UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Electronic Engineering Degree
Embedded Systems Specialization

Master's Thesis

Implementation of a Neonatal EEG Monitoring through Sonification on an Embedded Platform

Supervisors

Emanuel Popovici

Danilo De Marchi

Candidate

Marco Borzacchi

Italy, Turin
March 2020

Abstract

The aim of this thesis is to implement an embedded solution for portable device able to put into practice a biomedical research that has being developed in the Embedded Systems Group at University College Cork.
The idea is to realize an intuitive and persuasive solution for neonatal ElectroEncelaloGram monitoring systems used in healthcare facilities that enhances the classic approach by using Sonification and deep learning AI, providing information about neonatal brain health to all neonatal healthcare professionals, particularly those without EEG interpretation expertise.
Interpreting the EEG activity of a newborn, requires expert knowledge and experience and it is still hard to detect an anomaly, such as a seizure or abnormal EEG background activity, since the symptoms are often difficult to detect even for a specialist.
With the aid of an AI algorithm that has been already implemented, the percentage of detecting an abnormal brain functioning increases drastically and a contemporaneous Real Time Sonification processing transforms the complex EEG signal into a sound much easy to interpret, minimizing the possibilities to make an error.
The system has to be able to sample the EEG signals by using multiple channels, detect a possible anomaly, alert the medical staff, process the data in Real Time and stream video and audio on a device. The implementation presented in this thesis is mostly focused on Real Time signal processing and streaming, proposing a versatile and portable application that can be easily carried home and, in case of alert, can send data to a server in charge to process them and provide the streaming to the doctor that can decide if it is necessary to intervene immediately.
The equipment chosen for this purpose are:

- ADS1299 EEG (Texas Instrument): eight-channel, low-noise, 24-bit, simultaneous-sampling delta-sigma analog-to-digital converters (ADCs) with a built-in programmable gain amplifier (PGA), internal reference, and an onboard oscillator. The acquired samples are sent to a second platform through SPI.

- Rasberry Pi model 3: it is in charge to compute the received data through SPI and send them over a network to a Linux machine.

- Laptop (Linux machine): this machine acts as a server that captures the data, applies the Real Time Sonification algorithm and streams audio and video on a Python GUI.

It is necessary to underline that this implementation represents the first demo of the full research project, indeed, many choices have been made for simplifying the initial work, such as the use of Raspberry Pi, and further modifications are contemplated to optimize the application and provide a system that is the best trade off in terms of power, cost and performances. The concept idea and demo will help further optimisations for algorithms as well as for the entire embedded system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction, Motivations and Goal

The thesis proposal has been assigned after the publication of papers related to an innovative research project made in in University College Cork, Ireland (in the Embedded Systems Group and the Infant Centre). The project is called Neurobell and involves the development of a new EEG acquisition method capable to rapidly prevent seizures in newborns, by using Real Time signal processing, Sonificaton and AI algorithms. The final goal is to realize a portable, rapid and low-cost EEG monitoring system that merges the previous researches in a unique device.
Formerly the outset of this work of thesis, some ideas regarding the implementation of the final system had been developed and their outcomes are described in the following section. The involvement in this research, have been useful to evolve a new idea, crucial for the implementation of the final Demo that represents the main topic of this work of thesis.

1.1 Motivations

Neonatal brain injuries are a serious concern for clinicians and parents worldwide [1]. In particular, infants with seizures do not show any clinical manifestations.
The current method used to detect neonatal brain injuries states to continuously perform multi-channel EEG monitoring based on video visualization. Furthermore, interpretation of neonatal EEG requires a neurophysiologist with specific expertise in neonatal EEG that should work 24h per day to intervene in case of alarm. To fill the gap in the availability of appropriate expertise, clinical staff in the NICU are using a simpler form of EEG monitoring called amplitude integrated EEG, or aEEG [2]. However, many studies have demonstrated that aEEG has limited capabilities in detecting abnormal neonatal EEG activity. Consequently, researches are looking for an alternative method to overcome these limits. It has been showed that the human hearing input is better than the visual input with regard to assessing both the spatial and temporal evolution of the frequency characteristics of a signal [3]. Hearing is flexible and low-cost. It allows for faster processing than visual presentation and releases visual sense for other tasks [4] [5]. The actual EEG monitoring systems are very expensive and they can not be considered as portable. Starting from these main problems and considerations, an innovative idea have been elaborated and it will be discussed in the following paragraph.

1.2 Initial Concept

As discussed in the previous section, an innovative implementation of a EEG monitoring system is desired to detect and prevent seizures in newborns. The main goal is to realize a cheap and portable device that exploits new algorithms to overcome known issues present in the standard aEEG. The first idea introduced as starting point of this thesis was to implement a smartphone-based system in collaboration with a portable acquisition device that solves the issues of cost, portability and rapid assessment [6]. Android technology is widely available across the Globe. As of 2015, 37% of the population in developing nations own a smartphone [7]. At the beginning of this work, a back end Android Java application had already been implemented: an Android tablet (Samsung A6 tablet 2GB RAM, 8 cores @ 1.6GHz CPU, 7300mAh battery) receives the EEG signal via Bluetooth from the EEG acquisition board, and it is capable of real-time Sonification and visualisation of EEG on an Android device, as illustrated in Fig 1.1.

Figure 1.1: Neonatal Brain Stethoscope: EEG Sonification and Monitoring using Android Device.

However, all the front-end application, acquisition and Real Time, were not implemented yet but they were simulated through the extraction of processed samples stored in a text file. Consequently, the main goal of this work was to design a full demo able to put into practice an embedded systems implementation, and exploit the already existing application to realize a coexistent environment.
As a matter of facts, smartphones and tablets in general have a low power System on Chip built-in with a specific IP core able to easily perform signal processing, AI algorithms and parallel computations; therefore, these specifications meet the requirements to reach the desired performances.
However, after some initial considerations, it has been decided to add another computational board in between the smartphone/tablet and the EEG acquisition board in order to enable the edge computation (to deliver low latency nearer to the requests) and avoid unnecessary energy consumption on the smartphone/tablet side due by a continuous Real Time processing. The computational and the EEG boards communicate each other using SPI standard.

A Dragonboard 410c board was chosen for this purpose: this board uses a Qualcomm Snapdragon 410 processor and supports Android 5.1 Operating System. Con-

sequently, it has similar computational capabilities of a smartphone and the compatibility with Android, offers the possibility to integrate the already existing Java application in a larger environment.
The Dragonboard has been deeply studied during the first months of the work and it is well described in the chapter 3.2. The choice of using Android on this board resulted to be too difficult due to the absence of clear documentation and the complexity of building an Android Linux Kernel that integrates low level drivers to interface with external device using the SPI protocol.

The Android Linux Kernel needs to be compiled and built first, before being flashed into the chip: this means that in order to control and program a Snapdragon peripheral such as SPI, a full un-compiled high level operating system compatible with this board is required, and low level drivers need to be written first to be compatible with a specific and unique peripheral. In addition, the corresponding entry point needs to be added as well in order to interact with the device.

All these steps were really hard to be performed without internal Qualcomm documentation and the final choice to give up the Dragonboard was taken. It has been decided to proceed using a Raspberry Pi, much simpler to use and adapt to speed up the full demo implementation. It is necessary to underline that Raspberry Pi is not intended to replace the computational board in the end application: in fact, its hardware specifications do not meet the requirements such as low power and parallel computations but it can be considered useful for Demo purpose only.
During the Demo development a new idea has been elaborated: thinking about IOT(Internet of Things) architecture where the data processing is left to a big server side cloud, it has been thought to implement the Real Time algorithms on a single server machine in order to further decrement the whole power consumptions of the portable device. Furthermore, the server can receive data and stream EEG video and audio on a given IP, where doctors can access using their own credentials from whatever device (Laptop, Android or IOS smartphone, Tablet) removing also the problem of App compatibility and extending the application action range. The idea is better described in the following section.

1.3 New Concept

Moving forward with Raspberry Pi, it has been easier to set-up SPI communication between the Pi3 and the EEG acquisition board (ADS1299 EEG). The latter is a eight-channel, low-noise, 24-bit, simultaneous-sampling delta-sigma ($\Delta\Sigma$) analog-to-digital converters (ADCs) with a built-in programmable gain amplifier (PGA), internal reference, and an onboard oscillator [8]. The multiple programmable features, make the EEG acquisition board quite complex to program and the whole set-up has been described in chapter 4.1.
Once it has been ensured that the acquired EEGs were reliable, the Real Time algorithm implementation has begun. It is at this step that new concept came out: observing the difficulty to obtain Real Time behaviour on Pi3 due by hardware limitations, the idea of using a server type application has been elaborated. It is also true that starting idea was to use a quad-core Snapdragon that would have been more suitable for the treated computations. However, if on the one hand there was a computational deficiency, on the other there was a wake-up-call regarding the power

consumption that, in a real case scenario, would have drained the battery.
Thinking about IOT architecture, it has been possible to figure higher range application were the computational part is left to a server cloud that, at the same time, streams audio and video on an IP accessible through doctor credentials (Fig 1.2).

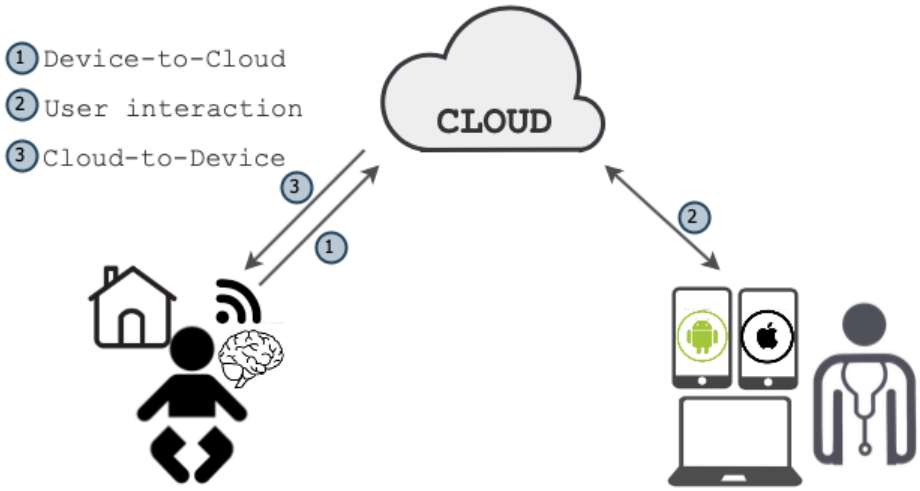

Figure 1.2: New application concept: EEG Sonification and Monitoring over the network through a server cloud.

In this case scenario, the portable device has to implement artificial intelligent algorithm only, and in case of danger, it sends a distress message over the network and it starts forwarding the acquired EEGs. Consequently, the Dragonboard may be substituted with a lower power microcontroller able to establish wifi, SPI communication and implement the AI. The main advantage of this configuration is that the application range is much more extended w.r.t. Bluetooth system and doctors can monitor the newborn brain activity directly from the hospital while the infant is at home with his family.
In addition, using this approach it would be possible to eliminate the barriers due to OS compatibility: in fact, the network back-end application may be develop to be compatible with all possible devices such as Laptop, Android or IOS phones, tablets and so on.
In this thesis, a laptop with GNU/Linux OS has been chosen to act as a server and also the back-end application (audio and video stream) has been implemented on the laptop as well. Consequently, the final demo flow shows the ADS1299 EEG board sampling the EEG (after being programmed by the Pi using SPI) and sending periodically packets of data to the Raspberry through SPI; the latter forwards the EEG to the laptop over wifi that starts processing them in Real Time and streams video and audio on a GUI.
The implementation of the full demo has been described in section 4.3.
This flow represents only a concept of the final application and the implementation is still open to updates and refinements.

Chapter 2

Research Project

The research projects, key point of this thesis, are treated in the following chapter. It makes a comparison between the current EEG Monitoring System and the idea developed by the researches, focusing on the biomedical instrumentation, effectiveness, ethical barriers, cost and power consumption.
Furthermore, algorithms behind the full application are discussed as well and their main characteristics and features are pointed out.

2.1 EEG Monitoring System

Clinical evaluation of electroencephalography (EEG) is crucial across neonatal, paediatric and adult patients presenting abnormal neurological behaviour. Previous publications have shown that using EEG monitoring to help the diagnosis of neonatal seizures remarkably increases the percentage of correct seizure detection w.r.t. diagnosing seizures based on clinical signs alone [9].
Main problem is related to the cost of instrumentation and the difficulty of finding expertises able to detect the seizures. Factors such as size, cost, preparation time and interpretation skills constrain the use of EEG monitoring in a range of scenarios [10]. Consequently, researchers are struggling to find new solutions able to overcome this limit. Developing new medical device that requires to place electronics on human subject (in particular newborns), must pass through several health and safety approvals and collides with ethical rules. This can drastically delay the process or, in the worst case scenario, blocking the research.

Therefore, the development of a method to quantitatively and accurately assess the performance of an EEG acquisition system and its components on clinical data without being dependent on human contributions is required [10].
Emulating the behaviour of original EEG signals requires high accuracy: beside the fact that EEGs have very small voltage amplitude (in a range of hundred micro volts), they are characterized by other influences such as noise and artifacts that need to be included in the model. Previous studies utilized sinusoid waveforms to assess acquisition accuracy. However, the performance varies based on the frequency of the input signal [11].
Prior to this thesis, an EEG system model, able to emulate the original behaviour, has been developed and the obtained results have guaranteed for the success outcome of the research. This experiment considers a novel hardware that generates

EEG potentials through a simulated skin-electrode interface, using readily-available laboratory hardware and low-cost components.
Electrical models of many dry and wet EEG electrodes have been used to precisely model the skin-electrode interface: the experiments have been performed on several healthy adult volunteers. The signals in the simulation process have been monitored at multiple stages in order to accurately detect individual losses caused by the simulation process, electrodes and acquisition system. Eventually, the efficient performance of the platform has been validated on clinically obtained neonatal EEG seizure data as per [10].
The simulation setup is showed in Fig. 2.1.

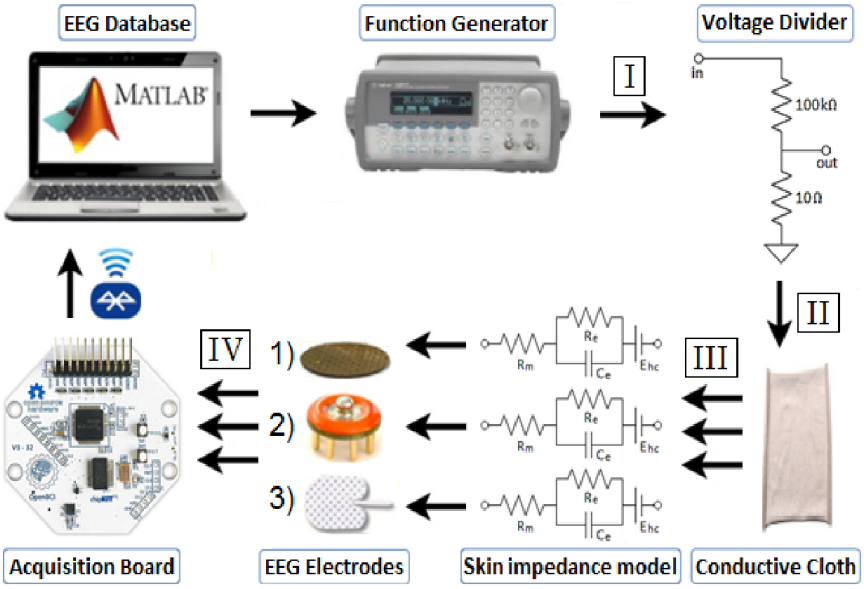

Figure 2.1: Simulation experiment setup [10]

Referring to the image above, in order to validate the correctness of the system, an anonymised neonatal EEG database from INFANT Research Centre has been exploited. These data are processed first on a Matlab application that transforms them in a 250Hz sampling rate, $\pm 0.5V$ amplitude, 0.5-100Hz bandwidth signal and applies a 50Hz notch filter to minimise the artefacts from the previous acquisition. EEG are extracted and grouped in epochs of 15 or 30 seconds before being sent to an Agilent 33220A AWG. After that, the signal is converted to analog and it is scaled by means a voltage divider in order to get closer to the original EEG, reaching around $\pm 100\mu V$.
The signal is passed on to a conductive cloth and the EEG electrodes are placed at the opposite end. The passive electrical impedance models of the skin are attached to the respective electrodes.
Each electrode output is sampled by a different channel on a acquisition board (OpenBCI Cyton Biosensing) that eventually, sends the data to the initial laptop through Bluetooth and comparisons with the initial signal are made.
Ideally, neonatal intracranial EEG would have been used, however this kind of data is not easily accessible. Although, EEG degradation due to skull artifacts is less severe with infants due to reduced skull impedance [12].
Another innovation introduced by this research project, is the use of dry electrodes

instead of wet. In fact, the current protocol used to acquire EEGs requires a non-negligible patient preparation: this entails the use of skin abrasives to abrade the top layer of the skin, and conductive gels to improve conductivity. Furthermore, skin abrasion can occasionally cause irritation [13]. Consequently, a novel solution that may be applied to neonatal field without irritating their delicate skin, has to be found in order to overcome ethical barriers that can delay the research process. For this purpose, dry electrodes have been taken in consideration. Even if wet electrodes can achieve less than 10 $K\Omega$ impedance (which is very low), dry ones can be used to obtain nice EEG acquisitions. In addition, they do not require abrasive or conductive gels but rely on novel mechanical designs to achieve high quality contact with the skin. Preliminary studies of alternative dry electrode technologies provided promising results [14].

