POLITECNICO DI TORINO

Master's Degree in Software Engineering

Master's Degree Thesis

EMGView8: a software for Intraoperative Nerve Monitoring

Supervisors

Prof. Marco GAZZONI

Prof. Paolo BARDELLA

Candidate

Simone POSELLA

Academic Year 2020 - 2021

Dedicated to my aunt Cinzia,
who saw this path begin,
and never saw its end.

Summary

Supervising and guiding the actions of a surgeon in the operating room and giving him/her as much information as possible are ones of the most innovative research goals in the medical field. Till approximately fifty years ago patients were often left with huge scars after surgery operation and a lot of damage has been caused to unaware patients due to human errors or poor equipment. This problems can now be avoided thank to the introduction of a new technique, based on the proper combination of instruments, electronic devices, and algorithms. As a result, operating rooms are becoming some of the most innovative and modern places of work. This thesis describes in details a software application and all of its main components used in conjunction with two hardware modules, for intraoperative nerve monitoring during a surgery exam. The presented software allows to monitor nerve stimulation response applied on a patient in order to avoid injuries caused by nerves resection during the operation. This software is called EMGView8 and is part of an Intraoperative Nerve Monitoring product called Nerveäna, distributed by Neurovision Medical Products, a company specialized in innovative surgical products, based in Ventura (California). The design of both software and hardware parts belongs to Neurovision together with OTBioelettronica s.r.l, an italian company based in Torino, specialized in R&D in the bio-electronic field.

Table of Contents

Acronyms VIII

1 Introduction 1

2 Intraoperative Nerve Monitoring Methodology 3
- 2.1 General overview . . . 3
- 2.2 IONM principles . . . 4
- 2.3 IONM methods . . . 5
- 2.4 State of the art of IONM devices . . . 7

3 System Overview 9
- 3.1 System requirements . . . 9
  - 3.1.1 The existing system . . . 9
  - 3.1.2 Hardware & firmware requirements . . . 11
  - 3.1.3 Software requirements . . . 12
  - 3.1.4 Use cases scenario . . . 13
- 3.2 Parameters . . . 19
- 3.3 System implementation . . . 23
  - 3.3.1 Hardware & firmware . . . 23
  - 3.3.2 Noteworthy features . . . 24
  - 3.3.3 Software implementation . . . 25
  - 3.3.4 Software modules . . . 26

4 Software Components 29
- 4.1 Hardware communication . . . 29
  - 4.1.1 Critical aspects . . . 30
  - 4.1.2 Communication protocol . . . 31
  - 4.1.3 Packet description . . . 44
  - 4.1.4 NerveanaComm classes . . . 46
  - 4.1.5 DataGeneratorNerveana . . . 47
  - 4.1.6 How events works . . . 58

4.1.7 Data unpacking 59
4.1.8 Data queues 62
4.2 LoaderNerveana: Data serialization 66
4.2.1 NerveanaContentFilter 66
4.3 NerveanaGUI: Graphic User Interface 72
4.3.1 Analysis of Startup form 73
4.3.2 Analysis of Main form 73
4.3.3 Analysis of ExamDetail form 88
4.3.4 Analysis of SelectDoctor form 89
4.3.5 Privacy 90
4.4 Database and NerveanaRegistry 91

5 Conclusions 94

Bibliography 96

Acronyms

IONM
: Intraoperative Nerve Monitoring

EMGView8
: EMG Viewer 8 channels

OR
: Operating Room

TMS
: Transcranial Magnetic Stimulation

NPI
: Nerveana Power Indicator

Chapter 1

Introduction

From the 60s and due to the increasing amount of hospitalized people and higher attention to hygiene inside operating room, surgery space starts to fill up with pre-sterilized products, disposable needles and other equipments such as gloves, masks, syringes and so on. With the incoming Digital Era operating room were integrated also with high technology devices, from operating tables to surgical lights or stimulator scissors.
In the future and with the further development of new algorithms, e.g. based on Artificial Intelligence, considering the enormous amount of available medical data that would be used to train machine learning algorithms, we will probably observe a progressive replacement of doctors with autonomous robots. [1]
I won't discuss here the moral code at the base of this new debate but the truth is: devices are key components and are strongly integrated in surgery operations.
The software I worked on for my thesis is part for a medical product used in patient monitoring and developed for Neurovision Medical Product. [2]
The systems monitors the state of the nervous system in real-time during surgery, alerting the surgeons of potential evolving neurologic injuries and allowing the application of corrective actions to prevent permanent deficits.

The main tasks of this software is described in the requirements section but they can be organized in three different groups:

- **Real-time signals visualization**: values received from hardware modules must be interpreted, converted to EMG wave form, and graphically visualized on a screen.

[1] Hollywood is taking this idea too deeply, see Alien Covenant of Ridley Scott for a brutal scene of a cesarean operation made by a surgery robot with a lot of available procedure installed. It's clearly exaggeration but it represents well this idea.

[2] http://www.neurovisionmedical.com/nmp/

- **Setting hardware configuration**: software must be capable to configure hardware internal parameters based on doctor preferences. A database must be included to store information about settings, doctors and occasionally patient.
- **Allow offline visualization of recorded exams**: real-time signals must be saved to files to be successively examined for research or legal purpose.

The design of each key point is described in Chapter 4, while in Chapter 3 I will look at a general overview of the product itself and how macro components are integrated with each other. While I won't present in details the technical aspects of the various hardware components which compose the full Nerveana system, I will however provide a brief overview of the complete device, dwelling on the components directly related to the software development.

Chapter 2

Intraoperative Nerve Monitoring Methodology

2.1 General overview

Intraoperative Nerve Monitoring (IONM) is a minimization risk tool developed in the last four decades to monitor the functional integrity of neural structures. It actually was experimented for the first time by Wilder Penfield, a Canadian neurosurgeon, in 1930 during an epilepsy surgery as a substitute test for cortical function, providing the first ever insight to monitoring human brain function. IONM development was made easier by technological advances in software algorithms and electronic equipment and became commercially available in 1981. A key role in the development of IONM was played by anesthesia. When a patient is under general anesthesia, he/she is paralyzed and heavily sedated to the point of unconsciousness. This can suppress neural activity, and IONM could not be safely provided in the Operating Room until enough advances were made in intravenous anesthesia.
IONM objective is to reduce the risk of damage to the patient and to provide a guidance for the surgeon during the operation; since IONM was introduced in the medical field, it has reduced the chance of paralysis, hearing loss, muscle weakness, and loss of other biological functions. A spontaneous electrophysiologic signals is received from the patient periodically or continuously through the course of the operation by the use of a stimulator and a set of recording electrodes. In general, a trained neurophysiologist (or a technician supervised by a neurologist), attaches the system to the patient and observes and documents the signals for the surgery session. IONM became more and more important through the years; The American Society of Neurophysiological Monitoring was founded in 1990 in order to serve the emerging field of neuro-monitoring .

2.2 IONM principles

A key factor of IONM is the **Evoked Potential (EP)**, an electric potential generated from a specific point of the nervous system after the presentation of a stimulus [1]. There are in general four main different types of evoked potentials, recorded from a different specific part of the nervous system after the generation of a stimuli:

- **Visual Evoked potentials (VEP):** used to examine the visual nerve and the optical pathway and detect optic nerve damage, neuritis or inflammation. Visual stimulus could be, for example, a flashing light.

- **Brainstem auditory evoked potentials (BAEP):** helpful in the diagnose of a hearing ability and detection of brainstem tumors and multiple sclerosis. An acoustic stimulus could be a speech sound transmitted by earphones.

- **Somatosensory evoked potentials (SSEP):** provide an evaluation of the central somatosensory pathway and the peripheral sensible nerves. The velocity of the transmission of the impulse to the brain or the spinal cord is measured after the occur of a transcutaneously stimulation, usually by an electric stimulus. SSEPs are usually used to detect spinous cord dysfunction or during surgery to examine neurological dysfunction.

- **Motor evoked potentials (MEP):** no sensory organ is stimulated but the motor cortex itself is excited. Motor Evoked potentials are mainly used intraoperatively to monitor the intactness of the motor system and to measure the transmittance between the brain and a muscle.

What interests us is the **MEP**, recorded from muscles following a direct stimulation of the exposed motor cortex. When a stimulus is generated, the proper observation of the registered response's shape allows us to discern if the nerve is injured or dysfunctioning.
An example of possible acquired EP signal is shown in 2.1 In response to the external stimulus applied at t=0 (1), the nervous system responds after a latency interval (2) with a depolarization phase (3), followed by a repolarization phase (4). It then recovers (5) to the unperturbed condition. Typical values for the measured potential are in the millivolt range, while the depolarization and repolarization phases typically last up to few tens of millisenconds.

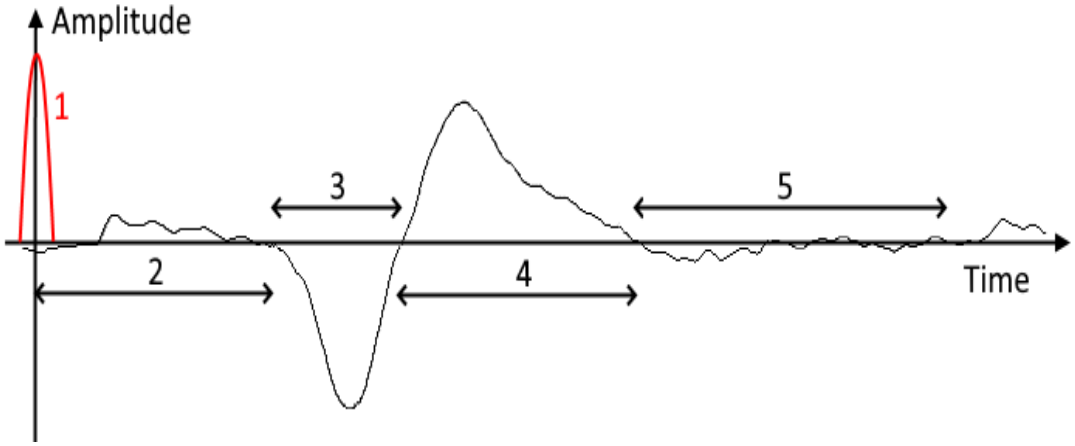


Figure 2.1: Example of MEP

1) Extended Stimulus 2) Latency interval 3) Depolarization phase
4) Repolarization phase 5) Recover

2.3 IONM methods

Two types of stimulations are tipically used for MEPs, namely **magnetic stimulation** and **electrical stimulation**. In both cases the procedure is the same: a particular area is stimulated and the nerve activity is checked using a set of sensors that follows the direction of the nerve itself. If a wrong signal shape is detected or no activity at all is present, something is wrong through the pathway.

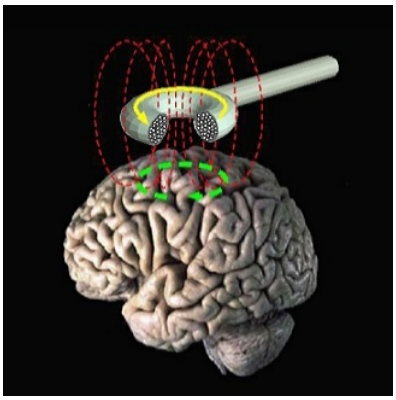


Figure 2.2: Transcranial Magnetic Field

Figure 2.3: Scorpion® (Neurovision)

Transcranial Magnetic Stimulation (TMS) is a non invasive form of brain stimulation. The TMS procedure includes a magnetic field generator, placed near

the head and via electromagnetic induction the brain receives electric currents (Fig 2.2). In detail an electric pulse generator, or stimulator, is connected to a magnetic coil, which is connected to the scalp. TMS has shown diagnostic and therapeutic potential in the central nervous system with a wide variety of disease states in neurology and mental health.
On the contrary, when the electrical stimulation is used, the stimuli is transmitted through cutaneous electrodes. The main advantage of this technique is an higher depth of penetration that allows direct spinal cord stimulation but the main limitation is the local discomfort that is caused. Electrical stimulators (Fig 2.3) produce constant current or high-voltage pulses of brief duration (from 2 ms to 50 ms) with a current up to 1 A. The voltage is kept constant during the stimulation, but the intensity of stimulation depends on the skin impedance.

Despite the many attempts in the 90s to monitor MEP following direct electrical stimulation of the spinal cord, MEP recorded over muscle is now the most widely adopted approach because of the relative simplicity of generating and recording MEPs. For several years the eletrical stimulation was studied on animals and researchers find out that an initial direct wave is followed by many indirect waves at periodic intervals. Direct wave represent the stimuli received from the neurosystem, while the indirect waves reflect indirect axons depolarization. Only the direct wave could be not enough to sollecitate the motoneuron to fire but the sum of multiple indirect waves can reach the threshold and trigger it.

To set up the use of IONM, in general, a trained operator connect one side of the device to the computer and the other side to the patient using recording electrodes. The device itself should be connected to a stimulator (or stimulating electrodes) used to sollecitate the nerve. The control of the stimulator can be duty of the surgeon or of the software that selectively activate the stimulation with timing. The software running on computer carries out one main task: process and display the electrophysiologic signals recorded by the electrodes. The neurophysiologist can thus observe and document the signals in realtime in the operating area during the surgery.

There are many surgery operations that can make use of IONM. In **Thyroidectomy and Parathyroidectomy** can be used to find the superior laryngeal nerve. In **Submandibular Gland Excision** can be utilized to locate the hypo glossal nerve by placing needle electrodes into the tongue. In **Neck and Skull Base Procedures** can be used to monitor the trapezius muscle and so on. In all mentioned cases the IONM procedure follows three steps:

- localize neural structure

- test structure functions
- detect neural injuries

2.4 State of the art of IONM devices

There are many companies on the market offering similar devices that differ mostly on:

- number of channels they are able to monitor: usually 8, 16, 32, 64 channels are used, while solutions with even more channels are also available.
- stimulation type: it depends mostly on the area we want to stimulate and on the surgery operation the device is meant to support.
- portability: the device can be stand-alone or modular and configurable on operating room trolley.

NeuroIOM (Fig 2.4) from Neurosoft is a 32-channel System for Intraoperative Neurophysiological Monitoring with Advanced Functionality for SEP and MEP acquisition. It provides auditory and visual evoked potentials, EMG, direct nerve stimulation, EEG and ECoG with transcranial electrical stimulator.

Figure 2.4: NeuroIOM

Figure 2.5: Cascade PRO

Figure 2.6: C2 Nerve Monitoring

Figure 2.7: NIM 3.0

The Cascade PRO IONM from Cadwell (Fig 2.5) monitors from 16 to 32 channels with one or two amplifiers and provides multiple stimulation. It is way less portable due to its number of accessories like: transcranial motor stimulation, electrical stimulation, earphones for auditory stimulation, extender pod cable shielding reduces noise and LED Goggles with disposable foam for visual stimulation.

C2 Nerve Monitoring (Fig 2.6) from Inomed offer six modalities for six different application area: thyroid surgery, facial/ent surgery, spine surgery, colorectal surgery, carotid surgery and brain mapping, all integrated in a single device with 4/8 monitoring channels and two dedicated stimulation channels.

The NIM 3.0 from Medtronic (Fig 2.7), for example, offer a semi-portable device integrated with a touchscreen display.
This solution monitors up to eight channels of nerve-muscle combinations during bipolar cautery offering also artifact detection software.

The new Nerveäna® system goals is to offer an high portability, thanks to the compactness of the hardware main module and monitoring up to eight channels. The system is designed to work stand-alone, notifying the operator with specific sounds alert and allowing the surgeon to maintain attention on the surgical field, or together with a touchscreen monitor. It includes also a stimulator with three different types of stimulation.

Chapter 3

System Overview

The purpose of this chapter is to introduce the entire system and give a general overview of the parts that compose it. Moreover, design choices will be discussed. I will not cover in detail the hardware design but I will focus mostly on the software part, that got me involved.

3.1 System requirements

3.1.1 The existing system

This project originates from the request from Neurovision Medical Product to update an old product called Nerveäna®, an integrated surgical tool composed by a nerve stimulator and an electromyographic monitor. There, the monopolar probe, continuously applies a stimulation pulse to soft tissue while the EMG monitor detects, interprets and records muscle response evoked by stimulation and once an evoked EMG is identified, the Nerveäna® produces an audio alarm allowing the surgeon to maintain attention on the surgical field.
In brief the old system is a one channel monitoring system that includes:

- A connector for stimulator probe
- Knob for setting stimulation value (up to 5 mA)
- Knob for increasing or decreasing signal amplification
- Knob for audio volume
- Knob for free-run EMG (chirp) level
- An impedance measurement function and monitor

- An integrated speaker
- USB cable connection to PC
- EMG input cable

Figure 3.1: Nerveäna® surgeon nerve locator with accessories

One of most important design choice about the old Nerveäna® system is the possibility to use two methods to locate the nerve according to the surgeon preferences. Basically the system can work in two modes: **Surgeon driven** or in **IONM mode**. In the first case the system is focused on the audio alert and for this it does not need any monitor or any adjustable height cart but can be placed on any fixed height surface. In the second mode the EMG signals are shown during the exam and a technical operator is required. In this case a cart is needed but due to the low number of accessories and the size of the hardware it doesn't look overwhelmed but small and compact. Image 3.2 and 3.3, taken from Nerveäna® user manual, show better the two cases.
While the parameters will be discussed on the next chapter I want to focus a moment on the impedance measurement. In order to ensure the success of neuromonitoring, the contact between nerve and electrode should also be guaranteed. The loss of action potentials or the inability to stimulate a nerve might not always be caused by nerve injury but also can be caused by a bad electrode-nerve-contact or by the misplacement of the electrode. Both failures can be discarded by using impedance measurement [2]. Impedance measurement can be started using the **Test function** by pressing the apposite test button and then the hardware creates a short circuit, induces a small electric current and it measures the impedance of each sensor.

