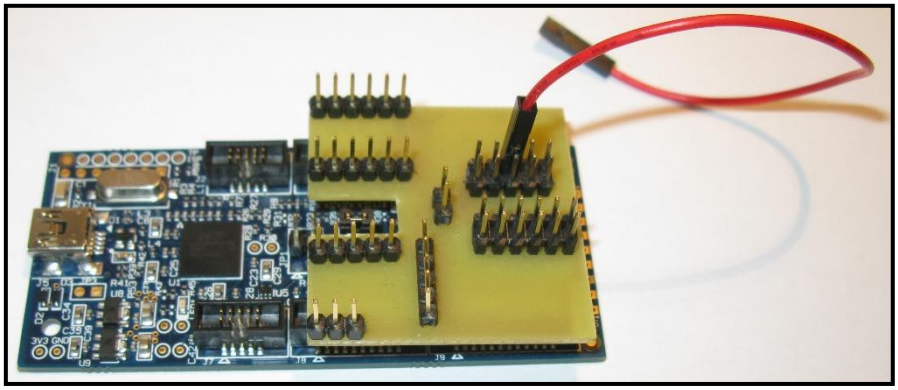


Fig. 4.6: LPC link2 with extension board.

As shown in *Fig. 4.6*, the board is designed to be connected to a link2 development board with the bottom layer's connectors shown at *Fig. 4.5*, which corresponds to 50mil pitch. The wires can be plugged at the top layer's connectors for testing purposes. The ADC's input is mapped to one header connector pin, but as this board design is only for development, the input can be heavily affected by electric noise. (*App. A1.*)

4.3. ADC extension board

Once the first development stage is finished and further improvements on ADC acquiring operation are needed, a new board must be developed. The target of this board is to reduce the amount of electric noise at the high speed ADC (HSADC) input pin. The design will be explained step by step throughout this section, specifying calculations and argumentations.

In order to perform the optimal sampling operation according to the microcontroller characteristics, voltages at the ADC's input pin must be comprised between 0.1V to 0.9V, so the dynamic range is 0.8V. The second limitation is the bandwidth of this signal. LPC4370's HSADC can operate up to 80MSps, which means that the maximum sampling frequency is 80MHz. Applying Nyquist theorem (*Eq. 4.1*), with F_s as sampling frequency, it is found that the maximum frequency that can be sampled without aliasing is 40MHz.

$$F_s > 2 \cdot F_{max} \tag{4.1}$$

To avoid signal aliasing due to higher frequency harmonics and reduce the impact of noise above Nyquist's limit, it is highly recommended to include a low pass filter with a cutoff frequency of F_{max}.

The first part of the system is the conditioning circuit, composed by a LNA and the demodulator (*Sec. 3*). This circuit is implemented at the radar receiver, so this part will not be considered in this design. An unbalanced signal of 0.4V of amplitude and 0.5V offset will be considered at the board input port. This signal is a low pass signal with a maximum bandwidth below 40MHz due to Nyquist theorem and previous considerations, so it must be filtered by an antialiasing filter with a cutoff frequency of 40MHz.

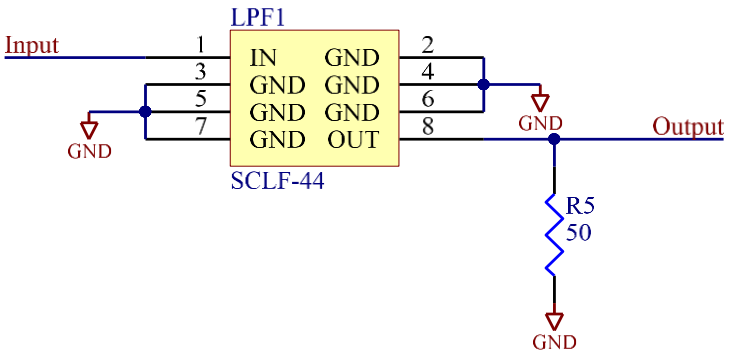


Fig. 4.7: Integrated filter design

Fig. 4.7 shows the filtering stage design. It is implemented using an integrated filter of mini-circuits SCLF series [21] with a cutoff frequency of 44MHz, which is more or less the value previous calculated. This integrated filter is designed to fit a characteristic impedance of 50Ω, so the input and output of the filter must be adapted to that value (output adapted with R5).

Next step of the design is to minimize the noise effect at the input due to the own microcontroller's ADC reference voltage variations. Inside the microcontroller, the voltage reference of the ADC is generated by a linear voltage regulator. Theoretically this reference voltage is constant and has no fluctuations, but as most of the microcontroller is digital circuitry, it can induce variations to this reference voltage (error in reference $\hat{e}_r$). $\hat{e}_r$ is a random variable with zero mean and variable variance due to microcontroller activity. So the reference voltage will be given by (*Eq. 4.2*).

$$V_{ADC_{ref}} = V_{ref} + \hat{e}_r \tag{4.2}$$

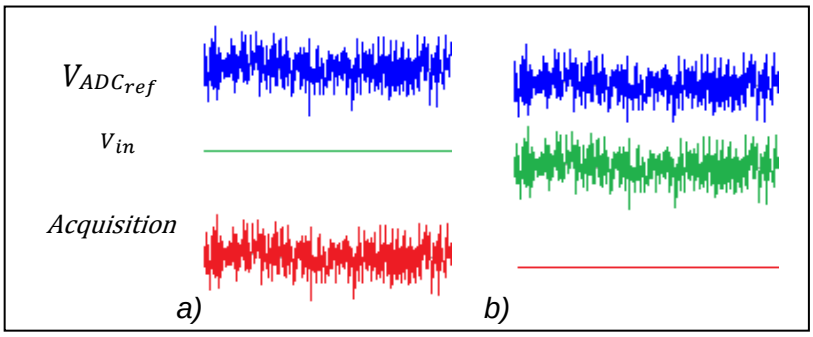


Fig. 4.8: Measure of DC voltage; a) bad acquisition, b) acquisition with error mixing.

In order to reduce the reference error effect on the samples, $\hat{e}_r$ must be mixed with the input signal (*Fig. 4.8.b*), otherwise the acquisition process will be noisy as shown in *Fig. 4.8.a.* LPC link2 has an output pin which allows using $V_{ADC_{ref}}$ in external designs for that purpose. This output pin returns the converter reference voltage but halved. The maximum error correction will be found in close values to this voltage. When the input voltage varies out of the center, correction effect is reduced. When not applying this method, the error is minimum at low input voltages, and maximum at high voltages. If this method is applied the error component is removed from the dynamic range center, but the halved error component appears at both edges. This is very useful when acquiring low power signals, as voltage values are close to the DR center.

To ensure that the impedance of the filter output is fixed to 50Ω, the next stage of the circuit must be isolated. The solution is to connect the filter output to an operational amplifier as follows. The high impedance of the amplifier input keeps constant the 50Ω resistance at the filter output.

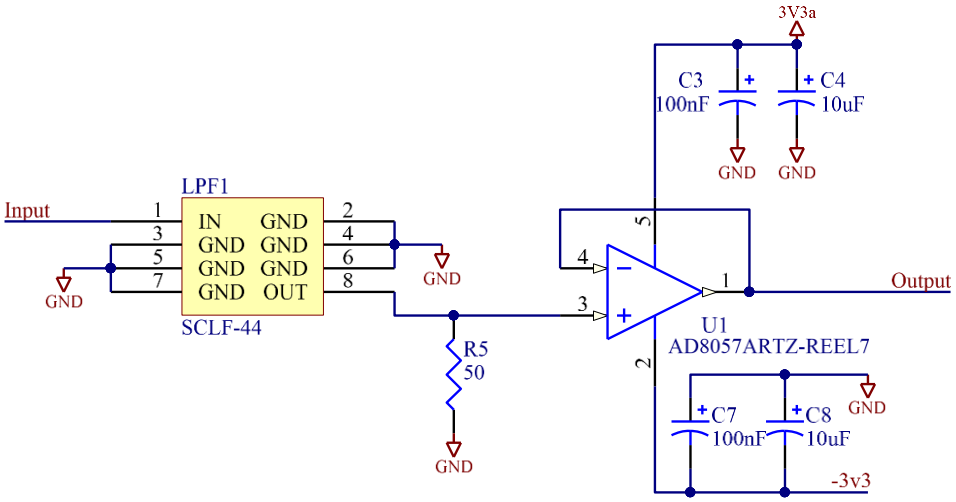


Fig. 4.9: Filter with the follower amplifier.

The last point of the design is to mix the error variable with the input signal coming from the filter, with some restrictions and compromises defined by the input port and the ADC. The signal at the output of the follower amplifier (now defined as V_{an}) is an unbalanced AC signal with a 0.5V offset voltage (*Eq. 4.3*), as previously defined. The second input of the circuit is the reference ($V_{ADC_{ref}}$), which includes the error component of the converter and its reference voltage of 0.5V (*Eq. 4.4*). At the output of this circuit (ADC input) an unbalanced signal is needed, which consists in a V_{sig}, 0.5V offset voltage and the error component (*Eq. 4.5*).

$$V_{an} = 0.5V + v_{sig} \tag{4.3}$$

$$V_{ADC_{ref}} = 0.5V + \hat{e} \tag{4.4}$$

$$V_o = 0.5V + v_{sig} + \hat{e} \tag{4.5}$$

The way to solve this problem is by using an operational amplifier in adder-subtractor configuration, as the direct (or proportional) adding of both inputs does not generate V_o and 0.5V must be subtracted from the sum of (*Eq. 4.3*) and (*Eq. 4.4*). The best solution is to subtract (*Eq. 4.3)* from (*Eq. 4.4*) and then add 0.5V. Doing this, the input and the ADC reference signals are separated from each other, and otherwise fast signals could

modify the reference and affect the conversion process. The main drawback is that v_{sig} will shift the phase by 180º, but it can be corrected by software at the post-processing stage.

As the offset signal has only DC component, a source power voltage can be used to generate it by voltage division in the adder-subtractor circuit, so next circuit is given by definition.

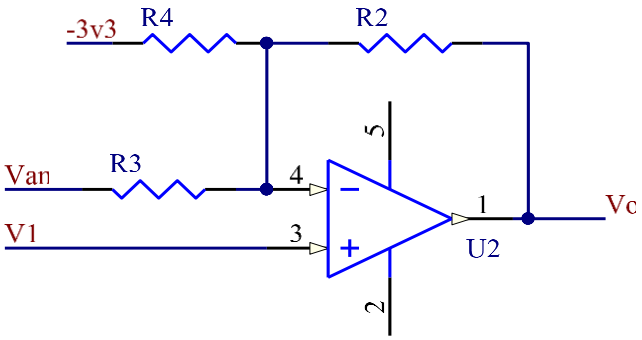


Fig. 4.10: Adder-subtractor amplifier configuration.

Looking at the *Fig. 4.10*, equation (*Eq. 4.6*) can be derived by applying Kirchhoff's law. Note that V_1 corresponds to the reference voltage $V_{ADC_{ref}}$, but it is expressed in this way in order to clarify what must be tuned after calculations.

$$\frac{V_1 - V_o}{R2} = \frac{V_{an} - V_1}{R3} + \frac{-3.3 - V_1}{R4} \tag{4.6}$$

When solving, this equation returns the result (*Eq. 4.7*).

$$V_o = V_1\left(1 + \frac{R_2R_3 + R_2R_4}{R_3R_4}\right) - V_{an}\left(\frac{R_2}{R_3}\right) + 3.3\left(\frac{R_2}{R_4}\right) \tag{4.7}$$

(*Eq. 4.5*) can be expressed in a different way (*Eq. 4.8*). Note that v_{sig} has been shifted the phase by 180º.

$$V_o = V_{ADC_{ref}} - V_{an} + 0.5V \tag{4.8}$$

Knowing that V_{offset} is 0.5V, the next values are found by combining (*Eq. 4.7*) and (*Eq. 4.8*).

$$V_{an} = V_{an}\left(\frac{R_2}{R_3}\right);\ \ 0.5 = 3.3\left(\frac{R_2}{R_4}\right) \tag{4.9}$$

$$V_{ADC_{ref}} = V_1(1 + 1.1515) \tag{4.10}$$

As V_1 is smaller than $V_{ADC_{ref}}$ as shown in (*Eq. 4.10*), it must be tuned in order to ensure that the error has no gain on the total contribution. This difference can be fixed using a voltage divider in $V_{ADC_{ref}}$.

The relationships between resistances are obtained from (*Eq. 4.9*), their values can be selected by creating a table with real resistances values close to the calculated ones. There is a tradeoff between low resistance values (high power consumption) and high values, which induce a slow response due to parasitic capacitances.

Some commercial resistors values are shown in the next table for a proper analysis.

R2 and R3	R4	Ideal R4	R2/R4 error (%)
560	3900	3696	0,1201
1000	6800	6600	0,0675
1200	8200	7920	0,0784
1500	10000	9900	0,0230
4700	33000	31020	0,1377
10000	68000	66000	0,0675

Table 4.1: R2, R3 and R4 relations and errors.

In *Table 4.1*, the relationship between R2, R3 and R4 is analyzed. Yellow indicates that the marked number is non-desirable due to the low resistance value. In red the value that is also non-desirable as the resistance value is high. In green the selected resistor values for this design, because it is the one providing the lowest percentage of error.

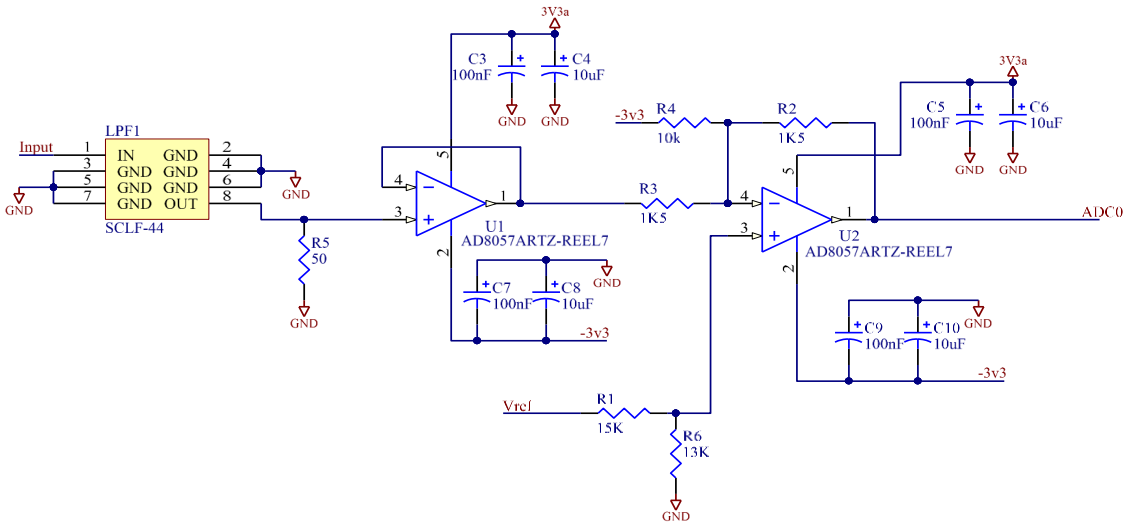


Fig. 4.11: Final design with selected values.

This board has also digital inputs and outputs in order to speed up the development work. The inputs allow the developer to start or stop sequences such as PC information dump, the ADC capture operation or other useful functions. The outputs also allow knowing which stage of the program is active or extracting digital signals helping the user to measure microcontroller timings (such as clocks, ADC speeds,…).

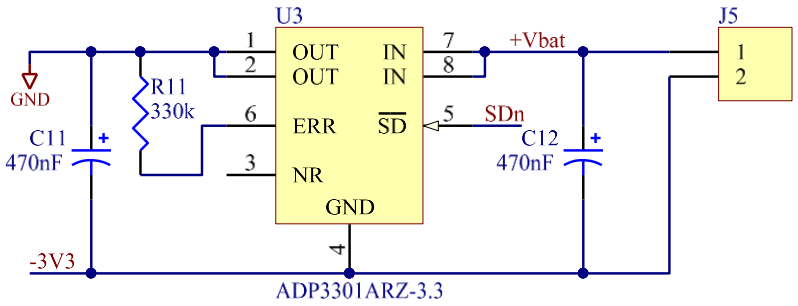
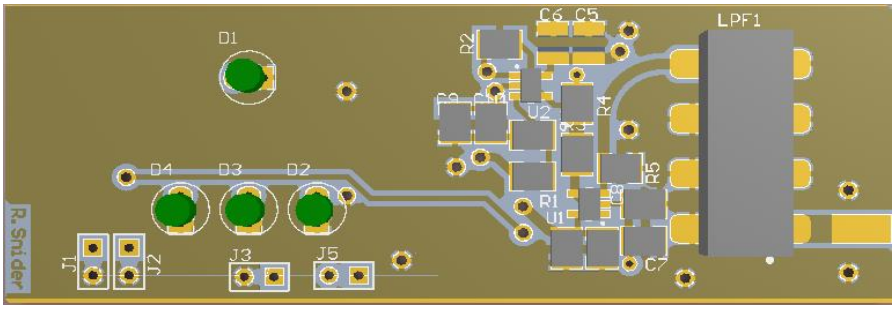


Fig. 4.12: Negative voltage generation circuit.

To obtain a negative voltage of -3.3V a voltage regulator (ADP3301ARZ-3.3 LDO in this case) and an independent power supply must be used (*Fig. 4.12*). The positive pole (J5's pin 1) must be connected to the input of the voltage regulator, and the negative pole (J5's pin 2) to ground. The output of the regulator will drive a voltage difference of 3.3 with respect to ground. So the output will be connected to the board's ground plane and the regulator ground pin will drive the -3.3V voltage to the corresponding components.

a)



b)

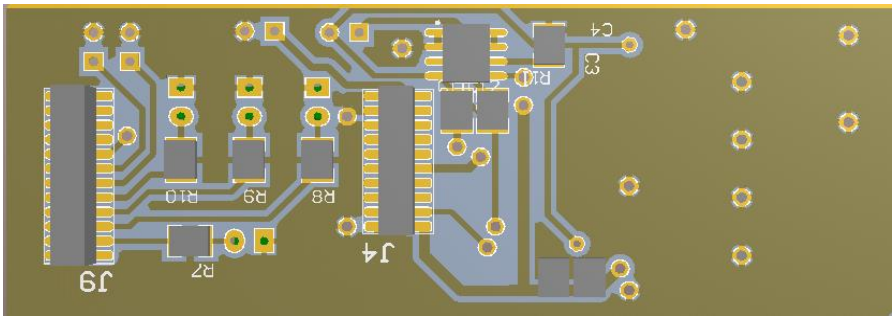


Fig. 4.13: PCB design of the board. a) Top layer, b) Bottom layer

Fig. 4.13 (a) and (*b)* shows the PCB design in Altium. Analog and digital parts have been separated. The analog part is placed on the right side of the top layer (*Fig. 4.13.a*). On the other hand, the digital part is situated on the left side of the bottom layer (*Fig. 4.13.b*). In addition, the use of separated power supplies for digital and analog minimizes the noise at the input of the ADC thanks to the use of the digital part.

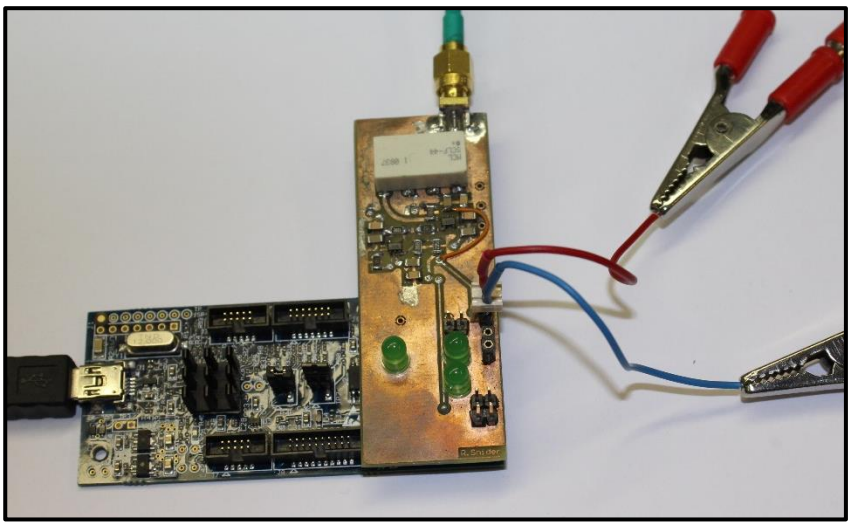


Fig. 4.14: ADC extension board typical connection.

Fig. 4.14 shows the developed PCB prototype. The analog input is injected to the input port by a coaxial cable with a SMA connector, the Link2 is connected to the computer by USB cable and finally an external supply is needed at connector J5. As the main power supply is provided by the USB connection, the negative one must be externally provided as previously commented. The voltage difference at J5 must be compressed between 3.4V and 16V in order to obtain a stable value of -3.3V in the circuit. (*App A.2.*)

5. Firmware developments

This section presents all the firmware developed for testing purposes and final version programs. This firmware must be programmed directly on the flash memory of the board (released version). Otherwise high speed instructions for data acquisition will not work and the ADC operation will become slow. The debug mode implies that the instructions are controlled by the JTAG master, and they are not real time. The debug code has also instruction redundancy and different structure (lower compilation level) to allow the developer to execute instructions step by step. The use of a lower compilation level also increases the latency of code execution.

5.1. High Speed Analog-to-Digital Converter peripheral

Several experiments were performed in order to test the correct operation of the ADC. The code was tested on the LPC link 2 development board, in addition to the Connector adapter extension board described in section (*Sec. 4.2*) and to the connection scheme shown in *Fig. 5.1*.

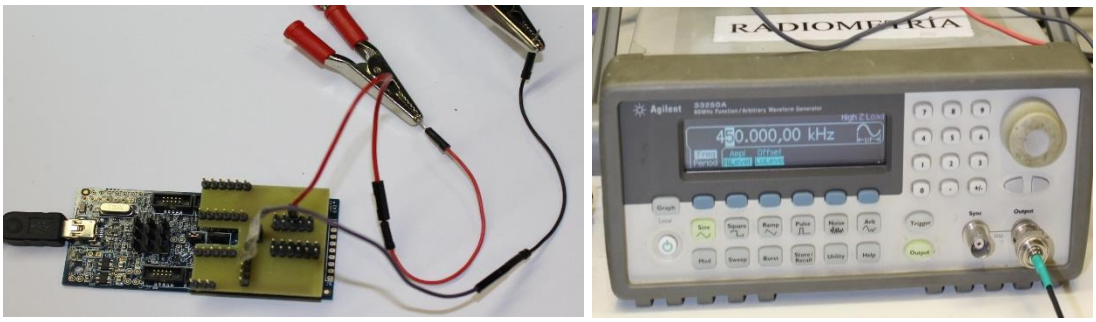


Fig. 5.1: Connection scheme of the board. Link2 connected to PC by USB, and the board to a waveform generator (right image) by BNC to banana connector.

The ADC module is initialized with a frequency of 80MHz. This clock is obtained by deriving the clock signal from USB0's PLL to two frequency dividers. The output of USB0 PLL is a signal of 480MHz, a first divider (x2) halves the input frequency (to 240MHz) and the last divider (x3) divides it by three, so the output frequency of this last divider is 80MHz.

Interrupt vector of the microcontroller has reserved a flag for ADC. This means that the microcontroller can be configured to execute specific code (Interrupt Service Routine,

”ISR”) when the activation of the ADC operation is finished. For testing purposes, some instructions are allocated in the ISR code, which saves into the RAM the acquired sample in every operation.

Each operation is started by a microcontroller register bit called trigger. The operations are configured by means of a vector named as in the descriptor table. These descriptor tables are a microcontroller set of registers in charge of the sampling process, in other words, samples that will be acquired in the operation can be configured in these registers indicating the delay between samples or which ADC peripheral channel will acquire each sample. Descriptors are useful when a multiple channel acquisition is needed, as descriptors can be used to switch to active channel. So for example, 2 successive samples can be taken at the 1st channel, and then 3 successive samples at the 2nd channel using 5 descriptors in total.

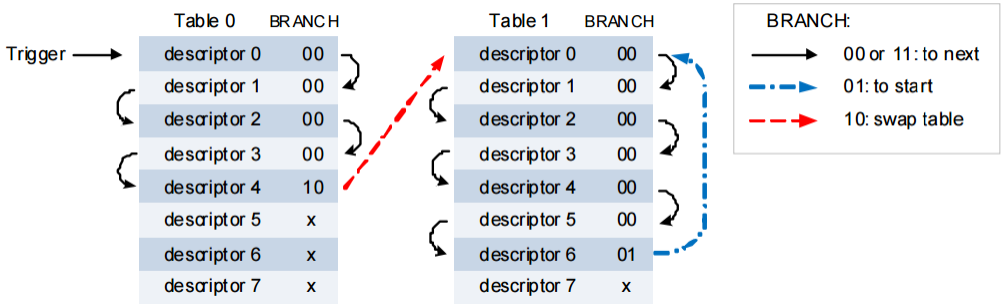


Fig. 5.2: Descriptor tables diagram.