To increase the chance of a research project success, different electrodes have been studied and a model has been extracted for each of them. The electrode types that have been considered are shown and numbered in Fig 2.1 and they are presenting below.

1. MicroTIPs: Micro Transdermal Interface Platforms (MicroTIPs) bypass the beyond layer of skin, which mostly consists of non-conductive skin cells [15]. This behaviour can be obtained by exploiting micro-machined badger structures that penetrates the Stratum Corneum managing to create direct contact with the internal layer of the epidermis that is conductive.
   Due to sub-millimetre height, the electrodes does not hurt the skin and let it bleed. Differently to skin abrasion, the MicroTIPs only penetrate the Stratum Corneum at a certain points, avoiding to damage the layer and decreasing the exposure to irritation and infection. The MicroTIPs are built by means the use of silicon wafers, which are eventually coated in gold. The electrodes are assembled with 3M RedDot packages, which are extensively used in clinics.

2. G.Sahara Dry Electrodes: G.Sahara electrodes, invented by G.Tec, show a pin based electrode made by a gold alloy. The electrode consists of 7mm or 16mm pins, which can sample high quality EEG through dense hair [16].

3. Wet Electrodes: Ambu Neuroline 700 and Cup electrodes were used with Ten20 conductive gel and NuPrep abrasive cream throughout this study as the typical wet electrode setup [10].

The EEG signal has to propagate through the skull and skin too before being captured by the electrodes. Consequently, a low electrode impedance is desired in order to reduce the hole filtering and increasing the acquisition reliability.
According to the International Federation of Clinical Neurophysiology (IFCN), contact resistance should be less than $5k\Omega$. However, current differential amplifiers preserve input impedances in the $G\Omega$ range, letting high skin-electrode impedances almost negligible [13].

Bio-impedance experiments have been done on adults in order to extract an electrical model of the skin-electrode interface. The extracted results have been used in the final simulation.
The impedance analysis has been made on 5 healthy adult volunteers with different skin and hair variety, with approval from the Institutional Clinical Research

Ethics Committee. The experiment has produced satisfying result and the resultant equivalent circuit is showed in Fig. 2.2.

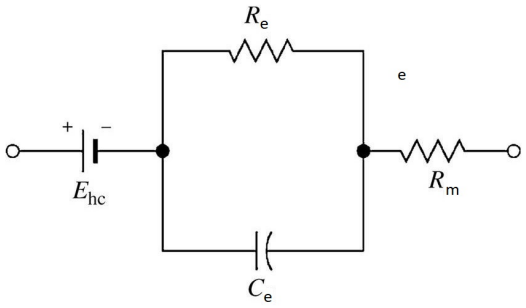

Figure 2.2: Bio-impedance equivalent circuit

Referring to the image above, the capacitance C can be evaluated by the following formula:

$$C = \frac{1}{(2\pi f|X|)}$$

The resistance R_e is thus extracted by means the experiment, E_{hc} is the half-cell potential between the electrode material and the ionic solutions surrounding it and R_m is the the electrode material resistance, which is negligible w.r.t. R_e.
At low frequencies, the circuit is dominated by the $R_e + R_m$ series combination. At high frequencies, it is dominated by the $R_m + C_e$ combination [17].
From the moment that Neonatal EEG activity is mostly low frequency, only R_e has been taken into account in the final Acquisition System [10].

At this point of the experiment, the model had been considered ready and first measurements could have been achieved. Since EEG signals have very tiny voltage (around $\pm 100\mu V$) and high slew-rate, low-noise high-resolution ADC is required. For this purpose, a custom system-on-chips (OpenBCI Cyton) have been developed appositely for EEG recording. This electronics exploits an ADS-1299 chip to acquire low-power (5mW/channel) and low noise ($1/muV_pp$) EEG [8].
It has been ascertained that the OpenBCI Cyton satisfies all the requirements imposed by International Federation of Clinical Neurophysiology.
Furthermore, in order to quantify the losses, the signal has been studied at stage I, II, III and IV as it is showed in Fig 2.1. This procedure guarantees to independently monitor the losses in the system due to the board and electrodes.

The output of the generator (Gen) ($\pm 0.5V$) was recorded using the Tektronix MSO 3032 (Stage I). The OpenBCI board recorded the following 8 channels as referenced in Fig. 2.1: Stage II after the resistor divider (Res), Stage III after the conductive cloth (Clo), Stage IV for wet frontal (WF), wet occipital (WO), G.Tec frontal (GF), G.Tec occipital (GO), MicroTIPs frontal (MF), MicroTIPs occipital (MO). The impedances corresponding to the electrode model of the frontal/occipital locations were used [10].

The power-line noise was evaluated by applying the Fast Fourier Transform and evaluating the average power at 50Hz. The amplitudes of the signals were normalized using standard z-score normalization and the SNR and Pearson correlation coefficient were calculated before and after applying a 50Hz notch filter:

$$Z_y = \frac{Y-\mu_y}{\sigma_y} \times \sigma_X + \mu_X$$

$$snr = 20 \log \frac{X}{|X-Y|}$$

$$r = \frac{1}{N-1} \sum_{i=1}^{N} \left(\frac{X_i-\mu_x}{\sigma_x}\right)\left(\frac{Y_i-\mu_y}{\sigma_y}\right)$$

where X is the emitted signal, Y is the received signal, Z is the normalized signal, μ is mean and σ is standard deviation [10].

The results have been extracted and comparisons between different electrodes have been made. As it showed in Fig 2.3, the various electrodes are compared focusing on skin-electrode resistance and capacitance (in blue and red, respectively) vs. frequency for 3 electrodes at frontal and occipital locations.

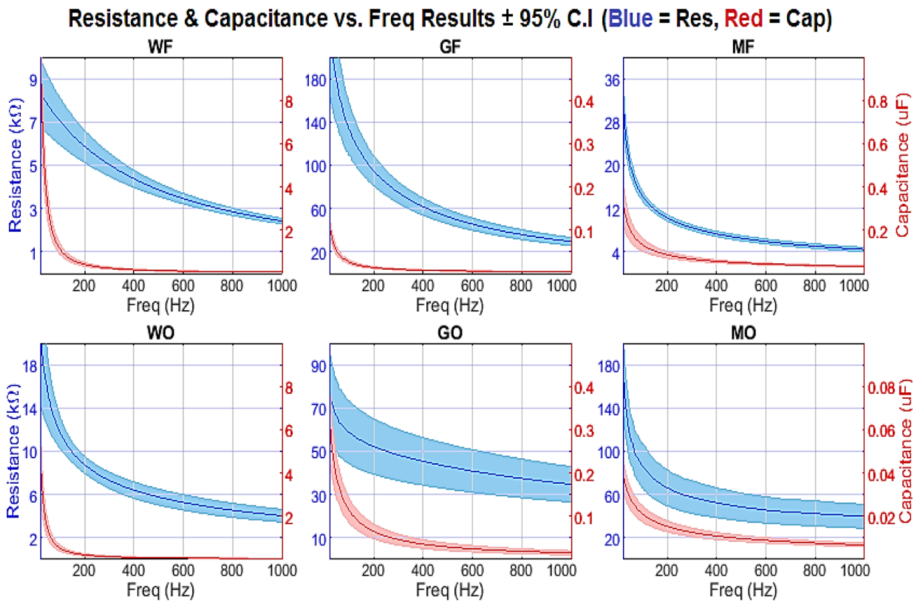

Figure 2.3: Skin-electrode resistance and capacitance [10]

Predictably, the wet electrode achieves the lowest impedance, mostly thanks to skin preparation via abrasive cream and better conductivity by using conductive gel.
The G.Sahara electrode achieved better results at the occipital region, over hair than on the frontal region.
Contrariwise, the MicroTIPs obtain lower impedances on the frontal region as the Stratum Corneum (SC) is effectively penetrated, drastically reducing the impedance. At the occipital region, the MicroTIPs are less efficient from the moment that the $500\mu m$ tips attempt to proceed through the dense hair and make contact with the Stratum Corneum.

At this point, as it has been mentioned before, the signal has been studied at every stage, thus correlation with the original signal and SNR have been evaluated. The calculations have also been made in unfiltered and filtered condition and for each electrode type. The records of correlations and SNR are reported in the Table 2.1 and Fig 2.4 respectively.

	Correlation ±95% Conf Int.		50 Hz noise (uV)
	Unfiltered	**Filtered**	
Generator	0.998 ± 0.005	0.998 ± 0.004	0.32 ± 0.11
Resistor	0.997 ± 0.015	0.997 ± 0.015	0.07 ± 0.11
Cloth	0.997 ± 0.015	0.997 ± 0.015	0.24 ± 0.11
Wet Front.	0.997 ± 0.016	0.997 ± 0.015	0.36 ± 0.15
Wet Occip.	0.997 ± 0.019	0.997 ± 0.015	0.51 ± 0.21
G.Tec Front	0.867 ± 0.555	0.982 ± 0.094	4.94 ± 2.11
G.Tec Occip.	0.978 ± 0.107	0.995 ± 0.019	1.54 ± 0.65
Micro Front	0.990 ± 0.041	0.996 ± 0.015	0.94 ± 0.40
Micro Occip.	0.881 ± 0.511	0.985 ± 0.076	4.59 ± 1.98

Table 2.1: Correlation and Power-Line Noise Results [10]

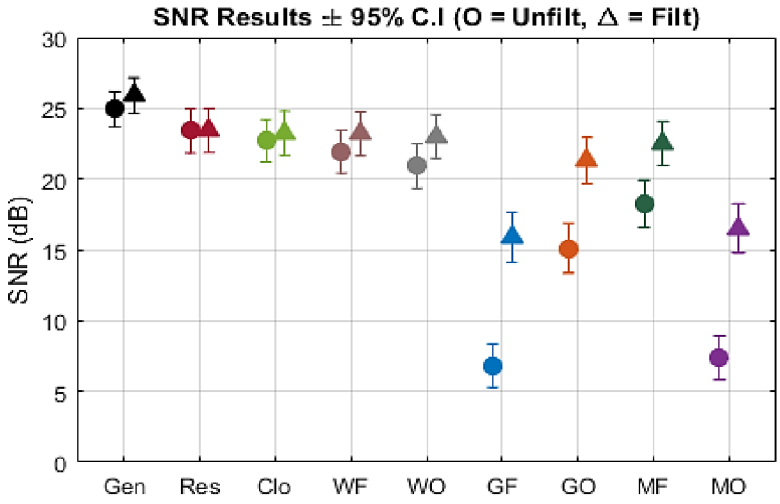

Figure 4. Simulation SNR results.

Figure 2.4: Simulation SNR results [10].

Again as expected, the signal extracted by means wet electrodes presents better correlation value and SNR w.r.t. other two electrode types. However, it is pointed out how the filtering may help the reconstruction of the original signal as far as the MicroTIPs and Sahara Dry electrodes is concerned.
Consequently, the experiment has been useful to verify the reliabilty and usability (despite lower performances and some assumptions) of using new electrode types that may be more adapt to use on delicate neonatal skin.

The main intent of this paragraph is to describe and validate an accurate and low-cost experimental platform, looking for new inventions that are more adapt for newborns. Results that have been obtained show minimal losses and high accuracy w.r.t. the original EEG signal. Filtering significantly helps the reformation of the original signal for those new electrode technologies that present higher impedance. The developed evaluation framework makes use of an extensive neonatal seizure EEG database and have allowed the researches to optimize and validate novel neonatal EEG monitoring systems.

2.2 Neonatal EEG Sonification algorithm

Neonatal encephalopathy is a clinically defined syndrome of disturbed neurologic function in the earliest days of life in an infant born at or beyond 35 weeks of gestation, manifested by a subnormal level of consciousness or seizures, and often accompanied by difficulty with initiating and maintaining respiration and depression of tone and reflexes. Nevertheless, it is estimated that only 34% of neonatal seizures [18] present clinical signs and the remaining part can only be diagnosed using electroencephalography (EEG) monitoring.
Failing the detection of such events implies a lack of treatment and so a potentially life-threatening outcomes. The standard approach to EEG signals is based on visual interpretation that requires significant training and a constant presence of experts. Usually this is not available on a 24-hour basis, 7 days a week and even if possible, it could be expensive.
To cope with it, methods for EEG Sonification are implemented to facilitate EEG interpretation in a quicker and easier way. In fact, the presence of a rhythm can be seen as a structure, more identifiable with hearing rather than with visual aids. However, the pivotal point of the Sonification algorithm that has been realized, is in its additional use of Amplitude Modulation (AM) and customised compression techniques. This aims to increase the sensitivity of the human ear to the presence of rhythm.

2.2.1 Phase Vocoder

The ordinary algorithm that is utilized to perform a Sonification is called Phase Vocoder (PV). It allows for the perception of such frequency changes over time in the audio [4][19] and, as final result, performs spectra to spectra mapping by preserving a horizontal phase coherence.
The pivotal feature of this algorithm is that it employs a short-time Fourier transform (STFT) analysis-modify-synthesis loop and is typically used for time-scaling of signals utilizing different time steps for STFT analysis and synthesis. In particular, the challenge of the developed PV algorithm used for that purpose is the conversion of the low frequency EEG signal (0.5-8Hz) into the audible frequency domain (250-4000Hz).

The algorithm is composed by four blocks that are shown in Fig. 2.5:

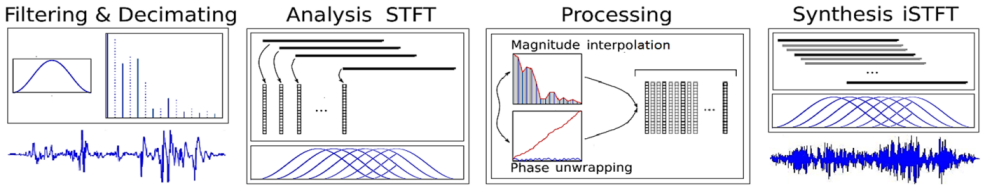

Figure 2.5: PV methodology: the EEG signal is processed by frames and reconstructed with coherent phase [18].

- **Pre-processing**: The original EEG signal is band-pass filtered between 0.5-

8Hz and down-sampled to 16Hz. These parameters were tuned towards maximising the perception of seizure frequency evolution [18].

- **Analysis**: This part of the process is in charge to divide the filtered signal into numerous time segments or blocks of samples and then multiplying those blocks by an envelope called a windowing function. The shape of these windows has an effect on the weighting of the resultant analysis. In this algorithm, the Short Time Fourier Transform (STFT) with a window length/shift of 64s/16s was used. The obtained results consist in the decomposition of the EEG segment into magnitude and phase for each frequency bin.
- **Processing**: The magnitude previously obtained is linearly interpolated by a variable time compression factor and saved at an 8kHz sampling rate. While, the phase is measured and unwrapped to track the cumulative phase variation in order to preserve the phase consistency across interpolated frames.
- **Synthesis**: In this final phase, an inverse STFT is applied, following by the overlap-add method using the same window from the analysis stage. The result consists in an 8kHz audio signal.

2.2.2 FM/AM algorithm

In contrast with the previous algorithm, the Frequency and Amplitude Modulation (FM/AM algorithm), the one that has been utilized in this work of thesis, implements a waveform to spectra mapping.

The spectra of the audio signal is obtained according to the waveform amplitudes of the EEG signal. This implies that a repetition of waveforms in the signal can be considered as a structure, perceived as a rhythm in the resultant audio [18].

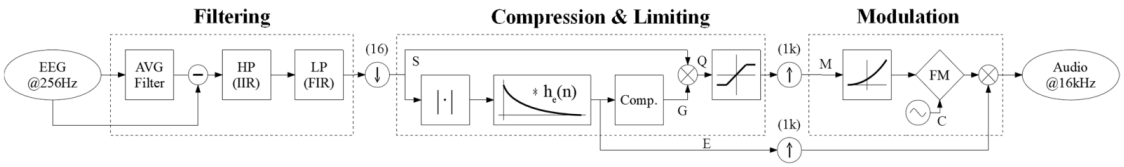

Figure 2.6: FM/AM methodology: the EEG signal is filtered, compressed and modulated [18].

Usually in telecommunication, the modulation process is used to vary a carrier wave (expressed as frequency or amplitude) proportionally to a modulating wave. In this case, the EEG is used as the modulating waveform. The developed algorithm is characterized by three main phases: the filtering, the compression and limiting and finally the modulation. The correspondent block diagram is shown in Fig 2.6.

The three phases will be discussed below.

1. **Filter Phase**
   From the assumption that EEG signal bandwidth is situated below 35 Hz, a series of cascaded filters was designed in order to remove background noise

and reduce the distortions possibly introduced by the record channels. In particular, the neonatal signal is characterized mostly by theta wavelets, whose frequency reaches at most 7.5 Hz, hence the latter has been considered as upper limit for the filtering stages. According to the algorithm [18], good results can be achieved by applying two cascaded filters, after removing the mean value from the original signal. The schematic of filtering stage is shown in the figure below.