Figure 3.2: Nerveäna® with fixed height cart

Figure 3.3: Nerveäna® with monitor

3.1.2 Hardware & firmware requirements

In the new system, the design must pay attention to audio alert and the same key features mentioned above but it must monitor up to eight channels. Compactness was an appreciated feature of the old system and the new one must take this into account also if the channels upgrade involves in a change of size or in the integration of one more hardware module. Summarizing the hardware and firmware requirements are:

- continuous monitoring of eight EMG channels
- change parameters on the fly
- connector to a stimulator
- a library of audio tones for event alert
- integrated battery and charge supplier's connector
- connection to computer for communication
- SD card space for data storage
- test function for impedance measurement

- different set of free-run voltage threshold levels (chirp)
- different set of stimulation levels
- compactness

3.1.3 Software requirements

The three macro requirements of EMGView8 software are described in Chapter 1 but can be exploited in points:

- **Hardware communication**:
  - **Receive real-time EMG data**: the hardware continuously monitors EMG channels and sends, via USB-C cable, values and information that must be elaborated and recorded.
  - **Real-time hardware set up**: the software must allow the user to change hardware parameters during the examination as if he/she were turning the knobs present on the hardware front panel.
- **Graphical User Interface**:
  - **Real-time signals visualization**: it must display all eight (or all active) channels of running EMG signals and display also the responses of channels after the stimulation.
  - **Visual Alert**: the GUI should alert the surgeon with visual warning, like red label message, when there are chance of risk, detected with algorithms.
  - **Display useful information**: the software should display to operator not only the EMG signals but also information received by hardware or detected with algorithms (voltage peak, impedance and so on)
- **Data storage**:
  - **Store hardware configuration**: the software must be able to store a set of different hardware configurations and load inside main module the chosen one during start up.
  - **Customize hardware configuration**: Each set of settings must be associated to the referring surgeon and must be customizable. Software must handle preferences of a set of surgeons.
  - **Store examination**: the real-time visualization must be saved to files to be successively opened for research or examination purpose.

  - **Privacy care**: the database should pay attention to privacy and sensitive data and must be accessible only using a password. According to local law sensitive data can be encrypted.

- **Offline examination**:

  - **Print report**: after the examination a report must be created, including surgeon personal data, hardware settings and most important events that happened during the operation.

  - **Visualize old exams**: all the stored exams must be accessible through the software, visualizing recorded signals and real-time hardware settings changes

3.1.4 Use cases scenario

EMGView8 software interact and connect three actors: operator, hardware main module and database and this section is meant to provide a view of the cases that involve the actors and explain how they interact to satisfy the requirements.

The following figure shows a scenario of EMGView8 that includes the four most important requirements.

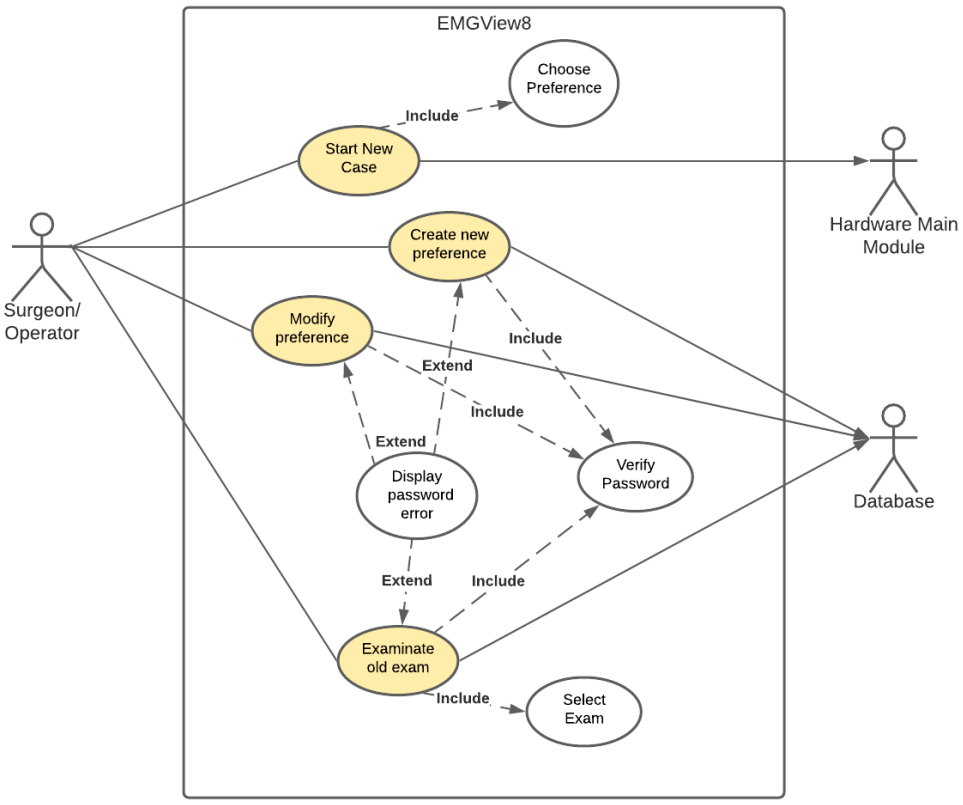


Figure 3.4: Use cases diagram

Start new case

Requirements: at least one doctor and a configuration contained in the database. To start a case the user must open the apposite window, select the Doctor who is operating and the configuration settings he/she want to load inside hardware module. When the operation is confirmed the software send initial settings and start command to hardware and the examination begin.

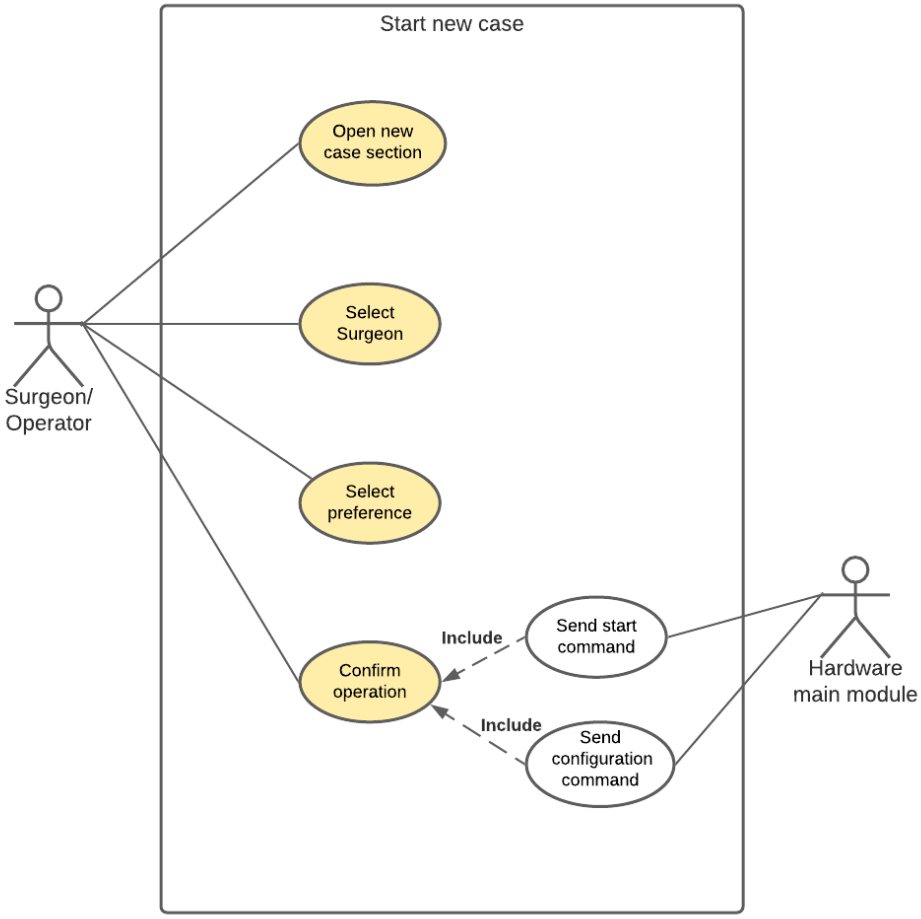


Figure 3.5: Start new case

Create new doctor

Requirements: no requirements are needed to create a doctor.
User just need to open the manage window (password required) and insert the required information. When the operation is confirmed a database connection is established and the data are stored.

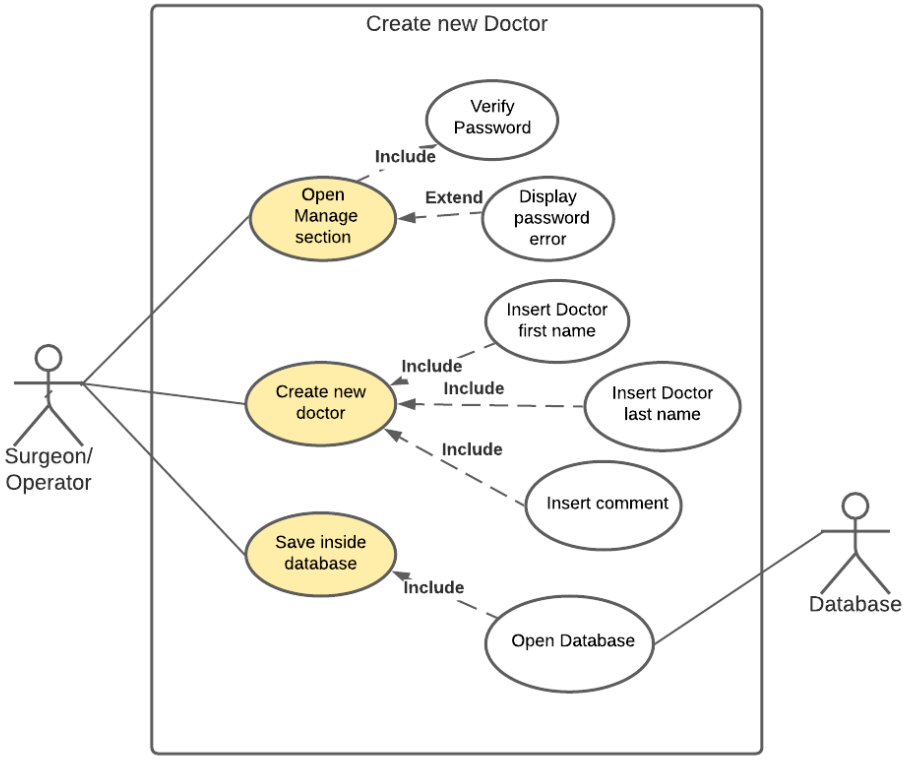


Figure 3.6: Create new doctor

Create new settings

Requirements: at least one active doctor is present inside database.
To create a new configuration the user need to open the manage window (password is required), select the doctor and create a new configuration. The new settings will be attached to the chosen surgeon. Then he/she can select and choose knobs and background parameters he/she prefer for the setting.

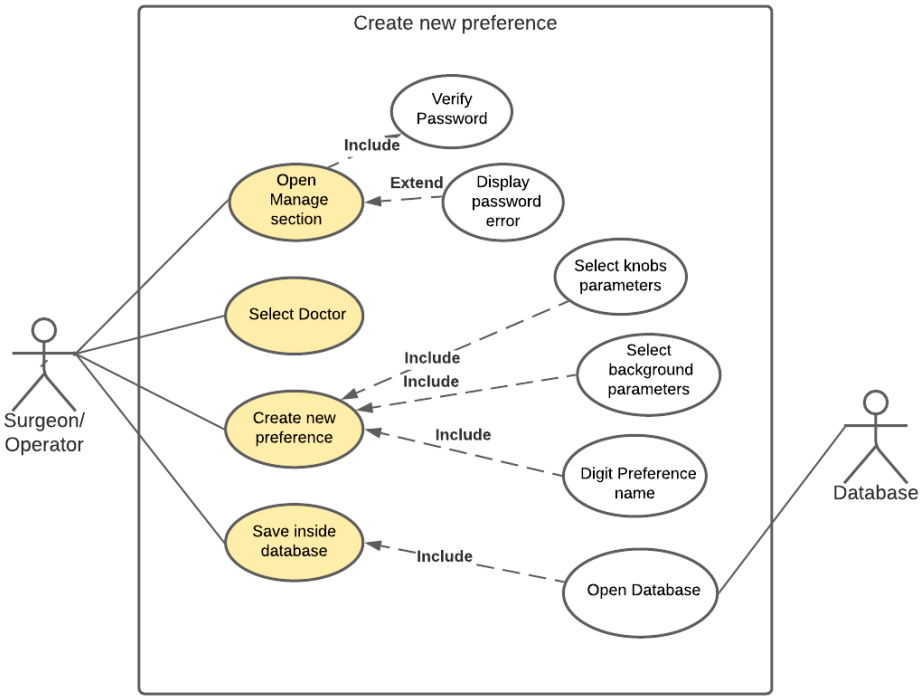


Figure 3.7: Create new preference

Modify preference

Requirements: there's at least one active doctor that contains a configuration settings.
To modify a preference the user must open the manage window (password is required), select the doctor and the preference and open the edit window. Make the changes he/she prefer, that could related to knobs parameters or to background parameters and confirm.

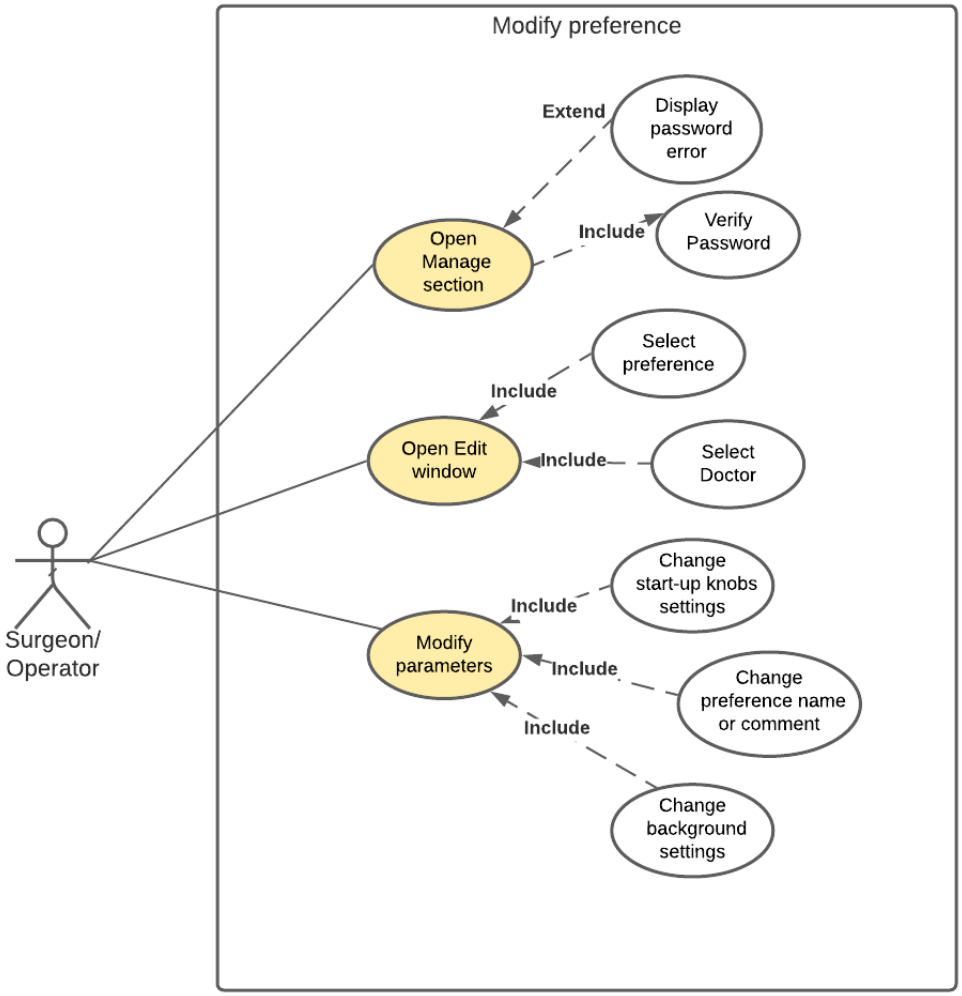


Figure 3.8: Modify preference

View old exam

Requirements: at least one exam is stored in the database. It could be added by import of an external file or due to an effective surgery operation.
To view an old file the user have to open the review windows, find for the file and confirm the selection. Another window should open, displaying the EMG signals and the preference settings used during the exam.