Fig. 5.2 illustrates the diagram of the descriptor vector. Each vector position corresponds to one register. When the operation is triggered, the ADC reads the first descriptor and, depending on the configured channel at the descriptor, samples one or other microcontroller pin. When the sample is acquired, the obtained value is saved into a FIFO memory. LPC4370’s FIFO has a capacity of 16 words of 32 bits, which is up to 32 samples in compact format (16+16 bits). Afterwards, as shown in *Fig. 5.2*, the first descriptor register can be configured to execute the next descriptor, making a table branch or restarting the descriptor table. Each descriptor also allows adding a delay between the sample acquisition and the descriptor command execution (next descriptor, table branch or reset), what makes fine tune the ADC sample rate possible without modifying the clock.

Several tests were made in order to test and compare different performances with the use of descriptor tables and modifying interrupts conditions.

Capture with 1 descriptor and operation end interrupt

The first test consisted in using one descriptor to acquire only one sample per operation. The objective of this test is to verify the correct sampling process using the peripheral one time per interrupt. The sample acquired and stored into a variable in the RAM memory and then the sampling process is reactivated. The maximum achievable capture rate using this method is around 100KSps.

The ADC is triggered inside the ISR function, so the time between samples may be not constant and depends on the processor activity. Then the acquired samples are not synchronous as the time instant of each sample cannot be deduced. (*App. B1.1.*)

Capture with 16 descriptors and operation end interrupt

The second test performed is meant to use more than one sample, using both available descriptor tables. Then 16 samples are acquired per operation, one per descriptor register. When the last descriptor is reached, ISR is automatically called. In the ISR code, the acquired samples are saved into an array located in the RAM. Using this test the microcontroller did not respond as expected. As the number of samples per operation is increased with respect to the previous test, theoretically the sample rate should increase in the same way. In practice the maximum sample rate is reduced regarding the previous test. , When 16 descriptors are used simultaneously at the same ADC channel, the maximum achievable speed is reduced to 25KSps.

Some delay is detected when the last descriptor is reached, as the interrupt notification is delayed a long time, approximately 40µs with respect to 12.5ns of the sample period (3,200 samples approximately). In the same way as the sample rate is extremely reduced, samples are captured in a bursty way.16 samples equally distributed on time at 80MSps and then a very large waiting time between other 16 samples. As bursty samples are not useful in radar applications (the synchronism at the sampling process must be maintained) and the sample rate is very low, this method is not feasible. (*App. B1.2.*)

Capture with 1 descriptor and FIFO full interrupt

The next test was devoted to continuously acquire samples using one descriptor in a loop (pointing to itself). The peripheral was configured to interrupt the microcontroller

main process only when the FIFO memory was full, so then 32 samples were captured between interrupts. During the ISR, the ADC continues sampling and storing the samples into the FIFO. The maximum achievable sample rate with this method is 1.2MSps. Now this value is limited by the time to store the acquired samples to an array in the RAM memory.

As the ADC peripheral is auto-triggering, samples are equally distributed in time and synchronous, so this method can be used for radar signal acquisition. (*App B1.3., B1.4.*)

5.2. ADC with Direct Memory Access peripheral

When more sample rate is needed, the response time between the interrupt service call and end of the FIFO emptying process, becomes a major limitation. Every interruption consumes hundreds of microprocessor instructions, used to store 16 words of 32bit, which corresponds to the 32 acquired samples.

The solution is to use Direct Memory Access (DMA). The DMA peripheral is in charge of transferring information, at very high speed, inside the microcontroller from a peripheral (or memory) to other peripherals (or memory addresses). The DMA allows reducing microprocessor activity due these operations, as it can move data at the same time that the processor is executing instructions.

LPC4370's DMA has up to eight channels available. Each channel can drive one up to 32-bit unidirectional transfer at a time, so the information movement is totally transparent to the microprocessor core. DMA can be configured to read from a peripheral (for example read the ADC's FIFO memory) and to write specific RAM memory address. The transport operation can be automatically triggered by the selected peripheral interrupt call. It can transfer a single word or be configured for a burst transfer, which allows transferring more than one word per operation. In the case of the ADC, DMA can transfer the full FIFO content into a specific memory address.

For radar data acquisition, a massive data transference is needed. DMA can manage up to 4095 transfers with no need to be reconfigured. When data dump is needed with more than 4095 32-bit transfers, DMA must be configured with configuration lists.

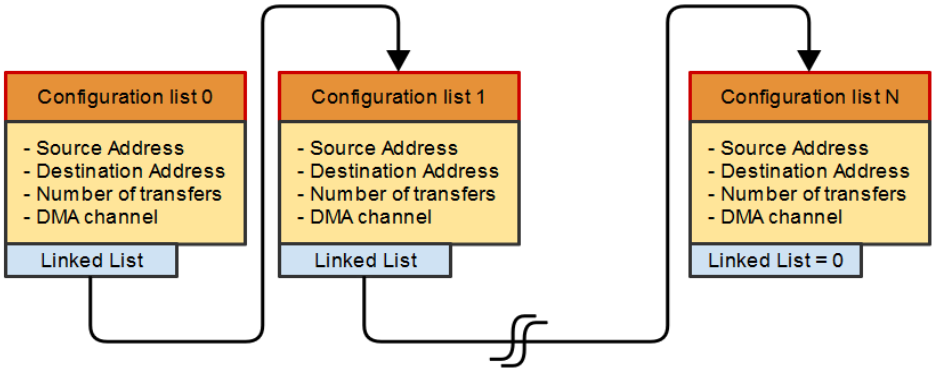


Fig. 5.3: Configuration array, linked list.

The configuration list (*Fig. 5.3*) is an array of N lists, in which every cell contains a set of attributes like the source and destination addresses of the transfer, the number of transfers that will be managed per operation or the channel that will be used. This corresponds to the required information to start every DMA operation.

When an operation is finished, an additional parameter (linked list) indicates the peripheral the new configuration list address for the next operation. Then, the DMA starts a new operation with these values. When the last scheduled operation is finished, a zero value linked to the list attribute informs the DMA that the last list is reached and the DMA closes the channel. The linked list attribute can also be auto-referenced in order to acquire samples in a continuous way.

DMA has an interrupt notification flag on microcontroller's interrupt vector. This notification can be automatically triggered at the end of each operation, or, if linked list are configured, at the end of the N-th operation.

Using this method, the maximum sample rate is the whole 80MSps, although in practice continuous acquisition at 80MSps is not possible. The sample rate is limited by the storage operation due to the great amount of generated data rate (1280Mbit/s which is 160MB/s). The RAM memory has a limited storage capacity (160KB), so it only supports a bursty signal acquisition. For a continuous acquisition, samples must be stored in an external device. So the maximum achievable rate will be limited by the maximum data rate of the communication protocol used for data storage.

In the case of quad SPI (QSPI) flash memories, the maximum speed of the QSPI peripheral is about 52MB/s. Taking into account that a sample is packaged on a 2 Bytes word, the maximum sample rate is 26MSps.

USB 2.0 peripheral supports communications up to 60MB/s, which in sample rate terms are 30MSps. This is the maximum theoretical value due to the flow control sequences and the operation modes, which reduces the useful bandwidth. This speed is only achievable in isochronous mode, but as this mode does not ensure data integrity, so it is not valid for radar applications. The bulk mode can work up to 20MB/s and lower rates if using the interrupt mode (8MB/s).

The external Memory Controller peripheral (EMC), allows using parallel data flow to an external SRAM, SDRAM or Flash memory device at about 83MB/s, which means 41MSps. This solution only allows mapping a low amount of addresses, which in storage terms is a low amount of Bytes (more or less 256MB per SDRAM chip). (*App. B.2.*)

5.3. Data acquisition, dump to PC using USB

In ground base radars the storage device can be a more complex system, such as a PC. Several tests programs, based on direct communication between the microcontroller and a computer using USB has been developed. This strategy allows avoiding the use of any type of memory installed on the acquisition board, as the acquired samples are directly dumped from the ADC to USB. The limitations of this method are found in the speed of the USB protocol but also regarding the host store capabilities.

In this section, some tests and methods will be discussed in order to achieve the maximum sampling rate, analyzing where the limitations are and how to avoid or reduce their impact. All the tests in this section were performed with a Computer Intel i5-4460 processor based with 16GB of RAM memory and a hard disk drive on SATA 3 (600MB/s) with 3TB of storage capacity.

USB as RS-232 emulator:

The first test was to use the USB bus as a RS-232 serial COM port emulator. NXP provides different libraries and drivers which allow the board to send and receive information coded in ASCII. To perform this test a PC software is needed, which is detailed in section (*Sec. 6*). The test consisted in linking the ADC and the USB to the DMA. The signal is sampled and then the DMA moves the FIFO content to the output buffer of the USB. Samples are packaged in memory in 16 bits (2 Bytes), so to represent each sample in ASCII 4 bytes or characters are needed in order to avoid protected

characters (like null or break line characters). Break line characters are used to inform the host that the transmission is over. When the device sends a null character this is not received by the host as it interprets that there is no information on it. So at the end every sample is by 4 bytes, each one representing 4 bits of the sample, “0x30” for “0b0000” representation, and “0x45” for 0b1111.

This method can only drive up to 128Kbps, and the limitations are imposed by the NXP’s driver itself. This maximum speed is very low in comparison with the maximum capabilities of the communication protocol. Taking into account that each sample is represented by 4 bytes (32 bits), the maximum sample rate that the bus can handle is 4KSps. As the results are extremely low, RS-232 emulation is a non-feasible method for direct data dumping.

USB bulk transmission using LibUSB

LibUSB is a library supported by windows and Linux that includes a generic driver that allows easily using endpoints [22] for communicating the device. These endpoints are configured in the USB device descriptors section. Using the drivers provided by this library, the communication speed is unlocked to the maximum speed depending on the selected operation mode. For bulk transfers the maximum data rate is 20MB/s.

The board configures two endpoints for PC communication. The first endpoint is configured for input data (endpoint address 0x1) and the second one for output data (address 0x82), both in bulk data mode. In the same way as in the previous method explained in this section, this program uses the DMA, which performs the internal ADC samples transport operation to the output USB buffers.

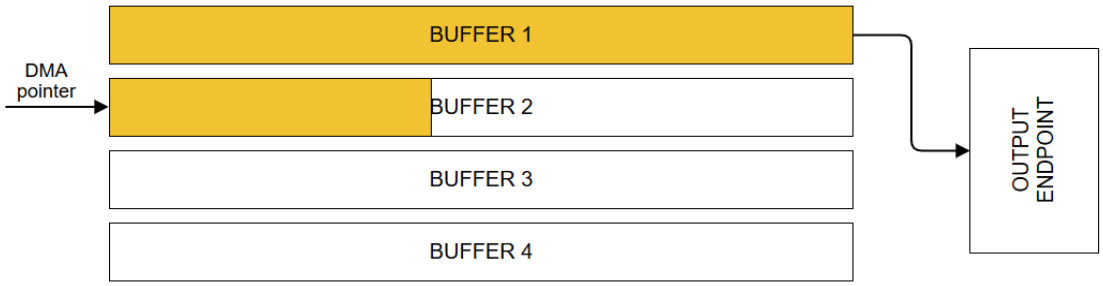


Fig. 5.4: Buffering operation.

There are 4 output buffers configured. When one buffer is filled by the DMA, the microcontroller dumps this full buffer to the output endpoint. *Fig. 5.4* illustrates the

buffering operation, in which when a buffer is completely filled, the DMA pointer switches to the next buffer and begins to fill it. At the same time, a service interrupt call occurs, and inside this routine, the microprocessor moves the buffer content to the output endpoint.

But the operation of the transmission is more complex than that, the output endpoint in bulk mode only accepts 512Bytes, so if the output buffer is greater than that dimension, it must be divided into more than one packet. When a packet is loaded at the output endpoint, the computer must accept the transmission before it can be transmitted.

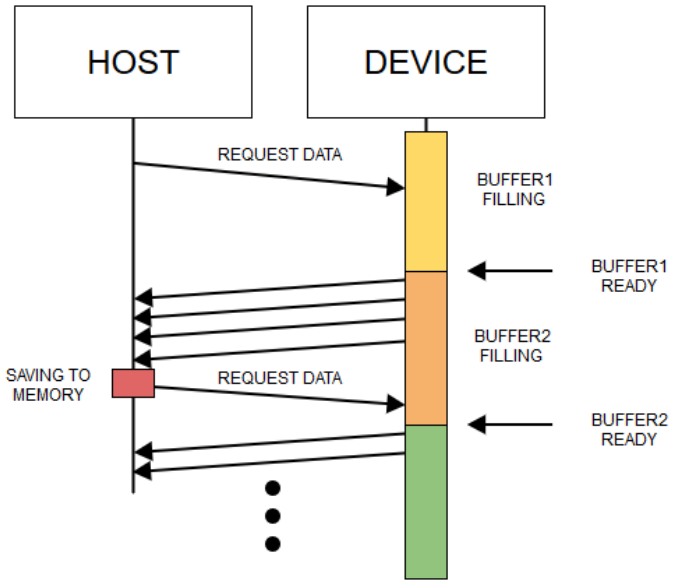


Fig. 5.5: Packet transmission protocol.

Fig. 5.5 illustrates the time flowchart of the transfer operation protocol between the host and the device. Data at the output endpoint is transferred to the host, only if this last has previously requested that data. When a request from the host to the device is performed, the device responds with information packets only when the buffer is filled. This figure also illustrate an example of a 2048Bytes buffer. When a request is performed and the buffer has already been filled, the device sends 4 packets of 512Bytes each one. When the host receives the data, it sends another request to handle the next buffer dump. (*App. B.3.*)

On the host side, the function used for performing the request is a locking bulk transfer function provided by LibUSB. This function waits until the number of bytes indicated in the function parameters are received, and then returns the received data.

Then the function used for data saving “fwrite()” [23], stores the content of the transference (in the example before, 2048 Bytes), to a previously created file. This function also locks the process thread until all the data is saved into the disk.

It can be deduced that if the host data saving process takes too much time, it may block the next device transmission. If the device pending transaction is not accepted and the DMA is pointing to the last available buffer (referring to the previous example, the pending buffer is buffer number 1 and the DMA is writing buffer 4), so if buffer 4 is filled before the data transaction corresponding to buffer 1 is performed, the DMA will overwrite buffer 1, 2 and 3. Then, when the transmission is performed, all the data from the 3 buffers will be lost. Therefore, in this case it is necessary to reduce the USB operations and data saving processing times in order to ensure that the DMA does not overwrite the buffers.

Theoretically this method allows sampling up to 10MSps, but in practice these values cannot be achieved. If the microcontroller uses locking USB transmission functions (also known as synchronous functions), working in bulk mode at this sample rate implies that endpoints will be always full. So the transmission functions will lock the microcontroller program flow and it will not be able to read the input endpoints and the end microcontroller will get frozen. If the computer delays one operation more than expected, the full set of samples of the buffers will be lost.

In the first test performed using this method, the maximum sample rate achieved was 250KSps. This sample rate was increased to 2MSps by modifying some timing parameters in following tests.

During the tests carried out to achieve the maximum sample rate, it has been found that the function “fwrite()”, working on a hard drive, is more efficient for a greater number of bytes written. The hard drive writing operation, for example, is faster when writing 2048 Bytes than when writing four times 512 Bytes due to physical restrictions of this type of storage device. For that reason, if the program developed on the computer side increases the number of packets before writing to a file, the efficiency between the number of written bytes and the required time increase. To increase the numbers of bytes to write to a file, the microcontroller USB buffers also needs to increase its size. Next graphic shows an analysis of buffer size versus link errors.

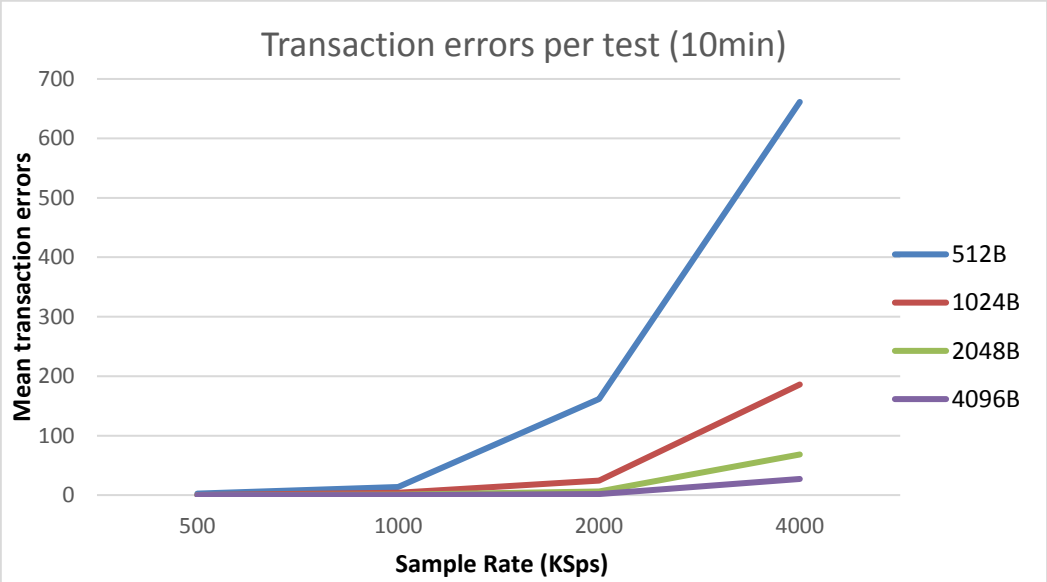


Fig. 5.6: USB tests transaction errors results.

The transaction error means that a DMA buffer overwrite due to computer delays. These errors randomly appear when increasing the sampling rate. When a transaction error occurs, it means that the DMA has overwritten all the 4 buffers, and the corresponding number of samples of these 4 buffers is lost per transaction error. As is shown in *Fig. 5.6*, transaction errors are proportional to the sample rate of acquisition, and inversely related to buffer size.

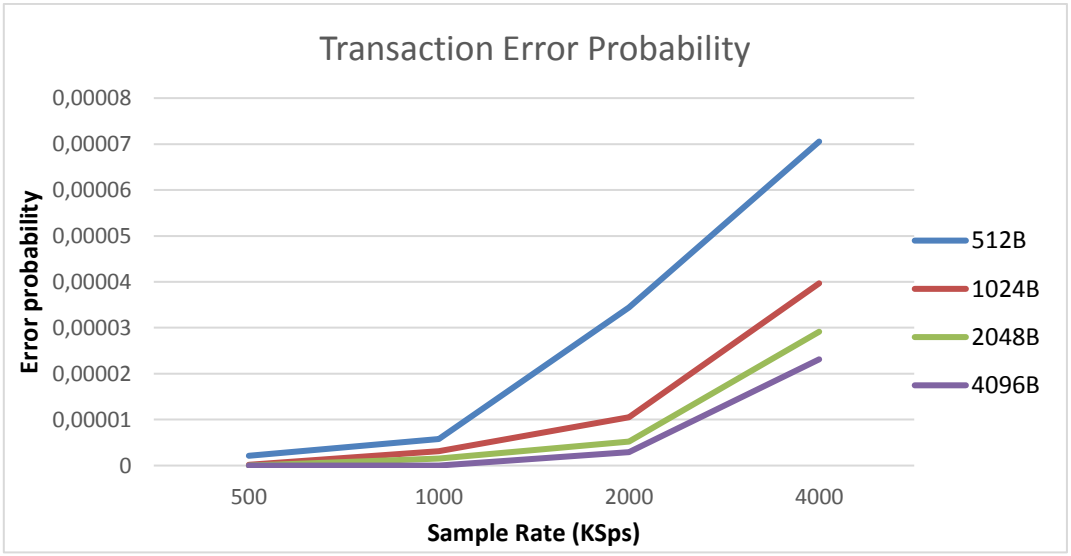


Fig. 5.7: USB transaction error probability.

Fig. 5.7 shows the relationship in terms of error probability. The equation for error probability calculation is shown in (*Eq. 5.1*). The transactions number is related to buffer

size and sample rate, for total observation time of 10 minutes. Looking at the figure it can be concluded that beyond a certain threshold value for a fixed sample rate, errors disappears.

$$P_{err} = \frac{N_{errors}}{Transactions} = \frac{N_{errors} \cdot N_{Bytes}}{t_{obs} \cdot f_{samp}} \quad (5.1)$$

The reason for reducing errors when increasing the buffer size is that more samples are saved into one “fwrite()” operation on the computer side. Although the hard disk storage time increases along with the number of stored bytes, the access time does not. The hard disk access time is a set value (time to search the address to store and physical limitations “T_{phy}”), and a variable value that depends on the computer activity “(T_{act}"), then the time efficiency taken to store a large number of bytes in relation with a lower number of Bytes is greater. For low quantity of data, access time is larger than writing time. The hard disk total writing time is given by (*Eq. 5.2*).

$$T_{tot} = T_{access} + T_w = (T_{phy} + T_{act}) + \frac{N_{Bytes}}{D_{rateHD}} \quad (5.2)$$

Next table shows numerically the maximum sample rate that can be achieved without errors in the tests performed in the laboratory with the computer described before.

Buffer Size	Maximum Sample Rate
512 Bytes	250 KSps
1024 Bytes	500 KSps
2048 Bytes	500 KSps
4096 Bytes	1 MSps
8192 Bytes	2 MSps

Table 5.1: Maximum sample rate due to buffer size.

Those results are taken from continuous sampling, but the operation can be bursty to obtain a better performance. In some cases there is no need of a continuous sampling process, for example in SAR applications, so samples can be acquired in a bursty way. Due to that, when for example 4096 samples are acquired on a first burst at a given sample rate, the sampling operation is halt and buffer content is sent to the computer. After a certain time the sampling operation is restarted and the sampling rate can be considerably increased.

5.4. Data acquisition, storing on external memory device

When a portable acquisition device is required, the system must be able to store into the board all the acquired digital samples. This can be achieved by using external memory devices, for example, serial flash memories. In this section, different peripherals, which allow communicating the microcontroller with external memory devices, will be analyzed and discussed.

LPC4370 has an available quad SPI interface for flash memories (SPIFI). This interface, as previously commented, allows storing up to 52MB/s. It also has available a normal SPI interface that allows a communication speed of 3.18MB/s, but it is extremely low due to application requirements and, due to that, SPI will be not analyzed. Finally, as commented in section (*Sec. 5.2*), this microcontroller has an EMC module that allows a communication rate of 83MB/s. LPC4370 uses other communication protocols but at lower speeds. For that reason, these will not be explained here.

5.4.1. Communication with quad SPI interface

Quad SPI interface, also known as SPI Flash Interface (SPIFI) [19], is designed to support fast communications with an external flash memory. In radar applications, this is the ideal peripheral. Fast communications allow storing up to 26MSps, which covers application requirements. But in practice, all the tests performed with this interface were not able to write data on the flash memory.

This peripheral is a special interface because the microcontroller uses it all the time to execute program instructions, so the program code is stored into the external flash memory provided by the board and all the instructions are fetched from there. This method is called “execute in place mode (XIP)”. Therefore, if the developed program needs to execute any instruction related to this peripheral, these instructions must be moved previously to the RAM memory. The SPIFI peripheral must be released when microcontroller writes data on it. Otherwise the instruction fetch process will block peripheral configuration, memory read or memory write operations. If the code is moved to the internal RAM and executed from there, then instructions that use SPIFI do not collide with the instruction fetch process.

- **Code relocation modes**

The code relocation operation is performed at the microcontroller booting stage, so the compiler and the linker modify the boot routine in order to inform which parts will be copied into the RAM when startup. There are three modes of code relocation [24] from flash to RAM. The first mode is meant to move specific functions by including the next header in the desired function.

```
#include <cr_section_macros.h>
__RAMFUNC(RAM)  void fooRAM(void) {...}
```

"fooRAM" is the name of the function that will be relocated.

To use other relocating methods, the linker needs to know which objects will be moved into the RAM memory, so the linker scripts must be created and added to IDE. NXP's IDE (LPCXpresso) provides a linker script helping tool, the developer can add linker functions only by creating a folder in the project folder with the name "linkscripts" (*Fig. 5.8*) and putting inside all the linker functions files. IDE will modify the main linker script in order to add the indicated functionalities or operations.

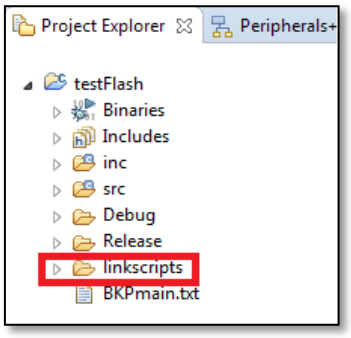


Fig. 5.8: Linker folder.