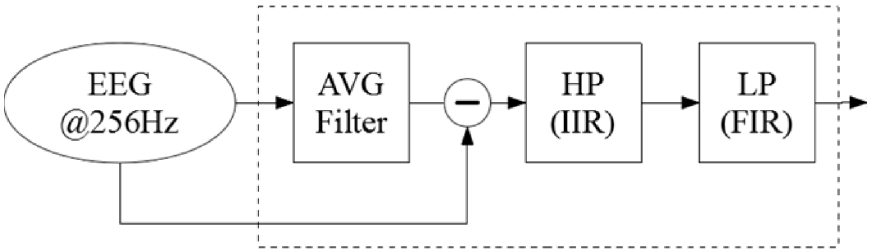

Figure 2.7: Filter stage of FM/AM Sonification algorithm

Going into more detail of the pre-processing stage, the average filter is the first one utilized: the DC is removed by applying a moving average filter to the original signal. Since the frequency resolution needed when dealing with an EEG signal is 0.5 Hz, a window length of 2 s has been chosen [18].
The second filter utilized is a bandpass one. This is achieved by means of a HPF IIR, followed by a LPF FIR. The signal is filtered between 0.5-7.5 Hz and down-sampled to 16Hz.
In addition, a Notch filter may be useful to remove the 50Hz component from the original signal, attributable to the power network interference.

The characterization of each filter will be discussed in chapter 6, where it is shown the implementation adopted for the full algorithm.

2. **Compressor phase**
This second phase to reduce the dynamic range of the input signal (S) (at Fig 2.6) with minimal distortion, since EEG signals often contain amplitude spikes that exceed the common amplitude range of $\pm 50\mu V$. For this purpose, the envelope of the signal is estimated, generating a smooth curve outlining its extremes, that then is used to control the gain reduction.

Compressors operate on a logarithmic scale, in decibels (dB). Thus, the signal E is firstly converted to his decibels version Edb as per formula below:

$$E_{db} = 20 \times log|E| \tag{2.1}$$

Sometime the amplitude of the envelope is under a specified threshold, implying that no gain reduction is applied to the signal. Contrary, when the threshold is exceeded, the amplitude of the signal is reduced proportionally to the compression ratio R, as in the following equation:

$$G_{db} = \begin{cases} 0, & \text{if } E_{db} < T_{db} \\ -(1 - \frac{1}{R})(E_{db} - T_{db}), & \text{otherwise} \end{cases} \tag{2.2}$$

For every 1 dB that the input signal overcomes T_{db}, the amplitude of the output will be reduced by an amount of $(1 - \frac{1}{R})$. As $R > 1$, the gain reduction G_{db} is always ≤ 0. G_{db} is then converted back to the linear domain (G) by using the reverse of 2.1. G is then used to control the gain of the input S and obtain the output Q as it is shown if Fig 2.6.
In this algorithm, a threshold T_{db} of -20dB ($5\mu V$) is chosen. R is chosen to be 1.5, meaning that a spike of $150\mu V$ would be reduced to $50\mu V$, keeping the output within the desired dynamic range [18].

In the Frequency Modulation stage, the signal could be affected by aliasing when it is utilized as modulator signal. To cope with it, the output of the compressor phase, indicated as Q, is amplified from $\pm 50\mu V$ to $\pm 1V$, as shown in equation:

$$M(Q) = \begin{cases} 1, & \text{if } Q > 50\mu V \\ 2 \times 10^4 Q, & \text{if } -50\mu V < Q < 50\mu V \\ -1, & \text{if } Q < -50\mu V \end{cases} \tag{2.3}$$

3. **Modulation phase**
Firstly, the modulator signal (M) is up-sampled to 16kHz in order to achieve an audio bandwidth. Then, an exponential transform is applied to M since the human hearing system perceives frequency on an exponential scale [20]. The value assumed by the frequency (F) depends on M, according to the following expression:

$$F(M) = 500 \times 10^M \tag{2.4}$$

So, when M equal to 0V, F is set to 500Hz; when M is -1V, then F goes down to 50Hz and if M is 1V, F goes up to 5kHz.

In the resulting output wave, W, the amplitude variations of the EEG are mapped to the frequency variations of the output signal. The AM phase is eventually performed when the envelope E is used to modulate the amplitude of W. Thus, the amplitude of the EEG signal is embedded in the output audio.

2.3 Deep Learning approach to Neonatal Seizure Detection

Nowadays the SVM-based classifier represents the state-of-the-art in neonatal seizure detection and its performance has been validated on a large clinical database to confirm robust functionality and a clinically acceptable level of accuracy of detection. The Support Vector Machine (SVM) system relies on a large set of engineered features and is well suited to binary classification problems like the seizure detection one.
Nevertheless, the study conducted by Alison O'Shea, Gordon Lightbody, Geraldine Boylan and Andriy Temko [21] aims to construct a deep convolutional neural network, trained on raw neonatal EEG waveforms, in order to compare its performance with the state-of-the-art SVM-based classifier.
What is new with respect to previous neonatal seizure detection studies, is the use of CNNs to learn features from the raw EEG data, in a similar manner to CNNs that have been applied to raw audio.
The small width of the convolutional filters facilitates the learning of high frequency patterns in the first layers, and low frequency patterns, with the increased receptive field, in subsequent layers.
The network is constructed according to a fully convolutional neural network (FCNN), without any fully connected layers at the back-end, conversely to the classical approach. This choice allows the application of the developed seizure detector to be adapted to a segment of EEG of any length and, additionally, it facilitates the interpretation of the learnt feature maps. This means that the network is able to localize the most discriminative waveforms in the original EEG, and offer higher representational power at lower computational cost.

According to [22], it is nevertheless possible to take advantage from the combination of both methods, due to the large contrast between SVM based on domain-knowledge feature extraction and purely data-driven CNN feature extraction.

2.3.1 FCNN-based seizure detector

The first phase is the pre-processing one, where the raw EEG data is sampled at 256Hz and then filtered with a band pass filter with cut off frequency of 0.5 and 12.8Hz. After, the signal is down-sampled to 32Hz and split into 8s epochs with 50% overlap. When the pre-processing is completed, the signal is used as input of the deep neural network, in this case constructed as a fully connected neural network (FCNN).

In general, FCNNs are composed by only convolutional, pooling and activation function layers in order to compute a deep nonlinear filter.
These building blocks allow to obtain a translational invariance, where the computed parameters in the FCNN are location independent. However, one very common problem in deep learning is related to the overfitting, in which a neural network has a very high variance and cannot generalize well to data it has not been trained on. To cope with this tricky situation, the approach adopted in the network is such that the fully connected layers (used commonly as the classification layers) are replaced with a global average pooling (GAP) classification procedure. This choice introduces regularization by reducing the number of learned parameters and improves

the performance of the network generalizing the data as much as possible.
Doing a deeper analysis on the architecture, the layers utilized are those specified in Table 2.2.

Layer Type	Shape	Output Shape	Parameters
Input	256	256x1	0
1D Convolution	32 filters 4x1 kernel Stride 1	32x253x1	160
1D Convolution	32 filters 4x1 kernel Stride 1	32x250x1	4128
1D Convolution	32 filters 4x1 kernel Stride 1	32x247x1	4128
Batch Norm.		32x247x1	64
Average Pooling	Pool 8 Stride 2	32x120x1	0
1D Convolution	32 filters 4x1 kernel Stride 1	32x117x1	4128
1D Convolution	32 filters 4x1 kernel Stride 1	32x56x1	4128
Average Pooling	Pool4 Stride 2	32x56x1	0
1D Convolution	2 filters 4x1 kernel Stride 1	2x53x1	258
GAP **Softmax**		2x1 2	0 0

Table 2.2: CNN architecture, all convolutional layers are followed by rectified linear units.

Even if the network is designed to be fully convolutional, there are no fully connected layers. The network layer values are calculated based on convolutional filters, including the final "decision making" layer at the output, while the internal and final representations of the data within the network are based on local spatial filters. This methodology gives greater importance to positional relationships of features when making decisions.
In addition to that, there are 6 convolutional layers in total, where each one can be seen as a higher level of feature abstraction of the previous layer. All of them use 1-dimensional convolutional filters of length 4, constructed using the Keras deep learning library [23].

In order to estimate the output of each feature map, the following procedure is applied:

$$h_i^k = max(0, ((W^k * x) + b))$$

Where i is the temporal index within a feature map, x is a matrix representing the input to a layer and W is the convolutional layer weight matrix.
Basically, the product $(W^k * x)i$ refers to convolutional calculation estimated at position i, for the k^{th} feature map, taking inputs from all feature maps in the previous layer. The activation is applied by taking the max between the output and 0.
Following the architecture in Table 2.2, the pooling layer is used to reduce the dimensionality of the data, and thus the number of parameters in the network, and provides spatial invariance. In particular, the pooling utilized is the average one, because convolution followed by average pooling can be seen as a filter-bank signal decomposition performed on the raw EEG signal. Precisely, the pooling windows used in this study are of width 8 with a stride of 2, halving the size of the input.

In the end, the final convolutional output is followed by GAP giving 2 values on which the softmax operation is performed to give the seizure and non-seizure class probabilities. In fact, using GAP, the two final feature maps are forced to correspond to the seizure or non-seizure class and this allows to obtain a clearer visualization and localization of the underlying processes, since each sample in the final layer can be traced back to a corresponding window of the input EEG signal.

In conclusion, the high pass filters are learnt in the first layers, which have narrow receptive fields, and low pass filters are learnt in the final layer, with the hierarchically increased width of receptive field.
As far as the training phase is concerned, the network is trained using cross-entropy as the loss function. Stochastic gradient descent is used with an initial learning rate of 0.003, reduced by 10% every 20 iterations. A batch size of 2048 is used for both training and validation. To reduce the risk of overfitting on training data, the regularization method is applied, using shared weights and sparse connectivity results in fewer trainable parameters than fully connected networks.

Eventually, to manage the output, a post-processing stage is request. A moving average smoothing of 61 seconds is applied to the output probabilities and the final probabilistic support for an epoch is computed as the maximum probability across the channels. Then, this percentage is compared with a threshold and every positive decision is expanded by 30 seconds from each side, to account for the delay introduced by the smoothing operator.

The pivotal aspect of this study is the replication of the process by which a healthcare professional interprets the EEG, achieved by the temporal relationship between the final layer and the raw input EEG.
What is more, the main advantage of using a fully convolutional network is that the weight matrix at each convolutional layer can be visualized, and so further research will concentrate on gaining new insights into the make-up of neonatal EEG signals based on these learned weight matrices [21].

Chapter 3

Instrumentation

The hardware devices used during the demo implementation, have been discussed in this chapter. A general study on these electronics have been made focusing on computation and power capabilities. In addition, their contribution in the thesis has been enhanced by underlining their role within the implementation development.

3.1 ADS1299 Texas Instrument

As discussed in the previous chapters, EEGs are very low voltage signals with an high slew-rate (variability of electrical quantity per time unit, Voltage in this case). Consequently, a low-noise high-resolution ADC is required. Furthermore, main goal of this work is to produce an end implementation able to satisfy requirements in therms of low power and portability.
One of the possible electronics able to carry out the initial demand is a ADS1299 from Texas Instrument. This is an eight-channel, low-noise, 24-bit, simultaneous-sampling delta-sigma ($\Sigma\Delta$) analog-to-digital converters (ADCs) with a built-in programmable gain amplifier (PGA), internal reference, and an onboard oscillator. The ADS1299 incorporates all commonly-required features for extracranial electroencephalogram (EEG) and electrocardiography (ECG) applications. With its high levels of integration and exceptional performances, the ADS1299 enables the creation of scalable medical instrumentation systems at significantly reduced size, power, and overall cost [8].

As it is shown in functional block in Fig. 3.1, the circuit can deal with up to 8 input signals (IN positive and IN negative each); these inputs go to a multiplexer whose control signals may disable or enable different kind of analysis such as temperature, supply, input short, and bias measurements. In addition, the MUX grants the capability to program any input electrodes as the patience reference drive.
On the multiplexer output, each signal is captured by a programmable gain amplifier (PGA): this allows the amplitude of the input signal to be multiplied by a factor of 1,2,4,8,12 and 24.

Therefore, the amplified EEGs are sampled by a $\Delta\Sigma$ ADC: these converters consist of an oversampling modulator followed by a digital/decimation filter that together produce a high-resolution data-stream output [24]. A possible block diagram of a $\Delta\Sigma$ converter is shown in Fig. 3.2. One of the main feature of this ADC

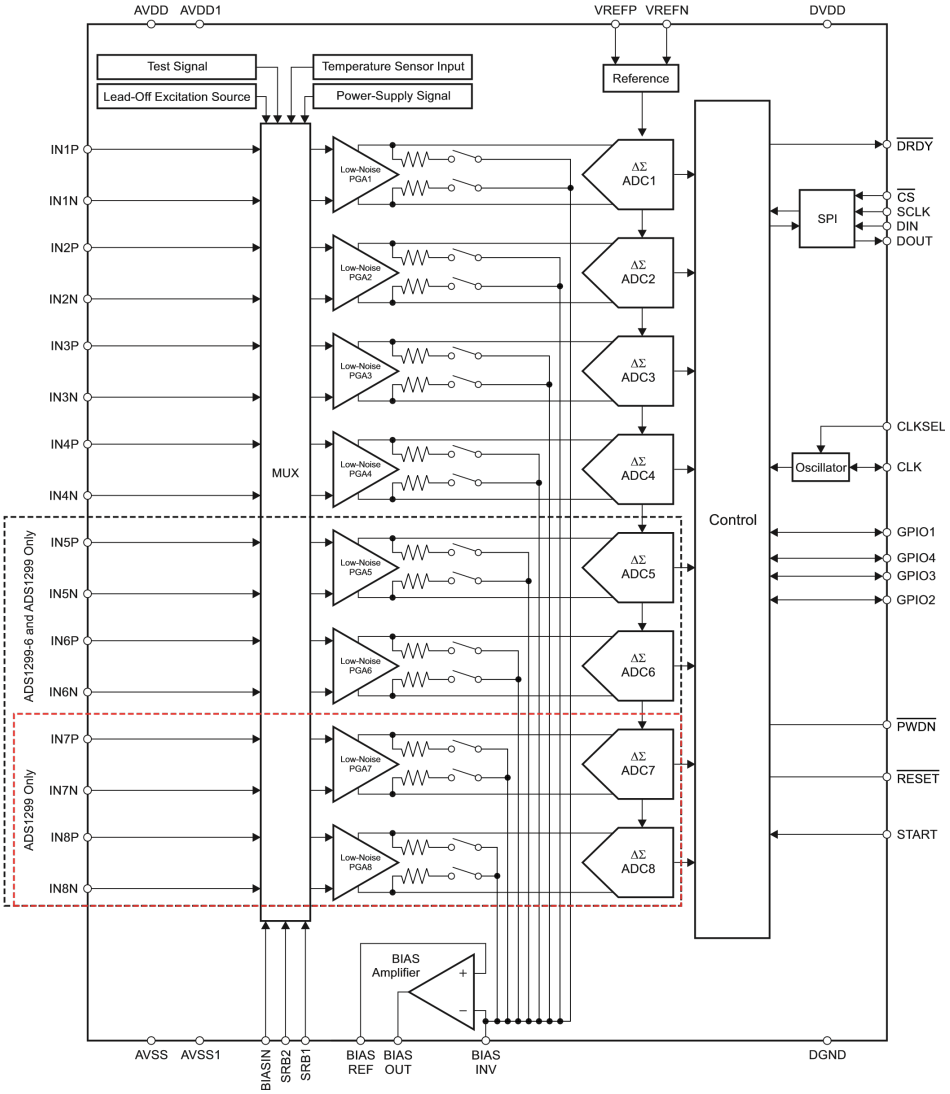

Figure 3.1: Functional Block Diagram ADS1299 [8]

is to use the oversampling technique: it allows to improve the signal-to-noise ratio by increasing the sampling frequency with respect the minimum value $2f_b$ (from Nyquist Theorem). Consequently, it lets possible to use cheap anti-aliasing filters because replicas are very far apart in the spectrum. $\Delta\Sigma$ ADCs are also fast circuits and the overall cost is reduced by moving the most complex operations (such as integration) in the digital domain where circuits are faster and cheaper.