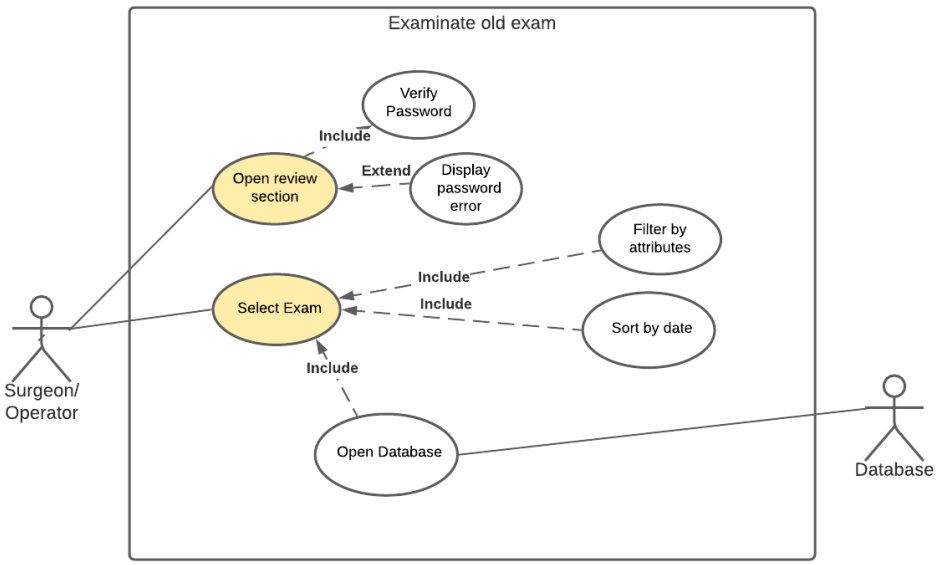


Figure 3.9: View old exam

3.2 Parameters

Before continuing with the implementation section we need to understand which parameters are involved in the process. I define two kind of parameters:

- **The main parameters**: under user control and adjustable during the entire operation.
- **The background parameters**: hidden parameters, settable before the operation starts. These parameters are related to hardware variables used for calculation.

Starting from the first case there are five main parameters that must be handled during an examination:

- **Stimulation level**: the hardware must communicate with the stimulator and set it's stimulation value. The stimulation value can be changed by the hardware main module or, in alternative, the operator can set it through the software. Stimulation level is a risk factor because a wrong value can expose the patient to a risk and lead to nerve injury.
- **Chirp**: this parameter, called also free-run EMG, is a threshold used to alert the surgeon if it's exceeded by a signal. In clinical use ranges between 50 and 150 uVolts.
- **Waveform amplification**: this parameter is related to the zoom level of the signal. Increasing or decreasing the waveform amplification means changing the multiplier for which the signal values are multiplied. The result is a better visualization of the signals.
- **Audio EMG**: this parameter is used to select one of the eight channels for Audio EMG, a direct voltage output to the audio speaker of the actual EMG signal.
- **Audio volume**: this is the master volume control for all audio signals. It changes audio level of the hardware integrated speaker.

One important value which enhances the Neurovision Medical Product over the other companies is the **Nerveana Power Indicator (NPI)**. As seen in section 2.2 the MEP complex and its components are the keys of intraoperative nerve monitoring and the company has decided to interpret the shape of the response as a number, called NPI. In fact the Nerveana Power Indicator is a value obtained calculating the integral of the nerve response and some of the background parameters are used by hardware to change how the NPIs are calculated. In details:

1. **Window delay**: this variable is used to determine the exact start moment for the NPI calculation after we send a stimuli. The default value is 1.5 ms and it's the low boundary of the integral but can be adjusted by the user depending on the distance between sensor and stimulator. Changing this parameters means moving the integration window and can affect the NPI value. The delay is shown in figure 3.10.
2. **Integration threshold**: This parameter is used to alert the operator when the NPI exceeds the threshold. It's not defined as a number but as a percentage and it's used to set the sensibility depending on position of sensors. Neurovision company did several studies during the years and accordingly to many results their researchers have defined a specified number for the 100% value of NPI. Changing the integration threshold means allowing to alert the surgeon with lower or higher value of the NPI with respect to the specified value. It's mostly

used if the stimulation pulse is deep inside the tissue or the stimulation point is too far or close to the sensor location and may happen that the recorded signal is not as wide as it should. An expert user can evaluate this situation and accordingly adjust this parameter. The 100% default threshold is shown in figure 3.10 as the black line that represent the usual response. Increasing the sensibility the user can be alerted also with the response shown with the red lines.

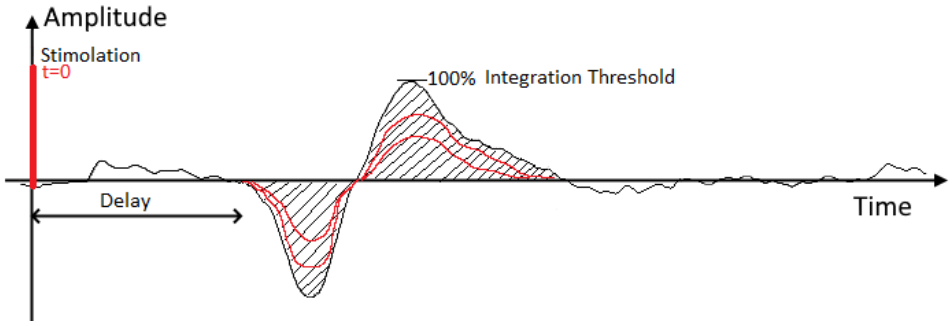


Figure 3.10: Parameters

3. **Stimulation type**: the stimulator type can be set to negative, positive or biphasic. In the first two cases the stimulation use pulses of monopolar shape with duration T. These are the most used and they are proposed by vendors all over the world. The third one instead use a bipolar pulse with the same amplitude and duration and from the most recent publications it seems to play an important role in the reduction of the artifacts. Biphasic refers of two phases: cathodic phase and anodic phase. At the cathodic phase, the action potentials are initiated by applied current pulse and neural reaction is elicited while the subsequent anodic phase cancels the charges accumulated on the electrodes. [3]

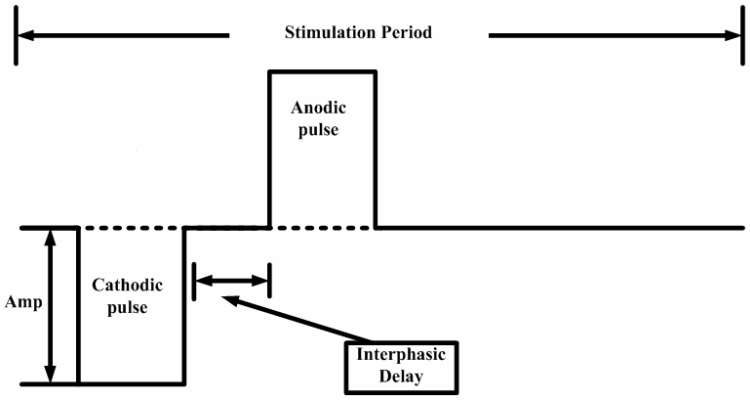


Figure 3.11: Biphasic stimulation

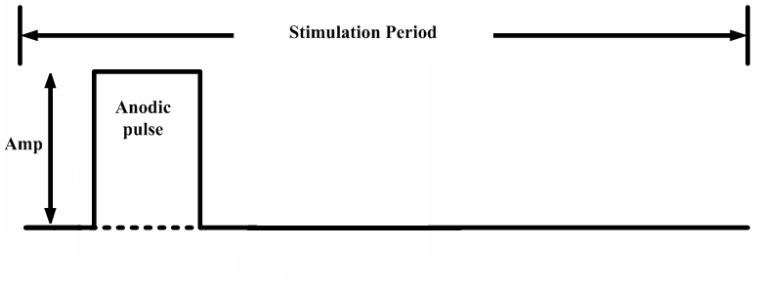


Figure 3.12: Positive stimulation

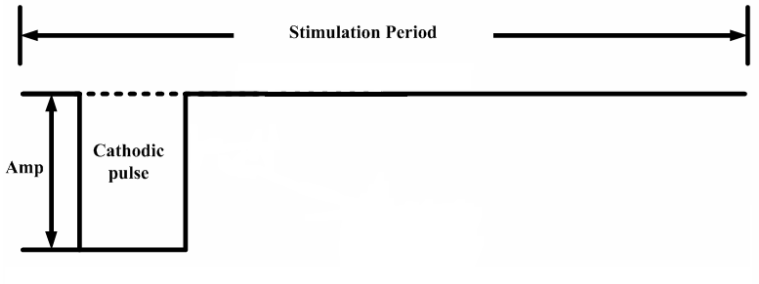


Figure 3.13: Negative stimulation

4. **Stimulation duration**: It defines the duration of the stimulus. Two values are available: 158 us and 300 us.

5. **Stimulation range**: this variable is related to the set of values the stimulator can be set to. Usually the stimulator range is 0 up to 5 mA but can be set with range 0 - 10 mA.

6. **Chirp level**: this parameters define the chirp values assigned to the ten different levels of the chirp knobs. Not a single set of chirp values is defined but there are different possibilities. Usually the chirp go from 30 to 150 uV and changing the parameters means moving up and down on the scale of predefined values. The chirp level select which scale to use. For example:
   [30 36 43 51 61 73 87 104 125 150],
   [40 48 57 68 81 97 116 139 167 200],
   [48 58 69 83 100 120 144 173 208 250],
   and so on..

7. **Impedance threshold**: the impedance value (calculated with Test function) of a specific channel is confronted with this variable. If the value is above the threshold the user is alerted.

3.3 System implementation

3.3.1 Hardware & firmware

Details of hardware and firmware parts are out of the scope of this thesis but what I want to show is the general point of view of the hardware design. The new Nerveäna® hardware system is composed by two modules:

Figure 3.14: Main module

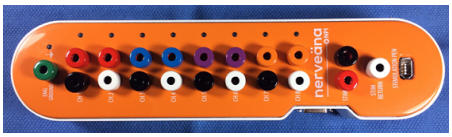


Figure 3.15: Patient Interface Box

Main module is the core component, it connects directly to a computer with a USB-C cable and to the patient box with a custom cable specifically designed. The front panel is composed with three knobs used to set the parameters described in section 3.2. The knobs has ten different levels and can be used in two ways: simply turning left or right to increase or decrease the level or by pushing and turning. This different use is required by the large number of parameters surgeon can set during the operation. The front panel is designed to contains three knobs that interact with five parameters.
Stimulation knob: used to set Stimulation level, if the knob is pushed the "Test" feature is activated. Push and turn don't produce any changes.
Signal Amplification knob: used to increase or decrease the multiplier of each signals. Turning the knob when is pushed allows to change the **Chirp threshold**.
Audio Volume knob: this knob is used to change the speaker volume. If it is pushed, the turning action allows the operator to choose the channel which is used for **Audio EMG**.

A led crowns surround each knobs with green led color to show the current level of the main parameter. If it's pushed the crown led color becomes yellow and shows the chosen level of the second parameter.

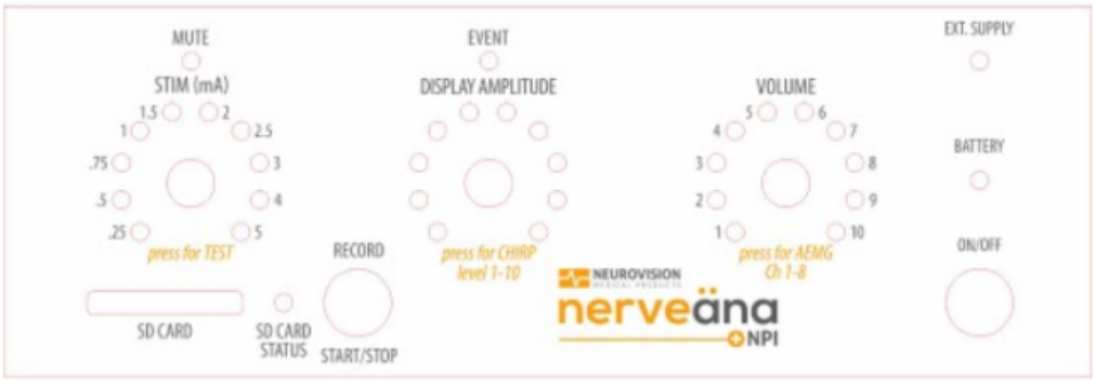


Figure 3.16: Front panel serigraphy

Patient Interface Box: it contains a single programmable board devoted to interact with main module and communicate EMG signals. It connects with the patient and, as shown in figure 3.15, it is composed by a set of connectors. In detail eight pairs are used for the eight EMG channels, one connector is used for the EMG ground and three are related to the electrical stimulator. Not all the connector have to be used during a surgery operation, only the ones that's needed.

3.3.2 Noteworthy features

I just want to mention two important features that are included with the new hardware module.

Lead-off detection

Detecting the connectivity of an electrode to a patient is essential. If there is any disruption between the body and the monitoring device, the reported results may not accurately correspond to the patient's physiology. Lead-off detection verifies that electrodes are properly connected, and immediately notifies the user if a fault is detected. This fault can be configured to alarm when an electrode is completely disconnected or when the connection is weak. The design of a electrode relies on the fact that the skin has a dry outer layer requiring gel in order to establish a strong conductive path from the patient to the system. Over time, these gels begin to dry out, changing the impedance characteristics between the electrodes and the patient. Also air gaps may develop between the electrode and skin, especially if there is hair on the skin surface, increasing the series impedance across the input path. [4]. The lead-off detection implemented in the new system constantly monitors that the voltage of the electrodes is between 0 and 3.3 V otherwise it alerts the operator to check the electrodes.

Exponential average filter

A design choice was to implement an high-pass filter that cut out all frequencies below 10 Hz. Instead of inserting a high-pass filter in hardware it was decided

to implement the filter at firmware level. This was done because if there were any changes the hardware component should have been replaced with a new one while, at firmware level, a simple change in firmware code would have solved the problem. The implemented filter is calculated subtracting the low pass filter called Exponential Average Filter, a moving average usually used in statistics. Given a series of values and a number of subset, the first moving average is calculated taking the average of the initials subset, then we shift excluding the first number and including the next one. In brief it can be seen as a smoothing of the data.

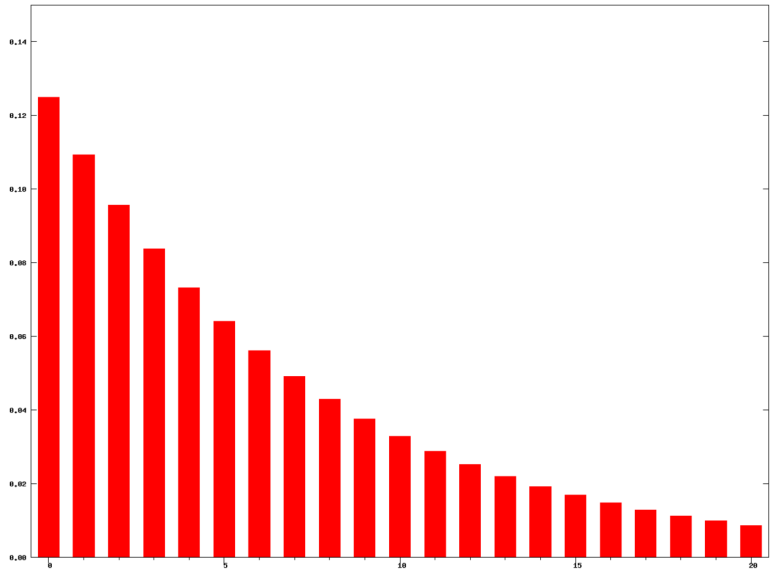


Figure 3.17: Exponential Average Filter with N=15

3.3.3 Software implementation

EMGView8 software is designed for Windows operative system on a native high definition monitor with 1920x1080 of resolution equipped with mouse or, in alternative, with a touchscreen. Its code is written in C++/C# programming language using Visual Studio 2017 Integrated Development Environment for simplicity of integration between the two different code sections. While the next chapter is entirely dedicated to the software implementation and all others technical aspects, this one want to cover the organization structure of classes, actors involved and the high level diagram. Development metodology used follows the Agile Development Principles:

- Deliver working software during the process for customer satisfaction
- Welcome changing requirements, even in late development

- Daily cooperation
- Simplicity and attention to best requirements and design

Stand-up or short feedback were used to report among the parties the work progress and define what to do in order to coordinate and adapt priorities.

The entire system is composed by three main actors: surgeon, EMGView8 software and hardware main module. Patient interface box is not considered because is not directly connected to the EMGView8 software. Also when controlled by main module to record the EMG signals, the values are communicated to the software by the main module. During software implementation and in particular in data storage, attention was paid to three important element: Surgeon, Exam and Preference.

- **Surgeon** he/she is identified by a small set of personal information and is the operator that uses the system. Each surgeon can create a list of preference, that belong to him/her only, and pick one up at the start of the surgery operation.
- **Preference** is a set of hardware settings and include all parameters that can be configured in hardware module. Those parameters are loaded inside hardware main module at the start up and while some of them remains fixed for the entire operation (like Impedance Threshold) others can be changed on the fly, like the parameters related to the knobs. The entire list of variables is described in chapter 4.
- **Exam** is the surgery operation itself. It's composed by the two previous elements (preference and surgeon attributes) and other important data, such as the patient personal information (optional attribute due to privacy) and the stored files of EMG signals.

It is important to define some of the acronyms and synonyms that will be used in next chapters:
Doctor/Surgeon/Operator: all terms refer to the main user.
Preference/Procedure/Preset/Settings/Configuration: are terms for the configuration settings saved inside the database and from which the user can choose before starting a new case.
Acquisition/Operation/Exam: is the set of files created after an exam, where all hardware data and information are stored.