The second mode is devoted to relocate particular libraries or different code compilations. This is useful if the developer wants to use a specific library functions executed from RAM. The library must be previously compiled; the generated compilation file with extension ".a" must be added to the linker script functions. To include it in the linker script, the folder "linkscripts" must contain the next four files (*Fig. 5.9*).

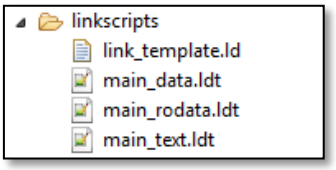


Fig. 5.9: Linker scripts files.

The first file “link_template.ld” is provided as a template by LPCXpresso. On the other hand, the remaining three files must be created and contain next code (note that the programing language is specific from the linker “FreeMarker” editor [25], [26]).

main_text.ldt:

```
*(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .text*)
```

main_rodata.ldt:

```
*(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .rodata .rodata.* .constdata
.constdata.*)
. = ALIGN(${text_align});
```

main_data.ldt:

```
*libMYLIBRARYPROJ.a:(.text*)
*libMYLIBRARYPROJ.a:(.rodata .rodata.* .constdata .constdata.*)
. = ALIGN(${text_align});
*(.data*)
```

“libMYLIBRARYPROJ.a” is the name of the library that will be relocated.

The third and last mode consists in relocating the most of the instruction code in the RAM memory. This operation does not copy the booting code for obvious reasons. The content of “main_text”, “main_rodata” and “main_data” must be modified.

main_text.ldt:

```
*cr_startup_*.o (.text.*)
*(.text.main)
*(.text.__main)
```

main_rodata.ldt:

```
*cr_startup_*.o (.rodata .rodata.* .constdata .constdata.*)
. = ALIGN(${text_align});
```

main_data.ldt:

```
*(.text*)
*(.rodata .rodata.* .constdata .constdata.*)
. = ALIGN(${text_align});
*(.data*)
```

Applying these changes, the microcontroller will dump the application from the flash to the RAM, including all declared constants and tables. In practice, analyzing all the tests performed using this last relocation mode, it has been found that the program is not completely dumped to RAM memory. Only the first 200 instructions are relocated, and next address positions where the code should be allocated appear with null content. This causes that the microcontroller gets frozen when attempting to execute it.

- **SPIFI operation modes**

The SPIFI memory interface allows working in two operation modes with different objectives, named “memory mode” and “command mode”. Memory mode allows the developer to use standard libraries of this microcontroller, and it is the easier option for beginners. On the other hand, the command mode is the hardest way to develop code, but it also gives more flexibility to designs.

The maximum storage capability that the memory mode can support is 128MB or, in other words, 1Gb. Taking into account that the memory mode shares flash memory for data and program code, useful data capacity will be a bit less than the maximum value. The memory mode also forces the user to use only one memory device, and it must be compatible with the libraries, otherwise the microcontroller will not recognize the flash memory and it will get frozen. This device needs to be initialized, configured and associated at the beginning of the communication process. The microcontroller has reserved a set of addresses to map the flash data into an internal bus, 0x1400 0000 to 0x17FF FFFF for 64MB mapping with debug capabilities, and 0x8000 0000 to 0x87FF FFFF for 128MB mapping without debug possibility.

The command mode, on the other hand, supports unlimited storage capabilities with little hardware modifications, but when this mode is activated the microcontroller does not accept XIP instructions anymore. Therefore, for using this mode is mandatory to relocate the full code of the program in the RAM memory. The command mode allows using any memory device compatible with QSPI protocol, more than a single device. This

fact is possible because SPIFI does not necessary needs to be associated to a memory device, so the developer needs to configure the correct commands (or operation codes) to communicate with the memory.

By using more than one memory device, the storage capacity constraints are removed. Theoretically LPC4370 does not support more than one memory device by its SPIFI peripheral, because this microcontroller only have one chip enable pin (CE). Fortunately in practice more than one chip enable can be emulated by using the original CE pin in conjunction with GPIO pins. Using the GPIO port and electronic switches, the signal coming from CE pin can be mapped into different memory devices, depending on the selected chip.

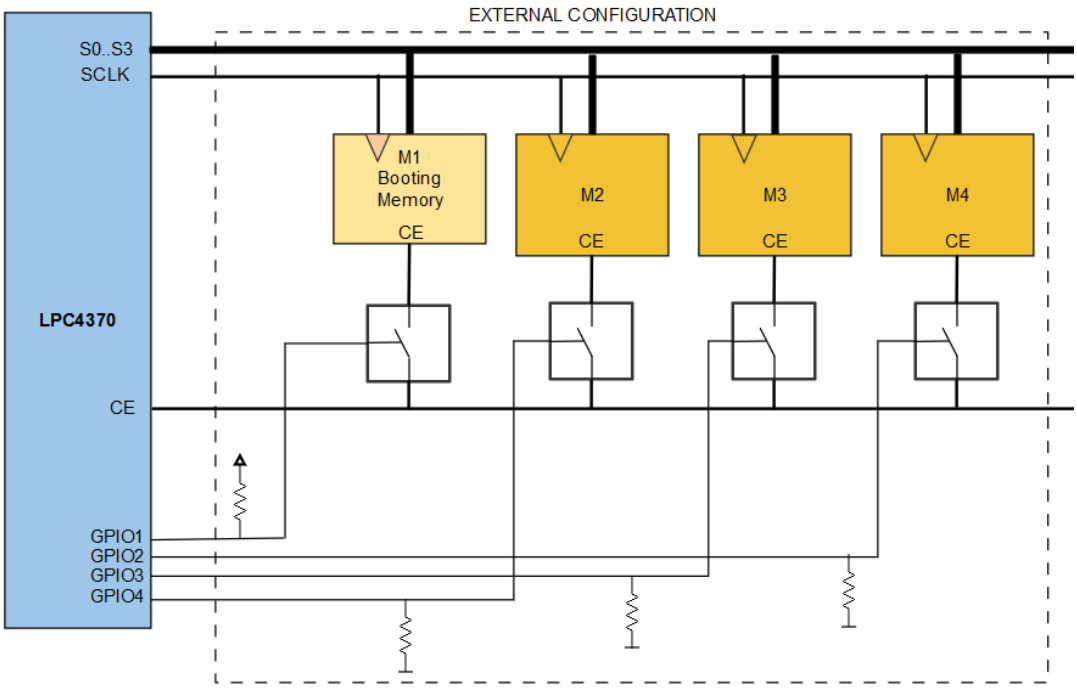


Fig. 5.10: SPIFI extension scheme.

Fig. 5.10 shows a proposal of design that allows extending the memory capacity from one device into 4 devices. It can be extended using this procedure to N devices. Bus data pins (S0 to S3) and the SPI clock (SCLK) pin of the microcontroller are shared by all the memory devices. The chip enable (CE) signal, on the other hand, is re-mapped from the GPIO port, connected to the electronic switches (these can be solid state relays, for example), to each memory device. Memory M1 can be different from M2 or M4, and it must be compatible with the microcontroller memory drivers. Note that there are GPIO's pull-up and push-down resistors. Their purpose is to ensure that memory M1 is selecting at the microcontroller startup. GPIOs are initialized as input pins, in high-

impedance state and, due to this, the value at the output must be externally forced to ensure that switch 1 is closed and the other switches are open.

When the microcontroller is booting, the SPIFI peripheral initializes memory M1 in Memory mode, and then the program code is completely dumped to the RAM. In this way, the Command mode can be initialized. M2 to M4 memory devices must be identical and they may not be compatible with drivers. For memory selection, the GPIO pin must acquire the value logical ‘1’ at the selected memory and logical ’0’ at the unselected ones. The CE is mapped to the desired memory and the content can be accessed by the microcontroller by means of the commands written on the SPIFI.

- **Tests and results**

It was not possible to obtain any conclusive result due to the lack of information and support provided by the manufacturer (NXP) of this peripheral. Existing documentation and applications notes have insufficient information content, and they do not allow developing a program able to save information into a flash memory using both operation modes.

The first test performed was to write a value coming from a microcontroller internal RAM variable to the flash memory at a given moment, working in memory mode and using LPCSpifiLib 1.03 [27]. In order to use provided functions, this library must be compiled and relocated in RAM memory at the booting stage, as previously commented. Then the library must be initiated and linked to the memory device. If the microcontroller is being booting from flash, the memory mode is already active and the memory initialized, but the library needs to know information about the device in order to start association. Therefore, the library must be initialized informing on the device which is connected to the MCU. Once the library is initialized, the developer may call read or write functions. Unfortunately, the microcontroller gets frozen when this function is executed.

The second test performed was meant to the same operation, but this time in command mode. The SPIFI peripheral must be reset in order to change the operation mode by writing ’1’ on the SPIFI’s STAT register bit 4 (reset), and then the current mode (memory mode) is aborted. To activate the command mode, a writing operation on the CMD register must be performed. In practice, when program the sets the STAT reset bit, the microcontroller gets frozen. If this operation is performed without copying the full application code to the RAM at the booting stage, the microcontroller gets automatically

frozen. The explanation is that the SPIFI stops the XIP operation, and the program counter points to a nonexistent address (the memory gets disconnected to the internal bus).

The third test was a variation of the previous one but performing the code relocation process. When the MCU boots and performs the relocation, only the first 200 instructions are copied and the microcontroller program counter points to null code addresses, and gets frozen again. There is no found reported explanation for that.

5.4.2. External Memory Controller

The EMC peripheral allow communications with external the memory device using a parallel data bus. This peripheral is designed for controlling external RAM memories, such as SRAM or SDRAM, and it also supports other memory types such as ROM or NOR flash. In case of SDRAM, the EMC includes a self-refresh function.

For LPC4370 100 pin packages, the EMC peripheral has available an 8 bits data bus and a 14 bits address bus, which allow mapping a total of 2^{14} addresses. The EMC can also support memories with row (2Kb, 4Kb and 8Kb addresses) and column access to improve storage capacity. Then the total number of addresses that this peripheral can handle is given by $2^{13} \cdot 2^{14}$, which corresponds to the combination of rows (8Kb) and columns that it can select. For an 8 bit memory, this means a maximum storage capacity of 256MB. For 256 pin packages, the EMC has a 32 bits data bus and a 24 bit address bus, allowing a total of 2^{24} addresses, what is a maximum storage of 128GB in 32 bit-word memories, but limited by the EMC design to 256MB per attached chip.

This peripheral has 4 chip enable (CE) pins, supporting up to 4 memory devices connected to it. Then the maximum addressable space is 896MB due to internal limitations.

Flash or static RAM (SRAM) memories do not accept row and column access, so the maximum storage capacity using parallel NOR flash and SRAM memories will be 16MB per chip, providing a total of 64MB in addressable space.

When using a synchronous dynamic RAM (SDRAM), a total space of 896MB with 4 memory devices can be achieved. When using dynamic mode, the peripheral performs a self-refresh operation in order to keep data integrity. This amount of data can only be achieved in the LPC4370's 256 pin version.

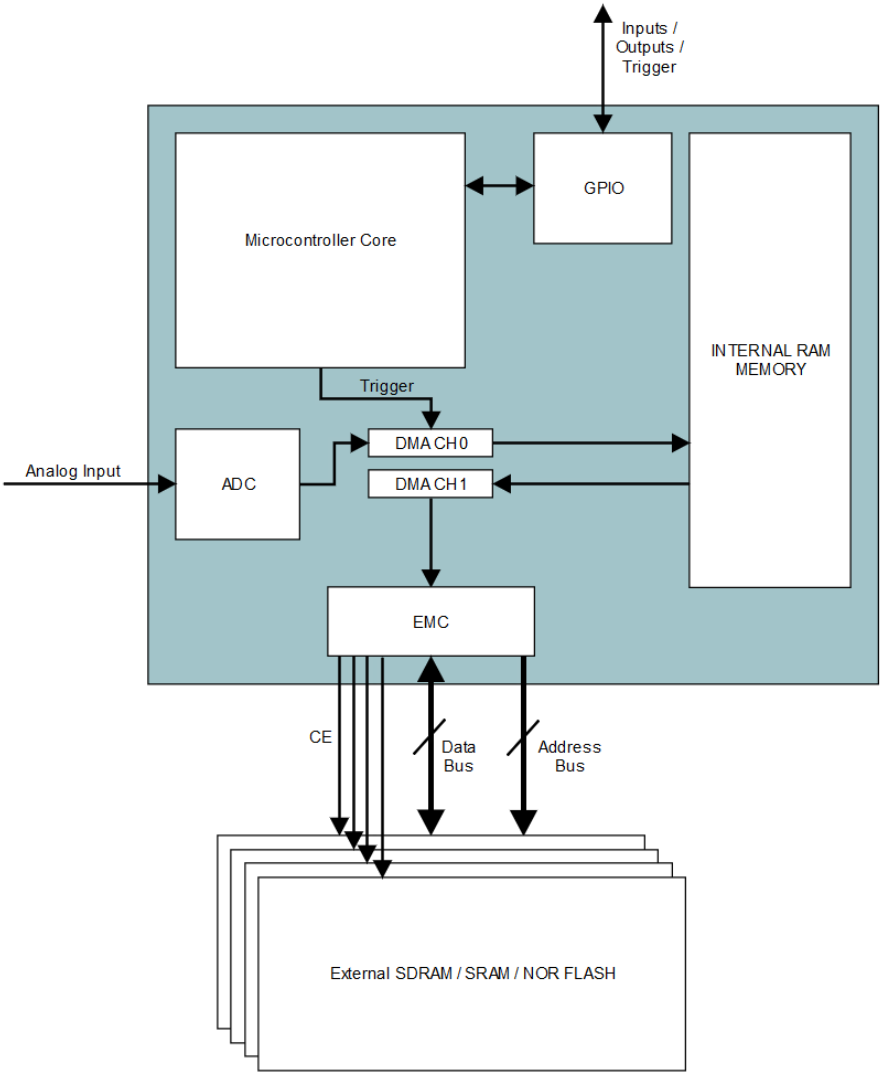


Fig. 5.11: EMC data acquisition working scheme.

The proposed design (*Fig. 5.11*) is to map the ADC into the EMC by using the DMA peripheral. The ADC samples the input signal and then, by using DMA channel 0, it dumps it to a buffer in the RAM memory. When the buffer has been filled, the DMA moves it from the RAM to the EMC peripheral, which moves out the data into external memory. The GPIOs can be used to interact with radar triggering signals or to map out the microcontroller digital signals for timing calculations. LPC Link2 does not support the EMC mapping, so a design that uses this peripheral must be included in a specific PCB design. Due to the low storage capacity that can be supported by this peripheral and the high complexity of designing and implementing a 256 pin BGA based board, it has been decided to explore other options of storing the acquired data.

6. Software developments

In USB communications, host system need to use an additional software to connect with the device. This software also allows performing all operations related with the data transmission. With this program the host can be associated with the USB driver, the driver is in charge of the communication between the library used in software, and the USB physical peripheral (or port). The developed software is based on LibUSB, an open source library for performing USB port control.

A program with two versions for three operative systems (OS) was developed. This program perform the communication with the microcontroller. The versions are for Windows, Linux and Raspbian (a Debian distribution for Raspberry Pi). Essentially the program is the same, but the operation is different for each OS software version. The Windows version has been programed in C#, an object oriented programming language, and the Linux version in ANSI C, that is non-object oriented language. Advantages of using object oriented languages in comparison with ANSI C, are the creating and starting program threads facilities, which allow performing multitasking inside the same program. As a drawback, using object oriented languages are sometimes difficult to develop programs.

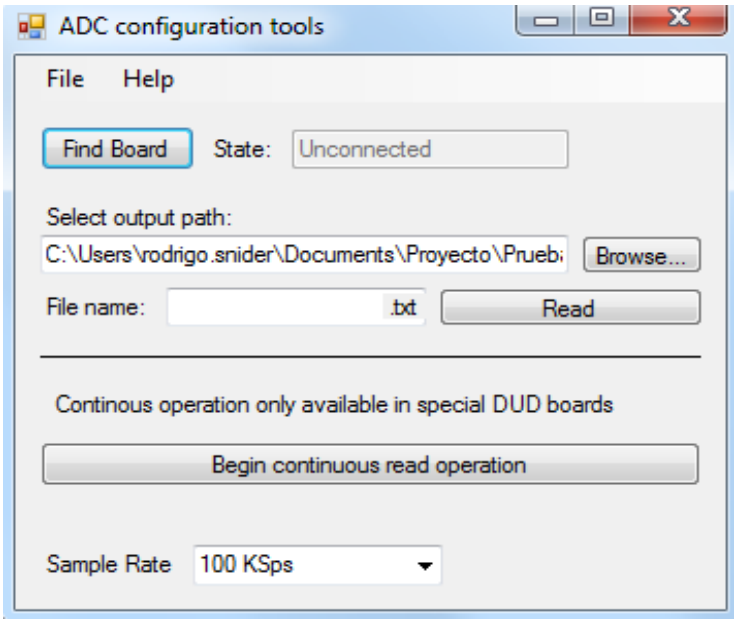


Fig. 6.1: ADC configuration tools GUI.

First developments were performed in Windows, using LibUSBdotnet library distribution for C#, interfaced with a graphical user interface (GUI) in Windows forms

format (*Fig. 6.1*). The program was developed by steps, when tests performed on the microcontroller code required a specific functionality in host side, it was implemented on the program. So at the end the program has multiple functionalities, for different board operation modes. (*App. C.1.*)

As can be appreciate in *Fig. 6.1*, the functionalities that this includes are: “Find Board”, which opens the USB port and associates the board to the program; “Read”, that reads the samples stored at internal RAM memory of the microcontroller, and saves it to the specified file; and “continuous read”, that forces the microcontroller to dump ADC samples directly to the USB, and then saves the received data on the specified file.

Functionalities as arrange received data, or change the format of the files was added also in order to test and analyze received data. When an operation as read or continuous read is performed and data is saved to the file, the user can select to arrange it, differentiating three types of arrangement. First is for read from RAM memory, it only arranges 64000 samples, that are the maximum samples that can be stored in RAM. For that if a continuous read file is arranged with that, output will be incomplete. Second is to arrange a “dump” file, its objective is to arrange a continuous file received in hexadecimal format coded in ASCII, actually this method is not used. The last is to arrange a binary file, it is used for arranging files obtained in continuous read operations, changing the format from binary file to ASCII hexadecimal. Arrangements are used for converting data in the file to a hexadecimal array, to allow easy importation in MATLAB. The program also allows plotting the selected file in MATLAB without necessity of opening MATLAB and manually charging the file. The last functionality is to check fails of a transmission, this is only for testing purposes and there are some limitations in order to use it correctly.

Checking errors algorithm is a probabilistic error founder, based on samples comparison. Supposing an input signal sampled at a certain rate, if a transference error happens, 4 buffers are lost (*Sec. 5.3.*), provoking that a large number of samples will be lost. The way to counter these errors is to compare the acquired signal samples with the next sample in a loop, if there is a difference greater than the expected value, it means that an error in transaction was happened. In this code, a value between 300 and -300 quantization levels between samples is considered a normal value, samples out of this range will be considered as an error.

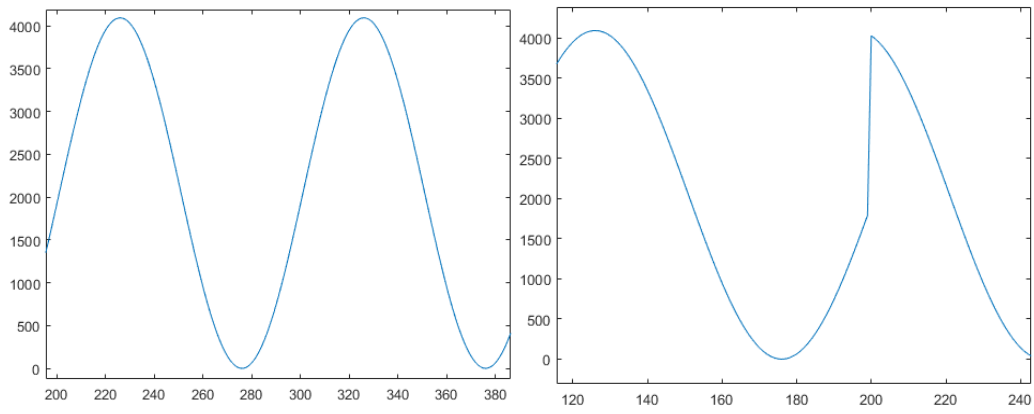


Fig. 6.2: Signal sampled; left, signal with no errors; right, signal with error.

Fig. 6.2 shows a typical error topology, the error is distinguished by the difference between the values of two samples. In tests the relations between the input frequency and the sample frequency was of 100 samples per period, this is used in order to avoid, by one hand slow signal variations, and by other hand fast variations which may cause non-detected errors. Error checking is only used in USB speed tests, analyzing the sinusoid, errors in packet transactions can be found in order to characterize the link limits. In other applications this algorithm has no sense, it only can be used for a constant frequency and amplitude sinusoid.

Fig. 6.3: Linux ADC config tools version.

Linux software only supports continuous data acquisition mode. This program also implements other functions like error checking, and sample rate selection. There was found in tests that is no real improvement between Windows and Linux versions, the maximum sample rate achieved by the Linux program is the same than for Windows. Linux program version does not implement any GUI, so the program is launched in terminal (*Fig. 6.3*).

Software developed in Linux can be easily converted to be able to run in Raspbian OS. When the program runs in Raspberry Pi environment, the maximum achievable sample rate of the microcontroller is reduced. (*App. C.2.*)

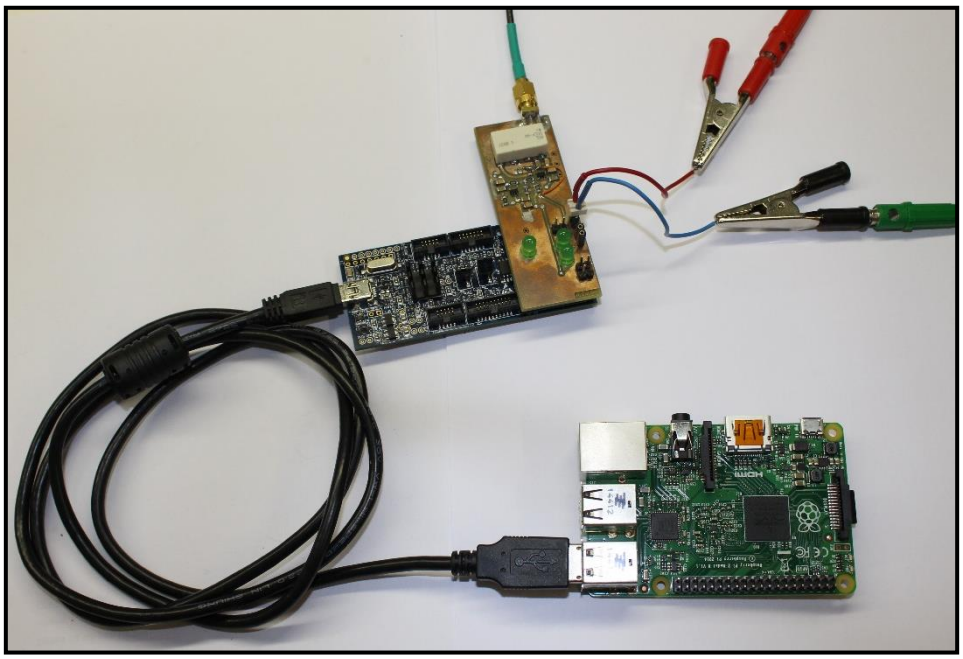


Fig. 6.4: Board to Raspberry connection, performed with USB bus.

This is because the Raspberry storage system is based on a SD card, and computer by other side is based on hard disk. SD card maximum transfer speed is around 10MB/s (Class 10), compared with hard disk, which transfer speed is around 160MB/s, it is 16 times lower. As using PC, with SD card the sample rate is limited by the storage speed at the host side, as previously commented. If the storage speed is reduced the maximum sample rate is reduced also. Using Raspberry Pi, the maximum achievable sample rate without errors is around 250KSps.

7. Acquisition analysis

7.1. Board performance

In this section, noise effect at the samples will be analyzed. Analysis is performed on the two developed boards in order to make a comparison between them. The main purpose of board analysis is to contrast the noise effect before and after a correct PCB design, implementing filter stages and noise reduction techniques. In order to do that, some considerations must be taken into account.

First consideration is that the noise power present at the digitalized signal will be lower bounded by the ADC itself, ADC is introducing a minimum noise power due to the process, and this will be reflected at the measures. Second consideration is that all the components that include the board, as signal conditioning circuit, will add some noise due to the supply fluctuations and non-idealities. Also the board itself is acting as an antenna, and for reducing it an accurate design must be performed. Final consideration is that non-ground shielded cables used in input signal connections, induces noise.