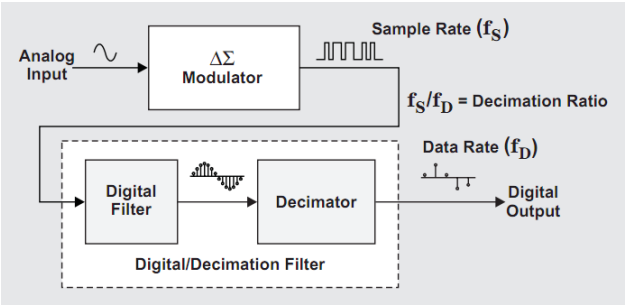

Figure 3.2: Block Diagram of a $\Delta\Sigma$ ADC

$\Delta\Sigma$ converter allows also to increase the dynamic range of the input by limiting the slew-rate of the signal: this behaviour can be achieves by the presence of an integrator before the conversion takes place. Other than improving the dynamic range, a $\Delta\Sigma$ ADC has an additional advantage. In fact, quantization noise is produced after integration, and it is subject to a high pass transfer function. Therefore, quantization noise is shifted to high frequencies, closer to the sampling one. This effects, known as noise shapping, permits to decrease the in-band power spectrum of quantization noise, or to use a simple reconstruction filter.
The output of a $\Delta\Sigma$ converter is a high rate bit stream. In order to let the digital domain able to deal with the incoming data, it is normally processed by a decimator, that generates the bit stream into a sequence of words at a much low rate. When the reconstruction of analog signal takes place, an interpolator recreates the high speed serial flow [25].

The $\Delta\Sigma$ ADCs used in the ADS1299 presents a configurable data rate varying from 250 SPS up to 16 kSPS (250, 500, 1000, 2000, 4000, 8000, 16000 SPS). The output of each ADC goes to a control logic that can be programmed and it represents the main interface between the user and the board.
The voltage supply reference for all the ADCs is extracted from the difference between VREFP and VREFN (VREF = (VREFP – VREFN) = 4.5 V) where VREFP and VREFN are dependent from the main supply AVDD and AVSS and they may vary between maximum ranges defined in Table 3.1.

VOLTAGE	MIN	MAX	UNIT
AVDD to AVSS	-0.3	5.5	V
DVDD to DGND	-0.3	3.9	
AVSS to DGND	-3	0.2	
VREFP to AVSS	-0.3	AVDD + 0.3	
VREFN to AVSS	-0.3	AVDD + 0.3	
Analog input	AVSS - 0.3	AVDD + 0.3	
Digital input	DGND -0.3	DVDD + 0.3	

Table 3.1: Absolute Maximum Voltage Range in ADS1299 [8]

where AVDD and AVSS have to be chosen in order to respect the equation $AVDD–AVSS = 5V$. DVDD and DGND are digital voltage supply and digital ground respectively and their difference is preferable to be chosen between 1.8V and 3.6V. Digital and Analog ground are connected.

Communication with the board is performed through SPI, either data exchange and programming. In addition, the hardware owns four general purpose input/output (GPIO) for generic usage. The internal oscillator generates a 2.048-MHz clock when enabled. The functional patient bias drive block, showed at the bottom of Fig. 3.1, grants the average on any electrode combo to be selected in order to generate the patient drive signal. Lead-off detection can be accomplished by using a current source or sink. A one-time, in-band, lead-off option and a continuous, out-of-band, internal lead-off option are available [8].

ADS1299 owns 24 8-bit registers in total, which three of them is readable only

(ID Register and 2 Lead-Off Status Registers) and the others are mostly R/W except very few bits that are read only. This means that the number of possible configurations is massive and this variety allows the ADS1299 to be used in several applications. The adopted configuration and the SPI setup at power-up will be discussed in the Implementation section.

The initial experiments have been performed using the full ADS1299 kit for evaluating the main chip: in fact this hardware is composed by the ADS1299 microchip rightly placed on a PCB (ADS1299EEG-FE) according to the datasheet[8] specifications plus the MMB0 Modular EVM motherboard that fully implement the SPI control interface, as it is shown in Fig. 3.3.

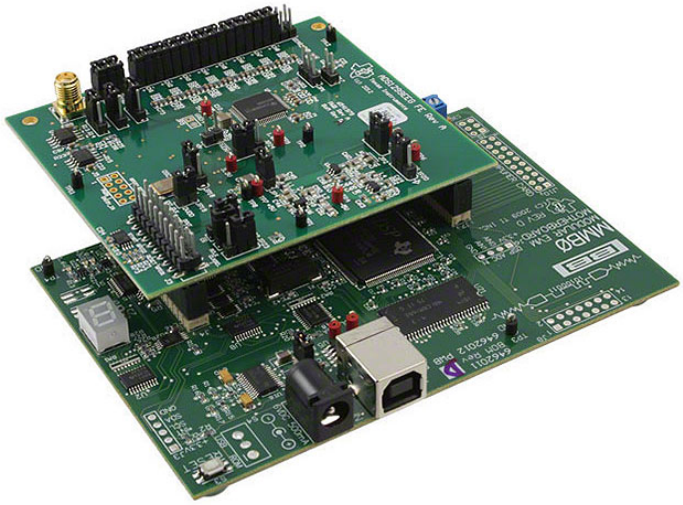

Figure 3.3: ADS1299EEG-FE Kit

The MMB0 motherboard allows the ADS1299EEG-FE to be connected to a computer via an available USB port [26]. A windows software comes out with the entire kit. This tool has been initially exploited to validate the chip and to find the best register configurations to acquire proper EEGs: in fact the software, offers to possibility to easily set the ADS registers using a simple GUI and this has helped to find the best trade off of settings. In addition, the ADS1299EEG FE printed circuit board has some jumpers that can be moved in order to change some HW characteristics such as Unipolar or Bipolar supply, digital supply (DVDD) equal to 3.3V or 1.8V and so on.

First test has been performed on a shrunk signal obtained by applying a voltage divider on a sinusoid generated by means an Agilent 33220A AWG. The amplitude has been limited between $-100\mu V$ and $+100\mu V$, similar to a real EEG and the signal has been sampled. The tool has been able to reconstruct the full signal; the noise introduced by an extreme low ratio divider of resistors and the function generator itself, has been reduced by applying filters available in the software. After all, the next test has been executed on real EEGs by applying two dry electrodes on the front and one under the hear to be used as negative/positive input and bias reference respectively. After that, the sampled signal has been filtered by employing 0.5Hz High-pass, 16Hz Low-pass and 50Hz Notch filters accordingly. Several

programmable settings have been tried to get closer to the expected EEG shape. Eventually, the acquired signals have been compared with respect to EEG waveforms provided by the INFANT Research Centre to validate the correctness.

However, the evaluation kit does not support the Real-time data processing, indeed it can not be used for the desired purpose. Furthermore, the motherboard has been disconnected and the full application has been implemented by programming the SPI interface and properly setting the boot-up sequence by means another board (Raspberry Pi in this case, but a firmware has been written and tested on Arduino as well).

Lastly, a much more compact PCB within the ADS1299 chip has been utilized: during the implementation of the full demo, Mark O'Sullivan (Phd student in University College of Cork) has been designing the acquisition board intended to be placed in the final portable appliance. The layout has been accurately chosen in order to customize the board for specific purpose only, significantly reducing the whole area and power consumption (e.g., those hardware characteristics modifiable through jumpers on ADS1299EEG FE printed circuit, have been removed in order to act in one way only). As a result, an extreme small acquisition board has been obtained.

3.2 Dragonboard 410c

This paragraph is focused on an embedded board that has been studied during the demo implementation. The main purpose of this electronic was to replace an Android tablet (Samsung A6 tablet 2GB RAM, Exynos 7870 @ 1.6GHz CPU, Mali-T730 GPU, 7300mAh battery) in performing the main computations such as Real Time Sonification, AI and signal processing.
The final idea was to let the Dragonboard sending already processed data to the tablet through Bluetooth; in this way, the tablet only needs to visualize the EEGs and reproduce the generated sound, drastically reducing the whole computation (hw + sw) and so the energy consumption. Furthermore, the developed Java application should have been split in 2 parts, one running on the Dragonboard and the other on a tablet/smartphone.

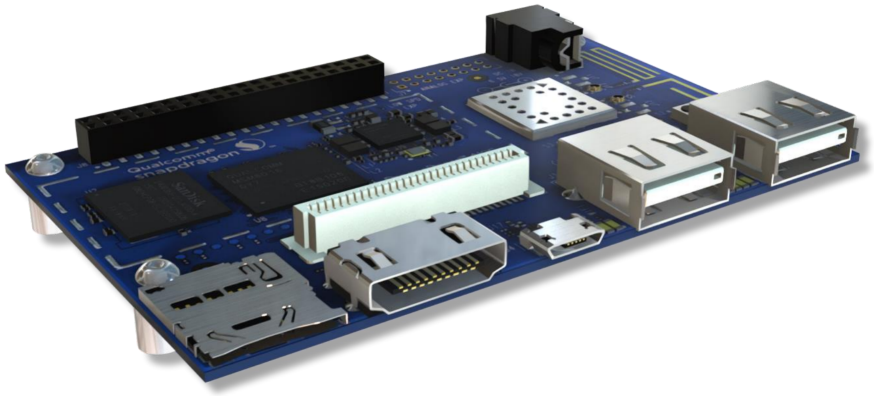

Figure 3.4: Dragonboard 410c Board

This concept has been thought to intensely improve the computation at the edge: this is a paradigm which brings computation and data storage closer to the location where it is needed, to improve response times and save bandwidth [27]. In particular, edge computing is any type of computer program that delivers low latency nearer to the requests in those applications in which Real-Time processing of data is required (as in this case).

Another reason to prefer an intermediary board w.r.t. a direct Android application is to avoid the tablet battery to drain too fast since the Android EEG acquisition app would have run constantly, with a significant power consumption due by high computation effort. Indeed, from the moment that both time and power are extensive, the outcoming energy results to be a killer for the tablet battery life, according to the simplified formula below, that expresses a direct proportional dependency between energy and the product of the other two quantities:

$$E = P \times t$$

Using a customized application on a intermediary board would have made the tablet/smartphone, already occupied in display and audio streaming, lasting for longer time.

The Dragonboard 410c is a 96Boards compliant community board based on Qualcomm Snapdragon 410 SoC. Making a comparison between the 2 SoCs (tablet with Exynos 7870 and Dragonboard with Snapdragon 410) it is possible to see how the performance/Watt ratio is suitable for Exynos. Various benchmarks performed on both the SoCs, see the Exynos winning in all the comparisons as per Table 3.2.

SoC	Samsung Exynos 7 Octa 7870	Qualcomm Snapdragon 410
Part number(s), S-Spec	Exynos 7 Octa 7870	MSM8916
Release date	Q1 2016	Q3 2013
Lithography	14 nm	28 nm
Cores	8	4
Threads	8	4
Base frequency	1.6 GHz	1.2 GHz
High performance cores	8x ARM Cortex-A53 @ 1.6 GHz	4x ARM Cortex-A53 @ 1.2 GHz
Cache memory	2 MB	1 MB
Thermal design power	3 W	3 W
GPU clock rate	700 MHz	400 MHz
GPU FP32 floating point	23.8 GFLOPS	19.2 GFLOPS
AnTuTu	46,000	21,020
Performance/ Watt	1,051 pts / W	463 pts / W
Performance/ Global Cost	17 pts / €	7 pts / €

Table 3.2: Specification comparison between Samsung Exynos 7 Octa 7870 and Qualcomm MSM8916 Snapdragon 410 [28]

As it is shown in table, the Exynos plenty wins the competition: even if Exynos has double of cores and HW threads w.r.t. Snapdragon (8 vs 4 respectively), higher base frequency (1.6 GHz vs 1.2 GHz), higher GPU clock rate (700 MHz vs 400 MHz), results to have a better performance/watt ratio and performance/cost ratio. Also the AnTuTu benchmark sees the Exynos winning against the Snapdragon with more than double benchmark points (46000 vs 21020).

However, Qualcomm Snapdragon 410 is 3 years older than Samsng Exynos 7870, with a Lithography process that is also quite old (14 nm Exynos, 28 nm Snapdragon) they can not be considered as real competitors. In addition, both the SoCs have been thought for low/middle class devices. Moving to another Qualcomm board (e.g. Dragonboard 810c) the competition would have been largely won by the Qualcomm SoC, having AI unit specialized in parallel computations and an higher performance/watt ratio.
Consequently, the Dragonboard 410c has not been thought to overcome the tablet in computation power, but it has been seen as an alternative to improve the edge computing and ease the tablet work, reducing the whole energy consumption. Furthermore, performances have been drastically increased during the last years and it is possible to figure a future development board with Snapdragon SoC (e.g. Qual-

comm Snapdragon 810) that perfectly fits in the end EEG portable application.

Dragonboard 410c owns many features among which the meaningful ones are pointed out as per [29]:

- **Memory/ Storage**: 1GB LPDDR3 533MHz, 8GB eMMC 4.51, SD 3.0.
- **Connectivity**: WLAN 802.11 b/g/n 2.4GHz, Bluetooth 4.1, On-board BT and WLAN antenna.
- **I/O Interfaces**: One 40-pin Low Speed (LS) expansion connector that includes UART, SPI, I2S, I2C x2, GPIO x12, DC power; One 60-pin High Speed (HS) expansion connector that includes 4L-MIPI DSI, USB, I2C x2, 2L+4L-MIPI CS.
- **Power, Mechanical and Environmental**: Power: +6.5V to +18V. Dimensions: 54mm by 85mm meeting 96Boards™ Consumer Edition standard dimensions specifications. Operating Temp: 0°C to +70°C.
- **OS-support**: Android 5.1, Linux based on Debian, Windows 10 IoT core.

These features respect the requirements given by the end EEG portable application: in fact, the board has SPI interface needed to program and communicate with the ADS1299, sufficient memory to store the O.S. and the Java Real Time software, Bluetooth 4.1 to exchange data with the tablet/smartphone, it is small, low power, with a discrete computation capability and it supports Android.
This last feature represents the most challenging obstacle that has been encountered during the demo development, inducing myself to move to another board and elaborate a new server-base demo concept. The performed steps to work with Android and the resulting complications are discussed in the following subsection.

3.2.1 Android and Linux

First of all, Android is an Operating System characterized by its own Libraries and a Linux Kernel. Furthermore, even if the Kernel structure is similar to a GNU/Linux O.S. (e.g. Debian, Ubuntu), it does not posses all the classic GNU shell utilities (e.g. apt-get) or the GNU C Library (glibc).

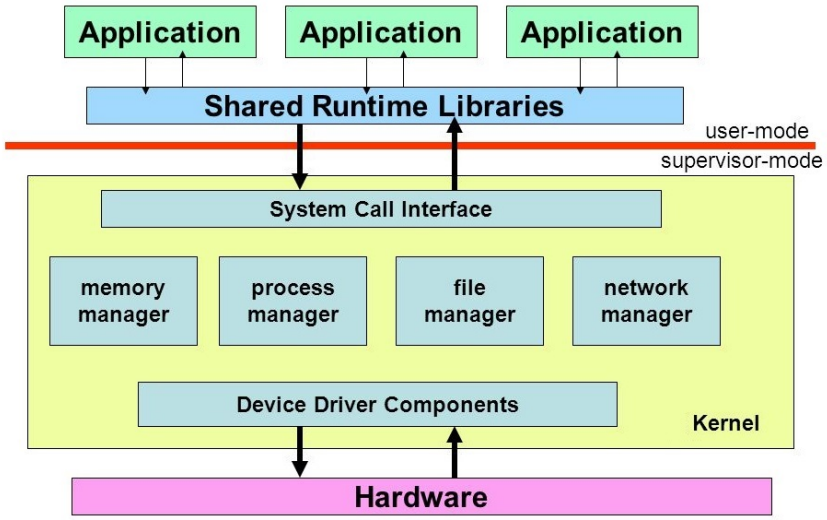

Figure 3.5: Modern OS Design

A modern Operating System can been split in two main parts as per Fig. 3.5:

1. **User Mode**: The system is in this mode when the OS is executing a user application on user interface. This section contains all the application that a user can run (e.g. browser, calculator, text editor), the application framework (e.g. activity manager, notification manager) and the various libraries proper of an operating system.

2. **Kernel or supervisor mode**: The O.S. boots up in Kernel mode and is able to run privileged instructions that can not be executed at user level. These commands are called System Calls and they depend on which Kernel is used. A user application may requires to execute commands at Kernel level (e.g. request to read or write the HW serial interface). The Application Programming Interface (API) allows a user level program to call one or more System Calls that are executed at Kernel space. Once it has been accomplished, the results can be extracted at user space. In addition, the Kernel space contains all the controllers that are closer to the Hardware (e.g. file manager, memory manager) and of course the device driver components.

Android has a Linux Kernel, this means that the structure is similar to a GNU/Linux O.S. and the System Calls may correspond as well. However, Linux is open source, this means that Google's Android developers can customize the Linux Kernel according to their requirements: this is one of the main differences by means it is possible to distinguish between an Android OS w.r.t. another (e.g. Samsung

Android and Xiaomi Android), even if they mount the same SoC.

Instead of executing ordinary Linux applications, Android makes use of Dalvik virtual machine to actually run applications written in Java. These apps are addressed to Android devices and specific application programming interfaces (APIs) rather than being targeted at Linux in general [30].

The APIs in the Kernel user space can be seen as a compound of System Call Interface and subroutines proper of a specific library (e.g. GNU C library glibc for GNU/Linux). The SCI (System Call Interface) is the cornerstone junction between an application and the Linux Kernel. In general, a System Call is a transition between user space to Kernel space and it can not be seen as a simple routine call. In fact, the execution switches from user space, characterized by its own stack, to Kernel space that owns another stack. At this point, depending on which System Call has been invoked, the address of the Kernel routine is extracted from a interrupt vector that contains all the addresses of System Call routines. The execution moves to that address (Program Counter jumps to that location) and the new stack will be used.