3.3.4 Software modules

To achieve and cover all aspects cited above the project was separated in parts to not mix the functionalities. In particular, the main project is composed by five sub-project:

- **NerveanaGUI**: it contains all the form that are displayed to the user and with whom he/she can interact. The startup form, the main form, the settings form, the manage form and so on. Moreover it contain few classes for custom control draw (led and knobs) to let user feels like he/she is using the hardware itself.
  - LedKnob class that contains drawing functions to simulate the hardware knob and used to draw custom controls in main form and offline examination form.
  - LedBulb class, used to stimulate the hardware led.
- **Utilities**: It contains some useful classes like database and report in order to open a connection and encapsulate the data and some other functions from different use. In detail:
  - Database class contains functions specific for database. Open, close, get doctor, update doctor, insert preset, update preset, insert exam, update exam, delete exam and so on. It contains two subclasses: Surgeon and Procedure.
  - Event class, used to create and store specific event happened during the real-time operation. This class supports the creation of the PDF report. When the user want to print a report, the software load data inside this class and then the object is passed to the apposite function to build the pdf file.
  - NerveanaRegistry, a class that define variables saved on local machine registry.
- **NerveanaComm**: this sub-project is the key for hardware communication. Written in C++ it contains all the functions that handle the communication, from opening the right port, create and send command to manage the incoming data. NerveanaComm is born from the adaptation of a library proprietary of OTBioelettronica that makes use of a class called DataGeneratorNerveana that indentify the device and associate a set of variables and functions.
- **LoaderNerveana**: it contains all the necessary for file handling. Examination file is stored and on opening each byte is interpreted in the correct way to reproduce and display all the data (EMG channels and settings). This class contains a subclass called NerveanaContentFilter, that is the real container of the values.
- **Soundtrack**: this sub-project contains all functions related to the EMG signal draw. Also this library is proprietary of OTBioelettronica used by many of its

products. The most important sub class of this library is the Track class used to encapsulate the NerveanaContentFilter and associate values with drawing functions.

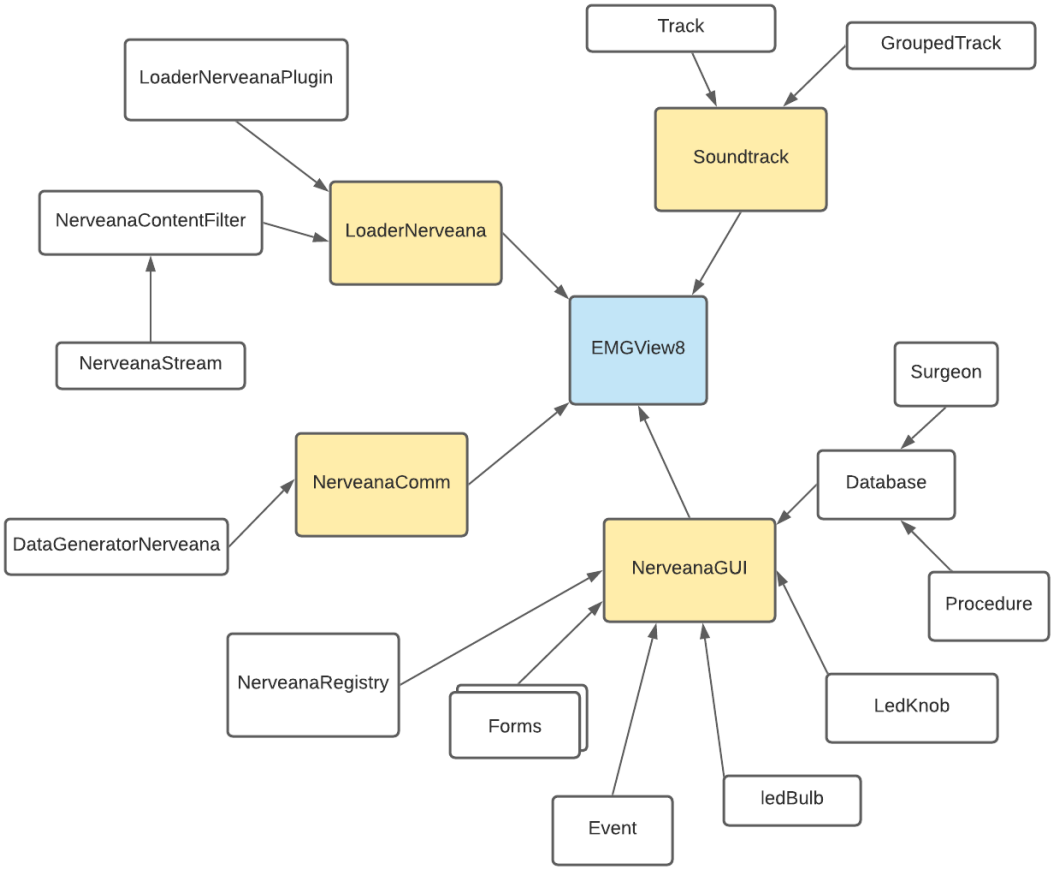


Figure 3.18: Class Diagram

Chapter 4

Software Components

This chapter is the core of my thesis, where the details of my work are explained, together with algorithms and all the implementation work. I defined four subsection, each of one related to the main sub-project of the EMGView8 software.

4.1 Hardware communication

The first subsection is about hardware communication. As said in a previous chapter the hardware communication was made possible with the use of a proprietary library of OTBioelettronica called OTComm. This library is written in C++ using Win32 API and implement class, structure and functions for communication and has been modified and extended through many years with the design and development of new devices. It allows to communicate with different protocol using wifi, bluetooth, LAN or USB cable and it implements a different class for each device. The name of those class is composed by the prefix **DataGenerator** followed by the name of the device (DataGeneratorSessantaquattro for example, DataGeneratorNerveana in our case). So the DataGenerator class contains all the functions useful for the specific device. What has been done is the convertion of all the useful part of this library into another one called: **NerveanaComm**.
OTBioelettronica devices are complex object that needs to be paired by the user with adapter and sensor in order to work, so a set of functions are related to the adapter connection and the saving of settings in .xml format for future loading. Nerveana system instead has a fixed structure and all those functions were ignored during the conversion. They were substituted by other functions used for communication, especially those functions that regards commands. NerveanaComm structure is shown in figure 4.1, not all the functions of DataGeneratorNerveana are included in figure but they are explained individually in next section.

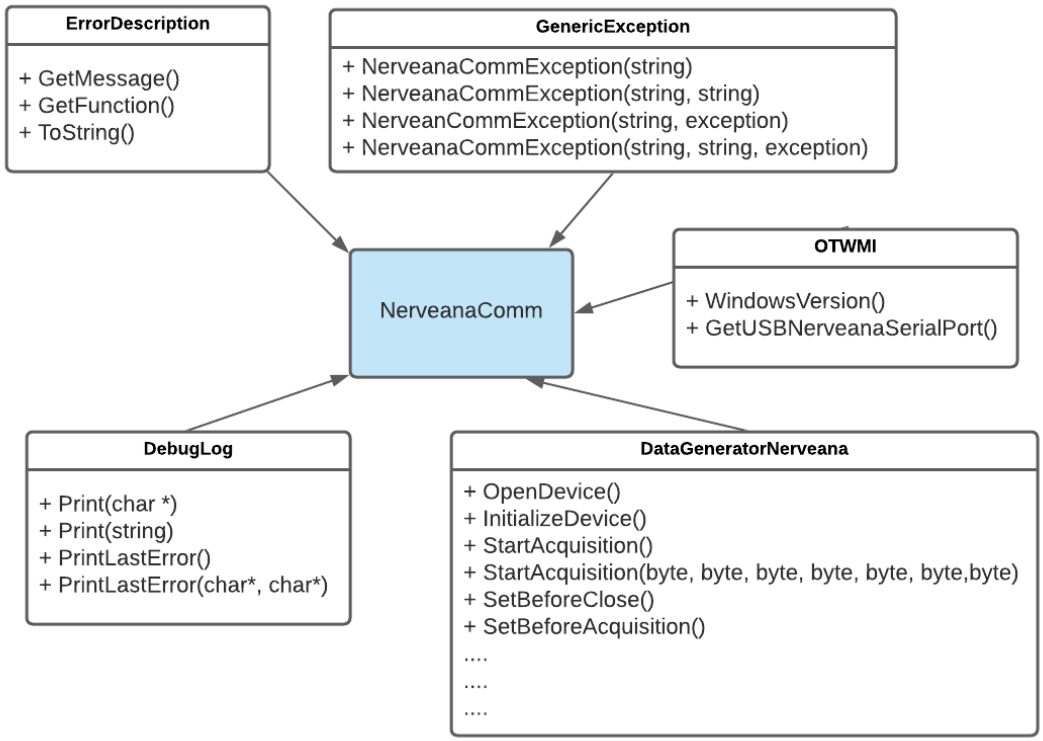


Figure 4.1: NerveanaComm structure

4.1.1 Critical aspects

Communication of a software for intraoperative nerve monitoring is not a joke. The purpose of this software is to be used in operating room, a place where anythings gone wrong causes problems. What if the software crash? What if the software freeze? These are case that must never happen. A priority for this project regards non-blocking functionality, especially during the real-time communication. For this reason asynchronous operation implemented with **OVERLAPPED I/O** were used. When a function is executed synchronously, it does not return until the operation has been completed. This means that the execution of the calling thread can be blocked for an indefinite period while it waits for a time-consuming operation to finish. Functions called for overlapped operation instead can return immediately, even though the operation has not been completed. This enables a time-consuming I/O operation to be executed in the background while the calling thread is free to perform other tasks. For example, a single thread can perform simultaneous I/O operations on different handles, or even simultaneous read and write operations on the same handle. This asynchronous communication paradigm is provided by the

operating system, refer to Charles Petzold's book *Programming Windows* for any technical interest.
To implement this choice three HANDLE variables which will be mentioned in next sections have been defined:

- **mDeviceHandle**: variable used to create the file and allow the asynchronous read/write operations from both software and hardware side.
  The FILE_FLAG_OVERLAPPED flag variable is required.
- **mCompletionPort**: variable created from mDeviceHandle and that associates mDeviceHandle with a port. When an asynchronous I/O operation on the file handle is complete, an I/O completion packet is queued in first-in-first-out (FIFO) order to the associated I/O completion port.
- **mAsyncIoThreadHandle**: handle to the thread in charge to communicate with the hardware and handle all the communication core functionality.

4.1.2 Communication protocol

First of all I want to give an overview of the protocol defined for the hardware main module because the functions that are gonna be explained in future sections make use of this. The USB communication between the eight-channel device and the PC uses a virtual COM implemented in the microcontroller.
The COM port specification are:

Parameter	Value
BaudRate	1000000
number of data bits	8
Parity bit	No
Stop bit	1
Flow control	No

A command string must be sent through the USB port to start the data transfer, to change the hardware configuration or the working mode. The command string starts with a **COMMAND&SIZE** byte and ends with a **CRC8** byte. The COMMAND&SIZE byte determines how many bytes are in the command string and their meaning.

COMMAND&SIZE

COMM2	COMM1	COMM0	SIZE4	SIZE3	SIZE2	SIZE1	SIZE0

COMM<2:0>: Type of command. Determines the meaning of the following bytes in the command string. The detailed description of each command type is in the following pages.

- 100 = Device working mode
- 011 = Advanced EMG settings
- 010 = Advanced stimulation settings
- 001 = Sound tones assignment and test
- 000 = Basic settings and communication

SIZE<4:0>: Size of the command string. It is the number of bytes in the command string including the COMMAND and SIZE byte themselves.

CRC8: It is the CRC estimation on 8 bits obtained from all the byte sent. The device reply with an error code in case something received is wrong. A list of possible errors and corresponding codes are listed in the following table:

Error type	Response code
Wrong CRC	0xCC
Parameters out of range	0x0F
Wrong command length	0xCE
Command unknown	0xC0
Syntax error	0xE0

Basic settings and communication

This command string can be used to start a data transfer and changing the basic settings of the hardware main module or to read the current settings. The bytes in the "Basic settings and communication" command string, are the following:

1. COMMAND&SIZE
2. CONTROL BYTE 0
3. CONTROL BYTE 1
4. CONTROL BYTE 2
5. CONTROL BYTE 3
6. CONTROL BYTE 4
7. FILENAME PREFIX 0
8. FILENAME PREFIX 1
9. FILENAME PREFIX 2

10. FILENAME PREFIX 3
11. FILENAME PREFIX 4
12. FILENAME PREFIX 5
13. TIMEDATE 0
14. TIMEDATE 1
15. TIMEDATE 2
16. TIMEDATE 3
17. CRC8

Not all the bytes are necessary, but at least the CONTROL BYTE 0 must be sent to start the data transfer.

- **CONTROL BYTE 0**

GETSET	FSAMP1	FSAMP0	NCH1
NCH0	MODE1	MODE0	END

GETSET: Describe the type of action

- 1 = GET settings. Regardless of the other bits and bytes sent values, this command requires the hardware current setting. The reply will be a sequence of 33 bytes indicating the current settings. This command can only be sent if the data transfer is not in progress, otherwise it will be discarded
- 0 = SET command. All the other bits and bytes are used to set new values to the hardware settings.

FSAMP<1:0>: Sampling frequency

- 11 = Not used
- 10 = Not used
- 01 = 4000 Hz
- 00 = Not Used

NCH<1:0>: Transferred channels

- 11 = 8 channels

- 10 = Not used
- 01 = Not used
- 00 = Not Used

MODE<1:0>: Working mode

- 11 = Test mode. Start transfer simulated signals and check the functionality of EMG
- 10 = Data transfer with Impedance check.
- 01 = Data transfer without Impedance check.
- 00 = Stop data transfer

END: specifies if there are more byte following

- 1 = there are no other bytes in the command except the CRC8 byte
- 0 = additional bytes follow

- **CONTROL BYTE 1**

LOFF2	LOFF1	LOFF0	STMTYP1
STMTYP0	SDSW	REC	BLANK

LOFF<2:0>: DC Lead-off comparator threshold

- 111 = 95% positive side, 5% negative side
- 110 = 92.5% positive side, 7.5% negative side
- 101 = 90% positive side, 10% negative side
- 100 = 87.5% positive side, 12.5% negative side
- 011 = 85% positive side, 15% negative side
- 010 = 80% positive side, 20% negative side
- 001 = 75% positive side, 25% negative side
- 000 = 70% positive side, 30% negative side

STMTYP<1:0>: Stimulation Type:

- 10 = Biphasic
- 01 = Positive
- 00 = Negative

SDSW: Enable or disable the REC button

- 1 = the REC button on the front panel is enabled
- 0 = the REC button on the front panel is enabled

REC: Starts/stops the acquisition on the MicroSD

- 1 = Recording on SD Card in progress
- 0 = Recording on SD Card is not in progress

BLANK: EMG signal Blanking

- 1 = Short-circuit the EMG inputs during stimulation
- 0 = Do not short-circuit the EMG inputs during stimulation

- **CONTROL BYTE 2**

VOL3	VOL2	VOL1	VOL0
STIM3	STIM2	STIM1	STIM0

VOL<3:0>: Audio volume level 1 to 10
STIM<3:0>: Stimulation level 0 to 10 (0 = no stim, all LEDs off)

- **CONTROL BYTE 3**

0	0	NOTCH_OFF	NTC_60_50
GAIN3	GAIN2	GAIN1	GAIN0

NOTCH_OFF 4: Enable or disable the notch filter

- 1 = Notch filter is OFF
- 0 = Notch filter is ON

NTC_ON 5: Enable or disable the notch filter

- 1 = Notch filter is set on 50 Hz
- 0 = Notch filter is set on 60 Hz

GAIN <3:0>: EMG gain from 1 to 10

- **CONTROL BYTE 4**

AEMG3	AEMG2	AEMG1	AEMG0
CHIRP3	CHIRP2	CHIRP1	CHIRP0

AEMG<3:0>: Channel for AEMG (Audio of one EMG signal) from 0

to 8 (0 = no AEMG and all LEDs off, 1 = Channel1, 2 = Channel2...) **CHIRP<3:0>**: Threshold level from 0 to 9 for the free run chirp function.

- **FILENAME PREFIX 0..5**: Five digits used as prefix for the filenames on the MicroSD card. The following 3 digits will be an incremental number.

- **TIMEDATE 0..3**: Value incremented every 0.25 s. The time 0 is considered 00:00 of January 1ST 2018.

Get settings

In case a command with the GETSET bit equal to 1 is send to the hardware, it replies with 33 bytes:

1. SIZE
2. CONTROL BYTE 0
3. CONTROL BYTE 1
4. CONTROL BYTE 2
5. CONTROL BYTE 3
6. CONTROL BYTE 4
7. FILENAME PREFIX 0
8. FILENAME PREFIX 1
9. FILENAME PREFIX 2
10. FILENAME PREFIX 3
11. FILENAME PREFIX 4
12. FILENAME PREFIX 5
13. TIMEDATE 0
14. TIMEDATE 1
15. TIMEDATE 2
16. TIMEDATE 3

17. BATTERY_LEVEL
18. LED_STATE
19. FIRM_VER0
20. FIRM_VER1
21. FIRM_VER2
22. STIM LEV DUR
23. NPI SETTINGS
24. CHIRP LEVELS
25. TONES EVENTS 1-0
26. TONES EVENTS 3-2
27. TONES EVENTS 5-4
28. TONES EVENTS 7-6
29. TONES EVENTS 9-8
30. TONES EVENTS 11-10
31. TONES EVENTS 13-12
32. TONES EVENTS 15-14
33. CRC8

Sound tones assignment and test

The hardware has the option to let choose the user which sound can be associated to a particular event. The "Sound tones assignment and test" command string allows to set a tones for an event. The "Test tone play" event force the hardware to play a tone as soon as it receives the command.

1. COMMAND&SIZE
2. TONES EVENTS 1-0
3. TONES EVENTS 3-2
4. TONES EVENTS 5-4

5. TONES EVENTS 7-6
6. TONES EVENTS 9-8
7. TONES EVENTS 11-10
8. TONES EVENTS 13-12
9. TONES EVENTS 15-14
10. CRC8

- **COMMAND&SIZE**: This byte can have values between 0x23 (Command = 1, Size = 3) and 0x2A (Command = 1, Size = 10). Refer to the COMMAND&SIZE byte description in the introduction to this section for further details.