First board to analyze is Connector Adapter board (*Sec. 4.2*), now called “board 1”, with the connection shown in *Fig. 7.1.a*. This board is not optimized in noise reduction, and has no additional filters, is only an interface between the link2 board and the wave generator. The second board to analyze is ADC extension board (*Sec. 4.3*), now called “board 2”, connected as shown in *Fig. 7.1.b*. This last case the board is optimized to reduce noise effect on samples. The comparison between two boards will show the signal-to-noise ratio (SNR) improvement due to filtering, better connections and better PCB design.

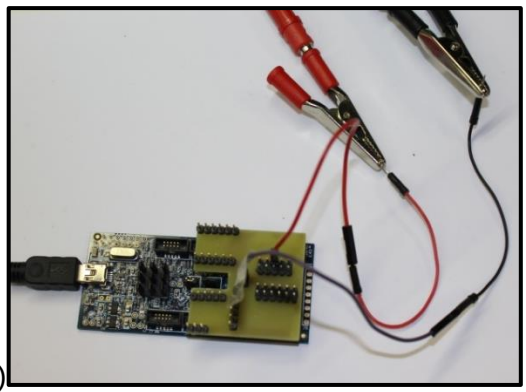
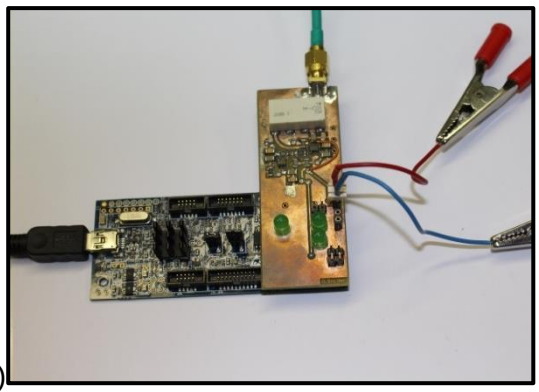
a)  b) 

Fig. 7.1: Connection of the 2 boards: a) Link2 with board 1, b) Link2 with board 2.

Input port of each board, is connected to the output of a waveform generator. Waveform generator has an output impedance of 50 Ω, which match to the filtering stage of board 2. Input impedance of the board 1 is the input impedance of the ADC pin, which corresponds to 1 MΩ. This causes that in order to measure the same voltage in both boards, generator must be configured at two times output voltage applied to the input of board 1 for board 2. (*App. D.1.*)

The objective of first test performed is to determine the noise power level for a continuous DC voltage at the input port. Waveform generator is configured in DC mode with an output voltage of 500 mV.

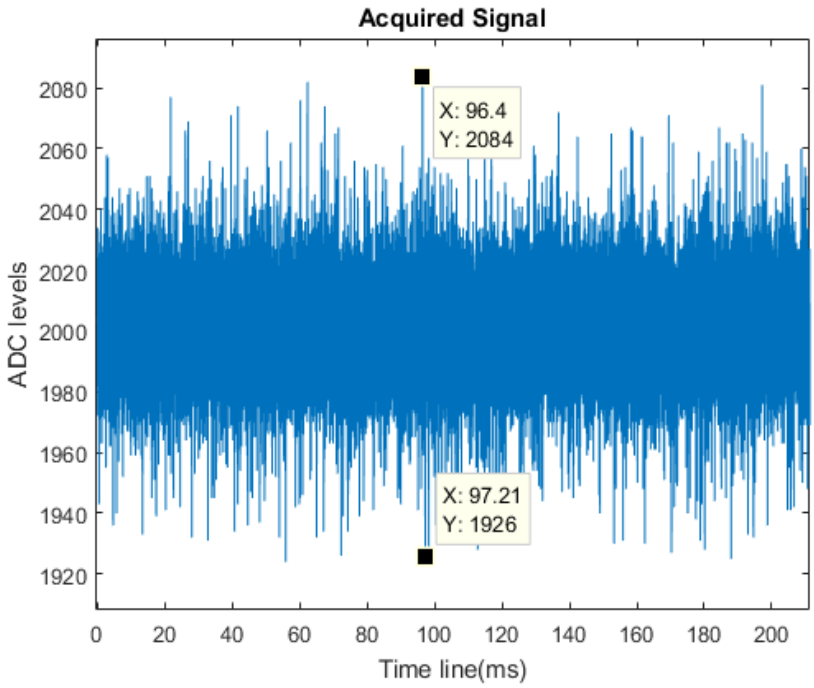


Fig. 7.2: board 1, acquisition of DC voltage.

Fig. 7.2 shows signal acquisition using board 1 for a short time period at 1MSps. As it can be observed, the signal value is fluctuating due to noise. Variations are from 1926 to 2084 levels, that it corresponds to 476.2 mV to 507 mV. Then the noise amplitude in this case is 15.4 mV. Note that the acquired DC signal is not perfectly centered on 500mV (actually is centered on 491.6mV approximately), this happens due to internal microcontroller design, voltage is acquired with a certain offset.

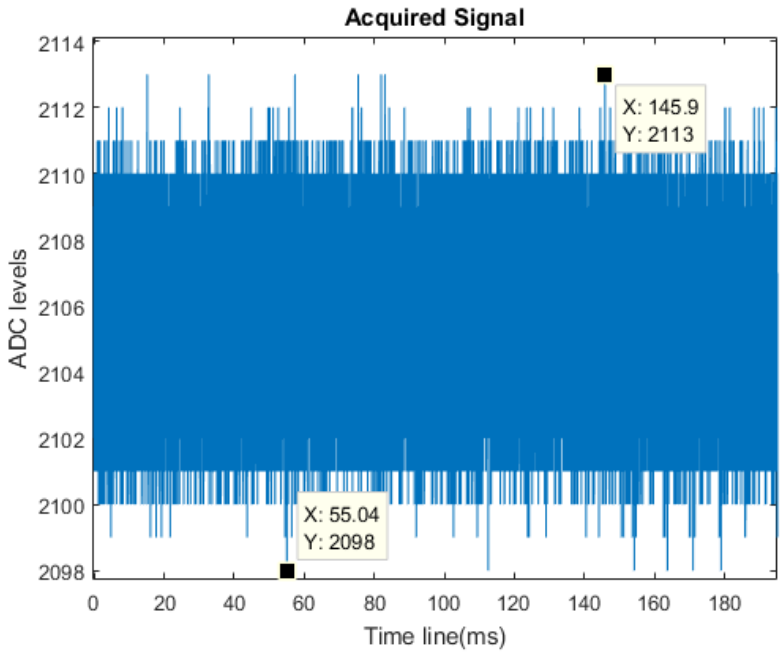


Fig. 7.3: board 2, acquisition of DC voltage.

The same test is performed using board 2 (*Fig. 7.3*). Note that SMA connector allows more robust connection between the board and the generator, reducing the noise due to connections. Compared to *Fig. 7.2*, the noise now is considerably lower; signal varies from 2098 to 2113 levels, which in voltage terms are 509.7 mV and 512.7 mV. This variation corresponds to 15 levels of noise amplitude, which is 3 mV. Signal is centered on a different voltage level in comparison with previous case; board 2 additional circuitry is modifying the voltage at the microcontroller input pin, adding an offset voltage due to resistor non-ideal values at operational amplifiers.

The objective of the second test performed is to analyze the SNR when applying a sinusoidal signal at the input port of both boards. A sine wave with 400 mV of amplitude, 500mV of offset and a frequency of 10 KHz is applied to the input. The analysis is performed by the calculation of power spectral density (PSD) [28] of the function. PSD is used in signal processing for describing a signal, it represent power distribution of each frequency contributions that the signal contains. As the acquired data is discrete, and ADC output is a finite number of samples, PSD cannot be calculated. Then it must be used an estimator for spectral density, named as periodogram; *Eq. 7.1* is the numerical calculation method.

$$S(f) = \frac{\Delta t}{N}\left|\sum_{n=0}^{N-1} x_n e^{-i2\pi n\Delta t f}\right|^2, \qquad -\frac{1}{2\Delta t} < f \leq \frac{1}{2\Delta t} \tag{7.1}$$

Periodogram can be calculated with MATLAB in two steps; first is to compute discrete Fourier transformation (DFT) of the acquired signal windowed with a length N number of samples; length value in this case corresponds to 1,000,000 samples. The window length affects to frequency resolution, and representation of the signal tones; greater length values imply better resolution and tone representation, but also longer computational times. Once the DFT is computed, then the result must be multiplied by its conjugate, and then divided by the total window length. Result can be plotted in logarithmic representation. (*App. D.2.*)

Spectral density estimation of the signal acquired using board 1 is shown in blue at *Fig. 7.4*, it can be appreciated the frequency response of the rectangular window on the 10 KHz tone.

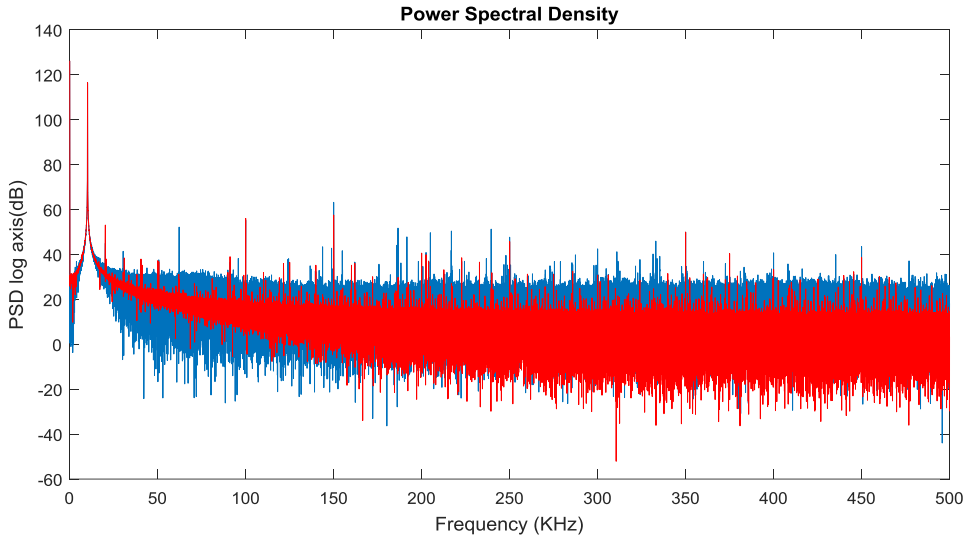


Fig. 7.4: Signal periodogram comparison between boards.

Analyzing the blue periodogram (board 1), the input sinusoid power representation is about 116.3 dB and the noise level is located at 32.5 dB approximately. The SNR can be calculated as the signal power, divided by the sum of all the noise power contributions (summation of all the DFT points without the input sinusoid and distortion, multiplied by the bandwidth that each point represents). MATLAB implements a function for this calculations “snr(x)”. The corresponding value for signal-to-noise ratio of board 1 is 42.45 dB. The signal periodogram in red (board 2) has a power representation for the input sine wave of 116.5 dB, its noise level is now 20.2 dB. The result of computing the SNR is 55.06 dB. It can be observed that the greater spurious tones (105, 150, 250 and 355 KHz) are shared by the two periodograms. It means that the designed filtering stage

circuit does not introduce it, and may be caused by the wave generator, the ADC itself or the environment test conditions.

Antialiasing filtering stage, shielded cables used on connections and a good PCB designing, reduces the amount of noise of the digitized signal. With the use of prototype board designed in this project, “ADC extension board”, the signal-to-noise ratio of the acquired signal is getting improved by 12.61 dB. This value is especially important when input signals has a very low power, for that case, improving SNR implies that signals with smaller power can be acquired and detected. The SNR is affected by the spurious components of the signal spectrum, and, eliminating these, the ratio may be improved. Also is affected by the noise due to the antialiasing filter, which cutoff frequency must be 5MHz for maximum filtering at 10MSps.

7.2. Analog input characterization

In this section, LPC4370’s ADC will be analyzed. Microcontroller’s ADC input pin will be mapped out using “ADC extension board”, it will filter the input signal and reduce the noise. For a correct characterization, it is necessary to use extension board in analysis, as the final application will include same design for analog input pin.

7.2.1. Spurious-Free dynamic range

Spurious-Free dynamic range (SFDR) [15], [29], is the power relation between a single-tone input signal and the largest spurious (non-signal) component inside the ADC bandwidth. ADC distortions produce harmonics on the input signal; this distortions can be caused by nonlinearities of the converter itself. For that, SFDR is an important parameter that defines the quality of the converter.

NXP provides a table with theoretical ADC SFDR parameter [19] which values are shown in *Table 7.1*, for two different sampling frequencies.

Sample Rate	SFDR (dB)
10 MSps	80
60 MSps	75

Table 7.1: SFDR values from datasheet

These values are measured by the manufacturer at indicated sample rate, an input sinusoid signal with a frequency of 1 MHz and a peak to peak voltage value equals to the dynamic range of the converter. Now, repeating this analysis at 10MSps with the same conditions, will give practical results about this parameter. (*App. D.2.*)

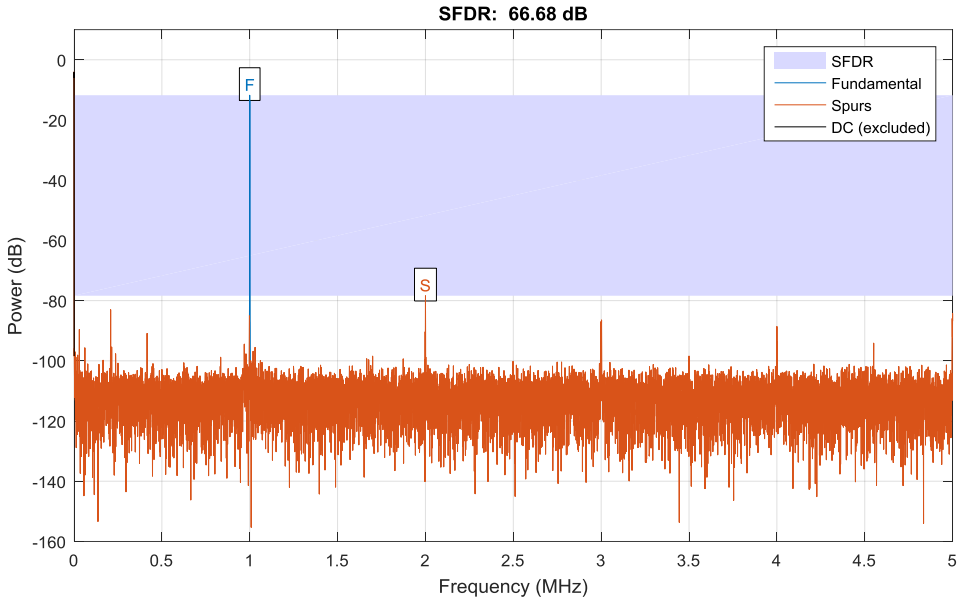


Fig. 7.5: measured SFDR for 10MSps.

Fig. 7.5 shows measured SFDR value using extension board. Its values differ with the expected value due to datasheet. At 10MSps SDFR value is 80 dB in the datasheet and the measured value is around 66.6 dB, 13.4 dB lower than expected. Waveform generator and ADC are both introducing distortion at harmonic frequencies of the input signal. So as the generator is introducing harmonics, it can be a part of the 13 dB difference between the datasheet and the test values.

7.2.2. Signal-to-Noise-and-Distortion ratio

This quality parameter, abbreviated as SNDR and usually SINAD [15], express the relation between the signal and noise, including the harmonic distortion power contribution at the calculations. Is a very important parameter for characterizing an ADC, the conversion produces distortion on the acquired signal.

SNDR is related with the effective number of bits, a characteristic that represent the number of bits used for represent the input signal. So it is a direct quality factor about the converter as it defines the useful bit resolution, then if the converter have N bit

resolution but K effective bits, K bits will be describing the useful signal, and N-K bits describing only the noise and distortion.

NXP provides a table with theoretical number of bits that ADC disposes, values are shown in *Table 7.12.* A comparison is performed repeating the analysis with the same conditions stated in previous section and the theoretical values.

Sample Rate	ENOB (bits)
10 MSps	10.4
60 MSps	10.1

Table 7.2: ENOB values from datasheet.

The procedure for SNDR calculation is similar to the SNR. The difference is, as on the SNR only the signal power and the noise is taken into account (distortion is discarded), in the SNDR distortion also is included as a power contribution. So the signal power is divided by the sum of all the non-useful signal power contributions (summation of all the DFT points without the input sinusoid, multiplied by the bandwidth that each point represents). . (*App. D.2.*)

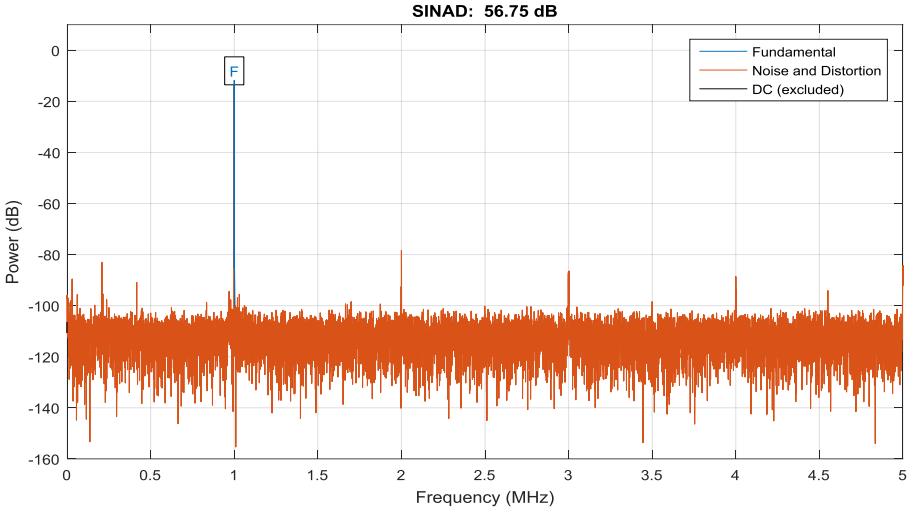


Fig. 7.6: Periodogram and SNDR calculation.

The conversion from SNDR (*Fig. 7.6*) to ENOB is calculated with *Eq. 7.2,* units of SNDR are dB and ENOB in bits. Found SNDR value is 56.75 dB,

$$ENOB = \frac{SNDR - 1.76}{6.02} \tag{7.2}$$

Substituting SNDR value at the equation is obtained 9.134 effective bits, a smaller value than proportioned by the manufacturer. It can be partially produced by the

antialiasing filtering stage, as the used cutoff frequency is 44MHz and the more convenient one should be 5MHz when sampling at 10MSps. Also the difference could be due to the additional circuitry that the extension board includes, or also the PCB design or connections.

7.2.3. Nonlinearity

ADC acquisition can have an erroneous representation of the input voltage along its dynamic range. This error is called nonlinearity, or linearity error. Nonlinearities can be classified as differential nonlinearities (DNL) and integral nonlinearity (INL). Both parameters are related. DNL is the error produced when the voltage range that defines each quantization level, is not uniform among each level. INL parameter is derived by the effect of DML along all the levels. So when different levels has different input voltage range representation, the output response becomes different from the ideal value. INL defines the difference between the real output function and the ideal function.

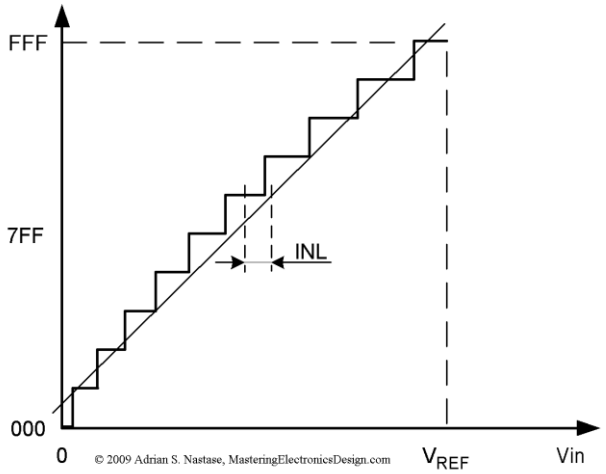


Fig. 7.7: INL, real vs ideal acquisition voltage ranges. *[31]*

Fig. 7.7 illustrates the INL error, the linear slope corresponds to the ideal ADC response. The real ADC response has different voltage range for different levels (DNL), due to the summation of these contributions the response is modified along the dynamic range, and INL appears.

In this section, LPC4370's ADC linearity will be analyzed. The way to do it is to generate a sawtooth signal, with a large amplitude value to cover all the dynamic range,

and a low signal frequency in comparison with the sample rate. When the ADC has captured the signal, the INL error can be found by subtracting one sawtooth slope, by its polynomial approximation of degree 1. (*App. D.3.*)

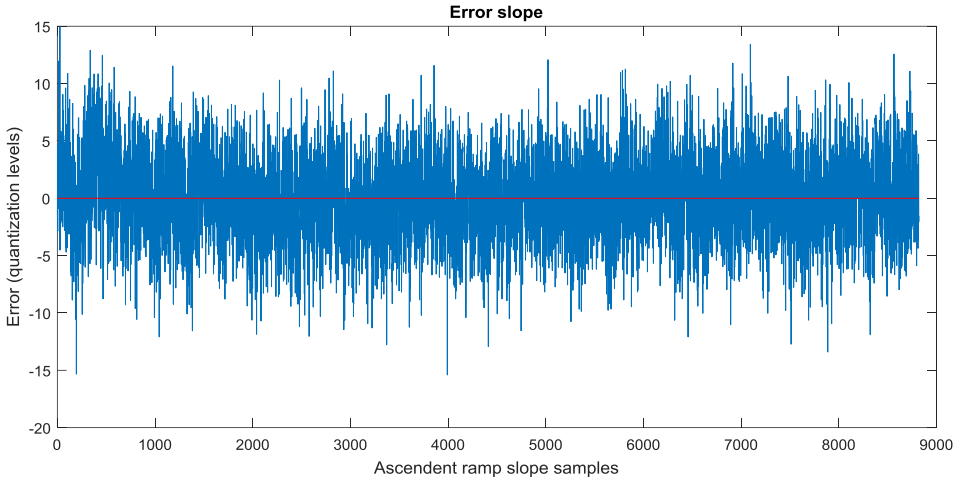


Fig. 7.8: INL representation on a sawtooth slope.

The subtraction operation of the sawtooth slope with the trendline, is plotted in blue at *Fig. 7.8.* In red is plotted the zero line, which corresponds to the zero error points. The INL error is the difference between the red and the blue lines. The blue is the captured error, its value fluctuates among samples due to the noise that includes the signal. Can be observed that the blue plot values are close to the red line, and 16 quantization levels corresponds more or less to the non-effective bits representation.

Error type	Value In bits (10MSps)
INL	1.1 LSB
DNL	0.7 LSB

Table 7.3: nonlinearity values from datasheet.

Datasheet provides theoretical values for nonlinearity parameters. These are shown at *Table 7.3.*As a conclusion of this analysis, the ADC does not introduces almost INL error as show the plot. Samples acquired inside the dynamic range of the ADC, are represented by a value close to the reality. As the acquisition is noisy, it cannot be analyzed the DNL of the converter. The error mean value (mean of blue plot) is -1.06e-12, it is a very low value and implies that the nonlinearity mean over all the dynamic range is almost 0. So as the error is around the ideal one, although the exact maximum INL cannot be calculated due to the noise presence, the values are near the datasheet ones

8. Conclusions and future lines

Researches on different acquisition modes and tests performed with the LPC4370 microcontroller, have shown the possibility of achieving the maximum sampling rate on pulsed radar applications, where burst acquisition is typically needed. For applications that needs a continuous acquisition mode, the maximum sample rate achieved is 2MSps. This rate can be increased if the duty cycle of the acquisition process is modified; duty cycle reduction increases the maximum achievable rate. These interesting conclusions results in a wide coverage to the needs of radar applications.

It must be taken into account, that LPC4370 is the first microcontroller that integrates a fast speed ADC. Hopefully in the close future other MCUs with fast signal acquisition capabilities will be developed with better data throughput and support for high speed massive storage devices, like solid state disks.

The speed of 2MSps is limited by the operation of the PC's hard disk drive, not by the microcontroller, so speeds up to 10MSps can be achieved using USB if the hard disk access times are reduced.

For ground radars a computer can be used, the software version that has a better performance for this type of acquisition is the Windows version using C# programming language. In a continuous acquisition operation mode, the maximum achieved speed has been 2MSps.

For radars mounted on small UAVs a computer can be used too, but, as commented in the previous sections, this is not the optimal solution due to the high mass and power consumptions. Some high capacity data storage strategies were investigated, for example as using SPIFI peripheral, but in the practice it was impossible to implement a prototype that uses it due to several problems and the still limited information provided by the manufacturer. The final solution for portable designs was to use USB to communicate with a Raspberry Pi, but with a reduced performance of 250KSps due to the SD card speed limits. In spite of the encountered difficulties, software prototypes and circuit schematics for increasing store capacity using SPIFI, as well as EMC circuit block diagrams are provided for future researches.