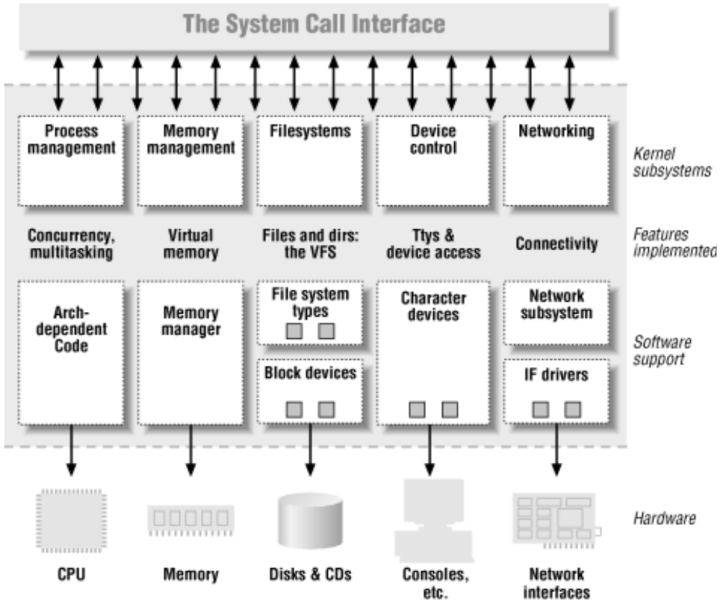

Figure 3.6: A split view of the Kernel [31]

System Calls are generally not invoked directly: in fact they are called via wrapper functions present in the system libraries. Making a comparison between typical Linux System Calls at [32] and a possible set of Linux Android ones at [33], it can be observed that many of them are exactly the same. This depends on the Kernel of course, that is still Linux.

Another important different between Android and GNU/Linux OS is concerned to drivers: a device driver is a loadable Kernel module that supervises between Hardware and the OS. Loadable modules are normally added during the boot or

they can be inserted or removed at runtime (e.g. on GNU/Linux can be used the shell command "insmod"). A device driver is a combination of C routines and data structures that can be accessed and used by other Kernel modules. These routines must use standard interfaces called entry points. By using this entry points, it is possible to interact with the device without knowing how the driver has been implemented and thus, how the device works.

As far as device drivers is concerned, Android shows slight differences w.r.t. typical Linux OS. In fact, on a GNU/Linux OS that runs on a laptop or on a development board such as Raspberry Pi, it is desirable that whatever peripheral is connected, it is recognizable and utilizable; this means that drivers for a generic interface (e.g. i2c or spi) must be written to deal with unlimited number of devices of the same type. Furthermore, peripheral interface registers need to be accessible at run time in order to fit with the connected device and react accordingly.

On Android OS this approach is different: since this OS is intended to run on a smartphone, where the various external peripherals are already connected when it delivers to the costumer, it is not necessary to create a generic driver that can deal with large number of devices of the same type. Contrariwise, a customized driver for specific peripheral is preferable to ameliorate the effectiveness of the software-hardware interaction that causes an improvement of the performances in therms of speed and power consumption. Consequently, each smartphone company that makes use of Android, must write specific device drivers depending on the chosen hardware peripherals such as Camera, microphone, speaker and so on.

As a result, in order to interact with an external SPI peripheral such as in this project case, a specific driver has to be written and the corresponding entry point needs to be added as well in the Kernel. In addition, the modified Kernel has to be re-built and flash into the board.

3.2.2 Working with Linux Android on Dragonboard 410c

As already mentioned, Dragonboard 410c supports three Operating Systems: Android 5.1, Linux based on Debian and Windows 10 IoT core. In this case, Android was appropriate since it offers the possibility to integrate the already existing EEG application. The board comes with Android installed by default; in case it is needed, there are two ways to install the Android Operating-system-image on the DragonBoard410 (as per [34]):

- **Installing the image from SD-card**: This is the easiest method to install a new Operating System on the DragonBoard and is recommended for users that are just getting started with the DragonBoard.
- **Installing from Host-pc**: This method is recommended for experienced users who will be downloading many iterative experimental versions of self-compiled OS's. Fastboot is a tool that communicates with the bootloader of the DragonBoard 410c and allows you to flash images onto the board, directly in its memory. Just like ADB (Android Debug Bridge), fastboot is included

in the Android SDK (Software Development Kit) [35].

Once the installation is completed, there are two ways to work with Dragonboard:

- **Working on the Dragonboard directly**

  This method represents the first one that has been utilized. It allows to work on the board by using keyboard, mouse and a monitor connected through HDMI port (exactly as on Raspberry Pi).

  First step to be performed in this case, is to download any application from Google store that allows to develop a Java app directly on the device. The app must be compatible with Android 5.1. In this work of thesis, "Jvdroid - IDE for Java" has been chosen. The primary idea was to develop a Java application to control a simple GPIO. Consequently, another app may be downloaded from the store to simplify the OS experience: "Termux". This application is a terminal emulator for Android that acts as the shell on a typical Linux environment but with a huge different: as mentioned before, Android does not own all the GNU shell utilities, indeed the number of commands that can be executed is limited. In addition, the navigation in root directories is forbidden by default; to overcome this issue, the Root of the Operating System is required. This guide [36] can be followed to complete the root on the Dragonboard (ADB is required for this).

  At this point, it is possible to give root permission to Termux and navigate inside the path */sys/class/gpio*. The files inside this directory can be seen as a virtualization of the real peripheral registers. The interaction between user and hardware is possible thanks to GPIO drivers that run in the Kernel, plus the device driver entry points that allows the peripheral to be accessible and visible at user space.

  In order to export a GPIO, an offset value of 902 must be added since the GPIO libraries of Dragonboard 410c dynamically allocates this offset. Consequently, if it is interested in exporting the GPIO 36 (pin 23 of the Dragonboard), write the GPIO number + the offset inside the file *export*: just type *echo* 938 > *export*. At this point a new folder has been created under the name *gpio*939. Navigate inside that folder to see new files such as *value*, *direction*, *edge* and so on. These elements represent the real GPIO properties. To enable GPIO output and make it high, type *echo out* > *direction* and then *echo* 1 > *value*. To read the GPIO input, *direction* must be *in* and the level can be picked up from the file *value*. To disable the GPIO, go back in the previous folder and type *echo* 938 > *unexport*.

  The command *echo* used in this way, allows to write a string inside a file (e.g. string "out" inside file *direction*). The same behaviour can be achieved on a

Java application that grants the root permissions. A Java app has been developed using "Jvdroid". The application on the Dragonboard screen appears like a normal app downloaded from the Google store. A simple button has been added on the GUI to turn-on and turn-off the GPIO.

This experiment is quite interesting and allows to get familiar with Android world. The same approach is not sufficient to work with a SPI peripheral: as explained before, a specific driver has to be written and the corresponding entry point needs to be added as well in the Kernel. In addition, Android Studio may be used to simplify the work and develop much more complex applications.

- **Using the Android Debug Bridge**

  ADB is a flexible command-line tool that allows to communicate with an Android device. The adb commands simplifies a large number of device actions, such as installing and debugging apps, and it provides access to a Unix shell that can be used to run a variety of commands on a device.

  It is a client-server program that includes three components: a client, which sends commands, a daemon (adbd), which runs commands on a device and a server, which manages communication between the client and the daemon [37].

  This method is the most effective to develop an application (by using Android Studio and installing the apk through adb) and deal with the Dragonboard. The GPIO experiment cited in the previous point, can be performed as well through adb without performing any Root of the OS. However, to develop an application that requires super user permissions, the Root may be needed.

The second method is what I personally recommend to use. However, if you want to perform some initial tests or it is the first time approaching with an Android application, the first method may be preferred.

As already discussed in the previous paragraph, Android does not posses a common driver that allows to interface with a generic SPI peripheral. This means that a specific driver and its correspondent entry point must be added to the Kernel to communicate with a distinct device, ADS1299 in this case.

In order to modify the Linux Kernel of Android OS on Dragonboard, there are some steps that needs to be followed (as per [38]):

1. **On a Ubuntu Machine, download HLOS software**: Open source HLOS (High Level Operating System) software for Qualcomm Snapdragon chipsets is available on the Linux Foundation hosted site *www.codeaurora.org*. First of all, ensure that the release is intended to download, matches the board support package. For example, if this is the chosen release:
   $linux\_android\_board\_support\_package\_vla.br\_.1.2.4 - 01810 - 8x16.0 - 2.zip$

then there will be a manifest called $LA.BR.1.2.4-01810-8x16.0.xml$ that has to be compatible with the Dragonboard 410c.
This series of letters and numbers is called the APSS (Application processor subsystem software) Build ID.

Before continuing with the next steps, make sure that there are at least 90GB of free space in the disk. Once it has been decided the manifest is wanted, navigate inside an empty folder and type the command:

```
$ repo init -u git://codeaurora.org/platform/manifest.git
-b release -m [manifest] -repo-url-git://codeaurora.org
/tools/repo.git
```

Then type the command:

```
$ repo synch
```

Once the download is completed, unzip the file. The extracted folder represents the full directory where it is willed to build Android images. Navigate inside the folder and type the commands:

```
$ chmod a+x DB410c_build.sh
$ ./DB410c_build.sh
```

This operation takes much time and it is required to build the Android HLOS first time. As per [38], source code clone and build takes about an hour or longer depending on the configuration of the Ubuntu machine; in reality, it can takes one full day or more depending on how many errors occur during this stages. In that case, search online and check how to fix them at every step.

2. **Add driver and entry point**: The generated directory represents a full unpacked Operating System that contains all the files required to build it and create the Android image. Between them, there are also source codes for the Kernel and so, all the drivers indispensable to interface with the Snapdragon hardware. Find these source codes and add drivers for the desired device (SPI on ADS1299 in this case). Finally, add the device entry point in a source code to the path:

   ```
   $ kernel/arch/arm64/boot/dts/qcom/
   ```

3. **Build the modified Kernel**: At this point, the modified version of the Kernel needs to be built. To perform this step, navigate in the main directory

   ```
   $ cd $BUILDROOT
   ```

   and type:

   ```
   $ make -j8 -> to build everything
   ```

   There is also specific command to build the Kernel boot image only:

   ```
   $ make -j8 bootimage -> to build kernel boot.img
   ```

4. **Flash the new Kernel image using fastboot**: At this last stage, the updated Kernel image has to be flashed into the board. Bring the Dragonboard 410c and hold down the VOL- key, connect the DC supply to the DragonBoard 410c, and press the power ON button. Plug the USB cable into the target. This will bring the device into fastboot mode. Navigate in the following directory:

   ```
   $ cd <$BUILDROOT>/device/out/host/linux-x86/bin
   ```

   and run:

   ```
   $ sudo fastboot devices
   ```

   in order to verify if the device has been recognized. Eventually, type the command:

   ```
   $ fastboot flash boot <path to boot.img>
   ```

   After all, the updated Kernel version has been flashed.

Flashing the Kernel only may be not enough. In order to complete the entire setup, follow the full guide at [38].

During this work of thesis, all these steps have been correctly completed, except for number 2. In particular, the total absence of documentation and the low knowledge in Kernel drivers, have made this phase too difficult to be performed. Furthermore, it has been finally decided to quit with this board and moving to Raspberry Pi 3. A restricted number of drivers can be added by following this video [39], but this array does not contain the desired one.

However, with this chapter, it is intentional to provide a starting point to work with the Dragonboard and leave a useful documentation to someone who wants to approach with this electronic.

3.3 Raspberry Pi 3 and Unix Laptop

The Raspberry Pi is a single board computer, offering features such as USB ports, GPIO pins, WiFi, Bluetooth, USB and network boot and is capable of performing most of the tasks that a regular PC can. Its compact size and affordable price make it very attractive for embedded application such as the present work of thesis.

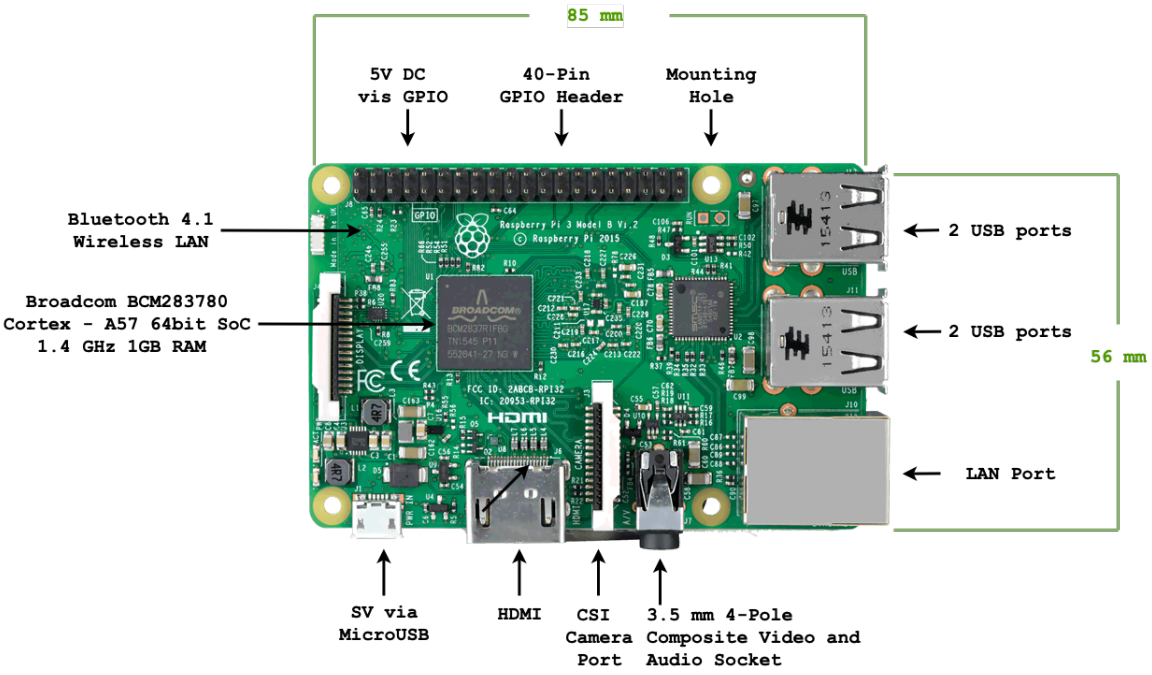

Figure 3.7: The Raspberry Pi 3 single board computer

The most notable specifications of the model 3B are:

- Quad Core 1.2GHz BCM2837 64bit CPU
- 1GB RAM
- 40-pin extended GPIO
- Integrated BCM43438 WLAN and BLE module

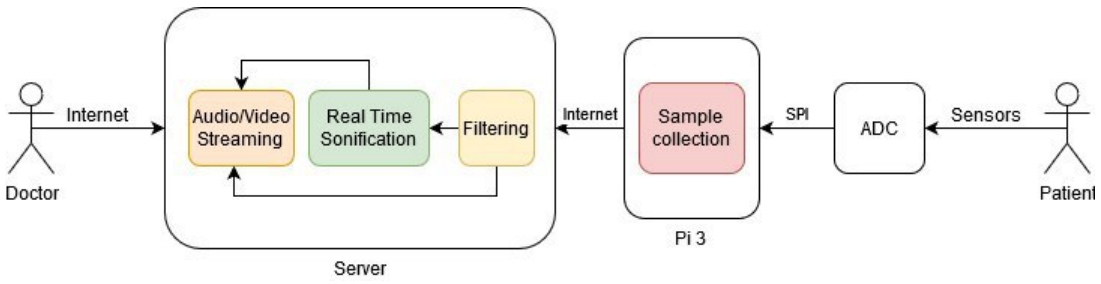

Figure 3.8: Demo Concept Diagram

In this demo concept, a server-client structure has been chosen. The client is represented by Raspberry Pi, which role is to receive the sampled data acquired by

the ADS1299, through SPI. Consequently, data packets are sent over the network every time t that depends on the sampling rate of the ADS1299. At this point, the data are collected by a server which processes them by applying filtering and Sonification algorithm while Real Time behaviour is ensured. Eventually, video and audio are streamed on a web page accessible by authorized doctors only, as it is shown in Fig. 3.8.

The picked server solution was to make use of a laptop PC running a GNU/Linux OS, and have it run a custom Python GUI handling the data processing instead of streaming on a web page.

The entire flow has been also emulated by connecting the laptop (server) to an external network w.r.t. the one of Raspberry Pi (client). In order to realize this scenario, the PC has been placed at home, 3 Km far from the Pi location (UCC University).
Since the PC was connected through a residential internet connection, the appropriate ports had to be forwarded on the host router to allow for communication to and from the server (Port Forwarding).

Port forwarding, or tunneling, is the behind-the-scenes process of intercepting data traffic headed for a computer's IP/port combination and redirecting it to a different IP and/or port [40]. To achieve it, the router settings of the home network need to be correctly set up by enabling the port forwarding option and setting the port number to unique value. In the routing table, this value is linked to the local IP address of the laptop, in order to guarantee that the incoming traffic directed to the server, is properly received. It is necessary to underline that the latter port number can not be chosen as 8080 or 8088, since they are already used for TCP protocol. Finally, packets that are sent by the Raspberry require to be addressed accordingly the public IP address of the home router.