- **TONES EVENTS X-Y**

EVNTX3	EVNTX2	EVNTX1	EVNTX0
EVNTY3	EVNTY2	EVNTY1	EVNTY0

EVNTX<3:0>: Tone ID corresponding to event X (odd values)
EVNTY<3:0>: Tone ID corresponding to event Y (even values)

Advanced stimulation settings

This command string allows to set the stimulus duration and the different levels assigned to the position of the stimulation level knob.

Duration ID	Stim. Duration (µs)
0	158
1	300

Lev ID	Stimulation levels (mA)
0	0, 0.25, 0.5, 0.75, 1, 1.5, 2, 2.5, 3, 4, 5
1	0, 0.25, 0.5, 1, 1.5, 2, 2.5, 3, 4, 5, 6
2	0, 0.25, 0.5, 1, 1.5, 2, 3, 4, 5, 6, 8
3	0, 0.5, 1, 1.5, 2, 3, 4, 5, 6, 8, 10
4	0, 0.5, 1, 2, 3, 4, 5, 6, 8, 10, 12
5	0, 0.5, 1, 2, 3, 4, 6, 8, 10, 12, 14
6	0, 1, 2, 3, 4, 6, 8, 10, 12, 14, 16
7	0, 1, 2, 4, 6, 8, 12, 14, 16, 18, 20

The "Advanced stimulation settings" command string allows to set is composed by the following bytes:

1. COMMAND&SIZE
2. STIM LEV DUR
3. CRC8

- **COMMAND&SIZE**: This byte can only have the value 0x43 (Command = 2, Size = 3). Refer to the COMMAND&SIZE byte description in this document introduction for further details.
- **STIM LEV DUR**

LEV3	LEV2	LEV1	LEV0
0	0	0	DURID

LEV<3:0>: Levels ID assigning different stimulation amplitudes to the 10 knob positions
DURID: Duration ID for the stimulation pulse

Advanced EMG settings

This command string allows to set the NPI threshold, the delay with respect to stimulus of the CAP integration window and the chirp thresholds.

NPI ID	NPI Threshold
0	70 % of standard NPI
1	80 % of standard NPI
2	90 % of standard NPI
3	Standard NPI default at start-up
4	110 % of standard NPI
5	120 % of standard NPI
6	not used
7	not used

NPI Delay ID	NPI window delay
0	1.5 ms
1	2.0 ms
2	2.5 ms
3	3.0 ms
4	4.0 ms
5	5.0 ms
6	10.0 ms
7	15.0 ms

Chirp Lev ID	Chirp Thresholds
0	30, 36, 43, 51, 61, 73, 87, 104, 125, 150 uV
1	40, 48, 57, 68, 81, 97, 116, 139, 167, 200 uV
2	48, 58, 69, 83, 100, 120, 144, 173, 208, 250 uV
3	58, 69, 83, 100, 120, 144, 173, 208, 250, 300 uV
4	68, 82, 98, 118, 141, 169, 203, 243, 292, 350 uV
5	78, 93, 112, 134, 161, 193, 232, 278, 333, 400 uV
6	88, 106, 127, 152, 182, 218, 261, 313, 375, 450 uV
7	98, 117, 140, 168, 202, 242, 290, 348, 417, 500 uV

Imped. ID	Impedance Value
0	No threshold
1	1 kΩ
2	2 kΩ
3	3 kΩ
4	4 kΩ
5	5 kΩ
6	6 kΩ
7	7 kΩ
8	8 kΩ
9	9 kΩ
10	10 kΩ
11	12 kΩ
12	14 kΩ
13	16 kΩ
14	18 kΩ
15	20 kΩ
16	25 kΩ
17	30 kΩ
18	35 kΩ
19	40 kΩ
20	45 kΩ
21	50 kΩ
22	60 kΩ
23	70 kΩ
24	80 kΩ
25	90 kΩ
26	100 kΩ
27	120 kΩ
28	150 kΩ
29	200 kΩ
30	220 kΩ
31	250 kΩ

The "Advanced EMG settings" command string is composed by the following bytes:

1. COMMAND&SIZE
2. NPI SETTINGS
3. CHIRP LEVELS

4. CRC8

- **COMMAND&SIZE**:
  This byte can only have the value 0x64 (Command = 3, Size = 4). Refer to the COMMAND&SIZE byte description in this document introduction for further details.

- **NPI SETTINGS**

0	NPIID2	NPIID1	NPIID0
0	DEL2	DEL1	DEL0

  NPIID<2:0>: NPI threshold ID.
  DEL<2:0>: NPI integration window delay ID.

- **CHIRP LEVELS**

IMP4	IMP3	IMP2	IMP1
IMP0	CHIRP2	CHIRP1	CHIRP0

  IMP<4:0>: mpedance threshold levels ID.
  CHIRP<2:0>: Chirp threshold levels ID.

Device Working mode

This command string allows to change the device working mode between Default mode, Diagnosis mode, Calibration mode and Firmware upgrade mode.
The "Device working mode" command string is composed by the following bytes:

1. COMMAND&SIZE
2. WORKING MODE
3. CRC8

- **COMMAND&SIZE**:
  This byte can only have the value 0x83 (Command = 4, Size = 3). Refer to the COMMAND&SIZE byte description in this document introduction for further details.

- **WORKING MODE**

0	0	0	0
0	0	WM1	WM0

WM<1:0>: Device working mode.

- 11 = Set the device in Firmware upgrade mode
- 10 = Set the device in Calibration mode
- 01 = Set the device in Diagnosis mode
- 00 = Set the device in Default mode by resetting it

4.1.3 Packet description

This section want to provide a compact vision of the data with which the packets coming from hardware are built. Each information is associated with an exact byte position inside the packet and this knowledge is used to unpack the container and obtain all the required data.
Some important specification before the table:

1. All bytes in range 202 - 14202 are related to the EMG channels and contains all the data about the signal monitoring.

2. All bytes in range 16212 - 16226 are the NPIs values calculated in hardware main module.

3. All bytes between 16229 and 16259 are the Impedance values detected by the hardware.

All other bytes are related to main and background parameters or other minors information, refer to the name column to understand the scope.

Byte Index	Name	Value	Size in bytes
0	Header	0x00AA	2
2	StimCurrent	-32768 to 32767	200
202	EMGCh1	-32768 to 32767	2000
2202	EMGCh2	-32768 to 32767	2000
4202	EMGCh3	-32768 to 32767	2000
6202	EMGCh4	-32768 to 32767	2000
8202	EMGCh5	-32768 to 32767	2000
10202	EMGCh6	-32768 to 32767	2000
12202	EMGCh7	-32768 to 32767	2000
14202	EMGCh8	-32768 to 32767	2000
16202	StimLev	0 to 10	1
16203	Gain	1 to 10	1
16204	Volume	1 to 10	1
16205	ChirpThres	0 to 10	1
16206	AEMGChan	0 to 8	1
16207	MuteLED	0 to 3	1
16208	EventLED	0 to 3	1
16209	Ext12VLED	0 to 3	1
16210	BatteryLED	0 to 3	1
16211	SDCardLED	0 to 3	1
16212	NPI1	0 to 65535	2

16214	NPI2	0 to 65535	2
16216	NPI3	0 to 65535	2
16218	NPI4	0 to 65535	2
16220	NPI5	0 to 65535	2
16222	NPI6	0 to 65535	2
16224	NPI7	0 to 65535	2
16226	NPI8	0 to 65535	2
16228	ActiveChan	0 to 255	1
16229	ImpedCh1P	0 to 65535	2
16231	ImpedCh1N	0 to 65535	2
16233	ImpedCh2P	0 to 65535	2
16235	ImpedCh2N	0 to 65535	2
16237	ImpedCh3P	0 to 65535	2
16239	ImpedCh3N	0 to 65535	2
16241	ImpedCh4P	0 to 65535	2
16243	ImpedCh4N	0 to 65535	2
16245	ImpedCh5P	0 to 65535	2
16247	ImpedCh5N	0 to 65535	2
16249	ImpedCh6P	0 to 65535	2
16251	ImpedCh6N	0 to 65535	2
16253	ImpedCh7P	0 to 65535	2
16255	ImpedCh7N	0 to 65535	2
16257	ImpedCh8P	0 to 65535	2
16259	ImpedCh8N	0 to 65535	2
16261	ImpedREF	0 to 65535	2
16263	AudioTone	0 to 255	1
16264	BatteryLevel	0 to 4096	2
16266	VoltExt	0 to 4096	2
16268	Curr12V	0 to 4096	2
16270	CurrBatt	0 to 4096	2
16272	CurrPatientMod	0 to 4096	2
16274	Time	0 to 65535	4
16278	UniqueHWID	0 to 65535	4
16282	CONTROL BYTE 0	0 to 255	1
16283	CONTROL BYTE 1	0 to 255	1
16284	CONTROL BYTE 2	0 to 255	1
16285	CONTROL BYTE 3	0 to 255	1
16286	CONTROL BYTE 4	0 to 255	1
16287	FILENAME PREFIX	-	6

16293	FIRM_VER	-	3
16296	STIM_LEV_DUR	0 to 255	1
16297	NPI SETTINGS	0 to 255	1
16298	CHIRP LEVELS	0 to 255	1
16299	TONES EVENT 1-0	0 to 255	1
16300	TONES EVENT 3-2	0 to 255	1
16301	TONES EVENT 5-4	0 to 255	1
16302	TONES EVENT 7-6	0 to 255	1
16303	TONES EVENT 9-8	0 to 255	1
16304	TONES EVENT 11-10	0 to 255	1
16305	TONES EVENT 13-12	0 to 255	1
16306	TONES EVENT 15-14	0 to 255	1
16307	EventFlags	0 to 255	1
16308	Dummy	0	74
16382	Footer	0x0055	2

4.1.4 NerveanaComm classes

Before explaining the main class DataGeneratorNerveana I want to briefly give a look at the other four classes that composes NerveanaComm:

- **ErrorDescription**: used by DataGenerator to encapsulate the exception and fire an error event.
- **DebugLog**: provide a set of function for debugging purpose and most of them are print functions. Custom message are allowed but also some specific print are defined:
  - LOG_ENTERING
  - LOG_EXITING
  - LOG_SUCCESS
  - LOG_MESSAGE(message)
  - LOG_ERROR(error)
  - LOG_EXCEPTION(exception)
- **GenericException**: inside this class a custom exception class called Nerveana† CommException is defined together with a set of exceptions, each of them with its own custom message. The defined exception are:
  - OpenDeviceFailedException

  - CloseDeviceFailedException
  - InitializeDeviceFailedException
  - StartAcquisitionFailedException
  - StopAcquisitionFailedException
  - InvalidHandleException
  - CompletionPortException
  - SendCommandFailedException
  - SetCommFailedException
  - GetCommFailedException
  - AcquisitionThreadException
  - ThreadCreationException

- **OTWMI**: contains functions helpful for windows management instrumentation. Two functions are defined: the first is WindowsVersion() and it's used to obtain the windows version installed on local machine and the second one is GetUSBNerveanaSerialPort() and as can be guessed from the name search for the correct COM port where the nerveana hardware is detected.

4.1.5 DataGeneratorNerveana

Now, I can finally focus on the main class **DataGeneratorNerveana** and describe the functions that belongs to it. DataGeneratorNerveana class handle the communication with hardware main module, including commands creation and commands sending and contains also a set of event that allow to pass the received values to the graphical interface or raise specific function call. DataGenerator and GUI are linked together so that while one is in charge of receiving packet from hardware the second one can draw the EMG signals on the main window. Or, viceversa, the graphical interface can capture the user intention to change knobs value or activate a features and using methods of the DataGenerator can create and deliver a command to the hardware.
DataGeneratorNerveana define an object that is initialized in main window and through its set of methods allow to control and communicate with the hardware main module. The events defined in the class are used to raise event from the object to the GUI and let it handle the situation and are the following:

- **StartAcquisition**: Warns the graphical interface that the acquisition is started.

- **StopAcquisition**: Warns the graphical interface that the acquisition is ended.

- **NewDataAvailable**: This event is the most important one and is used to encapsulate the data received from hardware and pass them to the graphical interface. Every time that a new packet has arrived this event is raised and the signals draw in the graphical interface are updated.
- **CloseDevice**: Warns the graphical interface that the device is closed.
- **Error**: Alert about an error encountered.
- **InizializeDevice**: Event raised when the device connection has been initialized.
- **OpenDevice**: Event raised when the device connection is opened.

DataGeneratorNerveana contains also a list of functions of different uses. For simplicity I divide the functions in three types:
Main functions, that contains all the most important functions that make use of the other two types to work
Command Creation functions, that contains all the functions that create command string depending of parameters or hardware features the user want to activate.
Command sending functions, that take care of delivering the command to the main hardware module. In figure 4.3 is shown the relationship between the main functions and the others while relationships between the creation and sending functions is shown in figure 4.3.

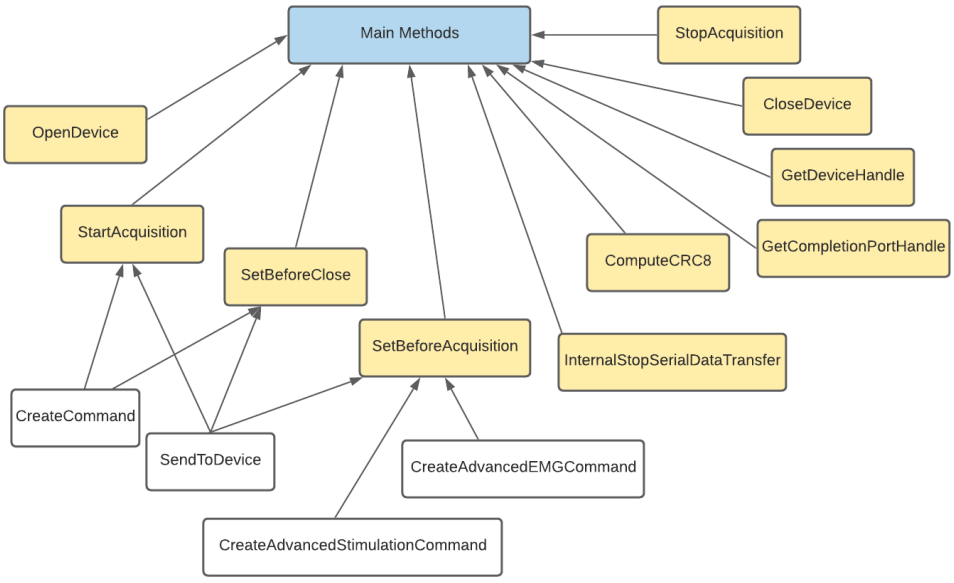


Figure 4.2: Relationship between main functions and others

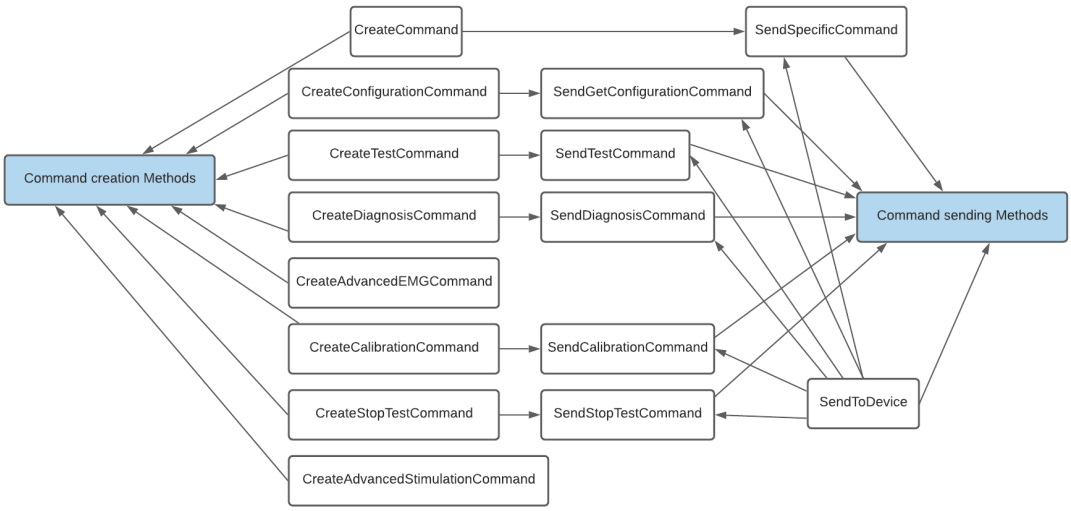


Figure 4.3: Relationship between creation and sending commands

Main functions

The main functions are used for any kind of purpose, they make use of the other two sets of functions in order to achieve their goal and they define the main functionalities of the communication library. Functions that belong to this section are:

- **OpenDevice**: This function creates the mDeviceHandle and overwrites the default port settings with the specification described at the beginning of section 4.1.2. Then it creates the mCompletionPortHandle and return.
- **StartAcquisition**: This function is used to start the acquisition. First it cancels any I/O operation on the mDeviceHandle, than it takes the settings variables passed to the function and call the CreateCommand function to create the array of byte. At the end it sends the string command using SendToDevice method.
- **SetBeforeClose**: This function is used before closing the communication with the device and has a simple purpose: mute the hardware. This is done creating a string command with "0" as index for the setting, in particular for the Audio Volume and send the string to the hardware.
- **SetBeforeAcquisition**: Before starting the communication with hardware, the settings needs to be configured to the one chosen by operator and to do so this function is used. Two string commands must be created, the first regards the stimulation settings, stimulation diration and stimulation

range, and the second is about the EMG settings, NPI integration threshold, NPI window delay and Chirp set of levels. The two commands are prepared using the functions **CreateAdvancedStimulationCommand** and **CreateAdvancedEMGCommand** and once they're ready, they are sent to hardware using SendToDevice.