Maximum speeds are measured for continuous operation, so sampling on intervals can increase the maximum sample rate. Also, it has been seen that the maximum sample rate, depends on the number of written bytes to the storage device at the PC

side. Increasing this number the write function efficiency also increases, so at the end the maximum sample rate increases too. Knowing these parameters a sample rate improvement can be performed in future developments.

So as a final conclusion, the developed prototype in the laboratory allows covering radar applications requiring continuous signal acquisition, with mid-low requirements in terms of sample rate, and high speed burst acquisition for pulsed radar applications. Hence, at the end a reasonable good digitizer can be used for low cost signal acquisition purposes.

The future lines of this project may be the following ones:

- Research on SPIFI peripheral, with support of the manufacturer to fill the encountered gaps of information. NXP offers a pre-paid supporting service, so a subscription would be the only choice to keep going in depth.

- Increase the sample rate using USB protocol. It can be achieved by modifying the software on the computer side or replacing the hard disk drive by a solid state disk. For software modifications, speed can be increased substituting the synchronous function “fwrite()” by an asynchronous function of the same type. If this function does not exist, the asynchronous behavior can be virtually simulated by using threading.

- Substitute the board antialiasing filter by a lower cutoff frequency filter, depending on the operating sample rate. In the case of analysis section, it would be good that these tests are repeated with a 5MHz antialiasing filter to be compared with datasheet values.

- Design a board that implements the final system in a reduced format. This design may have not be Link2 but based on LPC4370, improving the design to make it fit the application requirements.

- Improve the connection between the board and the Raspberry Pi. The current maximum sample rate of 250KSps can be improved by modifying the software. Also would be interesting to test if the use of an external hard disk connected to the Raspberry improves the speed.

- Configure the USB peripheral of the board to act as a host, allowing the connection of an external storage device as an external solid state disk.

List of acronyms

Acronym	Description
ADC	Analog-to-Digital Converter
AHB	AMBA High-performance Bus
ASIC	Application-Specific Integrated Circuit
BCD	Binary-coded Decimal
BGA	Ball Grid Array
CAN	Controller Area Network
CE	Chip Enable
CW	Continuous Wave
CPU	Central Processor Unit
DAC	Digital-to-Analog Converter
DC	Direct Current
DIP	Dual in-line Package
DMA	Direct Memory Access
DNL	Differential NonLinearity
DR	Dynamic Range
EEPROM	Electrically Erasable and Programmable Read Only Memory
EMC	External Memory Controller
ENOB	Effective Number Of Bits
FIFO	First Input First Output
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HDD	Hard disk drive
HSADC	High Speed Analog-to-Digital Converter
IDE	Integrated Development Environment
IC	Integrated Circuit
INL	Integral NonLinearity
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LCD	Liquid-Crystal Display
LED	Light-Emitting Diode

LFM	Linear Frequency Modulation
LNA	Low Noise Amplifier
LPF	Low Pass Filter
LSB	Least Significant Bit
MCU	Microcontroller Unit
MF	Multiple Frequency
MSB	Most Significant Bit
NVIC	Nested Vector Interrupt Controller
SDRAM	Synchronous Volatile Random Access Memory
SFDR	Spurious-Free Dynamic Range
SNDR	Signal to Noise and Distortion Ratio
SNR	Signal to Noise Ratio
SOP	Small Outline Package
SRAM	Static Random Access Memory
SSD	Solid State Disk
S&H	Sample and Hold
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCM	Pulse Code Modulation
PLL	Phase-locked Loop
PRF	Pulse Repetition Frequency
PSD	Power Spectral Density
PWM	Pulse Width Modulation
PXI	PCI extensions for Instrumentation
QFP	Quad Flat Package
QSPI	Quad Serial Peripheral Interface
RADAR	Radio Detection And Ranging
RAM	Random Access Memory
ROM	Read Only Memory
S&H	Sample and Hold
SAR[1]	Successive Approximation Register (ADC Sec. 2.3.) Synthetic Aperture Radar (referred to radar applications)
SPI	Serial Peripheral Interface

[1] Note that SAR acronym is used in more than one context.

SPIFI	Serial Peripheral Interface Flash Interface
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
XIP	Execute-In-Place

References

[1] P. M. Rainey, “Facsimile telegraph system”, U.S. Patent 1 608 527, July 20, 1921.

[2] Analog Devices Inc., Walter Allan Kestler. *The data conversion handbook.* 1st ed. Chapter 1 and 2. Burlington, USA: Newnes, Elsevier, 2005.

[3] National Instruments, datasheet. *Digitizer NI PCI/PXI – 5122 datasheet.* [Online] Available: http://www.ni.com/datasheet/pdf/en/ds-241

[4] ADLINK tech, product datasheet. *Digitizer PCI – 9820 datasheet.* [Online] Available:http://www.adlinktech.com/PD/marketing/Datasheet/PCI-9820/PCI-9820_Datasheet_en_1.pdf

[5] Albert Aguasca, Rene Acevo-Herrera, Antoni Broquetas, Jordi J. Mallorqui, and Xavier Fabregas; “ARBRES: Light-Weight CW/FM SAR Sensors for Small UAVs”, PMC, Mar 2013. Digital Object Identifier: PMC3658740.

[6] Clay Stewart, Vic Larson. *Synthetic aperture radar algorithms.* [Online] Available: http://dsp-book.narod.ru/DSPMW/33.PDF

[7] Bassem R. Mahafza. *Radar systems analysis and design using MATLAB.* 1st ed. Chapter 3. USA: Chapman & Hall, 2000.

[8] ARM limited. *ARM architecture reference manual.* ARM DDI 0100E. Cambridge, UK: ARM limited, 2000.

[9] Dogan Ibrahim. *Using LEDs, LCDs and GLCDs in microcontroller projects.* 1st ed. Chapter 1. USA: Wiley, 2012.

[10] Bit tech, Richard Swinburne. *Intel core i7 – Nehaem architecture dive.* [Online] Available: http://www.bit-tech.net/hardware/cpus/2008/11/03/intel-core-i7-nehalem-architecture-dive/5

[11] Peter Tilmanis. "Intro to computer systems". *RMIT University*, 2014. [Online] Available:https://www.dlsweb.rmit.edu.au/set/Courses/Content/CSIT/oua/cpt160/2014sp4/chapter/05/CPUArchitecture.html. [Accessed: 10 October 2016].

[12] Arnold S. Berger. *Hardware and computer organization: the software perspective vol.1.* 1st ed. Chapter 13. Burlington, USA: Newnes, Elsevier, 2005.

[13] Analog Devices Inc, Walt Kester, Dan Sheingold, James Bryant. *Analog-digital conversion handbook.* 1st ed. Chapter 1.

[14] Measurement Computing Corporation. *Data acquisition handbook.* 3rd ed. Chapter 2. USA: Measurement Computing Corporation, 2012.

[15] Clyde F. Coombs, Jr. *Electronic instrument handbook.* 2rd ed. Chapter 6. USA: McGraw-Hill, 1995.

[16] Texas Instrument, Data acquisition article. “How delta-sigma ADCs work”. [Online] Available: http://www.ti.com/lit/an/slyt423/slyt423.pdf [Accessed: 2 October 2016].

[17] NXP main website, products section. “*OM13054*: *LPC Link2 board”*. [Online] Available: http://www.nxp.com/products/reference-designs/lpc-link2:OM13054

[18] NXP comunity website, forums and questions. *Flash driver operation failed.* [Online] Available:https://www.lpcware.com/content/forum/flash-driver-operation-failed-program-operation-failed-validation-or-readback-compare#comment-1149834

[19] NXP, documents section, datasheet. “*LPC4370 product datasheet”.* [Online] Available: http://cache.nxp.com/documents/data_sheet/LPC4370.pdf?pspll=1

[20] NXP, documents section, user manual. “*UM10503: LPC43xx User Manual”.* [Online] Available: http://www.nxp.com/documents/user_manual/UM10503.pdf

[21] Mini-Circuits, datasheet. “*SCLF-44 surface mount filter datasheet*”. [Online] Available:https://www.digchip.com/datasheets/parts/datasheet/302/SCLF-44-pdf.php

[22] Keil website, USB component. *“USB communications”*. [Online] Available: http://www.keil.com/pack/doc/mw/USB/html/_u_s_b__endpoints.html

[23] Tutorials point, C libraries – “stdio.h”. “C library function fwrite()”. [Online] Available: https://www.tutorialspoint.com/c_standard_library/c_function_fwrite.htm

[24] LPCware, frequent asked questions. *“Relocating code to flash ram”.* [Online] Available:https://www.lpcware.com/content/faq/lpcxpresso/relocating-code-flash-ram

[25] Freemaker, documents. *“Template authors and programmer guide”.* [Online] Available: http://freemarker.org/docs/dgui.html

[26] Code-Red Tech, support page. *"Enhanced managed Linkscripts".* [Online] Available:http://www.support.code-red-tech.com/CodeRedWiki/EnhancedManagedLinkScripts

[27] LPCware, libraries and documents. "*LPCOpen LPClibSpifi 1.03_68 library*". [Online] Available: https://www.lpcware.com/system/files/lpclibspifi_1.03_68.pdf

[28] Cygnus research international. "*The data processing studio - spectral*" [Online] Available: http://www.cygres.com/OcnPageE/Glosry/SpecE.html

[29] HRL Lab. LLC, R.H. Walden; "Analog-to-digital converter survey and analysis", IEEE journal on selected areas in communication, Vol 17 Apr 1999. Digital Object Identifier**:** 10.1109/49.761034

[30] Maxim integrated, application notes. "*INL/DNL measurements for high speed ADCs*" [Online] Available:https://www.maximintegrated.com/en/appnotes/index.mvp/id/283

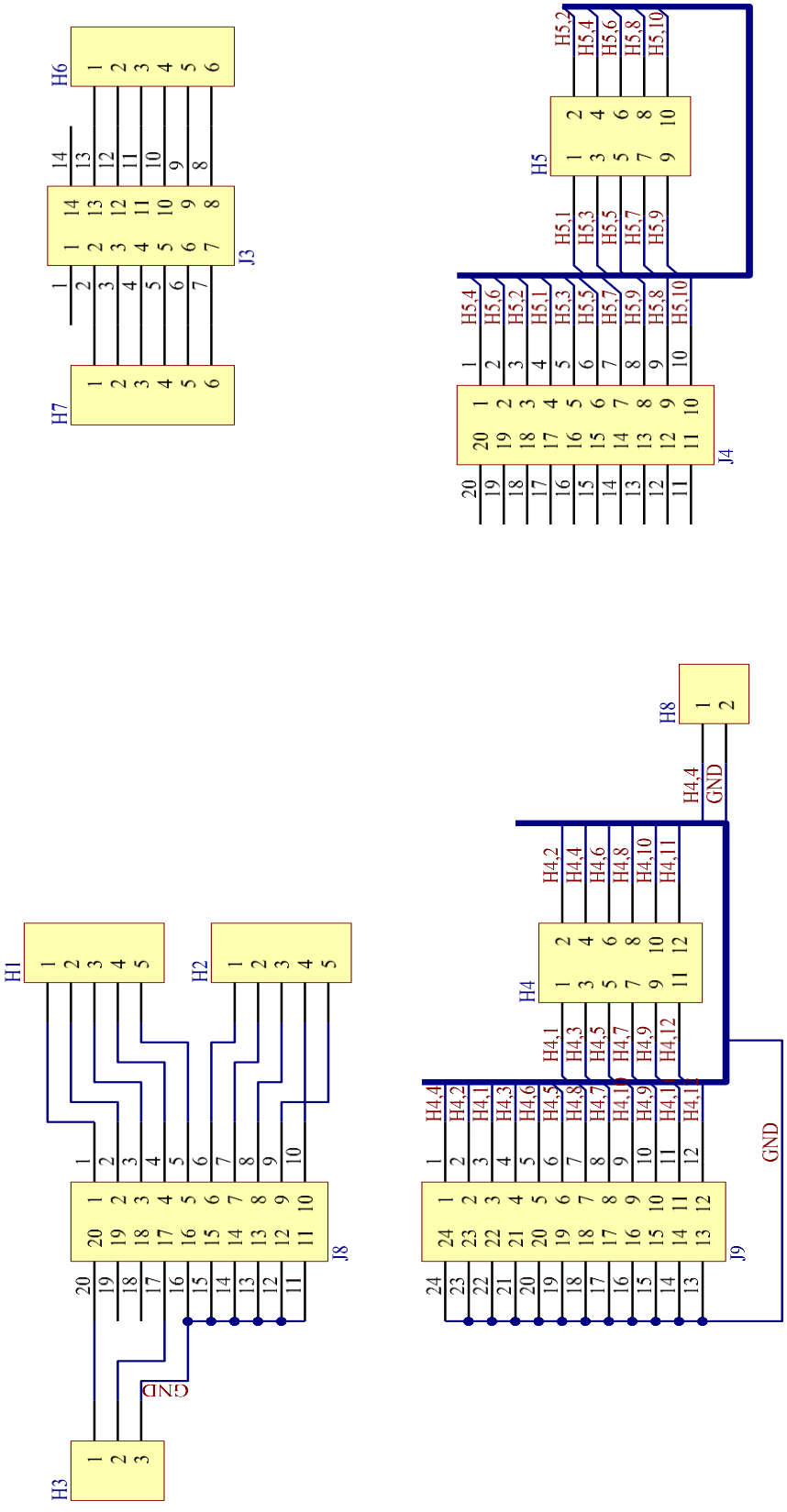
[31] Master in electronics design. "*ADC and DAC integral non-linearity*" [Online] Available:http://masteringelectronicsdesign.com/an-adc-and-dac-integral-non-linearity-inl/