Lastly, the correct behaviour was verified providing samples to the acquisition unit and verifying that they were correctly broadcast to the host via a VNC remote desktop connection.

It is fundamental to highlight that this implementation suggests just a concept idea, indeed, many choices have been made for simplifying the initial work, such as the use of Raspberry Pi or the laptop. Further adjustments are contemplated to optimize the application and provide a system that is the best trade off in terms of power, cost and performances.

Chapter 4

Implementation

This chapter is focused on the demo implementation from a Software point of view. Various phases of the work are discussed, pointing out the most significant achievements that have been obtained. In particular, diverse challenges are identified and possible solutions are proposed. Python is the programming language chosen to accomplish the purpose despite a series of Operating Systems interactions have been exploited to carry out a suitable application. Especially, Real Time behaviour has been classified as the most crucial dare and multiprocessing approach has been adopted and analysed as a potential solution. Eventually, a final demo implementation has been proposed after a series of refinements introduced during the work of thesis. It is important to specify that the software implementation and all the related functions, have been realized in generic way in order to deal with a number n of EEG signal. However, for experimental purpose, only one input signal has been taken in analysis to facilitate the application development. Small modifications at the code may be applied to enable the acquisition and processing of multiple n channels.

4.1 ADS1299 SPI Communication

The SPI (Serial Peripheral Interface) is a synchronous bus interface generally adopted to connect integrated circuits on the same board or to send data between microcontrollers and small peripherals such as sensors, shift registers and SD cards. It makes used of master-slave architecture where the master is unique and the number of slaves is not limited in principle but normally it does not exceed 4 or 5 devices. Main SPI protocol characteristics are:

- **Full duplex transmission**: it owns one wire for data from master to slaves and one wire for data from slaves to master. Master and slaves are always both transmitters and receivers.
- **4 wires**:
    1. SPICLK: the clock, always generated by the master.
    2. MOSI: Master Out Slave In, the output data line of the master, and input data line of slaves.
    3. MISO: Master In Slave Out, the input data line of the master, and output data line of slaves.

4. SPISS: the slave select, connected to the Chip Select (CS) input of slaves, and driven by the master. System Call might be multiple SPISS when connecting to several slaves.

- **Transmitting data**: A predefined number of bits is transferred in each transmission. One bit is sent per clock cycle using the NRZ encoding (Non Return to Zero). SPICLK is idle when there is no transmission. The Idle level depends on clock polarity:

  - CPOL = 0: clock is low when idle
  - CPOL = 1: clock is high when idle

  The Slave Select of the target slave is activated during the entire transmission. All other slaves with the SS not active keep their output (on MISO) in high impedance and ignore what happens on MOSI and SPICLK.
  Master generates n clock cycles where data is transmitted MSB first and it changes on a clock edge, and is sampled on the opposite clock edge:

  - CPHA = 0: data on trailing edge, sample on leading edge
  - CPHA = 1: data on leading edge, sample on trailing edge

  Slave Select is deasserted to complete the transmission.

- **Master and slaves shall agree on**:

  1. Number of bits in a transmission
  2. Clock frequency: clock generated by master, received by slaves. Slaves should support the master clock frequency.
  3. Clock Polarity (CPOL)
  4. Clock Phase (CPHA)

The ADS1299 SPI-compatible interface consists of four signal $\overline{CS}$, SCLK, DIN, and DOUT. The interface reads conversion data, reads and writes registers, and controls ADS1299 operation.

$\overline{CS}$ (SPISS) requires for activating the transmission: it has to go low before starting the communication and must remain low for the full data transaction. When $\overline{CS}$ is high, DOUT is in high impedance.

SCLK (SPICLK) represents the clock for serial communication. SCLK is a Schmitt-trigger input, but TI recommends keeping SCLK as free from noise as possible to prevent glitches from inadvertently shifting the data. Data are shifted into DIN on the falling edge of SCLK and shifted out of DOUT on the rising edge of SCLK [8].

DOUT and DIN are the MISO and MOSI respectively. First one provides data from ADS1299 to the master, second one from the master to the ADS1299.

Furthermore, another signal called data-ready ($\overline{DRDY}$), is used as a status signal to indicate when data are ready. $\overline{DRDY}$ goes low when new data are available. Full

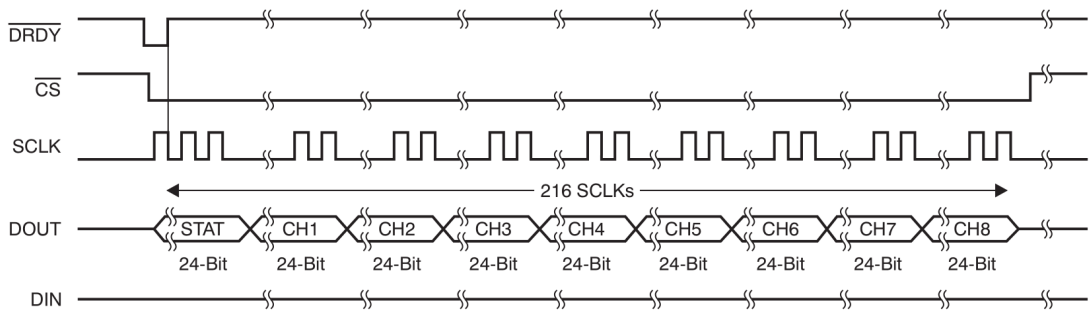

Figure 4.1: SPI Bus Data Output [8]

SPI data out transaction is showed in Fig. 4.1.

There are other 3 signals that has to be controlled in order to perform the Power-up sequence and start the acquisition:

1. START: this command starts data conversions. Pull the START pin high for at least 2 t CLK periods, or send the START command to begin conversions. When START is low and the START command has not been sent, the device does not issue a DRDY signal.

2. $\overline{RESET}$: the RESET command resets the digital filter cycle and restores all register all the default values for register settings.

3. $\overline{PWDN}$: when power-down signal is driven low, all the chip is powered off. In order to power the chip up again, drive the PWDN pin high.

These last three signals may be sent both through SPI or physically by using specific pins on the board.

Before starting to acquire samples, it is necessary to perform a initial flow at power up that requires many of the signals discussed until now, showed in Fig. 4.2.

As already mentioned, every piece of application has been implemented using Python. In this particular SPI implementation, the Python module called ”spidev” has been exploited for interfacing with SPI devices from user space via the spidev linux kernel driver. The function intended to initialize the SPI device is reported below:

```
spi = spidev.SpiDev()

def ads_setup():
    spi.open(0,0)
    spi.max_speed_hz = 2500000
    spi.mode = 0b10     #CPOL | CPHA
    GPIO.output(CS, GPIO.HIGH)
    GPIO.output(RST, GPIO.HIGH)
    GPIO.output(PWDN, GPIO.HIGH)
    GPIO.output(START, GPIO.HIGH)
    time.sleep(0.010)
```

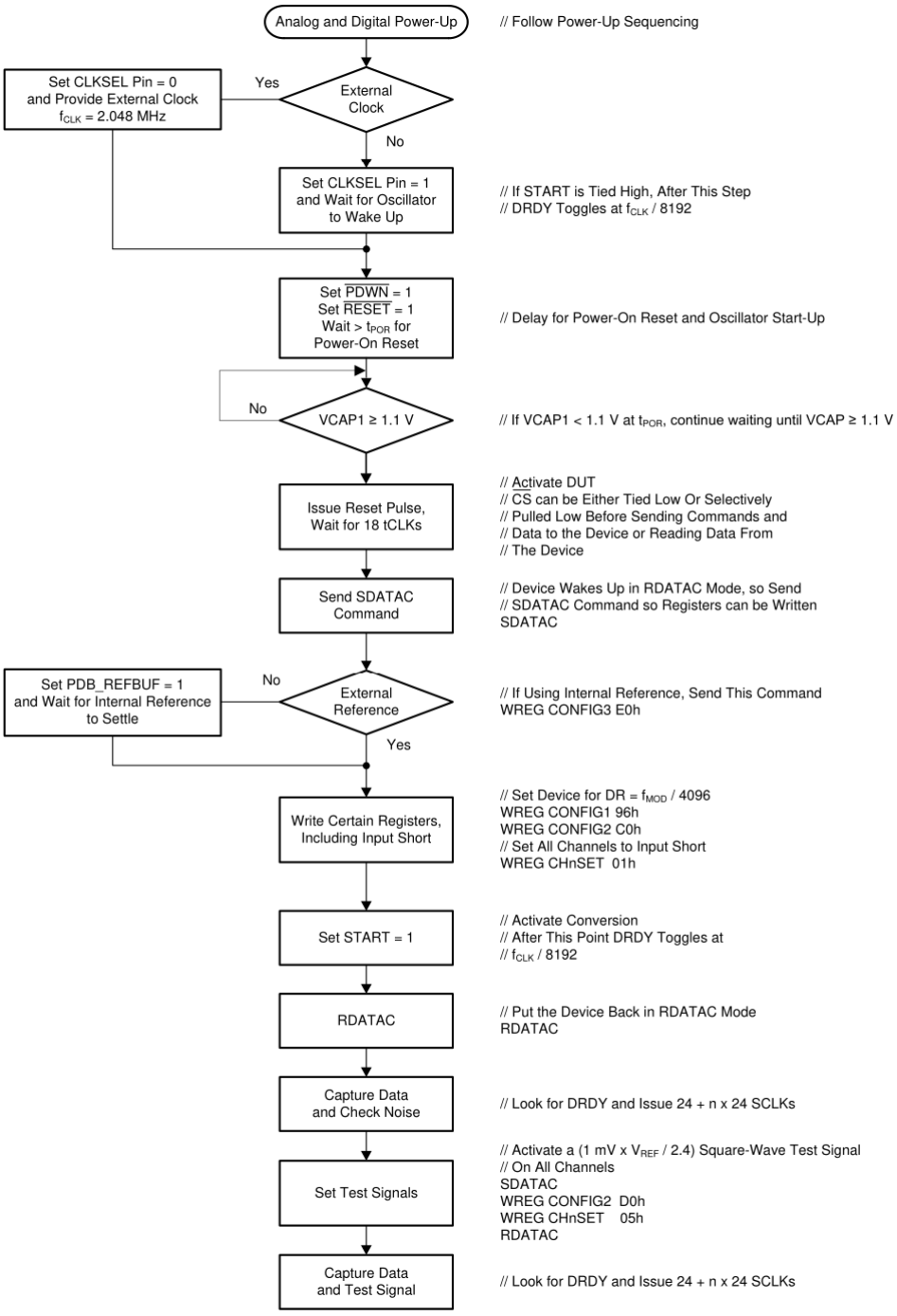

Figure 4.2: Initial Flow at Power-Up [8]

Function *open* of the module spidev, allows to connect to the specified SPI device, opening "/dev/spidev<bus>.<device>", *bus* 0 and *device* 0 in this case. Right after, other 2 parameters are set where *max_speed_hz* defines the maximum frequency for SCLK signal and *mode* provides the CPOL and CPHA of the serial communication. As per datasheet suggestion, CPOL = 1 and CPHA = 0 (clock high when idle, data on trailing edge and sample on leading edge).

Furthermore, after device initialization, the flow in Fig. 4.2 has been followed in

order to complete the power-up sequence. In particular, at the step "Write Certain Registers, Including Input Short" all ADS configurations need to be setup. To do that, a specific function has been written in order to facilitate the writing of internal ADS registers:

```
def ads_wreg(address, value):
    opcode = 0x40 + address
    GPIO.output(CS, GPIO.LOW)
    spi.xfer2([0x11])
    spi.xfer2([opcode])
    spi.xfer2([0x00])
    spi.xfer2([value])
    spi.xfer2([0x10])  # turn read data continuous back on
    GPIO.output(CS, GPIO.HIGH)
```

This function takes the *address* and *value* as arguments and write the value specified in the ADS1299 register at that address. This operation takes advantage of what in ADS datasheet is called *WREG* command (Write to Register): this is a a two-byte command followed by the register data input. The first byte contains the command and register address. The second command byte specifies the number of registers to write – 1.

Specifically, first command byte has formed by 010r rrrr, where r rrrr is the starting register address: this explains why in the code above, the opcode is chosen to be *address* + 0x40. Once the address is selected and the second byte is sent (0x0 to indicate zero number of registers to write – 1), the desired *value* is sent as well (spi.xfer2([value])). It is important to point out that before starting the communication, $\overline{CS}$ goes low to select the SPI device and *SDATAC* command is sent. The latter command is required to stop reading data in continuous mode, a setting that when enable, allows to constantly acquire EEG samples. At te end of the communication, $\overline{CS}$ returns high and *RDATAC* command is sent to turn read data continuous back on.

By using the created *ads_wreg* function, has been simple to initialize the ADS1299 device depending on the desired needs (all possible register configurations are showed in [8]). After the last step, EEG acquisition can finally begin.

In order to improve the computation at the edge, the $\overline{DRDY}$ signal as been treated as GPIO Interrupt source and a specific ISR (interrupt service routine) has been linked to it. From the moment that $\overline{DRDY}$ is active-low, the GPIO interrupt has been set to be triggered on the falling edge of the signal:

GPIO.add_event_detect(DREADY, GPIO.FALLING, callback=ISR)

The main role of the ISR is to collect samples coming from the channels and send them to the server. For experiment purpose, only one channel data are sent per time despite the application has been thought to deal with n number of channels. In addition, it has been chosen to send *N_SAMPLE* of data per time, to improve the filtering and Sonification Real Time processing at the expense of the latency.

In order to do that, the following function is called to act as ISR, every time that $\overline{DRDY}$ goes low.

```
def ads_acquire(to_client):
 GPIO.output(CS, GPIO.LOW)
 spi.xfer2([0x12])
 dataPacket = 0
 for i in range(0,9):      # up to 9, the first is the slave ID
  for j in range(0,3):  # up to 3 ==> 24 bit

   dataByte = (spi.xfer2([0x00]))
   dataPacket = (dataPacket << 8) | int(dataByte[0])

  if i == 1 :
   input = twos_comp(int(hex(dataPacket), 16), 24)

  dataPacket = 0

 GPIO.output(CS, GPIO.HIGH)
 input_V = input

 not_filtered[0][acquisitions] = input_V

 acquisitions += 1

 if acquisitions == N_SAMPLE :
  to_client.send(not_filtered)
  not_filtered = np.empty([1,N_SAMPLE])
  acquisitions = 0
```

As it is possible to observe from the code, before the serial transmission takes place, $\overline{CS}$ always goes low and it returns high only at the end of the communication. The RDATA command (0x12) is sent to the ADS1299 and the acquired data are delivered: they are accessible thanks to the return of the function *spi.xfer2([0x00])*.

The acquired data are then processed in two *for loops*: first one iterates the 8 channels plus the slave ID, the second accumulates the 24 bits in three times (1 byte per time). As already mentioned, only one channel is considered and this explains why there is a *if i == 1* condition that corresponds to channel 1. This behaviour is emphasized in Fig. 4.1.

Finally, N_SAMPLE acquisitions of the channel 1 are sent to the client process in charge to forward the data over the network. The Python module *socket* has been used for this purpose.

4.2 Real Time Processing

One of the most crucial challenge of this implementation is to achieve Real Time behaviour between Signal Processing, Sonification and Video-Audio Streaming. Each of this operation requires high computational activity (specially with data coming from 8 different channels) and it is hard to ensure a continuous flow by treating this services as unique piece of application.

Furthermore, the idea of splitting the operation in different instances, has been seen as a possible solution. In particular, the Real Time has been achieved by using multiple processes approach: various operations are treated as separate process in order to intensify their own activity, leaving the hard work to the OS scheduler.

The various processes exchange data through pipes: a pipe is a FIFO that allows to send data between 2 different processes (that do not share the same memory space like threads). In this application, each pipe has a default dimension of 2 elements. A process can write in a pipe only if it contains at most one element; similarly, a process can read a pipe if it contains at least one element. In all the other cases the current process stops until these conditions are verified. Using this approach it has been managed to guarantee Real Time without overloading one piece of application that requires high performances. In addition, the properties of the pipes have been exploited to ensure synchronization between processes other than Real Time.

Below, the piece of code that is required to fork main process (parent) in different subprocesses (children):

```
if __name__ == "__main__":
 # create 3 processes (all parallel children)
 for i in range (0,3):
  new_pid = os.fork()
  if new_pid == 0 and i == 0:
   RealTime.run(for_run, to_audio, to_plot)
  elif new_pid == 0 and i == 1:
   RealTime.play(for_audio)
  elif new_pid == 0 and i == 2 :
   RealTime.plot(for_plot)
 # only parent arrives here
 server(to_run)
```

Fork is a system call proper of Linux and in this case, a specific Python library has been used to exploit this OS property. Specifically, the Python module uses a wrapper from Python to C language that permits to handle the system call.