- **StopAcquisition**: This method simply closes the mCompletionPortHandle, stopping the communication.
- **CloseDevice**: This method closes the mDeviceHandle.
- **GetDeviceHandle**: Returns the mDeviceHandle variable.
- **GetCompletionPortHandle**: Returns the mCompletionPortHandle variable.
- **InternalStopSerialDataTransfer**: This function creates the stop command and send it to the hardware, then it cancel any IO operation on mDeviceHandle.
- **ComputeCRC8**: Implement the same CRC function used on hardware to check correctness of data.
- **OuternThread**: Starts a new thread and assign the ASyncIoSerialNerveana as working function.
- **AsyncIoSerialNerveana**: This function is the heart of NerveanaComm and take care of read/write operation on the communication socket. It's an adaptation from a function contained in OTComm library and it is the only unmanaged code part of the entire library. While all the rest of the DataGenerator class is managed code, that means that is internally translated into a pseudocode and interpreted at runtime, the AsyncIoSerialNerveana function is compiled directly into machine code.

Command creation functions

The Creation functions are used to build the correct array of byte to send to the hardware module. This array is also called command, because it's interpreted from the hardware and activate particular behaviour. The array of byte is written taking into account the protocol shown in section 4.1.2. There are two type of creation functions, those that make use of passed parameters and those who use predefined parameters and activate specific hardware functionalities. This section is growing with the incoming new request from the customer but at the time of writing of this thesis the functions that belong to this part are:

- **CreateCommand**: This method is the most general command creation, a functions to which are passed all variable related to the settings: workmode, stimulation, gain, audio, chirp, AEMG, blank, stimtype and create the array. It's defined as follows:

```
1  array<Byte>^ CreateCommand(byte workmode, byte stim, byte
   gain, byte audio, byte chirp, byte AEMG, byte blank, byte
   stimtype){
2      laststim = stim;
3      lastgain = gain;
4      lastaudio = audio;
5      lastchirp = chirp;
6      lastAEMG = AEMG;
7      lastblank = blank;
8      laststimtype = stimtype;
9      int commandlength = 17;
10     array<Byte>^ command = gcnew array<Byte>(commandlength);
11     command[0] = commandlength;
12     command[1] =
13         GetSets::Set << 7 |
14         FSamples::FSample_4k << 5 |
15         NChannels::NChannel_8 << 3 |
16         workmode << 1;
17     command[2] =
18         LOFFs::LOFF_95 << 5 |
19         stimtype <<3|
20         SDSWs::SDSW_enabled << 2 |
21         RECs::REC_notinprogress << 1 |
22         blank;
23     command[3] =
24         audio << 4 | stim;
25     command[4] = gain;
26     command[5] = AEMG<<4|chirp;
27     command[6] = (Byte)'N';
28     command[7] = (Byte)'V';
29     command[8] = (Byte)'0';
30     command[9] = (Byte)'8';
31     command[10] = (Byte)'_';
32     command[11] = (Byte)'_';
33     command[12] = 0;
34     command[13] = 0;
35     command[14] = 0;
36     command[15] = 0;
37     command[commandlength-1] = ComputeCRC8(command,
   commandlength-1);
38     return command;
39 }
```

```
40
```

- **CreateAdvancedStimulationCommand**: This method is used to create an array of bit for setting the background parameters related to the stimulator: stimduration (158 us or 300 us) and stimulation level (0-5 mA, 0-10 mA or higher). It takes the value passed and creates the command.

```
1     array<Byte>^ DataGeneratorNerveana::
      CreateAdvancedStimulationCommand(byte stimdur, byte stimlevID)
      {
2         int commandlength = 3; // [0010 0011] -> 0010 command -
      0011 size
3         array<Byte>^ command = gcnew array<Byte>(commandlength);
4         command[0] = commandlength +64;
5         command[1] = stimlevID << 4 | stimdur;
6         command[3 - 1] = ComputeCRC8(command, commandlength- 1);
7         return command;
8     }
9
```

- **CreateAdvancedEMGCommand**: This is the second and last method used to set background parameters, in this case: NPI integration threshold, NPI window delay and the chirp level, not the single value but the index of the set of values assigned to the ten knob position.

```
1     array<Byte>^ DataGeneratorNerveana::CreateAdvancedEMGCommand(
      byte NPIThreshold, byte NPIDelay, byte chirpLevel){
2         int commandlength = 100; //[0110 0100] -> 011 command -
      0100 size
3         array<Byte>^ command = gcnew array<Byte>(4);
4         command[0] = commandlength;
5         command[1] = NPIThreshold << 4 | NPIDelay;
6         command[2] = chirpLevel;
7         command[4 - 1] = ComputeCRC8(command, 4 - 1);
8         return command;
9     }
10
```

- **CreateDiagnosisCommand**: This method is used to create the command that set the hardware in diagnosis mode, a mode where the hardware send to the software a list of internal information. This command does not require any parameters or additional information.

```
array<Byte>^ DataGeneratorNerveana::CreateDiagnosisCommand(){
    int commandlength = 3; //[0010 0011] -> 0010 command - 0011 size
    array<Byte>^ command = gcnew array<Byte>(commandlength);
    command[0] = commandlength + 128; //device working mode
    command[1] = 01; //diagnosis mode
    command[commandlength - 1] = ComputeCRC8(command, commandlength - 1);
    return command;
}

```

- **CreateCalibrationCommand**: This is used to create the command that set the hardware in calibration mode.

```
array<Byte>^ DataGeneratorNerveana::CreateCalibrationCommand(){
    int commandlength = 3; //[0010 0011] -> 0010 command - 0011 size
    array<Byte>^ command = gcnew array<Byte>(commandlength);
    command[0] = commandlength + 128; //device working mode
    command[1] = 10; //calibration mode
    command[commandlength - 1] = ComputeCRC8(command, commandlength - 1);
    return command;
}

```

- **CreateConfigurationCommand**: This method creates the command used to receive the current hardware configuration setting and know all the parameters. No additional variable are required.

```
array<Byte>^ DataGeneratorNerveana::CreateConfigurationCommand() {
    int commandlength = 3;
    array<Byte>^ command = gcnew array<Byte>(commandlength);
    command[0] = commandlength;
    command[1] =
        GetSets::Get << 7 | //GET CONFIGURATION
        FSamples::FSample_4k << 5 |
        NChannels::NChannel_8 << 3 |
        WorkingModes::WM_NoImp << 1 |
        1; //END 1 no other bytes except CRC8
```

```
11          command[commandlength - 1] = ComputeCRC8(command,
    commandlength - 1);
12          return command;
13      }
14
```

- **CreateTestCommand**: This method creates the command for the Test mode, used to read the impedance values.

```
1       array<Byte>^ DataGeneratorNerveana::CreateTestCommand(){
2           int commandlength = 3;
3           array<Byte>^ command = gcnew array<Byte>(commandlength);
4           command[0] = commandlength;
5           command[1] =
6               GetSets::Set << 7 |
7               FSamples::FSample_4k << 5 |
8               NChannels::NChannel_8 << 3 |
9               WorkingModes::WM_Test << 1 |
10              1;
11          command[commandlength - 1] = ComputeCRC8(command,
    commandlength - 1);
12          return command;
13      }
14
```

- **CreateStopTestCommand**: This function creates the command to stop the Test mode.

```
1           array<Byte>^ DataGeneratorNerveana::CreateStopTestCommand
    (){
2           int commandlength = 3;
3           array<Byte>^ command = gcnew array<Byte>(commandlength);
4           command[0] = commandlength;
5           command[1] =
6               GetSets::Set << 7 |
7               FSamples::FSample_4k << 5 |
8               NChannels::NChannel_8 << 3 |
9               01 << 1 |
10              1;
11          command[3 - 1] = ComputeCRC8(command, commandlength - 1);
12          return command;
13      }
14
```

Command sending functions

The last functions section regards all the functions that actually send the command to the hardware module writing the buffer to the mDeviceHandle variable and they are twisted with the creation methods in order to create the correct command.The functions that belongs to this section are:

- **SendToDevice**: This is the general case where the command is passed as a buffer variable and the functions simply take care of it and write on the handle using the overlapped variable.

```
1   BOOL DataGeneratorNerveana::SendToDevice(byte *buf, int
    length){
2       static OVERLAPPED mOverlappedStrucSendCommand;
3       ZeroMemory(&mOverlappedStrucSendCommand, sizeof(
    OVERLAPPED));
4       mOverlappedStrucSendCommand.Internal = STATUS_PENDING;
5       SetLastError(0);
6       BOOL WriteFileRes = WriteFile(mDeviceHandle, buf, length,
     NULL, &mOverlappedStrucSendCommand);
7       return true;
8   }
9
```

- **SendSpecificCommand**: Instead of passing the command itself this functions was implemented to pass the command parameters to the functions and let it take care of creating the command. This is done using the CreateCommand general case functions and once it is created, a call to SendToDevice is made.

```
1   void DataGeneratorNerveana::SendSpecificCommand(byte workmode
    , byte stim, byte gain, byte audio, byte chirp, byte AEMG,
    byte blank, byte stimtype){
2       LOG_ENTERING;
3       {
4           array<Byte>^ buf_array = CreateCommand(workmode, stim
    , gain, audio, chirp, AEMG, blank, stimtype); // Decimal value
     to write to serial port
5           byte buf[128];
6           for (int ii = 0; ii < buf_array->Length; ii++)
7               buf[ii] = buf_array[ii];
8           SendToDevice(buf, buf_array->Length);
9           LOG_MESSAGE("Nerveana: Specific Command Sent");
10      }
```

```
11          LOG_EXITING;
12      }
13
```

- **SendDiagnosisCommand**: This method creates the diagnosis command calling the creation function and send it to the hardware.

```
1       void DataGeneratorNerveana::SendDiagnosisCommand()
2       {
3           LOG_ENTERING;
4           InternalStopSerialDataTransfer();
5           array<Byte>^ buf_array = CreateDiagnosisCommand();
6           byte buf[128];
7           for (int ii = 0; ii < buf_array->Length; ii++)
8               buf[ii] = buf_array[ii];
9           SendToDevice(buf, buf_array->Length);
10          {
11              try
12              {
13                  mMustStopAcquisition = false;
14                  FireOnStartAcquisition();
15              }
16              catch (Exception^exec)
17              {
18                  throw gcnew StartAcquisitionFailedException(
    __FUNCTION__, exec);
19              }
20          }
21          Sleep(100);
22          LOG_EXITING;
23      }
24
```

- **SendCalibrationCommand**: This method creates the calibrarion command calling the creation function and send it to the hardware.

```
1       void DataGeneratorNerveana::SendCalibrationCommand()
2       {
3           LOG_ENTERING;
4           {
5               array<Byte>^ buf_array = CreateCalibrationCommand();
6               byte buf[128];
7               for (int ii = 0; ii < buf_array->Length; ii++)
8                   buf[ii] = buf_array[ii];
9               SendToDevice(buf, buf_array->Length);
```

```
10              LOG_MESSAGE("Nerveana: Calibration Command Sent");
11          }
12          LOG_EXITING;
13      }
14
```

- **SendTestCommand**: This method creates the Test command calling the creation function and send it to the hardware.

```
1       void DataGeneratorNerveana::SendTestCommand()
2       {
3           LOG_ENTERING;
4           {
5               array<Byte>^ buf_array = CreateTestCommand();
6               byte buf[128];
7               for (int ii = 0; ii < buf_array->Length; ii++)
8                   buf[ii] = buf_array[ii];
9               SendToDevice(buf, buf_array->Length);
10              LOG_MESSAGE("Nerveana: Test Command Sent");
11          }
12          LOG_EXITING;
13      }
14
```

- **SendStopCommand**: This method creates the command to stop the Test mode calling the creation function and send it to the hardware.

```
1       void DataGeneratorNerveana::SendStopTestCommand()
2       {
3           LOG_ENTERING;
4           {
5               array<Byte>^ buf_array = CreateStopTestCommand();
6               byte buf[3];
7               for (int ii = 0; ii < buf_array->Length; ii++)
8                   buf[ii] = buf_array[ii];
9               SendToDevice(buf, buf_array->Length);
10              LOG_MESSAGE("Nerveana: Stop Test Command Sent");
11          }
12          LOG_EXITING;
13      }
14
```

- **SendGetConfigurationCommand**: This method creates the command to get the hardware current settings and send it to the hardware.

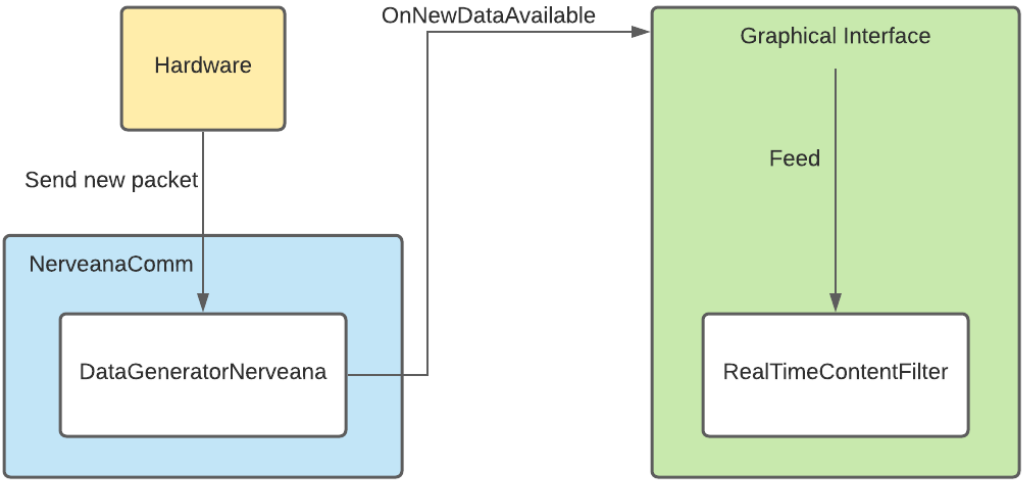
```
1   void DataGeneratorNerveana::SendGetConfigurationCommand() {
2       LOG_ENTERING;
3       {
4           array<Byte>^ buf_array = CreateConfigurationCommand()
    ;
5           byte buf[118];
6           for (int ii = 0; ii < buf_array->Length; ii++)
7               buf[ii] = buf_array[ii];
8           LOG_MESSAGE("Nerveana: CONFIGURATION Command Sent");
9       }
10      LOG_EXITING;
11  }
12
```

4.1.6 How events works

There are few important moments in the life of a packet. They are born in the hardware and through the USB cable they arrive in software where are captured and transformed in graphical data to be plotted on the main screen. They are exchanged using event triggers between the part that are involved in the process, starting from the hardware and the overlapped structure up to the main plot. This is done using events triggering and works as follows:

- A new packet comes from the hardware to the Overlapped structure
- DataGeneratorNerveana captures the packet and raise the OnNewDataAvailable event
- GUI captures the OnNewDataAvailable event and gets the values
- GUI feeds the RealTimeContentFilter with the new data

In detail the hardware module send four packets each second to the software and as show in section 4.1.3 the packet size is 16384 bytes. This events triggering works very quickly in order to not fill the buffer to the limit and do not cause a loss of information. The following image simply depict the functioning.