Appendices

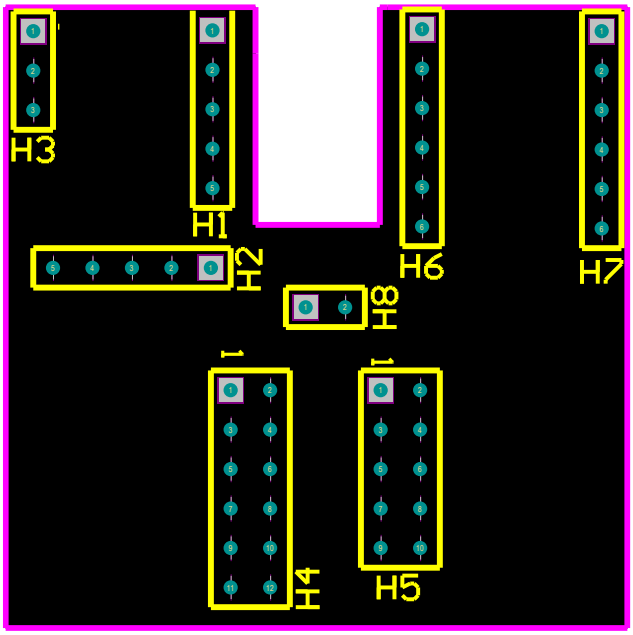
Appendix A
Hardware designs

A.1. Connector adapter board

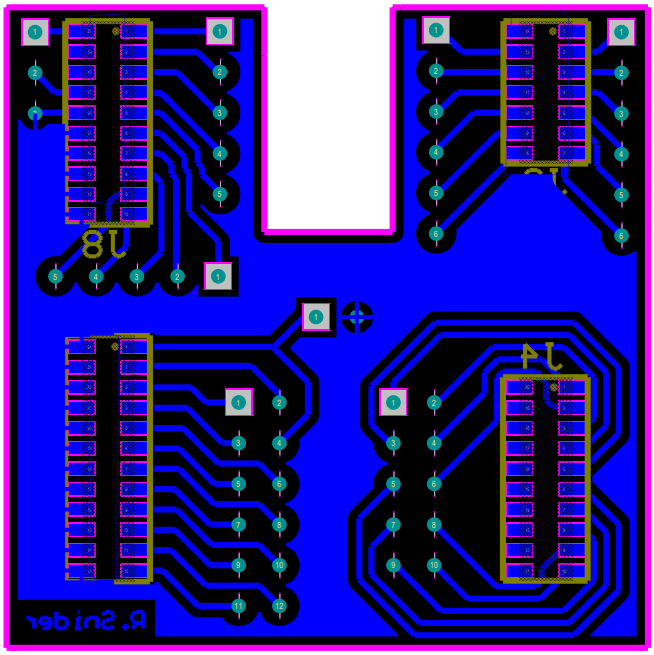
A.1.1. Schematic



A.1.2. PCB, top layer

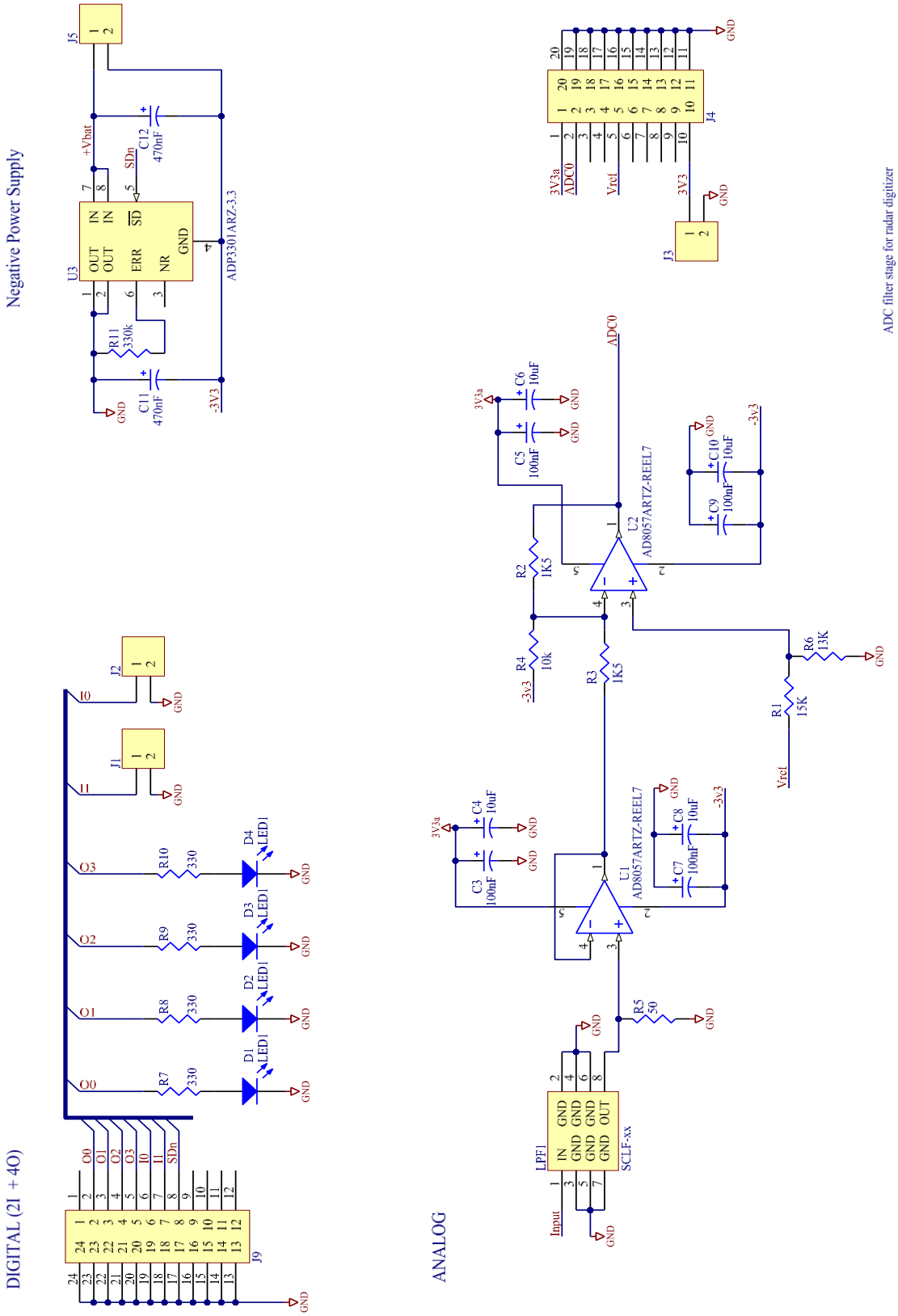


A.1.3. PCB, bottom layer

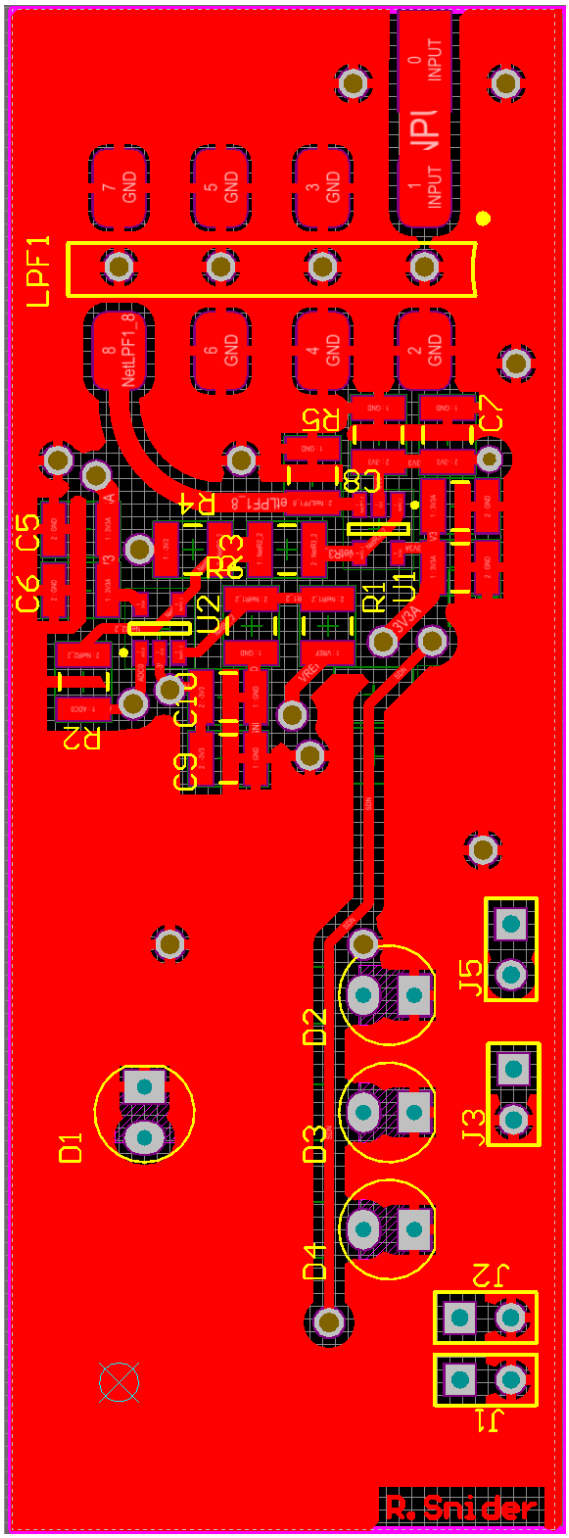


A.2. ADC extension board

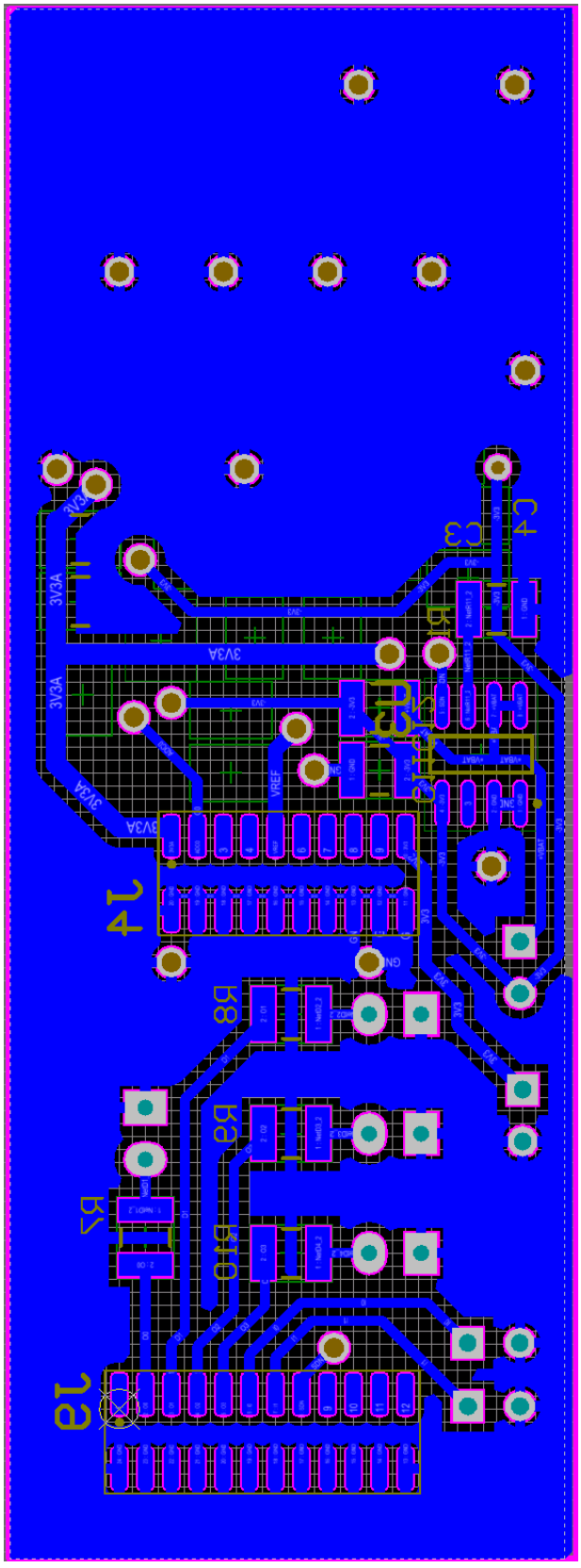
A.2.1. Schematic



A.2.2. PCB, top layer



A.2.3. PCB, bottom layer



Appendix B
Firmware, program codes

B.1. HSADC initialization

B1.1. Library “inits.h” with 1 descriptor

```
Description: This library aims to initialize all the used peripherals for data
acquisition.

Timers_Init: configure the Timer peripheral (Used in tests, useful)
GPIO_init: space for initiate the digital pins that will be used.
WDT_init: watch dog timer, for future lines.
HSADC_init: it configures the HSADC with one descriptor. When the configured
sample is taken, the microcontroller gets into converter’s ISR.

//////////////////
/* Rodrigo Snider Fiñana */
#ifndef INITS_H_
#define INITS_H_

#include "board.h"
#include <adc_func.h>

#define PORT(n) (n)
#define PIN(n) (n)

#define HSADC_CLOCK_RATE 80000000 //80MSps
#define WDT_INTERVAL 0xFFFFFF //0xFF - min 0xFFFF - max


//Functions Declarations
void Timers_Init(int);
void GPIO_init(void);
void WDT_init(void);
void HSADC_init(void);
////////Functions///////////////

//Timer0 initialization function
void Timers_Init(int freqmatch0){
      uint32_t timerFreq;

      Chip_TIMER_Init(LPC_TIMER0); //Timer 0 Init
      Chip_RGU_TriggerReset(RGU_TIMER0_RST);       //Reset and clock
                                                   settings
      while (Chip_RGU_InReset(RGU_TIMER0_RST)) {}
      timerFreq = Chip_Clock_GetRate(CLK_MX_TIMER0);
      //Timer0 Matching MR0 settings (TIMER1_HZ)
      Chip_TIMER_Reset(LPC_TIMER0);
      Chip_TIMER_MatchEnableInt(LPC_TIMER0, 0);
      Chip_TIMER_SetMatch(LPC_TIMER0, 0, (timerFreq / freqmatch0));
      Chip_TIMER_ResetOnMatchEnable(LPC_TIMER0, 0);
      Chip_TIMER_Enable(LPC_TIMER0);
      //IRQ enables
      NVIC_EnableIRQ(TIMER0_IRQn);
      NVIC_ClearPendingIRQ(TIMER0_IRQn);
}
//Init of GPIO pins
void GPIO_init(void){
```

```
	Chip_GPIO_SetPinDIROutput(LPC_GPIO_PORT, PORT(1), PIN(0));
	Chip_GPIO_SetPinDIROutput(LPC_GPIO_PORT, PORT(1), PIN(1));
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(0), false);
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(1), false);
}
/*Init WatchDog Timer reset mode*/
void WDT_init(void){
	Chip_WWDT_Init(LPC_WWDT);
	Chip_WWDT_SetTimeOut(LPC_WWDT, WDT_INTERVAL);
	Chip_WWDT_SetOption(LPC_WWDT, WWDT_WDMOD_WDRESET | WWDT_WDMOD_WDEN);
	Chip_WWDT_Feed(LPC_WWDT);
}
void HSADC_init(void){
	//Frequency settings, ADC enable
	//setupClock(HSADC_CLOCK_RATE);
	setadcCLK();
	Chip_HSADC_Init(LPC_ADCHS);
		//Config FIFO. (umbral de 15 muestras, empaquetadas de 2 en 2 en
	un word (4+12 + 4+12)
	Chip_HSADC_SetupFIFO(LPC_ADCHS, 15, true);
		//Trigger por software para pruebas (modificar para permitir trg
	externo)
	Chip_HSADC_ConfigureTrigger(LPC_ADCHS, HSADC_CONFIG_TRIGGER_SW,

	HSADC_CONFIG_TRIGGER_RISEEXT, HSADC_CONFIG_TRIGGER_NOEXTSYNC,
	HSADC_CHANNEL_ID_EN_ADD, 0x90);

	Chip_HSADC_SetACDCBias(LPC_ADCHS, 0, HSADC_CHANNEL_DCBIAS,
	HSADC_CHANNEL_NODCBIAS);
	//Se definen lindares de nivel para los registros A y B (provocan
	interrupciones) (no uso)
	Chip_HSADC_SetThrLowValue(LPC_ADCHS, 0, ((HSADC_MAX_SAMPLEVAL*1)/10));
	Chip_HSADC_SetThrHighValue(LPC_ADCHS, 0,((HSADC_MAX_SAMPLEVAL*9)/10));
	Chip_HSADC_SetThrLowValue(LPC_ADCHS, 1, ((HSADC_MAX_SAMPLEVAL*4)/10));
	Chip_HSADC_SetThrHighValue(LPC_ADCHS, 1,((HSADC_MAX_SAMPLEVAL*6)/10));
	//Set Power dependiendo de la Frecuencia y enable del ADC
	Chip_HSADC_SetPowerSpeed(LPC_ADCHS, false);
	Chip_HSADC_EnablePower(LPC_ADCHS);
	//Descriptor settings. (por ahora dos descriptores)

	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 0, (HSADC_DESC_CH(0) |
	HSADC_DESC_HALT |
	//HSADC_DESC_HALT |
	HSADC_DESC_MATCH(159) |   //Srate = (80/(delay+1))
	HSADC_DESC_RESET_TIMER));

	Chip_HSADC_EnableInts(LPC_ADCHS, 0, (HSADC_INT0_DSCR_DONE |
	HSADC_INT0_DSCR_ERROR));

	//Enable ADC interrupt y update de descriptors.
	//NVIC_EnableIRQ(ADCHS_IRQn);
	Chip_HSADC_UpdateDescTable(LPC_ADCHS, 0);
	Chip_HSADC_UpdateDescTable(LPC_ADCHS, 1);
   }

#endif /* INITS_H_ */
```

B1.2. Library “inits.h” with 16 descriptors

Description: This library takes the same structure than B1.1 but now it configures ADC with 16 description operation. After acquire the 16 samples, the microcontroller enters in ISR.

```
//////////////////
/* Rodrigo Snider Fiñana */
#ifndef INITS_H_
#define INITS_H_


#include "board.h"
#include <adc_func.h>
#include <cdc_vcom.h>
#include <USB_func.h>
#define a 1
#define PORT(n) (n)
#define PIN(n) (n)

#define HSADC_CLOCK_RATE 80000000 //80MSps
#define WDT_INTERVAL 0xFFFFFF //0xFF - minimo 0xFFFF - maximo


//Functions Declarations
void Timers_Init(int);
void GPIO_init(void);
void WDT_init(void);
void HSADC_init(void);
////////Functions///////////////

//Timer0 initialization function
void Timers_Init(int freqmatch0){
	uint32_t timerFreq;

	Chip_TIMER_Init(LPC_TIMER0); //Timer 0 Init
	Chip_RGU_TriggerReset(RGU_TIMER0_RST);	//Reset and clock
                                            settings
	while (Chip_RGU_InReset(RGU_TIMER0_RST)) {}
	timerFreq = Chip_Clock_GetRate(CLK_MX_TIMER0);
	//Timer0 Matching MR0 settings (TIMER1_HZ)
	Chip_TIMER_Reset(LPC_TIMER0);
	Chip_TIMER_MatchEnableInt(LPC_TIMER0, 0);
	Chip_TIMER_SetMatch(LPC_TIMER0, 0, (timerFreq / freqmatch0));
	Chip_TIMER_ResetOnMatchEnable(LPC_TIMER0, 0);
	Chip_TIMER_Enable(LPC_TIMER0);
	//IRQ enables
	NVIC_EnableIRQ(TIMER0_IRQn);
	NVIC_ClearPendingIRQ(TIMER0_IRQn);
}
//Init of GPIO pins
void GPIO_init(void){
	Chip_GPIO_SetPinDIROutput(LPC_GPIO_PORT, PORT(1), PIN(0));
	Chip_GPIO_SetPinDIROutput(LPC_GPIO_PORT, PORT(1), PIN(1));
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(0), false);
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(1), false);
```

```
}
/*Inicio del WatchDog Timer en modo reset*/
void WDT_init(void){
	Chip_WWDT_Init(LPC_WWDT);
	Chip_WWDT_SetTimeOut(LPC_WWDT, WDT_INTERVAL);
	Chip_WWDT_SetOption(LPC_WWDT, WWDT_WDMOD_WDRESET | WWDT_WDMOD_WDEN);
	Chip_WWDT_Feed(LPC_WWDT);
}
void HSADC_init(void){
	//Setting de frecuencias y enable del ADC
	//setupClock(HSADC_CLOCK_RATE);
	setadcCLK();
	Chip_HSADC_Init(LPC_ADCHS);
		//Config FIFO. (umbral de 15 muestras, empaquetadas de 2 en 2 en
	un word (4+12 + 4+12)
	Chip_HSADC_SetupFIFO(LPC_ADCHS, 15, true);
		//Trigger por software para pruebas (modificar para permitir trg
	externo)
	Chip_HSADC_ConfigureTrigger(LPC_ADCHS, HSADC_CONFIG_TRIGGER_SW,

	HSADC_CONFIG_TRIGGER_RISEEXT, HSADC_CONFIG_TRIGGER_NOEXTSYNC,
	HSADC_CHANNEL_ID_EN_ADD, 0x90);

	Chip_HSADC_SetACDCBias(LPC_ADCHS, 0, HSADC_CHANNEL_DCBIAS,
	HSADC_CHANNEL_NODCBIAS);
	//Se definen lindares de nivel para los registros A y B (provocan
	interrupciones) (no uso)
	Chip_HSADC_SetThrLowValue(LPC_ADCHS, 0,((HSADC_MAX_SAMPLEVAL*1)/10));
	Chip_HSADC_SetThrHighValue(LPC_ADCHS, 0,((HSADC_MAX_SAMPLEVAL*9)/10));
	Chip_HSADC_SetThrLowValue(LPC_ADCHS, 1,((HSADC_MAX_SAMPLEVAL*4)/10));
	Chip_HSADC_SetThrHighValue(LPC_ADCHS, 1,((HSADC_MAX_SAMPLEVAL*6)/10));
	//Set Power dependiendo de la Frecuencia y enable del ADC
	Chip_HSADC_SetPowerSpeed(LPC_ADCHS, false);
	Chip_HSADC_EnablePower(LPC_ADCHS);
	//Descriptor settings. (por ahora dos descriptores)

	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 0, (HSADC_DESC_CH(0) |
	HSADC_DESC_BRANCH_NEXT

	HSADC_DESC_MATCH(1) |

	HSADC_DESC_INT | HSADC_DESC_RESET_TIMER));

	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 0, (HSADC_DESC_CH(0) |

	HSADC_DESC_BRANCH_NEXT |

	HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 1, (HSADC_DESC_CH(0) |

	HSADC_DESC_BRANCH_NEXT |

	HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 2, (HSADC_DESC_CH(0) |

	HSADC_DESC_BRANCH_NEXT |
```

```
HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 3, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 4, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 5, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 6, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 7, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_SWAP |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 0, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 1, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 2, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 3, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 4, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 5, (HSADC_DESC_CH(0) |

HSADC_DESC_BRANCH_NEXT |

HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
```

```
	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 6, (HSADC_DESC_CH(0) |

	HSADC_DESC_BRANCH_NEXT |

	HSADC_DESC_MATCH(1) | HSADC_DESC_RESET_TIMER));
	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 1, 7, (HSADC_DESC_CH(0) |

	HSADC_DESC_HALT | HSADC_DESC_MATCH(1) |

	HSADC_DESC_INT | HSADC_DESC_RESET_TIMER));


	Chip_HSADC_EnableInts(LPC_ADCHS, 0, (HSADC_INT0_DSCR_DONE |

	HSADC_INT0_FIFO_FULL | HSADC_INT0_DSCR_ERROR));
	//Enable ADC interrupt y update de descriptors.
	NVIC_EnableIRQ(ADCHS_IRQn);
	Chip_HSADC_UpdateDescTable(LPC_ADCHS, 0);
	Chip_HSADC_UpdateDescTable(LPC_ADCHS, 1);
}



#endif /* INITS_H_ */
```

B1.3. Library “inits.h” with 1 cyclic descriptor and FIFO interruption

Description: This library takes the same structure than B1.1 but now it configures ADC with 1 descriptor and FIFO interruption. After acquire the 32 samples that FIFO supports, the microcontroller enters in ISR.

```
//////////////////
/* Rodrigo Snider Fiñana */
#ifndef INITS_H_
#define INITS_H_


#include "board.h"
#include <adc_func.h>

#define PORT(n) (n)
#define PIN(n) (n)

#define HSADC_CLOCK_RATE 80000000 //80MSps
#define WDT_INTERVAL 0xFFFFFF //0xFF - minimo 0xFFFF - maximo


//Functions Declarations
void Timers_Init(int);
void GPIO_init(void);
void WDT_init(void);
void HSADC_init(void);
////////Functions///////////////

//Timer0 initialization function
void Timers_Init(int freqmatch0){
	uint32_t timerFreq;

	Chip_TIMER_Init(LPC_TIMER0); //Timer 0 Init
	Chip_RGU_TriggerReset(RGU_TIMER0_RST);	//Reset and clock
                                            settings
	while (Chip_RGU_InReset(RGU_TIMER0_RST)) {}
	timerFreq = Chip_Clock_GetRate(CLK_MX_TIMER0);
	//Timer0 Matching MR0 settings (TIMER1_HZ)
	Chip_TIMER_Reset(LPC_TIMER0);
	Chip_TIMER_MatchEnableInt(LPC_TIMER0, 0);
	Chip_TIMER_SetMatch(LPC_TIMER0, 0, (timerFreq / freqmatch0));
	Chip_TIMER_ResetOnMatchEnable(LPC_TIMER0, 0);
	Chip_TIMER_Enable(LPC_TIMER0);
	//IRQ enables
	NVIC_EnableIRQ(TIMER0_IRQn);
	NVIC_ClearPendingIRQ(TIMER0_IRQn);
}
//Init of GPIO pins
void GPIO_init(void){
	Chip_GPIO_SetPinDIROutput(LPC_GPIO_PORT, PORT(1), PIN(0));
	Chip_GPIO_SetPinDIROutput(LPC_GPIO_PORT, PORT(1), PIN(1));
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(0), false);
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(1), false);
}
/*Init WatchDog Timer reset mode*/
```

```
void WDT_init(void){
	Chip_WWDT_Init(LPC_WWDT);
	Chip_WWDT_SetTimeOut(LPC_WWDT, WDT_INTERVAL);
	Chip_WWDT_SetOption(LPC_WWDT, WWDT_WDMOD_WDRESET | WWDT_WDMOD_WDEN);
	Chip_WWDT_Feed(LPC_WWDT);
}
void HSADC_init(void){
	//Setting de frecuencias y enable del ADC
	//setupClock(HSADC_CLOCK_RATE);
	setadcCLK();
	Chip_HSADC_Init(LPC_ADCHS);
		//Config FIFO. (umbral de 15 muestras, empaquetadas de 2 en 2 en
	un word (4+12 + 4+12)
	Chip_HSADC_SetupFIFO(LPC_ADCHS, 15, true);
		//Trigger por software para pruebas (modificar para permitir trg
	externo)
	Chip_HSADC_ConfigureTrigger(LPC_ADCHS, HSADC_CONFIG_TRIGGER_SW,

	HSADC_CONFIG_TRIGGER_RISEEXT, HSADC_CONFIG_TRIGGER_NOEXTSYNC,
	HSADC_CHANNEL_ID_EN_ADD, 0x90);

	Chip_HSADC_SetACDCBias(LPC_ADCHS, 0, HSADC_CHANNEL_DCBIAS,
	HSADC_CHANNEL_NODCBIAS);
	//Se definen lindares de nivel para los registros A y B (provocan
	interrupciones) (no uso)
	Chip_HSADC_SetThrLowValue(LPC_ADCHS, 0, ((HSADC_MAX_SAMPLEVAL*1)/10));
	Chip_HSADC_SetThrHighValue(LPC_ADCHS, 0,((HSADC_MAX_SAMPLEVAL*9)/10));
	Chip_HSADC_SetThrLowValue(LPC_ADCHS, 1, ((HSADC_MAX_SAMPLEVAL*4)/10));
	Chip_HSADC_SetThrHighValue(LPC_ADCHS, 1,((HSADC_MAX_SAMPLEVAL*6)/10));
	//Set Power dependiendo de la Frecuencia y enable del ADC
	Chip_HSADC_SetPowerSpeed(LPC_ADCHS, false);
	Chip_HSADC_EnablePower(LPC_ADCHS);
	//Descriptor settings. (por ahora dos descriptores)

	Chip_HSADC_SetupDescEntry(LPC_ADCHS, 0, 0, (HSADC_DESC_CH(0) |
	HSADC_DESC_BRANCH_FIRST |
	//HSADC_DESC_HALT |
	HSADC_DESC_MATCH(159) |   //Srate = (80/(delay+1))
	HSADC_DESC_RESET_TIMER));

	Chip_HSADC_EnableInts(LPC_ADCHS, 0, (HSADC_INT0_FIFO_FULL |
	HSADC_INT0_DSCR_ERROR));

	//Enable ADC interrupt y update de descriptors.
	//NVIC_EnableIRQ(ADCHS_IRQn);
	Chip_HSADC_UpdateDescTable(LPC_ADCHS, 0);
	Chip_HSADC_UpdateDescTable(LPC_ADCHS, 1);
   }


#endif /* INITS_H_ */
```

B1.4. HSADC interruption service routine (for B1.3. library)

Description: This function corresponds to the ADC's interruption routine. It is thrown when FIFO is full, then this functions saves the FIFO content to a RAM array (samplevector). When it arrives to 32000 words (64000 samples) it stop saving samples.

```
/* Rodrigo Snider Fiñana */
void ADCHS_IRQHandler(void)
{

	uint32_t sts, data, sample;
	uint32_t i;
	static bool on;

	//Reads first FIFO value
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(1), true);

	//Save to ram (samplevector array)
	if(numsamples<(32000-16)){
		data = Chip_HSADC_GetFIFO(LPC_ADCHS);
			while (!(data & HSADC_FIFO_EMPTY)) {//Enters if FIFO is
			    not empty.
			samplevector[numsamples] = data;
			// Next sample
			numsamples++;
			data = Chip_HSADC_GetFIFO(LPC_ADCHS);
		}
	}

	// Get ADC interrupt status on group 0 (TEST)
	sts = Chip_HSADC_GetIntStatus(LPC_ADCHS, 0) &
	Chip_HSADC_GetEnabledInts(LPC_ADCHS, 0);

	// Clear group 0 interrupt statuses
	Chip_HSADC_ClearIntStatus(LPC_ADCHS, 0, sts);
	fifofull=true;
	Chip_GPIO_SetPinState(LPC_GPIO_PORT, PORT(1), PIN(1), false);
}
```