As it is showed in the code above, main process splits in 3 different subprocesses. The fork system call, returns the PID (Process ID) that is 0 for the child and it is a value different from zero for the parent. Consequently, it is checked if the current PID is 0 to call the relative child routine (that depends also on the index i) and only the parent will arrive to the end since each routine is enclosed in a while(1) (children will never exit from it). In addition, to each routine (process) is also passed the head and/or the tail of a Pipe as parameter, depending on the needs.

4.2.1 Filter Processing

As discussed in Paragraph 2.2.2, the Filter phase is a fundamental part in the entire flow that is required for both Sonification and Video Streaming to remove noise and artifacts influences on the EEG input signals. Consequently, it is desired that the incoming EEG is subjected to three different filters: Low-Pass, High-Pass and Notch.

All filter transfer functions are derived from analog prototypes in the S-domain to the respective digital transfer functions in the Z-domain by using the Bilinear Transform (BLT) [41]. Thus, the digital transfer function (in the Z-domain) can be defined as:

$$H(z) = \frac{A(z)}{B(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}} \tag{4.1}$$

Starting from this generic filter transfer function, it is now necessary to evaluate various a and b coefficients depending on which filter has been taken into account. Before defining them, it is required to introduce some parameters:

- F_s : Sampling Frequency (e.g. 250Hz in this implementation).
- f_0 : Center Frequency or Corner Frequency that depends on the filter type.
- ω_0 : Normalized pulsation defined as $\omega_0 = 2\pi(\frac{f_0}{F_s})$.
- Q : Quality Factor, that has been chosen equal to 0.7071.
- α : defined as $\alpha = \frac{sin(\omega_0)}{Q}$.
- z_i : initial condition for the filter.

Python filter function that has been implemented in this work of thesis, takes the input signal, sampling frequency, the various corner frequencies of the three filter types and the initial condition z_i as arguments:

```
def eegFilter(Sig, Fs, FHP, FLP, FN, zi)
```

The filter coefficients have been evaluated in the following way:

High Pass Filter:

$$\begin{array}{lll} a_0 = 1 & a_1 = \frac{2\pi f_0}{Fs} - 1 & a_2 = 0 \\ b_0 = 1 & b_1 = -1 & b_2 = 0 \end{array}$$

Low Pass Filter:

$$a_0 = 1 \qquad a_1 = 0 \quad a_2 = 0$$
$$b_i = \frac{w_i}{\sum_i^{N+1} w_i}$$

where $\overline{w}$ is equal to the sinc of n, smoothed by using Hanning window (half cosine); n is defined as a vector with length N+1, containing elements between $\frac{-Nf_0}{Fs}$ and $\frac{Nf_0}{Fs}$; N is set to 64. Consequently, $\overline{b}$ is a vector of dimension N + 1.

Notch Filter:

$$a_0 = 1 + \alpha \quad a_1 = -2cos(\omega_0) \quad a_2 = 1 - \alpha$$
$$b_0 = 1 \qquad b_1 = -2cos(\omega_0) \quad b_2 = 1$$

and normalization on Notch coefficients has been applied.

In addition, all the coefficients are merged in 2 single coefficients by applying polynomial multiplication. The evaluated values, are passed to the following Python function:

output, zi = signal.lfilter(coeffB, coeffA, Sig, zi = zi)

where *lfilter* is a function belonging to *scipy* Python module that performs digital filtering to the input signal *Sig* given the coefficients of the polynomials B(z) and A(z), *coeffB* and *coeffA* respectively, and the initial condition z_i. Finally the output filtered signal and the new filter condition are returned and the latter one will be used in the following filtering.

The implemented function *eegFilter* has been used in one of the three child processes within Sonification algorithm. This operation is called every time a new packet of data is sent from the client to the server. Consequently, the generated output is used to be processed by Sonification and, at the same time, it is sent to the Video Streaming child process through a Pipe.

4.2.2 Sonification

First of all, the filtered signal needs to be resampled before applying Sonification. In order to do that, a specific function has been implemented:

```
def resample(x, fs_out, fs_in, zi) :
 USF = int(fs_out/gcd(fs_out, fs_in)) #upsample factor
 DSF = int(fs_in/gcd(fs_out, fs_in))  #downsample factor

 if USF > 1 :
  x_us, z_up = up(x, USF, z_up)
 else :
  x_us = x
 if DSF > 1:
  y, z_down = down(x_us, DSF, z_down)
 else :
  y = x_us

 zi = [z_up,z_down]

 return y,zi
```

This function takes as input the actual sampling frequency (e.g. 250 Hz), and the desired output frequency. By evaluating the upsample factor and the downsample factor, it is possible to determine if the incoming signal requires to be upsampled first or/and downsampled after, depending on the input and output frequency ratio. The associated upsample and downsample functions, have been implemented by exploiting the Hermite Windows.

Furthermore, the Sonification may be applied. The first step of this algorithm, as per Fig. 2.6, is the compressor phase in charge of reducing the dynamic range of the input signal with minimal distortion. To achieve it, the envelope of the signal is evaluated as the convolution of the absolute value of the downsampled signal, using an exponential impulse response $h_e(n)$:

```
def envelope (x, fs, As, Ds, S, Rs, zi):
 A = int(np.max(round(As*fs),0))
 D = int(np.max(round(Ds*fs),0))
 R = int(np.max(round(Rs*fs),0))
 S = int(np.max(np.minimum(S,1),0))

 NCH = x.shape[0]
 L = x.shape[1]

 if A+D+R > 0:
  #FIR Envelope
  if A+D > 0:
   e_dB = np.zeros(A+D)
   e_dB[:A] = np.linspace(-90, 0, A)
   e_dB[A:A+D] = np.linspace(0, 20*math.log10(S),D)
   e = np.power(10, e_dB/20)
   e = e/sum(abs(e))
  else:
   e = [1,0]
```

```
    deltadB = 90.0/20
    alphaR = math.pow(10, - deltadB/R)
    y, zi = signal.lfilter((1-alphaR)*e*2,[1,-alphaR], np.power(x,2), zi
        = zi)
    y = np.sqrt(x)
  else:
    y = abs(x)
  return y, zi
```

By implementing the above Python function, it has been possible to apply the envelope on the input signal and generate a smooth curve enhancing its extremes, that is exploited to control the gain reduction. Compressors operate on a logarithmic scale, for this reason dB conversions have been employed.

Successively, a new operation has been implemented to limit the envelope output: referring to the Paragraph 2.2.2, it is explained how the signal must be limited after compression. This operation is needed since the amplitude of the envelope may be under a certain threshold, resulting with a missing gain reduction. Contrariwise, when a threshold is exceeded, the amplitude of the signal is reduced proportional to the compression ratio R. Consequently, a function called *soft_limit_indep* has been implemented and this applies what is showed in formula (2.2).
Similarly, the signal is amplified as per equation (2.3) to overcome the aliasing problem that can be issued in Modulation stage. This step has been included in *soft_limit_indep*.

The previous *envelope* function has been embedded in *soft_limit_indep* as well, in order to deal with a unique operation that returns both soft limited and envelop output.

Third and last phase is Modulation, implemented as follows.

```
  y_rs, z2 = Sample.resample(soft_limited, int(round(FS_OUT*r)), FDS,
      z2, first_round)
  env_rs, z3 = Sample.resample(env, int(round(FS_OUT*r)), FDS, z3,
      first_round)

  w = y_rs
  w [w > audio_lim]= audio_lim
  w [w < -audio_lim] = -(audio_lim)
  w = (w / audio_lim + 1)/2

  w = 30.0*np.power((3000/30), w) / (FS_OUT/2)
  z = np.cos(np.pi * np.cumsum(w) + np.pi* state_mod)/audio_lim

  state_mod = (state_mod + np.sum(w))%2

  z = z*env_rs

  z, foo, z4 = son.soft_limit_indep(z, FS_OUT, 1.0/(math.sqrt(2)), 1.5,
       1, 1.0/20, 0, 1.0/20, z4, first_round)
```

Firstly, a resample is performed in both branches (*env* and *soft_limited*), returned by the function *soft_limited* in order to achieve the final audio sampling rate. The signal *y_rs* and *env_rs* are obtained by resampling *soft_limited* and *envelop* signals respectively.

After resampling, *y_rs* can get outside the boundaries [-*audio_lim* , *audio_lim*] despite the fact the original signal was inside the limits. Following operation forces $|y\_rs| < audio\_lim$. At the same time, a normalization is applied to the new signal w in order to let it fitting within the interval [0,1].

Furthermore, exponential transformation and FM modulation are performed on the signal since the human hearing system perceives frequency on an exponential scale, obtaining the signal z. Moreover, the current state of the modulation (*state_mod*) is saved in order to process the frame of audio continuously for every next iteration.

The AM modulation is performed by multiplying the previous *env_rs* with the current signal. In this case, the envelop is used to modulate the amplitude of the actual signal.

Finally, another envelope and soft limit are applied in order to ensure that final signal won't be outside the [-1,1] interval. This last step has been performed to avoid the phenomena of clipping, something that may happen when pushing the gain of a signal past the capabilities of the gear handling that signal [42].

At this point, as at the end of the Filter processing, the final output is sent to the audio streaming process through Pipe, ensuring Real Time.

4.2.3 EEG Video and Audio Streaming

Once the EEG has been filtered and processed by Sonification, the two branches of it are sent to Video and Audio child process respectively. This link between different processes is possible by the presence of Pipes that ensure Real Time behaviour.

Video and Audio operations are quite expensive in therms of computational power and, other than ensuring a fluent application flow, they require to run in concurrency. For this reason, they are treated as separated parallel processes that run at the same time. In particular, the Pipe transaction happens at the end of the first child process, that in charge of filtering and Sonification.

- **Video Streaming Routine**

```
def plot(for_plot):
 line_F = []
 sync = 0
 while True:
  filtered = for_plot.recv()
  line_F = live_graph.live_plotter(filtered,line_F,identifier='EEG
      Signal')
```

The above piece of application shows the plot routine enclosed in a *While True*. As it is possible to observe, the function takes the tail of one of the Pipe and it uses the attribute *recv()* to receive the data.

It is important to remember that this process is created together with the other two children but during the first iteration of the routine, it will hang until the Pipe (dimension = 2) is filled with some data. This means that the Video will be displayed with certain latency (same as Audio).

Once first data is received, the EEGs start displaying on a Python GUI created by using the module *pyplot*. This procedure is contained in the *live_plotter*, specifically implemented fort this purpose.

This operation has been quite challenging to perform, since the latter function needs to take into account the frame rate of the laptop display other than ensuring a fluid streaming. For this reasons, internal delays present in *live_plotter*, have been let modifiable in order to adapt the streaming on different devices, depending on the frame rate, computational capabilities of the graphic board and the dimension/frequency of the data buffer.

- **Audio Streaming Routine**

```
def play(for_audio):
 p = pyaudio.PyAudio()

 def callback(data, frame_count, time_info, flag):
  global first_round_audio

  com_data = for_audio.recv()

  if first_round_audio:
   time.sleep(audio_delay)

  first_round_audio = 0
  data = com_data.astype(np.float32)

  return (data, pyaudio.paContinue)

 # open STREAM using CALLBACK
 stream = p.open(format=pyaudio.paFloat32,
      channels=1,
      rate=FS_OUT,
      output=True,
      frames_per_buffer=N_RESAMPLE,
      stream_callback=callback)

 # start the STREAM
 stream.start_stream()

 while stream.is_active():
  time.sleep(0.001)
```

The module pyaudio has been exploited for playing EEG sound on the laptop. Once the object has been instantiated, the corresponding stream is constructed. A stream in a object oriented language, is a sequence of data in which all the specified methods are applied subsequently to each frame of the flow. In this case, it acts as an input stream, that behaves like a source for the audio drivers [43].

The stream invokes a specific callback that is required to get audio data. As per video streaming, the tail of a Pipe is passed to the process, and the action of reading the Pipe is located inside the callback: in this way, once the Pipe is filled, the callback returns audio data and the stream makes sure that the corresponding sound will be played.

This process remains busy until the audio data have been all played. To ensure Real Time and continuous flow without interruption, it is necessary to keep the Pipe always full with at least one packet of data. In this way, once the stream has been consumed, the callback can provide a new packet right after without interrupting the flow. To ensure this behaviour, a small delay can be added in the first iteration.

This two procedures run in parallel thus video and audio are streamed at the same time. An example of a frame of the GUI is showed in Fig. 4.3.

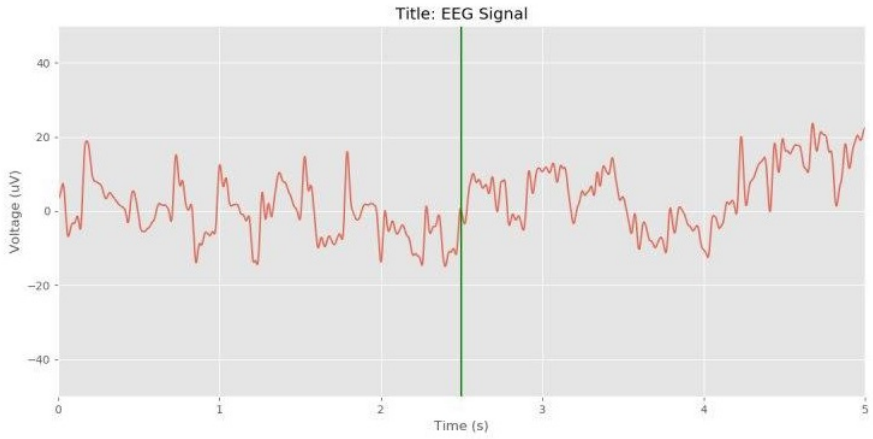

Figure 4.3: Python GUI for Video and Audio Streaming

This is a dynamic graph that slides from the right to the left. Audio and Video are synchronized on a green line (as per figure above) and they will keep matched as soon as the Pipe properties are respected. An initial delay to shift the phase of the audio signal is necessary to guarantee synchronization.

As discussed, audio and video are subjected to not-negligible latency. In particular, the minimum delay they are both affected, is given by this formula:

$$t_{lat} = t_{acq} + t_{net} + t_{filt} + t_{son}$$

where t_{acq} depends on the length of the data buffer (related to the sampling rate of the ADS1299), t_{net} is the delay introduces by the network transmission, t_{filt} and t_{son} are delays imported by filter and Sonification algorithms respectively.
In order to ensure Real Time and keep the Pipes always filled, it is necessary that:

$$t_{acqu} > t_{net} + t_{filt} + t_{son}$$

As it is possible to observe, the latency may be very large (> 1 second or more). However, this application does not require quick response: in fact, from the moment that newborns do not show any clinical manifestations, it is not necessary being synchronize between infant movements and audio/video streaming.

4.3 Full Demo

In this paragraph it is summarized the flow of the entire implementation. All the various application instances have been linked together and the corresponding diagram is showed in Fig. 4.4.

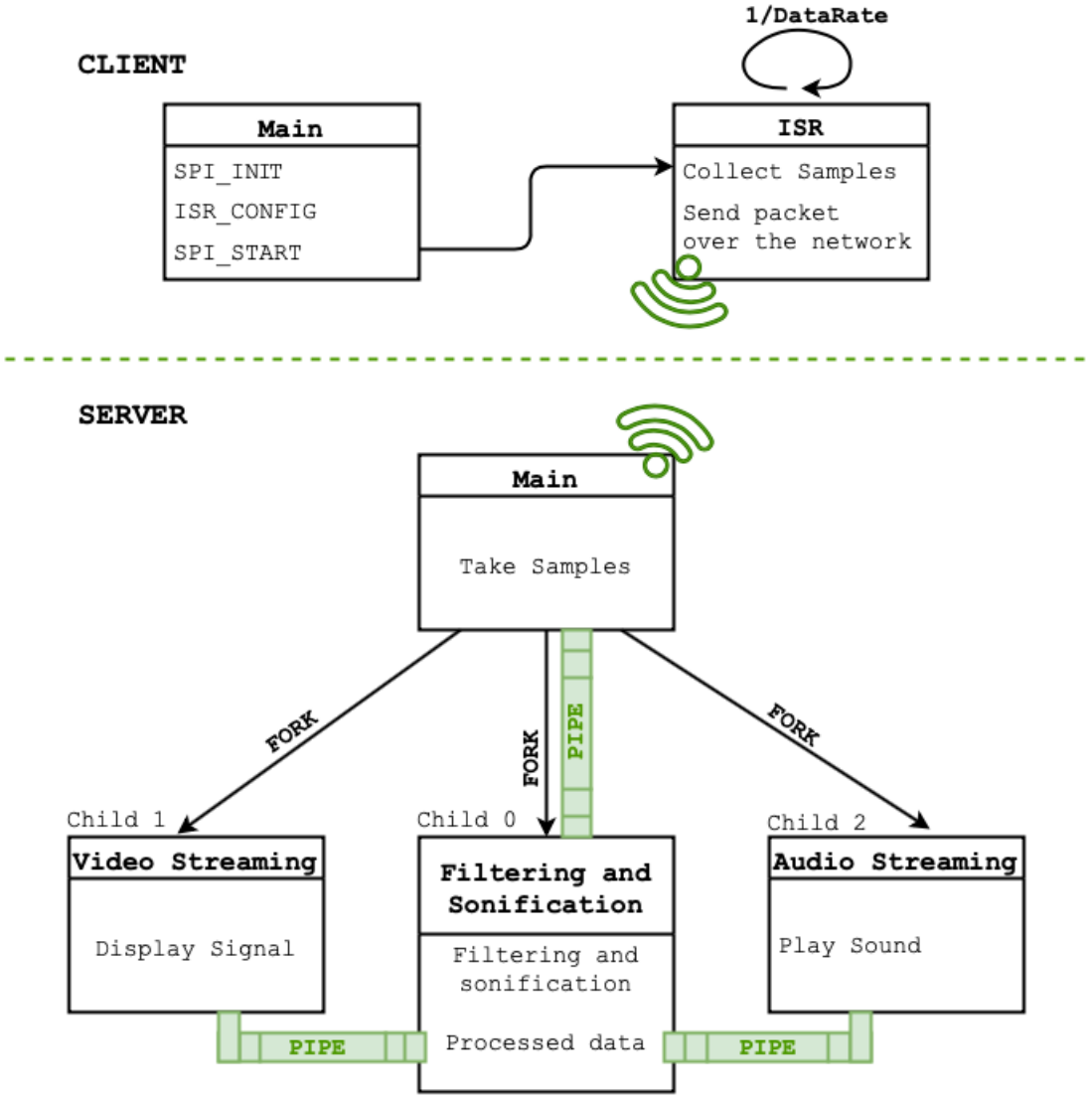

Figure 4.4: The Full Demo flow scheme

Referring to the figure above, the client is represented by a Raspberry Pi3, in charge of getting EEG data from ADS1299 through SPI and forward them over the network (environment setup in Fig. 4.5). The implementation on Raspberry is characterized by a main code that initializes SPI communication and configures the interrupt handler. The latter routine is called every time a falling edge of the signal $\overline{DRDY}$ is detected by a Pi GPIO.