4.1.7 Data unpacking

At the beginning of this process the packet is captured from the main form in charge to draw the EMG signals and all other useful information, and the data are unpacked and stored inside different structure. While the next section explain how the data are handled, this one is a step before and show how the unpack is done. There are three functions that unpacks the data and are obviously related to the three most important information:

1. **EMG channels**
2. **NPIs values**
3. **Impedance**

GetDataMatrixFromPacket

This function take all the EMG channels values from the packet, passed as an argument from the raised event OnNewDataAvailable as seen in previous section. Its purpose is to encapsulate all the data inside a matrix and to do so it make use of a set of variables that points to the exact byte position of the EMG channels inside the packet. For example a variable called *mShiftChannelEMG1* is defined and represent the starting point for the first EMG channel and, as can be deduced from section 4.1.3, its value is 202. After 2000 byte, at position 2202 the packet contains values of the second EMG channels and this position is stored inside the

mShiftChannelEMG2 variable. This goes on for a bunch of variables but not only for the EMG channels, a whole series of variables are defined to know where is located each precise information. This part will be covered and discussed more in detail in the next section.
What the function do next is iterating for the next one thousand sample and collect the data. Successively they are converted to double and multiplied for a gain factor, due to zoom feature (refer to chapter 4.3), and for a conversion factor of *0.2861* to obtain the uV value.
In the following code sections there is a function called *FindNextIndex* that will be discussed in chapter 4.3.

```
1  private double[][] GetDataMatrixFromPacket(NewDataAvailableEventArgs
       packet, double gain, int zoomIndex)
2  {
3      double[][] dataAvailable = new double[m_AcquisitionTracks.Count
       ][];
4      for (int ii = 0; ii < dataAvailable.Length; ii++)
5      {
6          int maxValueAvailable = 1000;
7          if (zoomLevels[zoomIndex] * 4000.0 < maxValueAvailable)
8              maxValueAvailable = (int)(zoomLevels[zoomIndex] * 4000.0)
       ;
9          dataAvailable[ii] = new double[maxValueAvailable];
10         FindNextIndex();
11         for (int jj = 0; jj < 2 * maxValueAvailable; jj += 2)
12         {
13             int pos = mShiftChannelEMG1 + lastK * 2000 + jj;
14             double val = packet.Data.data8[pos] + 256 * packet.Data.
       data8[pos + 1];
15             dataAvailable[ii][jj / 2] = (val < 32768 ? val : val -
       65536) /gainArray[(int)gain-1] *0.2861;
16         }
17     }
18     lastK = -1;
19     return dataAvailable;
20 }
```

GetDataMatrixFromPacketNPI

This function is used to collect the NPIs values from the packet. As the previous function it make use of a variable, called *mShiftChannelNPI1* that point to the first NPI byte inside the packet and differently from the previous function it iterate for eight times, that are the number of channels to which each NPI refers. Then it collect the two byte value (NPIs sent as a int16 type) and the value is converted,

no additional conversion are required. Also in this case there's a function called *FindNextNPIIndex* that will be discussed in chapter 4.3.

```
private int[] GetDataMatrixFromPacketNPI(NewDataAvailableEventArgs
    packet)
{
    int[] dataAvailable = new int[m_AcquisitionTracks.Count];
    for (int ii = 0; ii < dataAvailable.Length; ii++)
    {
        int pos = mShiftChannelNPI1 + FindNextNPIIndex(ii + 1) * 2;
        int val = packet.Data.data8[pos] + 256 * packet.Data.data8[
    pos + 1];
        dataAvailable[ii] = val < 32768 ? val : val - 65536;
    }
    return dataAvailable;
}
```

GetDataMatrixFromPacketImpedance

The last but not the least is the function for the impedance values. It has the same behaviour of the previous function but it use the *mShiftChannelImpedence* variable and also convert the value received in kOhm.

```
private int[] GetDataMatrixFromPacketImpedence(
    NewDataAvailableEventArgs packet)
{
    int[] dataAvailable = new int[8];
    for (int ii = 0; ii < dataAvailable.Length; ii++)
    {
        int pos = mShiftChannelImpedence + FindNextNPIIndex(ii + 1) *
     4;
        int val = packet.Data.data8[pos]+ 256*packet.Data.data8[pos
    +1];
        dataAvailable[ii] = (int)(val * 5.8 / 1000);//((val < 32768 ?
     val : val - 65536));
    }
    return dataAvailable;
}
```

All other real time informations, like led blinking, time, battery levels or number of active channels are of course contained inside the packet but they're taken by the GUI only when's needed using the appropriate variable for the number of bytes to shift.

4.1.8 Data queues

In order to understand how the data are handled I want to explain where they are contained. The class defined for this purpose is **RealTimeContentFilter** and also if it is not part of the communication library is used to contains the data in the GUI part. The RealTimeContentFilter class contains many variables and functions but the most important are three linked lists that are used to contains data together with the **Feed** function. As seen in section 4.1.3 the hardware build packets with a lot of information and software does not need to store all of it during the real time visualization. What's very important to the user and requires a special management at each new packet are the EMG channels, NPIs and Impedance values. That's why the three linked list are:

- LinkedList<double[]> **dataQueue**: used to store the EMG channels that must be drawn on the main screen.

- LinkedList<int> **dataQueueNPI**: the values of NPIs calculated by hardware.

- LinkedList<int> **dataQueueImpedance**: Impedance values detected by hardware.

When the OnNewDataAvailable event is captured the Feed function is called and the values are passed to the RealTimeContentFilter that add the three different type of data to the three list. DataQueueNPI and dataQueueImpedence lists are handled with FIFO (first in first out) queue management, using a variable to determine the maximum buffer limit for the list, when the limit is reached a pop operation is performed.
Unlike the other two the dataQueue made use of another queue management and the important assumption on its task is: the user wants to look at the signal as long as possible. Using a FIFO queue means having the EMG channels draw shift left (or right, depending on draw) at each new packet. This is absolutely fine with Impedance value and NPIs drawn on screen because the user just want to keep looking at the last value but for EMG channels this is bad. The user want to monitor at the channels and if the signal is moving he/she need to keep tracking the signal with the eyes while it's shifting out of the window. To overcome this problem the idea is to keep a slots window and overwrite each slot at its specific index location.
Suppose the user wants to monitor at the last five packets. As long as the buffer is empty the Feed function keep filling the buffer.

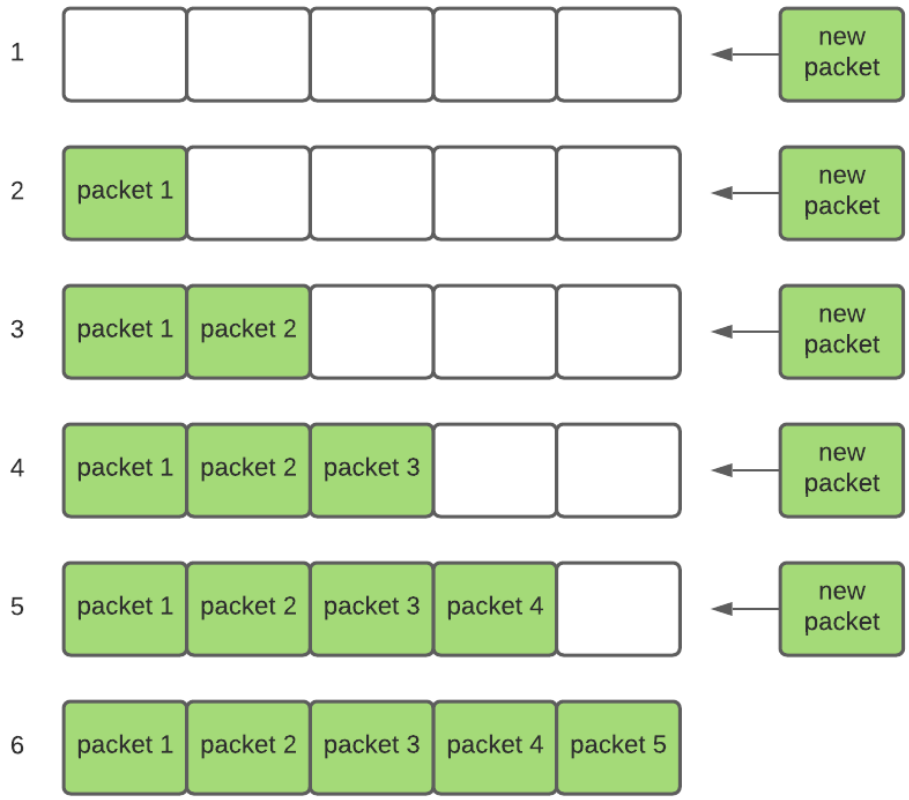


Figure 4.4: Buffer filling

Once the buffer is full the new packet does not push in and pop out the packet 1 but instead, it overwrites the packet 1 as shown in figure 4.5. The consequence of this choice is that when a packet comes and is put inside the list, it stays at the same exact position until a new packet takes its place. Supposing a slots window of five slot, packet 1 at position 1 stands still for five packet and do not ever shift. The operator does not need anymore to keep tracking the signal with the eyes.

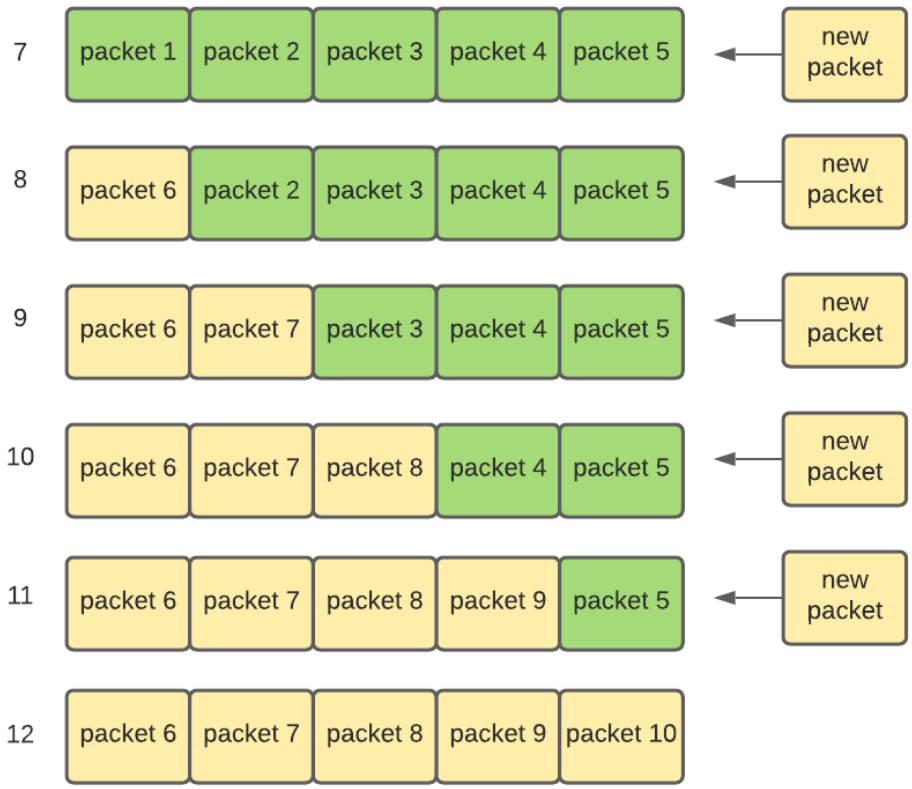


Figure 4.5: Buffer overwriting

The code of the function used to feed the three lists is the following

```
public void Feed2018a(double[] data, int NPI, int Impedence, double
    VoltagePeak, int Delay)
{
    this.VoltagePeak = VoltagePeak;
    this.Delay = Delay;
    if (current_index == feedBufferLimit) empty = false;
    if (empty){
        //At the beginning you need to initialize enough node in the
    linkedList
        if(dataQueue.First == null || dataQueue.Count <
    feedBufferLimit)
        {
            for (int i = 0; i < feedBufferLimit-1; i++)
            {
                LinkedListNode<double[]> lastNode = new
    LinkedListNode<double[]>(new double[200]);
```

```
13                  dataQueue.AddFirst(lastNode);
14                  dataQueueNPI.AddFirst(0);
15              }
16              empty = false;
17              dataQueue.AddFirst(data);
18              dataQueueNPI.AddFirst(NPI);
19              dataQueueImpedence.AddFirst(Impedence);
20
21          }
22      }
23      else {
24          if(current_index == feedBufferLimit) current_index = 0;
25          int i = 0;
26          LinkedListNode<double[]> nodeToSubstitute = null;
27          for (LinkedListNode<double[]> node = dataQueue.First; node !=
        null; node = node.Next)
28          {
29              if (i == current_index)
30              {
31                  nodeToSubstitute = node;
32                  break;
33              }
34              i++;
35          }
36          dataQueue.AddAfter(nodeToSubstitute, data);
37          dataQueue.Remove(nodeToSubstitute);
38          i = 0;
39          LinkedListNode<int> nodeNPIToSubstitute = null;
40          for (LinkedListNode<int> node = dataQueueNPI.First; node !=
       null; node=node.Next)
41          {
42              if (i == current_index)
43              {
44                  nodeNPIToSubstitute = node;
45                  break;
46              }
47              i++;
48          }
49          dataQueueNPI.AddAfter(nodeNPIToSubstitute, NPI);
50          dataQueueNPI.Remove(nodeNPIToSubstitute);
51          if (dataQueueImpedence.Count > feedBufferLimit)
52          {
53              dataQueueImpedence.Clear();
54          }
55          dataQueueImpedence.AddFirst(Impedence);
56      }
57      current_index++;
58  }
```

4.2 LoaderNerveana: Data serialization

Data storage in an important task for future evaluation. The EMGView8 software must allow to open an old examination and show data and information exactly as they were during the surgery operation and this could be possible thanks to **NerveanaContentFilter**. This class is one of the two class the composes the **LoaderNerveana** sub-project, together with the **LoaderNerveanaPlugin** that contains the most important function for data loading used from the GUI. NerveanaContentFilter is the real container of the data given the filenames of the stored raw data, saved during the real-time visualization. Let's start with order.

4.2.1 NerveanaContentFilter

This class is used to encapsulate the data for an offline visualization. The longer the surgery operation, the more are the files that are stored, that's why it's associated with a set of files. It uses a subclass called **NerveanaStream** to open a file stream and manage file variable like number of packets, packet size, header position and so on. NerveanaContentFilter define also a list of static variable that identify the position of a specific information inside the packet. How it was shown in section 4.1.3 the packet has a specific byte position for each important information and when data are loaded in software it must obtain the exact information it needs for each packet of the file in order to reconstruct the same visual examination as if it was real-time. Defining a set of position it ensure an easy pick inside the packet for the specific value. Overwhelming the list it's easy with dozens of position but not all the details are useful for an offline visualization. Only index about information the software must display are defined as position variable and are the following:

- PositionStimulationByte : 16202
- PositionGainByte : 16203
- PositionVolumeByte : 16204
- PositionChirpByte : 16205
- PositionAEMGByte : 16206
- PositionMuteByte : 16207
- PositionEventLED : 16208
- PositionExt12VByte : 16209
- PositionBatteryLed : 16210

- PositionSDCardLevByte : 16211
- PositionActiveChannels : 16228
- PositionAudioToneByte : 16263
- PositionBatteryLevel : 16264
- PositionTime : 16274
- PositionUniqueHWID : 16280
- PositionEventFlags : 16309

From the list some important positions are missing, in particular EMG channels, NPIs and Impedance but they're defined directly inside the functions that make uses of position. **GetShiftsPerPacketDependingOnTrackIndex** function take an index value and return the exact location of the value inside the packet and is used during the creation of a NerveanaContentFilter to determine which data will be encapsulated in the object.

```
static public int GetShiftsPerPacketDependingOnTrackIndex(int index)
{
    switch (index)
    {
        case ChannelIndex.Header:
            return 0;
        case ChannelIndex.StimCurrent:
            return 2;
        case ChannelIndex.Gain:
            return PositionGainByte;
        case ChannelIndex.EventLed:
            return PositionEventLED;
        case ChannelIndex.StimLev:
            return PositionStimulationByte;
        case ChannelIndex.SDCardLev:
            return PositionSDCardLevByte;
        case ChannelIndex.Mute:
            return PositionMuteByte;
        case ChannelIndex.Volume:
            return PositionVolumeByte;
        case ChannelIndex.Ext12V:
            return PositionExt12VByte;
        case ChannelIndex.BatteryLed:
            return PositionBatteryLed;
        case ChannelIndex.AudioTone:
            return PositionAudioToneByte;
```

```
27          case ChannelIndex.NPI_1:
28              return 16212;
29          case ChannelIndex.NPI_2:
30              return 16214;
31          case ChannelIndex.NPI_3:
32              return 16216;
33          case ChannelIndex.NPI_4:
34              return 16218;
35          case ChannelIndex.NPI_5:
36              return 16220;
37          case ChannelIndex.NPI_6:
38              return 16222;
39          case ChannelIndex.NPI_7:
40              return 16224;
41          case ChannelIndex.NPI_8:
42              return 16226;
43          case ChannelIndex.Time:
44              return 16276;
45          case ChannelIndex.EMG_1:
46              return 202;
47          case ChannelIndex.EMG_2:
48              return 2202;
49          case ChannelIndex.EMG_3:
50              return 4202;
51          case ChannelIndex.EMG_4:
52              return 6202;
53          case ChannelIndex.EMG_5:
54              return 8202;
55          case ChannelIndex.EMG_6:
56              return 10202;
57          case ChannelIndex.EMG_7:
58              return 12202;
59          case ChannelIndex.EMG_8:
60              return 14202;
61          case ChannelIndex.IMP_1P:
62              return 16230;
63          case ChannelIndex.IMP_2P:
64              return 16233;
65          case ChannelIndex.IMP_3P:
66              return 16237;
67          case ChannelIndex.IMP_4P:
68              return 16241;
69          case ChannelIndex.IMP_5P:
70              return 16245;
71          case ChannelIndex.IMP_6P:
72              return 16249;
73          case ChannelIndex.IMP_7P:
74              return 16253;
75          case ChannelIndex.IMP_8P:
```

```
76              return 16257;
77          case ChannelIndex.IMP_1N:
78              return 16231;
79          case ChannelIndex.IMP_2N:
80              return 16235;
81          case ChannelIndex.IMP_3N:
82              return 16239;
83          case ChannelIndex.IMP_4N:
84              return 16243;
85          case ChannelIndex.IMP_5N:
86              return 16247;
87          case ChannelIndex.IMP_6N:
88              return 16251;
89          case ChannelIndex.IMP_7N:
90              return 16255;
91          case ChannelIndex.IMP_8N:
92              return 16259;
93          case ChannelIndex.AEMG:
94              return PositionAEMGByte;
95          case ChannelIndex.Chirp:
96              return PositionChirpByte;
97          default:
98              return 0;
99      }
100 }
```

From the function the index value is compared to the value defined inside the **ChannelIndex** class that belong to a specific information. All the available channel indexes are the following.

```
1  static public class ChannelIndex
2  {
3      public const int Header = 0;
4      public const int StimCurrent = 1;
5      public const int Gain = 2;
6      public const int EventLed = 3;
7      public const int StimLev = 4;
8      public const int SDCardLev = 5;
9      public const int Mute = 6;
10     public const int Volume = 7;
11     public const int PowerLed = 8;
12     public const int Ext12V = 9;
13     public const int BatteryLed = 10;
14     public const int AudioTone = 11;
15     public const int NPI_1 = 12;
16     public const int NPI_2 = 13;
17     public const int NPI_3 = 14;
18     public const int NPI_4 = 15;
```

```
19      public const int NPI_5 = 16;
20      public const int NPI_6 = 17;
21      public const int NPI_7 = 18;
22      public const int NPI_8 = 19;
23      public const int Time = 20;
24      public const int EMG_1 = 21;
25      public const int EMG_2 = 22;
26      public const int EMG_3 = 23;
27      public const int EMG_4 = 24;
28      public const int EMG_5 = 25;
29      public const int EMG_6 = 26;
30      public const int EMG_7 = 27;
31      public const int EMG_8 = 28;
32      public const int IMP_1P = 29;
33      public const int IMP_2P = 30;
34      public const int IMP_3P = 31;
35      public const int IMP_4P = 32;
36      public const int IMP_5P = 33;
37      public const int IMP_6P = 34;
38      public const int IMP_7P = 35;
39      public const int IMP_8P = 36;
40      public const int IMP_1N = 37;
41      public const int IMP_2N = 38;
42      public const int IMP_3N = 39;
43      public const int IMP_4N = 40;
44      public const int IMP_5N = 41;
45      public const int IMP_6N = 42;
46      public const int IMP_7N = 43;
47      public const int IMP_8N = 44;
48      public const int AEMG = 45;
49      public const int Chirp = 46;
50  }
```

Using a value between 0 and 46 allows to load the data specified by the ChannelIndex: 21 is used to load data about the first EMG channel, 12 is used for the NPI calculated for the first channel, 2 is used to load the gain values during the entire operation and so on. To be totally clear, the NerveanaContentFilter does not contain all the data related to a surgery operation but just the one specified by the index. Multiple content filter object are defined and each of them is related to a specific information, creating a new NerveanaContentFilter means crossing all the saved filenames, taking the value of each packet and store them inside an array.
The software uses a NervenaContentFilter for the EMG channels, one for the impedance values, one for the NPIs and so on. The arrays are created when the user choose to open an old file, in this case it select the exam with the apposite window as shown in chapter 4.3 and the list of *".nrv"* files is created.
The class contains also another important function called **LoadData** related to the content filter and that given start and end time it return an array of type double

containing all the values of the content filter information contained inside the set of files.