B.2. DMA's library "GPDMA.h"

```
Description: This library aims to initialize GPDMA peripheral. Supports
different operation modes.

GPDMA_init: simple peripheral init.
GPDMA_M2Mtransfer: used in tests, memory to memory transfer. (test only)
GPDMA_ADC2Mtransfer: inits a transfer of indicated length, from ADC to a
                     given memory position. (test only)
GPDMA_initADV: peripheral init and transaction from ADC to destination (test
                     only)
GPDMA_capture: transaction from ADC to destination, using one list only,
                     maximum samples 4095.
ADC_DMA_capture: transaction using linked list, ulimited number of samples.
                     (limited by MCU RAM to allocate the vector).

//////////////////
/* Rodrigo Snider Fiñana */
#ifndef GPDMA_H_
#define GPDMA_H_
#define Ph(n) (n)

void GPDMA_init(void);
void GPDMA_M2Mtransfer(uint32_t *src, uint32_t *dst, uint32_t burst);
void GPDMA_ADC2Mtransfer(uint32_t *dst, uint32_t burst);
extern uint32_t makeCtrlWord(const GPDMA_CH_CFG_T *GPDMAChannelConfig,
uint32_t , uint32_t,      uint32_t, uint32_t);
void GPDMA_capture(uint32_t *dst, int samples);
void ADC_DMA_capture(uint32_t *dst, uint32_t samples);
void gpdmaTurnOff(void);


DMA_TransferDescriptor_t arrayLLI[20];

typedef struct{
      uint32_t SrcAddr;   /**< Source Address */
      uint32_t DstAddr;   /**< Destination address */
      uint32_t NextLLI;   /**< Next LLI address, otherwise set to '0' */
      uint32_t Control;   /**< GPDMA Control of this LLI */
}GPDMA_vector;

void GPDMA_init(void) {

      Chip_GPDMA_Init(LPC_GPDMA); //Clk settings

      NVIC_DisableIRQ(DMA_IRQn);
      NVIC_SetPriority(DMA_IRQn, 0x2);

      LPC_GPDMA->INTTCCLEAR = 0x1; //Clr int flags for CH0
      LPC_GPDMA->INTERRCLR = 0x1;  //Clr error flags for CH0

      /* Setup the DMAMUX */
      LPC_CREG->DMAMUX &= ~(0x3 << (Ph(7) * 2));
      LPC_CREG->DMAMUX |= 0x3 << (Ph(7) * 2);      /* peripheral 7 ADC
      Write(0x3) */
      LPC_CREG->DMAMUX &= ~(0x3 << (Ph(8) * 2));
```

```
	LPC_CREG->DMAMUX |= 0x3 << (Ph(8) * 2);		/* peripheral 8 ADC
	read(0x3) */

	LPC_GPDMA->CONFIG = 0x1;  //Enable bit for DMA
	while (!(LPC_GPDMA->CONFIG & 0x01)); //Wait until GPDMA is enabled

	NVIC_EnableIRQ(DMA_IRQn);
}

void GPDMA_M2Mtransfer(uint32_t *src, uint32_t *dst, uint32_t burst){
	uint8_t channel;
	//Free channel searching
	channel = Chip_GPDMA_GetFreeChannel(LPC_GPDMA, 8);  //Peripheral ID 8:
	ADCHS read
	//Channel connection FIFO to variable
	Chip_GPDMA_Transfer(LPC_GPDMA, channel, (uint32_t) src, (uint32_t)
	dst, GPDMA_TRANSFERTYPE_M2M_CONTROLLER_DMA, burst);

}

void GPDMA_ADC2Mtransfer(uint32_t *dst, uint32_t burst){
	uint8_t channel=0;
	//Free channel searching
	//channel = Chip_GPDMA_GetFreeChannel(LPC_GPDMA, 8);		//Peripheral
	ID 8: ADCHS read
	//Channel connection FIFO to variable
	Chip_GPDMA_Transfer(LPC_GPDMA, channel, 8, (uint32_t) dst,
	GPDMA_TRANSFERTYPE_P2M_CONTROLLER_DMA, burst);

	//Chip_GPDMA_Stop(LPC_GPDMA, channel);
}
void GPDMA_InitADV(uint32_t *dst){

	int size = 500;

	Chip_GPDMA_Init(LPC_GPDMA); //Clk settings

	NVIC_DisableIRQ(DMA_IRQn);
	NVIC_SetPriority(DMA_IRQn, 0x2);

	LPC_GPDMA->INTTCCLEAR = 0x1; //Clr int flags for CH0
	LPC_GPDMA->INTERRCLR = 0x1;  //Clr error flags for CH0

	/* Setup the DMAMUX */
	LPC_CREG->DMAMUX &= ~(0x3<<(Ph(7) * 2));
	LPC_CREG->DMAMUX |= 0x3<<(Ph(7) * 2);/* peripheral 7 ADC Write(0x3) */
	LPC_CREG->DMAMUX &= ~(0x3<<(Ph(8) * 2));
	LPC_CREG->DMAMUX |= 0x3<<(Ph(8) * 2);/* peripheral 8 ADC read(0x3) */

	LPC_GPDMA->CONFIG = 0x1; //Enable bit for DMA
	while ( !(LPC_GPDMA->CONFIG & 0x01) ); //Wait until GPDMA is enabled


	//GPDMA now is enabled, Next code is to configure channel 0 for ADC
	dump to memory.
	LPC_GPDMA->CH[0].SRCADDR = (uint32_t) &(LPC_ADCHS->FIFO_OUTPUT);
	//Source address, ADC FIFO
```

```
	LPC_GPDMA->CH[0].DESTADDR = (uint32_t) dst;
		//Destination address, variable
	LPC_GPDMA->CH[0].LLI = 0;

	//Configuration registers for channel 0

	LPC_GPDMA->CH[0].CONTROL =  (size << 0)  |          // Transfersize
(does not matter when flow control is handled by peripheral)
	 (0x3 << 12)  |          // Source Burst Size
	 (0x3 << 15)  |          // Destination Burst Size
	 (0x2 << 18)  |          // Source width // 32 bit width
	 (0x2 << 21)  |          // Destination width   // 32 bits
	 (0x1 << 24)  |          // Source AHB master 0 / 1
	 (0x1 << 25)  |          // Dest AHB master 0 / 1
	 (0x0 << 26)  |          // Source increment(LAST Sample)
	 (0x1 << 27)  |          // Destination increment
	 (0x1 << 31);             // Terminal count interrupt disabled

	LPC_GPDMA->CH[0].CONFIG |=  (0x1 << 0)        |     // Channel Enable
	 (0x8 << 1)   |          // Source Peripheral //Ph8 -> ADC read
	 (0x0 << 6)   |          // Destination Peripheral //Ph0 -> Memory
	 (0x2 << 11)  |          // Flow Control (P2M DMA control)
	 (0x1 << 14)  |          // Int error mask
	 (0x1 << 15);            // ITC - term count error mask

	NVIC_EnableIRQ(DMA_IRQn);
}

	//GPDMA must be initialized, capture a set of samples and saves it to
	the vector indicated.
void GPDMA_capture(uint32_t *dst, int samples){

	NVIC_DisableIRQ(DMA_IRQn);
	//configure channel 0 for ADC dump to memory.
	LPC_GPDMA->CH[0].SRCADDR = (uint32_t) &(LPC_ADCHS->FIFO_OUTPUT);
	//Source address, ADC FIFO
	LPC_GPDMA->CH[0].DESTADDR = (uint32_t) dst;
	//Destination address, variable
	LPC_GPDMA->CH[0].LLI = 0;

	//Configuration registers for channel 0

	LPC_GPDMA->CH[0].CONTROL =  (samples << 0)  |       // Transfersize
(does not matter when flow control is handled by peripheral)
	 (0x2 << 12)  |          // Source Burst Size
	 (0x2 << 15)  |          // Destination Burst Size
	 (0x2 << 18)  |          // Source width // 32 bit width
	 (0x2 << 21)  |          // Destination width   // 32 bits
	 (0x1 << 24)  |          // Source AHB master 0 / 1
	 (0x1 << 25)  |          // Dest AHB master 0 / 1
	 (0x0 << 26)  |          // Source increment(LAST Sample)
	 (0x1 << 27)  |          // Destination increment
	 (0x1 << 31);             // Terminal count interrupt disabled

	LPC_GPDMA->CH[0].CONFIG |=  (0x1 << 0)        |     // Channel Enable
	 (0x8 << 1)   |          // Source Peripheral //Ph8 -> ADC read
	 (0x0 << 6)   |          // Destination Peripheral //Ph0 -> Memory
```

```
        (0x2 << 11)  |          // Flow Control (P2M DMA control)
        (0x1 << 14)  |          // Int error mask
        (0x1 << 15);            // ITC - term count error mask

    NVIC_EnableIRQ(DMA_IRQn);
}

void ADC_DMA_capture(uint32_t *dst, uint32_t samples){

    uint32_t numLLI = 0, meansamples = 0, lastsamples = 0;
    uint32_t i = 0;
    if(samples%4000 == 0)numLLI = samples/4000;        //Divide the number
    of wanted samples by 3328.
    else numLLI = samples/4000 + 1;                          //numLLI is
    the number of linked list that we must use.

    meansamples = samples/numLLI;
                            //Number of samples for each LLI
    lastsamples = samples-meansamples*(numLLI-1);//Number of samples for
    the lastLLI

    //Filling the linked list array with all the parameters (ADDRESSES and
    CONTROL)
    for(i=0; i < (numLLI); i++){
        arrayLLI[i].dst = (uint32_t) dst + (uint32_t) (i*meansamples*4);
        arrayLLI[i].src = &(LPC_ADCHS->FIFO_OUTPUT[0]);
        arrayLLI[i].lli = &arrayLLI[i+1];                    //Direction
        for the next LLI array position.

        arrayLLI[i].ctrl = (meansamples << 0)  |             //
        Transfersize (does not matter when flow control is handled by
        peripheral)
         (0x2 << 12)       |          // Source Burst Size
         (0x2 << 15)       |          // Destination Burst Size
         (0x2 << 18)       |          // Source width // 32 bit width
         (0x2 << 21)       |          // Destination width   // 32 bits
         (0x1 << 24)       |          // Source AHB master 0 / 1
         (0x1 << 25)       |          // Dest AHB master 0 / 1
         (0x0 << 26)       |          // Source increment(LAST Sample)
         (0x1 << 27)       |          // Destination increment
         (0x0 << 31);             // Terminal count interrupt disabled
    }
    arrayLLI[numLLI-1].lli = 0x0;   //Indicate the last LLI by writing a 0

    //Last control register must contain the number of samples for the
    last chunk, and enable the count interrupt
    //to indicate that the overall sampling operation is finished.
    arrayLLI[numLLI-1].ctrl = (lastsamples << 0) |           //
    Transfersize (does not matter when flow control is handled by
    peripheral)
     (0x2 << 12)       |          // Source Burst Size
     (0x2 << 15)       |          // Destination Burst Size
     (0x2 << 18)       |          // Source width // 32 bit width
     (0x2 << 21)       |          // Destination width   // 32 bits
     (0x1 << 24)       |          // Source AHB master 0 / 1
     (0x1 << 25)       |          // Dest AHB master 0 / 1
     (0x0 << 26)       |          // Source increment(LAST Sample)
```

```
        (0x1 << 27)       |          // Destination increment
        (0x1 << 31);                   // Terminal count interrupt Enabled


      //Chip_GPDMA_SGTransfer(LPC_GPDMA, 0, arrayLLI,
      GPDMA_TRANSFERTYPE_P2M_CONTROLLER_DMA);

      //Filling first LLI array to the registers in order to initialize the
      system.
      LPC_GPDMA->CH[0].SRCADDR = arrayLLI[0].src;
      LPC_GPDMA->CH[0].DESTADDR = arrayLLI[0].dst;
      LPC_GPDMA->CH[0].CONTROL = arrayLLI[0].ctrl;
      LPC_GPDMA->CH[0].LLI = (uint32_t)(&arrayLLI[1]); // must be pointing
      to the second LLI as the first is used when initializing
      LPC_GPDMA->CH[0].CONFIG |=  (0x1 << 0)        |     // Channel Enable
        (0x8 << 1)   |          // Source Peripheral //Ph8 -> ADC read
        (0x0 << 6)   |          // Destination Peripheral //Ph0 -> Memory
        (0x2 << 11)  |          // Flow Control (P2M DMA control)
        (0x1 << 14)  |          // Int error mask
        (0x1 << 15);            // ITC - term count error mask

}
void ADC_DMA_captureUSB(uint32_t *dst0, uint32_t *dst1, uint32_t *dst2,
uint32_t *dst3, uint32_t samples){

      uint32_t numLLI = 0, meansamples = 0, lastsamples = 0;
      uint32_t i = 0;


      //Filling the linked list array with all the parameters (ADDRESSES and
      CONTROL)
      arrayLLI[0].dst = (uint32_t) dst0;
      arrayLLI[1].dst = (uint32_t) dst1;
      arrayLLI[2].dst = (uint32_t) dst2;
      arrayLLI[3].dst = (uint32_t) dst3;

      for(i=0; i < (4); i++){
            arrayLLI[i].src = &(LPC_ADCHS->FIFO_OUTPUT[0]);
            arrayLLI[i].lli = &arrayLLI[i+1];                  //Direction
            for the next LLI array position.

            arrayLLI[i].ctrl = (samples << 0)      |            //
            Transfersize (does not matter when flow control is handled by
            peripheral)
             (0x2 << 12)       |          // Source Burst Size
             (0x2 << 15)       |          // Destination Burst Size
             (0x2 << 18)       |          // Source width // 32 bit width
             (0x2 << 21)       |          // Destination width   // 32 bits
             (0x1 << 24)       |          // Source AHB master 0 / 1
             (0x1 << 25)       |          // Dest AHB master 0 / 1
             (0x0 << 26)       |          // Source increment(LAST Sample)
             (0x1 << 27)       |          // Destination increment
             (0x1 << 31);             // Terminal count interrupt disabled
      }
      arrayLLI[3].lli = &arrayLLI[0]; //Indicate the last LLI returns to the
      first.
```

```
	//Chip_GPDMA_SGTransfer(LPC_GPDMA, 0, arrayLLI,
	GPDMA_TRANSFERTYPE_P2M_CONTROLLER_DMA);

	//Filling first LLI array to the registers in order to initialize the
	   system.
	LPC_GPDMA->CH[0].SRCADDR = arrayLLI[0].src;
	LPC_GPDMA->CH[0].DESTADDR = arrayLLI[0].dst;
	LPC_GPDMA->CH[0].CONTROL = arrayLLI[0].ctrl;
	LPC_GPDMA->CH[0].LLI = (uint32_t)(&arrayLLI[1]); // must be pointing
	   to the second LLI as the first is used when initializing
	LPC_GPDMA->CH[0].CONFIG |=  (0x1 << 0)        |     // Channel Enable
	 (0x8 << 1)   |          // Source Peripheral //Ph8 -> ADC read
	 (0x0 << 6)   |          // Destination Peripheral //Ph0 -> Memory
	 (0x2 << 11)  |          // Flow Control (P2M DMA control)
	 (0x1 << 14)  |          // Int error mask
	 (0x1 << 15);            // ITC - term count error mask

}

void gpdmaTurnOff(void){
	LPC_GPDMA->CH[0].LLI = 0;
	LPC_GPDMA->CH[0].CONFIG |=  (0x0 << 0)        |     // Channel Disable
	 (0x8 << 1)   |          // Source Peripheral //Ph8 -> ADC read
	 (0x0 << 6)   |          // Destination Peripheral //Ph0 -> Memory
	 (0x2 << 11)  |          // Flow Control (P2M DMA control)
	 (0x1 << 14)  |          // Int error mask
	 (0x1 << 15);            // ITC - term count error mask
}


//OPT: Create a LLI for USB only!



#endif /* GPDMA_H_ */
```

B.3. ADC direct to USB main code

Description: This is the main program code of the Direct to USB mode. The microcontroller initializes ADC, GPDMA and USB in bulk mode. The working operation is explained at section 5.3. This program uses Libusb for ARM cortex, NXP distribution.

Note: The DMA is initialized to point to 4 buffers, these buffers are used in USB communications and its transfer is triggered at DMA's interrupt routine. The main loop objective is to control USB state, a ASCII '1' forces the start of data dumping, and ASCII '2' stops the dumping process.

```
//////////////////
/* Rodrigo Snider Fiñana */

#if defined (__USE_LPCOPEN)
#if defined(NO_BOARD_LIB)
#include "chip.h"
#else
#include "stdio.h"
#include "board.h"
#include <inits.h>
#include <libusbdev.h>
#include <GPDMA.h>
#endif
#endif
#include <cr_section_macros.h>

#define FREQTIMER0M0 1000
#define PACKET_BUFFER_SIZE        1024//4096
#define LUSB_DATA_PENDING       _BIT(0)

static uint8_t g_rxBuff[PACKET_BUFFER_SIZE];
static uint8_t g_txBuff[8192]; //4096
static uint8_t g_txBuff1[8192];
static uint8_t g_txBuff2[8192];
static uint8_t g_txBuff3[8192];
//volatile uint32_t samplevector[32000];
volatile uint32_t numsamples = 0;
volatile bool unhandledReq = false;
volatile bool usbDumpConnected = false;
volatile uint8_t pendingChunks = 0;
volatile uint8_t count = 0;

uint32_t sampleados = 0;
void samplingADC(void);
void usbTransferControl(void);
int rateTable[6] = {100,250,400,500,1000,2000,4000};//Srate = (80/(delay+1))

int main(void) {

      //Read clock settings and update SystemCoreClock variable
      SystemCoreClockUpdate();

      //Init section (Board, gpio, Timers, ADC, WDT)
```

```
	Board_Init();
	GPIO_init();
	GPDMA_init();
	libusbdev_init(USB_STACK_MEM_BASE, USB_STACK_MEM_SIZE);

	Board_LED_Set(0, false);

	LPC_GPDMA->SYNC = 1;	//Sync GPDMA enabled.
	while (1) {
	//	Chip_WWDT_Feed(LPC_WWDT); //Reset WDT

		usbTransferControl();	//Handles PC-board events.

	}
	return 0; //Fin del programa
}

void usbTransferControl() {
	//Handles every PC->Board interaction events.
	//Optimized for High Speed
	int i;

	if (libusbdev_Connected()) {
		if (libusbdev_QueueReadDone() != -1) {//Enter here when
		receiving data.
			libusbdev_QueueReadReq(g_rxBuff, PACKET_BUFFER_SIZE);
			//Process the data request, save the received data at
			g_rxBuff.

			if (g_rxBuff[0] == '0'); (Unimplemented in this version)
			if (g_rxBuff[0] == '1') {
				HSADC_init((int)rateTable[g_rxBuff[1]-'0']);
				samplingADC();
				usbDumpConnected = true; //When receiving '1' enter
				in Dump-Data mode
				Board_LED_Set(0,true);
				//GPDMA_capture(&g_txBuff[0], 128);
				//Starts DMA capture
				ADC_DMA_captureUSB(&g_txBuff[0], &g_txBuff1[0],
				&g_txBuff2[0], &g_txBuff3[0], 2048); //PCK Bytes/4
			}
			if (g_rxBuff[0] == '2') {
				usbDumpConnected = false; //When receiving '2'
				exits Dump-Data mode
				//Clear unsent buffers
				Board_LED_Set(0,false);
				gpdmaTurnOff();
				for (i = 0; i < 8192; i++) {
					g_txBuff[i] = 0;
					g_txBuff1[i] = 0;
					g_txBuff2[i] = 0;
					g_txBuff3[i] = 0;
				}
			}

			g_rxBuff[0] = 0;	//Clear Buffer RX
		}
```

```
	}
}


void samplingADC(void) {
	LPC_ADCHS->DSCR_STS = ((0 << 1) | 0);  //Descriptor table 0
	LPC_ADCHS->TRIGGER = 1;                //SW trigger ADC
}

//////////////Interrupt service Handler Functions//////////////////////////
void TIMER0_IRQHandler(void)           //Not Used
{
	if (Chip_TIMER_MatchPending(LPC_TIMER0, 0)) {
		LPC_TIMER0->IR = TIMER_IR_CLR(0);
		LPC_ADCHS->TRIGGER = 1;
	}
}
void ADCHS_IRQHandler(void)                  //Not Used
{
	uint32_t sts;
	// Get ADC interrupt status on group 0 (TEST)
	sts = Chip_HSADC_GetIntStatus(LPC_ADCHS, 0)
			& Chip_HSADC_GetEnabledInts(LPC_ADCHS, 0);
	// Clear group 0 interrupt statuses
	Chip_HSADC_ClearIntStatus(LPC_ADCHS, 0, sts);
}

void DMA_IRQHandler(void) {
	uint32_t actualLLI;
	static bool on1, on2;

	if (usbDumpConnected == true) {              //If USB is in dump mode
		actualLLI = LPC_GPDMA->CH[0].LLI;      //Look at LLI in order to
		know what is the previous full USB buffer and send to PC
		if (actualLLI == (uint32_t) &arrayLLI[0]) {
			while (libusbdev_QueueSendDone() != 0);
			while (libusbdev_QueueSendReq(g_txBuff2, 8192) != 0);
		}
		if (actualLLI == (uint32_t) &arrayLLI[1]) {
			while (libusbdev_QueueSendDone() != 0);
			while (libusbdev_QueueSendReq(g_txBuff3, 8192) != 0);
		}
		if (actualLLI == (uint32_t) &arrayLLI[2]) {
			while (libusbdev_QueueSendDone() != 0);
			while (libusbdev_QueueSendReq(g_txBuff, 8192) != 0);
		}
		if (actualLLI == (uint32_t) &arrayLLI[3]) {
			while (libusbdev_QueueSendDone() != 0);
			while (libusbdev_QueueSendReq(g_txBuff1, 8192) != 0);
		}
	}

	LPC_GPDMA->INTTCCLEAR = LPC_GPDMA->INTTCSTAT;
	//GPDMA_capture(&g_txBuff[0], 128);    //Restart DMA operation
	//Board_LED_Set(0,true);
}
```

Appendix C

Software, program codes

“ADC config tools”

C.1. Windows version

C.1.1. Main class

```
/*  Rodrigo Snider Fiñana
 *  Version: 0.5
 * */

using System;
using System.IO;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using ADC_Config_Tools_Libusb.Help;
using ADC_Config_Tools_Libusb.USB;
using System.Diagnostics;

namespace ADC_Config_Tools_Libusb
{
    public partial class MainForm : Form
    {
        /////////ATTRIBUTES////////
        private ReadWrite usbport = new ReadWrite();
        private ContinousRW usbContPort = new ContinousRW();
        private About AboutForm = new About();
        private bool matlabExisting = false;
        private MLApp.MLApp MatLab;
        public Thread _Tfilewriter;
        public Thread _Tbufwriter;
        private bool dumpState = false;

        //////////METHODS////////
        //Builder:
        public MainForm()
        {
            InitializeComponent();
            setTextTemp();

        }

        //Buttons:
        private void butConnect_Click(object sender, EventArgs e)
        {
            usbport.OpenUSB();
            usbport.initEp();

            if (usbport.state) textState.Text = "Connected to LPC.";
            else textState.Text = "Unable to connect.";
        }
        private void butRead_Click(object sender, EventArgs e)
        {    //NEXTSTEP: Look if path exists.
```

```
 //When a click is performed, PC sends to the device a '0' and
 the device sends the buffer content.
byte[] send = { (byte)'0' };
byte[] temporal;
byte[] rec = new byte[32000 * 4]; //32000 * 2 samples, 2 bytes
 per sample = 32000*2*2
string line;
bool noexists = false;
int i, j;

refreshTemp();  //Refresh the temporary file with the last
 configuration of Path and filename.

if (testConnection() == false) goto exit;   //If LPC is not
connected, don't perform read operation

 if (File.Exists(this.textPath.Text + @"\" +
        this.textFileName.Text + ".txt") == false) noexists =
        true;    //If the file don't exist, create it.
else     //If already exists, ask for overwrite.
{   //Display the overwrite messagebox
    string message = "File already exists. Overwrite?";
    string caption = "Warning";
    MessageBoxButtons buttons = MessageBoxButtons.YesNo;
    DialogResult result;

    result = MessageBox.Show(message, caption, buttons);

    if (result == System.Windows.Forms.DialogResult.Yes)
        noexists = true;   //Overwrite!
    else noexists = false;  //No overwrite!
}
if (noexists)    //If don't exist or the user has pressed to
 overwrite enters
{
    File.Delete(this.textPath.Text + @"\" +
        this.textFileName.Text + ".txt");       //Delete if
        exists.
    usbport.Write(send);
    // Send request for bulk sram data ('0').

    //////////////READ OPERATION//////////////
     //Testings with limited and known length
     for (i = 0; i < (32000 * 4 / 512); i++)
        //32000*4 bytes, formed by 512 B chunks.
    {
         temporal = usbport.Read512();           //read 512 bytes

         line = BitConverter.ToString(temporal).Replace("-",
        string.Empty);          //All temporal to string in Hex
        format.
         using (StreamWriter file = new
        StreamWriter(this.textPath.Text + @"\" +
        this.textFileName.Text + ".txt", true)) //Print chunk
        into the selected file.
         {
             file.WriteLine(line);
```

```
            }

        }
        messageFinish();    //Display success message.
    }

exit: ; //GOTO HERE IF LPC NOT CONNECTED


}
private void butBrowse_Click(object sender, EventArgs e)
{//Browse the folder to create the output file.
    folderBrowserDialog1.Description = "Select the desired folder
     output path:";
    //folderBrowserDialog1.
    folderBrowserDialog1.RootFolder =
     Environment.SpecialFolder.Desktop;
    if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
    {
        this.textPath.Text = folderBrowserDialog1.SelectedPath;
    }
}
private void butContRead_Click(object sender, EventArgs e)
{
    byte[] temporal;
    byte[] rec = new byte[32000 * 4]; //32000 * 2 samples, 2 bytes
     per sample = 32000*2*2
    string line;
    bool noexists = false;
    usbport.rateIndex = this.rateBox.SelectedIndex;
    int i, j;

    refreshTemp();  //Refresh the temporary file with the last
    configuration of Path and filename.

    if (testConnection() == false) goto exit;   //If LPC is not
      connected, don't perform read operation
    if (!dumpState)
    {


        if (File.Exists(this.textPath.Text + @"\" +
           this.textFileName.Text + ".txt") == false) noexists =
           true;    //If the file don't exist, create it.
        else     //If already exists, ask for overwrite.
        {   //Display the overwrite messagebox
            string message = "File already exists. Overwrite?";
            string caption = "Warning";
            MessageBoxButtons buttons = MessageBoxButtons.YesNo;
            DialogResult result;

            result = MessageBox.Show(message, caption, buttons);

            if (result == System.Windows.Forms.DialogResult.Yes)
               noexists = true;   //Overwrite!
            else noexists = false;  //No overwrite!
        }
```

```
            if (noexists)    //If don't exist or the user has pressed to
               overwrite enters
            {
                dumpState = true;
               File.Delete(this.textPath.Text + @"\" +
               this.textFileName.Text + ".txt");       //Delete if
               exists.
                this.butContRead.Text = "Abort continuous read";

                ////////////////////READ OPERATION////////////////////
                usbport.continuousWEn = true; //Enables cont operation

               usbport.setFilePath(this.textPath.Text + @"\" +
               this.textFileName.Text + ".txt");

                _Tfilewriter = new Thread(usbport.continuousWriteOPBin);
                _Tfilewriter.Priority = ThreadPriority.Highest;
               this._Tfilewriter.Start();  //Starts the cont operation
               thread (continuousWriteOP())
            }

        }
        else
        {
            dumpState = false;
            usbContPort.continuousWEn = false;
            usbport.continuousWEn = false; //Disables cont operation.
            this.butContRead.Text = "Begin continuous read operation";
        }
    exit: ; //GOTO HERE IF LPC NOT CONNECTED

    }

    //Tool strips:
    private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
    {
        AboutForm.ShowDialog();
    }
    private void arrangeAFileToolStripMenuItem_Click(object sender,
         EventArgs e)
    {
        string message = "Do you want to arrange this file? \n" +
         this.textPath.Text + @"\" + this.textFileName.Text + ".txt";

        string caption = "File";
        MessageBoxButtons buttons = MessageBoxButtons.YesNo;
        string line;
        char[] buffer = new char[4];
        char[] discard = new char[5];
        int a, i = 0;


        if (MessageBox.Show(message, caption, buttons) ==
           System.Windows.Forms.DialogResult.Yes)
        {
```

```
            if (File.Exists(this.textPath.Text + @"\" +
               this.textFileName.Text + "_ARR.txt"))

            File.Delete(this.textPath.Text + @"\" +
               this.textFileName.Text + "_ARR.txt");

            using (StreamWriter fileW = new
               StreamWriter(this.textPath.Text + @"\" +
               this.textFileName.Text + "_ARR.txt", true)) //Print into
               the selected file.
            {
               using (StreamReader fileR = new
               StreamReader(this.textPath.Text + @"\" +
               this.textFileName.Text + ".txt", true))
                {
                    while (fileR.Peek() != -1)
                    {
                        if (fileR.Peek() == '\r')
                        {
                            fileR.Read(discard, 0, 2);
                        }
                        a = fileR.Read(buffer, 0, 4);

                        fileW.WriteLine(buffer);
                    }
                }
            }
            //plot2Matlab();
        }
    }
    private void plotAFileToolStripMenuItem_Click(object sender,
         EventArgs e)
    {
        bool correct = false;
        DialogResult Result = DialogResult.None;
        while (correct == false && Result != DialogResult.Cancel)
        {
            openFileDialog1.FileName = null;
            openFileDialog1.InitialDirectory = this.textPath.Text;
            Result = openFileDialog1.ShowDialog();
            if (Result == DialogResult.OK)
            {
                if (openFileDialog1.FileName.Contains("_ARR.txt"))
                {
                    setPlot2Matlab(openFileDialog1.FileName);
                    correct = true;
                }
                else
                {
                    string message = "Please, select an arranged file.
                      (*_ARR.txt)";
                    string caption = "Error!";
                    MessageBoxButtons buttons = MessageBoxButtons.OK;
                    MessageBox.Show(message, caption, buttons);
                }
            }
        }
```

```
}
private void checkFailsToolStripMenuItem_Click(object sender,
     EventArgs e)
{

    string caption = "Errors";
    long lastcount = 0, numsample = 0;
    int numerror = 0;
    byte[] buffer = new byte[2];
    if (File.Exists(this.textPath.Text + @"\" +
     this.textFileName.Text + ".txt"))
    {
         FileStream FSr = new FileStream(this.textPath.Text + @"\" +
           this.textFileName.Text + ".txt", FileMode.Open,
           FileAccess.Read);
        using (StreamWriter fileW = new
        StreamWriter(this.textPath.Text + @"\" +
        this.textFileName.Text + "_err.txt", true)) //Print into the
        selected file.
        {
            using (BinaryReader fileR = new BinaryReader(FSr))
            {
                while (true)
                {
                    fileR.Read(buffer, 1, 1);
                    fileR.Read(buffer, 0, 1);
                    numsample++;
                    if (buffer[0] == 255 && buffer[1] == 255) break;
                        //If it's End Of File, break the loop.
                    else
                    {
                        if (lastcount >= buffer[0] * 256 + buffer[1]
                        + 300 || lastcount <= buffer[0] * 256 +
                        buffer[1] - 300)
                        {
                            numerror++;
                            fileW.WriteLine(numsample.ToString());
                        }
                        lastcount = buffer[0] * 256 + buffer[1];
                    }

                }
                string message = "Number of errors: " + numerror;
                MessageBoxButtons buttons = MessageBoxButtons.OK;
                MessageBox.Show(message, caption, buttons);
            }
        }
    }

}
private void arrangeSelectedDumpFileToolStripMenuItem_Click(object
     sender, EventArgs e)
{
    string message = "Do you want to arrange this dump file? \n" +
     this.textPath.Text + @"\" + this.textFileName.Text + ".txt";

    string caption = "File";
```

```
        MessageBoxButtons buttons = MessageBoxButtons.YesNo;
        string line;
        char[] buffer = new char[4];
        char[] discard = new char[5];
        int a, i = 0;


        if (MessageBox.Show(message, caption, buttons) ==
        System.Windows.Forms.DialogResult.Yes)
        {
            if (File.Exists(this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt"))
                File.Delete(this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt");
            using (StreamWriter fileW = new
                StreamWriter(this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt", true)) //Print into
                the selected file.
            {
                using (StreamReader fileR = new
                      StreamReader(this.textPath.Text + @"\" +
                      this.textFileName.Text + ".txt", true))
                {
                    while (fileR.Peek() != -1)
                    {
                        if (fileR.Peek() == '\r')
                        {
                            fileR.Read(discard, 0, 2);
                        }
                        a = fileR.Read(buffer, 2, 2);
                        a = fileR.Read(buffer, 0, 2);

                        fileW.WriteLine(buffer);
                    }
                }
            }
            //plot2Matlab();
        }

    }
    private void arrangeBinFileToolStripMenuItem_Click(object sender,
         EventArgs e)
    {
        string message = "Do you want to arrange this dump file? \n" +
         this.textPath.Text + @"\" + this.textFileName.Text + ".txt";

        string caption = "File";
        MessageBoxButtons buttons = MessageBoxButtons.YesNo;
        string line;
        byte[] buffer = new byte[2];
        int a, i = 0;


        if (MessageBox.Show(message, caption, buttons) ==
                System.Windows.Forms.DialogResult.Yes)
        {
```

```
            if (File.Exists(this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt"))
                File.Delete(this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt");

            FileStream FSr = new FileStream(this.textPath.Text + @"\" +
                this.textFileName.Text + ".txt", FileMode.Open,
                FileAccess.Read);

            using (StreamWriter fileW = new
                StreamWriter(this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt", true)) //Print into
                the selected file.
             {
                 using (BinaryReader fileR = new BinaryReader(FSr))
                 {
                     while (true)
                     {
                         fileR.Read(buffer, 1, 1);
                         fileR.Read(buffer, 0, 1);
                         if (buffer[0] == 255 && buffer[1] == 255) break;
                             //If it's End Of File, break the loop.

                         fileW.WriteLine(BitConverter.ToString(buffer).Re
                             place("-", string.Empty));
                     }
                 }
             }
             //plot2Matlab();
        }


    }

    //Auxiliar Methods:
    private void plot2Matlab()
    {
        if (matlabExisting == false)
        {
            MatLab = new MLApp.MLApp();
            matlabExisting = true;
        }

       MatLab.Execute("Mat = readtable('" + this.textPath.Text + @"\" +
                this.textFileName.Text + "_ARR.txt');");
        MatLab.Execute("Mat = hex2dec(char(table2array(Mat)));");
        MatLab.Execute("plot(Mat)");

    }
    private void setPlot2Matlab(string route)
    {
        if (matlabExisting == false)
        {
            MatLab = new MLApp.MLApp();
            matlabExisting = true;
        }
```

```
        MatLab.Execute(@"cd 'C:\Users\rodrigo.snider\Documents\Visual
                Studio 2013\Projects\ADC Config Tools Libusb\ADC Config
                Tools Libusb'");
        MatLab.Execute("PlotFunct('" + route + @"')");
    }
    private void refreshTemp()
    {
        string temPath =
                System.Environment.GetFolderPath(Environment.SpecialFolde
                r.CommonApplicationData);

        temPath = temPath + @"\configurations.tmp";
        if (File.Exists(temPath)) File.Delete(temPath);
        using (StreamWriter file = new StreamWriter(temPath, true))
                //Print into the selected file.
        {
            file.WriteLine(this.textPath.Text);
            file.WriteLine(this.textFileName.Text);
        }
    }
    private void setTextTemp()
    {
        string temPath =
                System.Environment.GetFolderPath(Environment.SpecialFolde
                r.CommonApplicationData);
        temPath = temPath + @"\configurations.tmp";
        if (File.Exists(temPath))
        {
            using (StreamReader file = new StreamReader(temPath))
            {
                this.textPath.Text = file.ReadLine();
                //this.textFileName.Text = file.ReadLine();
            }
        }
    }
    private bool testConnection()
    {
        if (this.textState.Text != "Connected to LPC.")
        {
            string message = "Connection is not performed, please open
                      USB port first";

            string caption = "Board not asociated";
            MessageBoxButtons buttons = MessageBoxButtons.OK;
            DialogResult result;

            // Displays the MessageBox.

            result = MessageBox.Show(message, caption, buttons);
            return false;
        }
        return true;
    }
    private void messageFinish()
    {
        string message = "Operation finished succesfully!";
```

```
            string caption = "Done!";
            MessageBoxButtons buttons = MessageBoxButtons.OK;
            DialogResult result;

            // Displays the MessageBox.

            result = MessageBox.Show(message, caption, buttons);
        }

    }
}
```