The routine is served with a certain timing that strictly depends on the data rate set on the ADS1299. The handler is in charge of receiving data through SPI and send the relative packet to the Server.

The server has been implemented by means a Laptop with GNU/Linux based OS installed. The main code in this case is in charge of creating three children processes that run their own routines, while the parent starts receiving network data sent by the Raspberry Pi. Once a packet of data is received, it is forwarded to first child process through a Pipe.

Child 0 applies Filtering and Sonification algorithms to the incoming input and it delivers the processed data after filtering and after Sonification to the other children processes, video streaming and audio streaming respectively. The exchange is again performed through the usage of Pipes that guarantee Real Time and synchronization.

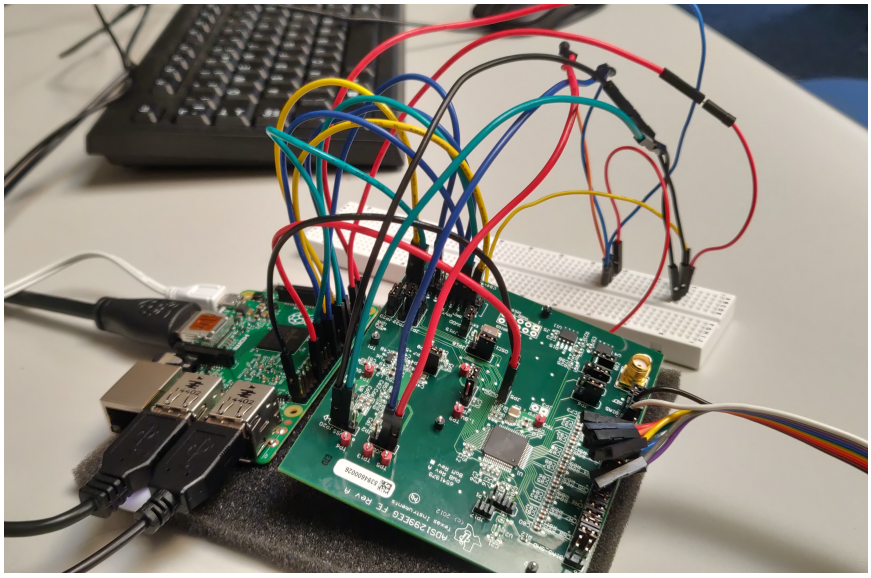

Figure 4.5: Picture of the ADS1299 connected to the Raspberry Pi.

Child 1 is video streaming process that displays the incoming data on a Python GUI in Real Time. A continuous flow has been achieved by progressively displaying the incoming data with a certain frequency, limited by the Frame Rate of the display.

At the same time, Child 2 is taking care of playing EEG sound. The Stream property of Python has been exploited to exchange sequence of data that act as source for audio drivers. In addition, this process needs to match the timing of Video child in order to synchronize Audio and Video on the green line in Fig. 4.3.

This implementation guarantees a large range of functioning as far as the timing equations a the end of paragraph 4.2.3 are respected.

Chapter 5

Conclusions

Relevant training and high medical experience is required to visually interpret neonatal EEG signals. Previous studies made by the Embedded Systems Group in University College of Cork, have designed a solution based on Sonification algorithm and Machine Learning to facilitate the seizure detection in newborns that do not show any clinical manifestation.

In particular, through the exploiting of Sonification, it would be easier to diagnose this kind of disease by associating an audio to the EEG monitoring, much more simple to interpret. The main concept was to realize a low-power portable electronic able to put into practice what has been conceived by the researchers.

This work of thesis proposes the first Demo solution to implement the concept. Therefore, many choices have been made to simplify the initial work and further improvements are necessary to optimize the final application.

First of all, the usage of Raspberry Pi can not be seen as a real solution since this board is characterized by high power consumption. Consequently, it is not possible to embed this device in a portable application of which battery life must last as long as possible. Desired behaviour might be achieved with a low-power microcontroller that needs to establish SPI communication with ADS1299 chip and forward the data over the network.

What is more, speaking of internet transmission, it would be useful to utilize compression protocol in order to reduce the dimension of data packets that are exchanged.

Another further improvement would be to choose a microcontroller that mounts an Hard Real Time Operating System to intensify the speed of response and so, maximize the computation at the edge.

Speaking about server side, it would be preferable to use a real server instead of laptop to increase parallel computations and reduce the application latency. The multiprocesses approach may be revised as well by substituting processes with threads that are characterized by an inexpensive creation and context switch: furthermore, they could ease the full application that can better deal with higher number of chan-

nels and devices at the same time. In addition, audio and video streaming should be performed on an IP accessible by the doctors instead of a Python GUI.

The Demo has been implemented by considering a generic number of channels. However, only one of them has been tested in the Real Time application. Further works, need to validate the effectiveness also in the case of larger number of channels that still must respect the timing limit given by Real Time.

Finally, Machine Learning algorithm must be implemented in the portable device as well, in order to automatically detect seizure: in this way, it is possible to aware the doctors who can instantly check the EEG waveform and sound on a smartphone/tablet/laptop and promptly intervene in case of seizure confirmation.

Bibliography

[1] J. Rennie and G. Boylan. *Treatment of neonatal seizures*. Vol. 92. Arch Dis Child Fetal Neonatal Ed, 2007, pp. 148–150.

[2] M. Toet and P. Lemmers. *Brain monitoring in neonates*. Vol. 85. Early Hum Dev, 2009, pp. 77–84.

[3] S. Simpson H. Khamis A. Mohamed and A. McEwan. *Detection of temporal lobe seizures and identification of lateralisation from audified EEG*. Vol. 123. Clinical Neurophysiology, 2012, pp. 1714–1720.

[4] A. Temko. *Real Time Audification of Neonatal Electroencephalogram (EEG) Signals*. Ed. by UK Patent Office Patent GB1412071. 2014.

[5] G. Boylan A. Temko W. Marnane and G. Lightbody. *Clinical Implementation of a Neonatal Seizure Detection Algorithm*. Vol. 70. Decision Support Systems, 2015, pp. 86–96.

[6] Emanuel Popovici Senior Member IEEE Jonatan Poveda Mark O'Sullivan and IEEE Andriy Temko Senior Member. *Portable Neonatal EEG Monitoring and Sonification on an Android Device*. 2017.

[7] A. Kogan R. Friedman and Y. Krivolapov. *On Power and Throughput Tradeoffs of WiFi and Bluetooth in Smartphones*. Vol. 12. IEEE Transactions on Mobile Computing, 2013, pp. 363–1376.

[8] Texas Instrument. *ADS1299-x Low-Noise, 4-, 6-, 8-Channel, 24-Bit, Analog-to-Digital Converter for EEG and Biopotential Measurements*. 2017.

[9] I. Ali C. A. Ryan B. P. Murphy D. Murray G. Boylan and S. Connolly. *Defining the gap between electrographic seizure burden, clinical expression and staff recognition of neonatal seizures*. Vol. 93. Archives of Disease in Childhood - Fetal and Neonatal Edition, 2008, pp. 187–191.

[10] Andrea Bocchino Conor O'Mahony Geraldine Boylan Andriy Temko Mark O'Sullivan Emanuel Popovici. *System Level Framework for Assessing the Accuracy of Neonatal EEG Acquisition*. 2017.

[11] N. Ford S. Wyckoff L. Sherlin and D. Dalke. *Validation of a wireless dry electrode system for electroencephalography*. Vol. 12. Journal of NeuroEngineering and Rehabilitation, 2015, p. 95.

[12] C. Ramon W. J. Freeman P. B. Colditz and S. Vanhatalo. *Spatial patterning of the neonatal EEG suggests a need for a high number of electrodes*. Vol. 68. NeuroImage, 2013, pp. 229–235.

[13] G. S. Russell T. Ferree P. Luu and D. M. Tucker. *Scalp electrode impedance, infection risk, and EEG data quality*. Vol. 113. Clinical Neurophysiology, 2001, pp. 536–544.

[14] A. Bocchino C. O'Mahony D. Costello E. Popovici M. O'Sullivan J. Poveda and A. Temko. *Comparison of electrode technologies for dry and portable EEG acquisition," in IEEE International Workshop on Advances in Sensors and Interfaces*. 2017.

[15] A. Ciarlone G. Giannoni A. Kenthao C. O'Mahony K. Grygoryev and P. Galvin. *Design, fabrication and skin-electrode contact analysis of polymer microneedle-based ECG electrodes*. Vol. 26. Journal of Micromechanics and Microengineering, 2016.

[16] B. Allison C. Guger G. Krausz and G. Edlinger. *Comparison of dry and gel based electrodes for P300 Brain–Computer Interfaces*. Vol. 6. Frontiers in Neuroscience, 2012.

[17] J. Bronzino. *Chp 47: Biopotential Electrodes*. 2007.

[18] E. Popovici S. Mathieson G. Boylan A. Temko S. Gómez M. O'Sullivan. *On sound-based interpretation of neonatal EEG*.

[19] G. Boylan J. O'Toole A. Temko W. Marnane and G. Lightbody. *Neonatal EEG audification for seizure detection," in IEEE International Conference of Engineering in Medicine and Biology Society (EMBC)*. 2014.

[20] C. Micheyl J. McDermott M. Keebler and A. Oxenham. *Musical intervals and relative pitch: Frequency resolution, not interval resolution, is special*. Vol. 128. The Journal of the Acoustical Society of America, 2010.

[21] Geraldine Boylan Andriy Temko Alison O'Shea Gordon Lightbody. *Neonatal Seizure Detection Using Convolutional Neural Networks, Irish Centre for Fetal and Neonatal Translational Research, University College Cork*.

[22] Geraldine Boylan Alison O'Shea Gordon Lightbody and Andriy Temko. *Investigating the Impact of CNN Depth on Neonatal Seizure Detection Performance*.

[23] *"Keras"*. 2015. URL: `https://github.com/fchollet/keras/`.

[24] Texas Instruments Incorporated Bonnie Baker. *How delta-sigma ADCs work*. 2016.

[25] Claudio Passerone. *Analog and Digital Electronics for Embedded Systems*. clut, 2015.

[26] Texas Instruments. *User's Guide: EEG Front-End Performance Demonstration Kit*. 2016.

[27] *"What is Edge Computing: The Network Edge Explained"*. 2018. URL: `https://www.cloudwards.net/what-is-edge-computing`.

[28] *"Samsung Exynos 7 Octa 7870 vs Qualcomm MSM8916 Snapdragon 410"*. URL: `https://gadgetversus.com/processor/samsung-exynos-7-octa-7870-vs-qualcomm-msm8916-snapdragon-410/`.

[29] Inc Qualcomm Technologies. *DragonBoard 410c Hardware Manual*.

[30] *Android is Based on Linux, But What Does That Mean?* 2017. URL: `https://www.howtogeek.com/189036/android-is-based-on-linux-but-what-does-that-mean/`.

[31] *Linux Device Drivers*. URL: `https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch01.html`.

[32] *Linux Programmer's Manual*. URL: http://man7.org/linux/man-pages/man2/syscalls.2.html.

[33] *Syscalls supported by the Bionic C library*. URL: https://android.googlesource.com/platform/bionic/+/cd58770/libc/SYSCALLS.TXT.

[34] Inc Qualcomm Technologies. *DragonBoard 410c Android User Guide*.

[35] *What Is Fastboot and How Do You Use It?* URL: https://android.gadgethacks.com/how-to/know-your-android-tools-what-is-fastboot-do-you-use-it-0155640.

[36] *Fix SuperSU on DragonBoard 410c (root)*. URL: https://forum.xda-developers.com/general/rooting-roms/guide-fix-supersu-dragonboard-410c-root-t3313746.

[37] *Android Debug Bridge (adb)*. URL: https://developer.android.com/studio/command-line/adb.

[38] Inc Qualcomm Technologies. *Software Build and Installation Guide, Linux Android*.

[39] *Interfacing I2C Grove Digital Light Sensor on DragonBoard 410c*. URL: https://www.youtube.com/watch?v=1c3Epva9hCc.

[40] *What is Port Forwarding?* URL: https://whatismyipaddress.com/port-forwarding.

[41] Robert Bristow-Johnson. *Cookbook formulae for audio equalizer biquad filter coefficients*. Ed. by Doug Schepers (W3C). 2019.

[42] iZotope Contributor Nick Messitte. *Audio Clipping and Creative Uses of Limiting*. 2018. URL: https://www.izotope.com/en/learn/audio-clipping-and-creative-uses-of-limiting.html.

[43] Prashant Mishra. *Streams in Computer Programming: Definition and Examples*. URL: https://study.com/academy/lesson/streams-in-computer-programming-definition-examples.html.

Chapter 6

Acknowledgement

First and foremost, I wish to show my gratitude to Professor and Supervisor Emanuel Popovici who led me during my Erasmus period: he always believed in me and he gave me the strength to complete my University career and to approach with the working life. I will never forget his support.

Furthermore, I would like to thank Danilo Demarchi, my Supervisor in Politecnico di Torino: despite the distance, he always supported me showing remarkable patience and extreme knowledge. I really appreciated his guidelines that have contributed in making this study possible.

This work of Thesis could not be completed without the efforts and support of other group members: it is for this reason that I would like to pay my special regards to PhD students Sergi Gómez Quintana and Mark O'Sullivan, without whom I would haven't reached this goal.

Voglio poi ringraziare tutti gli altri professori avuti in questa carriera universitaria. Ringrazio in particolare il mio Professore di Laurea Triennale Andrea Fabbri che per primo mi ha trasmesso la passione per l'Elettronica e per il mondo degli Embedded Systems, senza il quale non avrei scelto questo percorso.

Un ringraziamento speciale ai miei colleghi universitari di Roma, Torino e Cork con cui ho trascorso gli anni più belli della mia vita. Ricorderó per sempre i momenti vissuti insieme, le ansie condivise e quel senso di angoscia che sono consapevole mi mancherá in futuro. Voglio ringraziarli di esserci stati nel momento più bello ma anche più difficile della mia vita in cui ho dovuto fare i conti con una crescita personale e tutte le difficoltà che ne conseguono. Per questo sarebbe riduttivo chiamarli semplicemente colleghi, ed è quindi che voglio ricordarli per nome: Simone, Alessandro, Florencia, Ilaria, Elena, Federica, Caso, Ibra, Andrea, Marcello, Riccardo, Hairo, Enrico, Francesco, Damiano e tutti gli altri.

Ringrazio inoltre i miei amici di una vita, coloro con cui sono cresciuto. Sono consapevole che nonostante la distanza e le occasioni che il futuro ci porrá davanti, riusciremo ad essere parte integrante l'uno della vita dell'altro, come sempre fatto: Valerio, Gamma, Giorgio, Andrea, Angelo, Matteo, Nicola, Claudio.

Un ringraziamento che viene dal cuore va alla mia ragazza Martina che mi ha sostenuto in questo obbiettivo. Lei é la mia certezza, l'unica su cui posso contare ogni giornata che mi ama e mi sopporta nonostante i miei difetti. I traguardi che ho raggiunto in questo ultimo periodo non sarebbero stati possibili senza di lei.

Infine, ringrazio la mia famiglia che ha reso questo mio percorso possibile, dandomi il supporto emotivo di cui avevo bisogno. Senza di loro non sarei la persona che posso dire di essere oggi, con i traguardi raggiunti. Sono fiero degli insegnamenti ricevuti e faró di tutto pur di trasmetterli anche ai miei figli, vi voglio bene.