C.1.2. ReadWrite class

```
/*  Rodrigo Snider Fiñana
 *  Version: 0.5
 * */
using System;
using System.IO;
using LibUsbDotNet;
using LibUsbDotNet.Main;
using System.Threading;
using System.Diagnostics;

namespace ADC_Config_Tools_Libusb.USB
{
    class ReadWrite
    {

        private static UsbDevice MyUsbDevice;
        private static UsbDeviceFinder MyUsbFinder = new UsbDeviceFinder(0x1FC9,
             0x8A);
        private UsbEndpointReader reader;
        private UsbEndpointWriter writer;
        private string filePath;
        private byte[] bufferRx1 = new byte[512 * 10];
        private byte[] bufferRx2 = new byte[512 * 10];
        private bool buffer1 = false;
        private bool eventBuf = false;
        public bool continuousWEn = false;
        public bool state = false;
        public int rateIndex = 0;

        public ReadWrite()
        {
        }
        public void OpenUSB()
        {
            //LPC link2's VendorID and ProductID
            MyUsbFinder.Vid = 0x1FC9;
            MyUsbFinder.Pid = 0x8A;

            //Open a conection, if is already open throw an exception.
            MyUsbDevice = UsbDevice.OpenUsbDevice(MyUsbFinder);
            if (MyUsbDevice == null) Console.WriteLine("Could not connect");
            else
            {
                this.state = true;
                Console.WriteLine("Connected to Board!");
            }
            //Configuration for a whole device.
            IUsbDevice wholeUsbDevice = MyUsbDevice as IUsbDevice;
            if (!ReferenceEquals(wholeUsbDevice, null))
            {
                // Select config #1
                wholeUsbDevice.SetConfiguration(1);
                // Claim interface #0.
                wholeUsbDevice.ClaimInterface(0);
            }



        }
```

```
public void close()
{
    if (MyUsbDevice != null)
    {
        if (MyUsbDevice.IsOpen)
        {
                                // If this is a "whole" usb device
             (libusb-win32, linux libusb-1.0)
            // it exposes an IUsbDevice interface. If not (WinUSB) the
            // 'wholeUsbDevice' variable will be null indicating this is
            // an interface of a device; it does not require or support
            // configuration and interface selection.
            IUsbDevice wholeUsbDevice = MyUsbDevice as IUsbDevice;
            if (!ReferenceEquals(wholeUsbDevice, null))
            {
                // Release interface #0.
                wholeUsbDevice.ReleaseInterface(0);
            }

            MyUsbDevice.Close();
        }
        MyUsbDevice = null;

        // Free usb resources
        UsbDevice.Exit();

    }

}
public void initEp()
{
    // open read and write endpoint 1 R(0x81) W(0x01).
    this.reader = MyUsbDevice.OpenEndpointReader(ReadEndpointID.Ep01);
    this.writer = MyUsbDevice.OpenEndpointWriter(WriteEndpointID.Ep01);
}
public byte[] Read()
{
                  //This function reads from ENDPOINT 1 '4096' Bytes, in
     512 bytes packets (Internally by libusb), returns the readed bytes.
    ErrorCode ec = ErrorCode.None;
    byte[] readBuffer = new byte[8192]; //8192 -> ?//4096 -> 1MSps
    int bytesRead;

    //Read all the pending bytes
    try
    {
        reader.Read(readBuffer, 100, out bytesRead);        //Read with
            100ms of timeout.
        if (bytesRead == 0) throw new Exception("No more bytes!");
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine((ec != ErrorCode.None ? ec + ":" :
            String.Empty) + ex.Message);
    }

    return readBuffer;  //Returns bytes in vector form.
}
public byte[] Read512()
```

```
{
                //This function reads from ENDPOINT 1 '4096' Bytes, in
     512 bytes packets (Internally by libusb), returns the readed bytes.
    ErrorCode ec = ErrorCode.None;
    byte[] readBuffer = new byte[512]; //512
    int bytesRead;

    //Read all the pending bytes
    try
    {
        reader.Read(readBuffer, 100, out bytesRead);        //Read with
            100ms of timeout.
        if (bytesRead == 0) throw new Exception("No more bytes!");
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine((ec != ErrorCode.None ? ec + ":" :
            String.Empty) + ex.Message);
    }

    return readBuffer;  //Returns bytes in vector form.
}
public void Write(byte[] buffer)
{
    //Write 'buffer' to USB ENDPOINT 1.
    ErrorCode ec = ErrorCode.None;
    int byteswriten;

    //Write the specified buffer
    try
    {
        ec = writer.Write(buffer, 2000, out byteswriten); //Write with 2s
            of timeout.

    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine((ec != ErrorCode.None ? ec + ":" :
            String.Empty) + ex.Message);
    }
}
public void setFilePath(string filePath)
{
    this.filePath = filePath;
}

public void continuousWriteOPBin()
{
    //Function used by isolated thread, different from main.
                //This function creates a binary text file, and enters
     to a loop, where saves the USB received data to the text file.

    byte aux = Convert.ToByte(rateIndex + '0');
    byte[] send = { (byte)'1', aux };
    byte[] temporal;
    byte[] eof = { 255, 255 };
```

```
        FileStream FS = new FileStream(filePath, FileMode.CreateNew);
        //File handler, creates a file if the especified file don't exists.

        Write(send);  //Write to USB '1' + value corresponding to the
        selected data rate. Indicates the MCU to beginning tx data.

        using (Process p = Process.GetCurrentProcess()) p.PriorityClass = P
                       rocessPriorityClass.RealTime;    //Set maximum
                priority to this thread

        using (BinaryWriter file = new BinaryWriter(FS))
        {
            while (this.continuousWEn)  //Loop until connection is closed by
                the main thread.
            {

                temporal = Read();      //Reads USB data
                file.Write(temporal);   //Save in binary form to the file.
            }
            file.Write(eof);    //Write 0xFF,0xFF to the file (End of file).
            file.Close();       //Close and save the file.
        }
        send[0] = (byte)'2';

        Write(send);           // Send ('2') to MCU to indicate that the
          connection is closed.
    }

    //Unused methods (only for tests)//
    public void continuousWriteOP()
    {
        byte[] send = { (byte)'1' };
        byte[] temporal;
        string line;

        Write(send);
        // Send request for dump directly data ('1').
        using (StreamWriter file = new StreamWriter(filePath, true)) //Print
        chunk into the selected file.
        {
            while (this.continuousWEn)
            {
                temporal = Read();
                line = BitConverter.ToString(temporal).Replace("-",
                       string.Empty);
                file.WriteLine(line);
            }
        }
        send[0] = (byte)'2';

        Write(send);
        // Send cancel for dump directly data ('2').
    }
    public void contPutBuffer()
    {
        Int32 pos = 0;
        byte[] bufaux;
        while (this.continuousWEn)
        {
            for (pos = 0; pos < 10; pos++)
```

```
                {
                    bufaux = Read();
                    bufaux.CopyTo(bufferRx1, (pos * 512));
                }
                buffer1 = true;
                eventBuf = true;
                for (pos = 0; pos < 10; pos++)
                {
                    bufaux = Read();
                    bufaux.CopyTo(bufferRx2, (pos * 512));
                }
                buffer1 = false;
                eventBuf = true;
            }
        }
        public void contPutFile()
        {
            byte[] send = { (byte)'1' };
            byte[] temporal;
            byte[] eof = { 255, 255 };
            FileStream FS = new FileStream(filePath, FileMode.CreateNew);

            Write(send);
            // Send request for dump directly data ('1').
            using (BinaryWriter file = new BinaryWriter(FS)) //Print chunk into
            the selected file.
            {
                while (this.continuousWEn)
                {
                    if (eventBuf)
                    {
                        eventBuf = false;
                        if (buffer1 == true)
                        {
                            file.Write(bufferRx1);
                        }
                        else
                        {
                            file.Write(bufferRx2);
                        }
                    }
                    temporal = Read();
                    file.Write(temporal);
                    Thread.Sleep(10);
                }
                file.Write(eof);
                file.Close();
            }
            send[0] = (byte)'2';
            Write(send);

        }
        /// <summary>
        /// ////////////////////////////////
        /// </summary>
        ///

    }

}
```

C.2. Raspberry Pi version

```
//Rodrigo Snider Fiñana; v0.5a for linux

#include "stdio.h"
#include <libusb-1.0/libusb.h>
#include "string.h"
//#include "ncurses.h"
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>

//Vendor ID and Product ID indicated in USB descriptors (Change if needed).
#define ProID 0x8A
#define VenID 0x1FC9
//////////////////////////////////////////////////////

//Text Colours definition
#define GRN "\x1B[32m"
#define RED  "\x1B[31m"
#define NRM  "\x1B[0m"
//////////////////////////

//Function declarations
int menu();
int chkErrors(char*);
//////////////////////



//Main function (argv not used)
int main(int argc, char *argv[]){

	//////////Local variables section////////
	int result, cnt = 0, srate;
	FILE *ptr_file;
	libusb_device_handle *dev_handle;
	libusb_context *ctx;
	int aux, i, e;
	char name[50];
	unsigned char data[8192], datao[2];
	int actuallengths = 0, actuallengthr = 0;
	//////////////////////////////////////////



	//Init sequence
	libusb_init(&ctx);
	aux = setpriority(PRIO_PROCESS, 0, -20);
	printf("%d", aux);
	///////////////



	//Main loop
	while (1){
		result = menu(cnt);		//Prints menu, returns the selection.
```

```
if (result == 1){
                                        //Connect usb OPTION.
        printf("Connecting...");
        dev_handle = libusb_open_device_with_vid_pid(ctx, VenID,
                ProID);             //Open specified device
        if (dev_handle != NULL){
                //libusb_set_configuration(dev_handle, 1);
                printf("%sSuccessfully connected.\n", GRN);
                                //Enters here if device is connected.
                cnt = 1;//Indicates that device is connected


        }
        else {
                printf("%sConnection fail.", RED);
                cnt = 0;
        }
        printf("%s\n", NRM);
}
else if (result == 2){
                                        //Dump data to file OPTION.
        if (cnt == 0) printf("Board not connected, try to connect
                first.\n");//Ends if device is not connected.
        else{
                printf("Select a name for the file: ");
                scanf("%s", name);
                //Scan the filename
                strcat(name, ".bin");
                //ADDs .bin extension (can be anyone)
                ptr_file = fopen(name, "w+b");                  //Open
                the specified file R&W (creates if not exists)

                do{    //Sample rate selection, loop to avoid
                incorrect values.
                        printf("%sSelect sample rate:\n", RED);
                        printf("(0) 100KSps ; (1) 250KSps ; (2) 400
                                KSps ; (3) 500KSps ; (4) 1 MSps ; (5) 2
                                MSps ; (6) 4 MSps\n");
                        scanf("%d", &srate);
                        printf("%s", NRM);
                } while (srate != 0 && srate != 1 && srate != 2 &&
                srate != 3 && srate != 4 && srate != 5 && srate != 6);


                libusb_claim_interface(dev_handle, 0);   //Start a
                device interface
                printf("Reading...\n");
                printf("Press a key to end reading process...\n");

                //prepare to send 10: beginning of data dump process
                (INDICATES TO MCU START DUMP)
                data[0] = '1';
                data[1] = '0' + srate;;
                i = libusb_bulk_transfer(dev_handle, 0x1, data, 2,
                        &actuallengths, 10000);// write 10


                while (kbhit() == 0){      //USB reading loop, when
                key is pressed exits.
                        //for(i=0;i<1000000;i++);
```

```
                                    libusb_bulk_transfer(dev_handle, 0x81, data,
                                     4096, &actuallengthr, 0); //Read package
                                     fwrite(data, 1, 4096, ptr_file);
                                    //write package to the file
                                     //break;
                                }
                                //data[0] = 0xFF;
                                //data[1] = 0xFF;
                                //fwrite(data,2,1,ptr_file);

                                //prepare to send 2: finish data dump process
                                (INDICATES TO MCU STOP DUMP)
                                data[0] = '2';
                                actuallengths = 0;
                                i = libusb_bulk_transfer(dev_handle, 0x1, data, 1,
                                      &actuallengths, 1000);// write 2

                                //Write EOF in File
                                data[0] = 0xFF;
                                data[1] = 0xFF;
                                fwrite(data, 1, 2, ptr_file);
                                //write package to the file

                                //Close and save file
                                fclose(ptr_file);
                                printf("File was saved.\n");

                                //Error checking. ONLY FOR TESTS
                                printf("checking for errors...\n");
                                aux = chkErrors(name);
                                printf("errors: %d\n", aux);


                                //close libusb procedure
                                libusb_release_interface(dev_handle, 0);
                                //release device
                                libusb_close(dev_handle);
                                //close handler
                                libusb_exit(ctx);
                                //close libusb


                                break;
                                //finish program
                            }
                        }
                        else printf("Introduce a valid value.\n");
                        printf("\n");

                }
                return 0;

}

int menu(int state){ //Function that prints Menu, and returns the user selection.
        int result;
        printf("-----------MENU-----------\n");
        if (state == 0)printf("%s1. Connect USB.\n", RED);
        else printf("%s1. Connect USB.\n", GRN);
        printf("%s2. Dump data to file.\n", NRM);
```

```
	printf("\n");
	printf("Select an option: ");
	scanf("%d", &result);
	return result;
}

int kbhit(){		//Function that returns a value different from 0 if key is
	pressed.
	struct timeval tv;
	fd_set fds;
	tv.tv_sec = 0;
	tv.tv_usec = 0;
	FD_ZERO(&fds);
	FD_SET(STDIN_FILENO, &fds);
	select(STDIN_FILENO + 1, &fds, NULL, NULL, &tv);
	return FD_ISSET(0, &fds);
}


int chkErrors(char* filename){	//Function that checks errors (TESTS ONLY)
	FILE *ptr_file;
	unsigned char vector[2];
	int lastvalue;
	int error = 0, r;
	long int pos = 1;
	ptr_file = fopen(filename, "r+b");

	r = fread(vector, 1, 2, ptr_file);

	lastvalue = vector[1] * 0x100 + vector[0];
	while (vector[1] != 0xFF && vector[0] != 0xFF){
		if ((vector[1] * 0x100 + vector[0]) >= (lastvalue + 300)){
			error++;
			printf("%li \n", pos);
		}
		else if ((vector[1] * 0x100 + vector[0]) <= (lastvalue - 300)){
			error++;
			printf("%li \n", pos);
		}
		lastvalue = vector[1] * 0x100 + vector[0];
		r = fread(vector, 1, 2, ptr_file);
		pos++;
	}
	fclose(ptr_file);
	return error;
}
```

Appendix D
MATLAB scripts

D.1. Signal plot and error detection

```
clear
Mat1 = readtable('sen5kcont_ARR.txt');    %Take arranged file
Xpos = hex2dec(char(table2array(Mat1)));  %Convert to decimal
sps = 8e+6;

%%%%Algorithm for error detection

Errores = 0;
j=1;
tic
for i=1:(length(Mat)-1)
    if(Mat(i)>Mat(i+1))
        if(Mat(i)-Mat(i+1)>300)Errores=Errores+1;
            PosError(j)=i;
            j=j+1;
            end
    else
        if(Mat(i+1)-Mat(i)>300)Errores=Errores+1;
            PosError(j)=i;
            j=j+1;
        end
    end
end
toc

Errores
PosError
plot(Mat)
length(Mat)
```

D.2. Periodogram, SNR, SFDR and SNDR

```
clear
Mat1 = readtable('1ktri10Mreb_ARR.txt'); %Take arranged file
Xpos = hex2dec(char(table2array(Mat1))); %Convert to decimal

SF = 10000000;                              %sampling frequency
Soffset = 100;                             %Erase first 100 samples
Wlen = 61340;                             %W Length of 400k samples
Ypos = Xpos(Soffset:Wlen+Soffset-1);   %cut the signal, Sspan samples

Yvol= Ypos*0.8/4096.+0.1;                 %convert to voltage
%Signal processing:

Y = fft(Ypos,Wlen);                       %fourier transform
Pyy = Y.*conj(Y)/Wlen;                    %power spectral density
f = SF/Wlen*(0:(Wlen/2-1))/1000;      %define frequency axis for
plotting PSD

yaxis = Pyy(1:Wlen/2);                    %take half of Fourier transform
%plotting the PSD in logaritmic axis
figure(1)
plot(f,10*log10(yaxis))                    %logaritmic plot
title('Power Spectral Density')
xlabel('Frequency (KHz)')
ylabel('PSD log axis(dB)')

figure(2)
taxis = (0:1000/SF:1000*(Wlen-1)/SF);  %Time axis definition
plot(taxis, Ypos)                          %plot Signal
title('Acquired Signal')
ylabel('ADC levels')
xlabel('Time line(ms)')

sinad(Yvol,SF);   %compute SNDR
figure(3)
snr(Yvol,SF);     %compute SNR
figure(4);
sfdr(Yvol,SF);    %compute SFDR
```

D.3. Nonlinearity calculation

```
Mat1 = readtable('1ktri10Mreb_ARR.txt');
Xpos = hex2dec(char(table2array(Mat1)));

SF = 10000000;                              %sampling frequency
Soffset = 100;                             %Erase first 100 samples
Wlen = 61340;                             %W Length of 400k samples
Ypos = Xpos(Soffset:Wlen+Soffset-1);    %cut the signal, Sspan samples

figure(1)
plot(Ypos)
Ramp = Ypos(1077:9897);                   %Take only one ramp
P=polyfit((1:length(Ramp)),Ramp',1);      %approximation to a deg. 1
                                           polynom.
Aprox = P(1)*(1:length(Ramp))+P(2);       %create line

figure(2)
error = Ramp'-Aprox;                       %INL
plot(error);                               %plot INL
mean(error)                                %Mean error
```