LoaderNerveanaPlugin

This class is the other half part of this sub-project and is used in effect to load the content filter and encapsulate it into another class which is more manageable: **Track** class. This class is strictly related to the graphical user interface and will be discussed in section 4.3.2 but using it as a black box the following code could be discussed.

```
1  public TrackCollection LoadTracks(string[] filenames)
2  {
3      List<string> filelist = new List<string>();
4      foreach(string s in filenames)
5      {
6          filelist.Add(s);
7      }
8      TrackCollection tracks = new TrackCollection();
9      for (int ii = 0; ii < 47; ii++)
10     {
11         NerveanaContentFilter ner = new NerveanaContentFilter(
       filelist, ii);
12         {
13             Track tr = new Track();
14             tr.ContentFilter = ner;
15             tr.Duration = ner.SignalDuration;
16             tr.Title = ner.ChannelName;
17             tr.UnitOfMeasurement = ner.Units;
18             tr.RangeMax = ner.RangeMax;
19             tr.RangeMin = ner.RangeMin;
20             tr.Start = 0;
21             tr.isPlayback = true;
22             tracks.Add(tr);
23         }
24     }
25     return tracks;
26 }
```

This above function is used to load the data stored into a list of tracks given a set of files. The data are loaded defining a set of NerveanaContentFilter with different index value, in detail using value between 0 and 46 as specified in the ChannelIndex class. The content filter is then assigned to the track content filter and the object is added to the list and returned to the graphical interface, who did the function call. The following figure explain how the parts discussed are related.

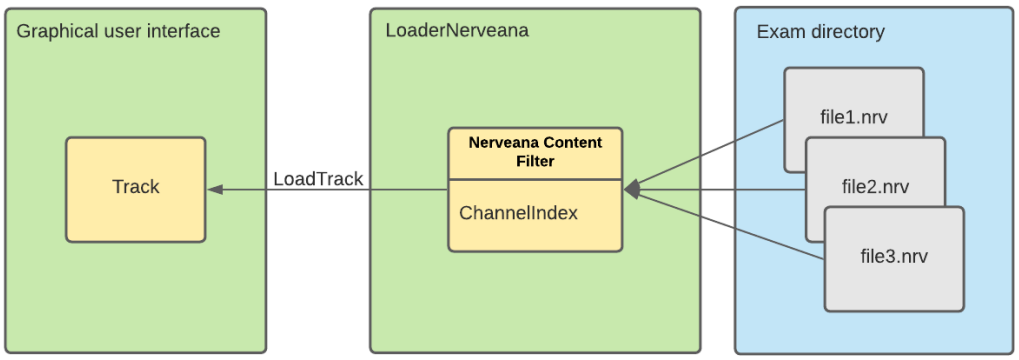


Figure 4.6: Loader Nerveana Relationships

4.3 NerveanaGUI: Graphic User Interface

This third section contains all the visualization part, from forms to draw library, that are used in the graphical user interface. In general this third sub-project define many forms or window used to allow a set of action to the operator, in particular the action related to the use cases defined in section 3.1.4. The forms has been designed using the toolbox included in Visual Studio 2017 and other custom graphic objects created with drawing primitives in order to be as similar as possible to the front panel of the main hardware module. Not all the windows will be discussed but only the most important. The following list show almost all the forms contained in the project

- **frmMain**: is used as main window for both the real-time and the offline visualization. It include controls to let the user interact with the hardware parameters and display the EMG channels. Implement the most part of this sub-project, from alert algorithm to information display and is the core of the GUI part.

- **frmStartup**: is the first screen that appear on launch, it allows the user to access a specific part of the software, starting a new exam or manage preference.

- **frmSelectDoctor**: is the real management window which the user can use to create new doctor, new preference or make changes.

- **frmSelectExam**: is the window that show all the offline exam stored of the local pc and allow to choose one of them for the offline visualization.

- **frmConfiguration**: is the classic option tab that allow to make changes related to the software, like theme or track colors, or access specific feature, like change password or change backup path.

- **frmConfirmCode**: is the window used to verify user identity asking for the password.

- **frmChangePassword**: allows the user to set a new password.

- **frmExamDetail**: this form appears at the end of a surgery exam and allows to change personal data or to export a PDF report.

- **frmInsertUpdateDoctor**: is a simple window used to set or change doctor personal data.

- **frmInsertUpdatePreset**: allows to select or change preference parameters during the creation of a new set of hardware settings.

- **frmMedicalDislaimer**: shows and ask to accept a message of responsibility for the use of the EMGView8

4.3.1 Analysis of Startup form

4.3.2 Analysis of Main form

The first window I want to analyze is the main form, in charge to draw signal during real-time communication or offline examination, allow different action to the user related to hardware command or to data visualization and display information received from hardware or read from files.

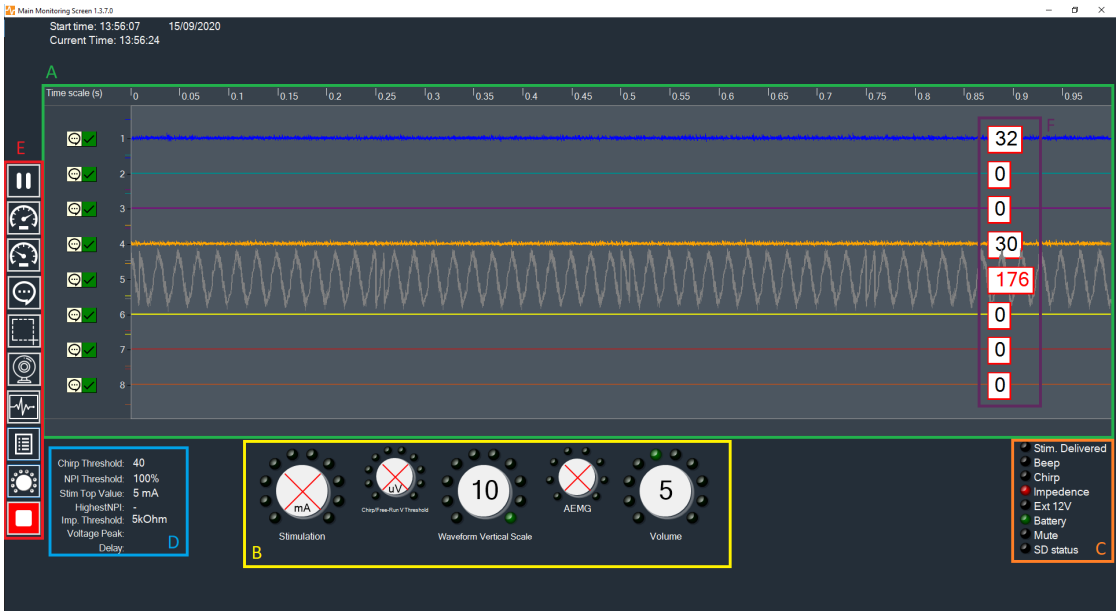


Figure 4.7: Main form

The main form appear in real-time as in figure 4.7 and the main sections shown are:

A **Soundtrack**: this is the custom control defined to draw the EMG channels and the response to the stimulation impulse. All about this class is explained in section 4.3.2

B **Control Panel**: this panel contains a set of custom controls called virtual knobs that allow the user to change parameters through the software, refer to chapter 4.3.2 for all the information about.

C **Led Panel**: this panel contains custom leds that have the same blinking behaviour of the hardware leds.

D **Information Box**: box where some informations are displayed during the real-time visualization

E **Action Panel**: this last panel contains a set of button which are related to particular features or graphic adjustment.

F **NPI Panel**: this panel contains the NPI value of each channel, drawn in a rectangle and updated at each new packet.

Control and Led Panel

As said in previous chapters is important to let the user feels like he/she is using the hardware and this is done mostly for one reason: **usability**. Also if the target user is a technician operator, the software wants to be simple to use and understand. The double use mode (surgeon driven or IONM mode) requires that the operator learn how to use hardware module and how knobs can be turned or pushed in order to activate specific functionalities. Activating those functionalities or reproducing the same behaviour using virtual knob does not require the operator to be trained again from scratch. Without forgetting that the software must run on a touchscreen. For this purpose two custom controls have been defined:

- **Knob**
- **Led**

Virtual knob must represent the knob as they are in hardware main module and are drawn like in figure 4.8; a central circle show the value of the knob index, while a crown of led surround it and is illuminated in green at the same index position. Leds that surround the knob are interactive. The user can click on it to set specified index position in hardware. This is done using the create command method defined in section 4.1.5 and passing the selected values.

Figure 4.8: Virtual Knob

In control panel five knobs are defined: stimulation knob, chirp knob, waveform vertical scale, AEMG knob and volume knob, each of them related to a specific main parameters. In respect to the hardware the virtual knobs cannot be pushed, that's why the three physical knob (with turn and push use) have been substituted with five virtual knobs. To better distinct the unit of measurements the stimulation and chirp virtual knobs show the current mA and uV values. Remembering the touchscreen use a feature has been implemented, allowing to change the knob using a press and drag action. When a push is detected the virtual knob changes as in

figure 4.9 and moving the finger right the value is increased, on the left instead the value decreases.

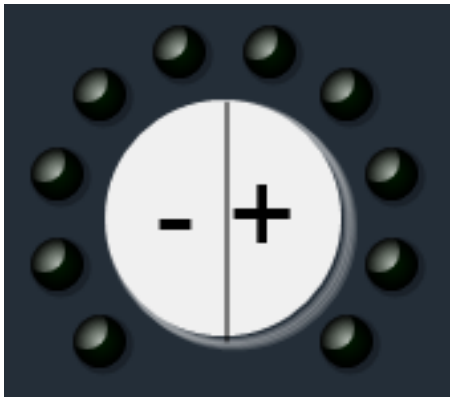


Figure 4.9: Pushed virtual knob

Led custom control is also used singularly and not associated to a knob in the led panel on the right-bottom part on the main form. Led in hardware have blinking behaviour depending on the current situation: battery charging, stimulation delivered, SD present or not, chirp threshold reached and so on and EMGView8 want to respect that behaviours in GUI. This is done mostly checking value of the event flag contained inside the packet at position 16208 as defined in section 4.2.1. Depending of that information the stimulation led, beep led and chirp led are updated. SD led, mute led, external supply charger led and battery led are instead are updated using other position defined in the same section.

Information Box

The information box purpose is to show to user information related to the settings he/she has chosen. Before a surgery exam start the operator choose the preference that better fit his/her needs through the frmSelectDoctor. Those settings are in part shown on screen.

Action Panel

On the left side on the main form there is the action panel, that contains different button for activate specific function. In order:

1. **Pause** button: used to freeze the real-time visualization. Communication keep running but the update of the draw of the EMG signals is stopped until the pause button is clicked again.

2. **Zoom In** button: used to enlarge the signal visualization. By default 25 ms of signal are shown bu the scale can by changed on the fly.

3. **Zoom out** button: used to reduce the signal visualization and increase the amount of time visualized on screen, Together with the zoom in button is used to set the X-axis time scale.

4. **Comment** button: during real-time could happens that the operator needs to keep track of a specific time moment, maybe just after some change or to simply track the surgery procedure. To allow that this button is inserted and when clicked a new window opens asking for a comment. The inserted text is stored in a list and used to produce the PDF report at the end of the exam.

5. **LarynxView** button: this button was inserted after the request to open a software used to communicate with a laryngeal webcam. A click simply start a process of the defined software, provided by Neurovision.

6. **Test** button: as seen in chapter 3, the hardware can start the impedance measurement with the activation of a test function. Via hardware this can be done holding pushed the stimulation button while via software can be done clicking this button, that using the SendTestCommand method activate the test function.

7. **Show NPI** button: hide/show NPI values

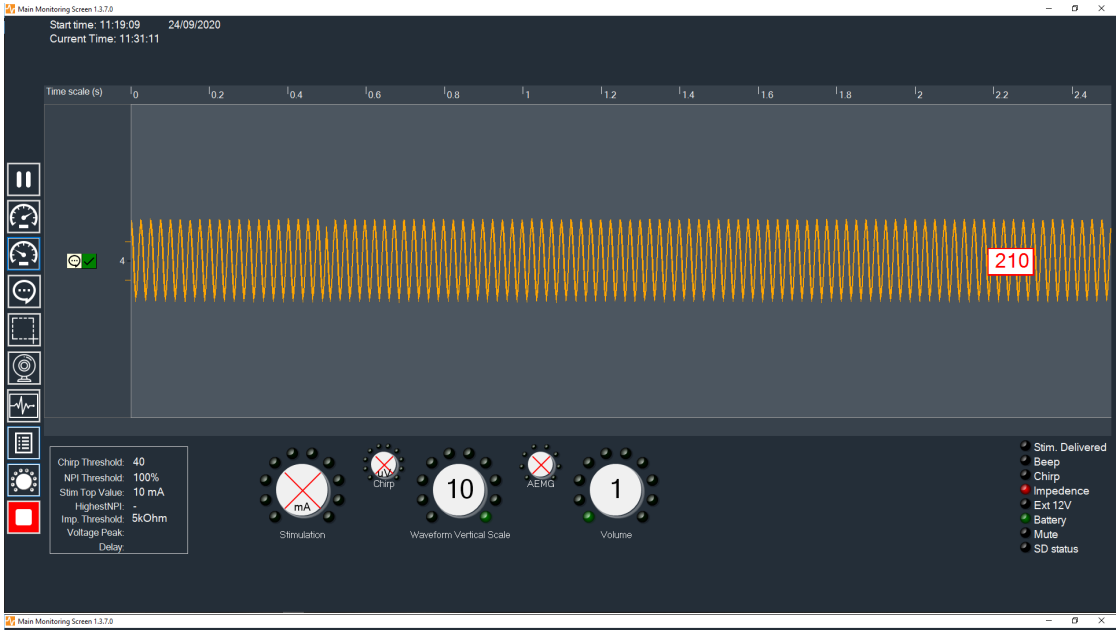
8. **Hide control panel** button: hide/show control panel

9. **Stop** button: stop the communication and open the frmExamDetail to save the exam and export the PDF report.

How the zoom works

From the action panel two buttons are related to zoom functionalities, changing the horizontal time scale during real-time communication. As seen in section 4.1.8 a queue of double array manage the incoming packets and as we have seen at the beginning it was handled like a FIFO stack and successively modified to overwrite the older packet. How the two things are related? Well, the queue has a bound limit called *feedBufferLimit* that identify the maximum number of packets that are stored inside the list. This variable is changed from the zoom in and out buttons and increases when the user wants to look at more packets at a time or decreases if he/she prefer to have a closer look to the incoming packet. Knowing that each second four packets come from the hardware when change the *feeBufferLimit* accordingly. Zoom level can change from 0.25 s (one packet in buffer) up to 10 s (40 packet in the queue). The following images shows a single active channel

feed with a sinusoidal signal thanks to a signals generator, with two different zoom levels. The first plot 2.5 seconds that are the equivalent of 10 packets while the seconds show 0.05 seconds that is a fifth of a packet.



Soundtrack

Like the OTComm library also Soundtrack library is proprietary of OTBioelettronica, used by the company for its software with the specific purpose of drawing the EMG signals. It provides a set of classes and functions that handle the signals

drawing, from colors to vertical scale and is born with the intent to reproduce an aspect similar to a multi-track audio editor like Audacity® or other sound editor. The Soundtrack library define a custom control called Soundtrack that is used as a container for the each signal draw which I will refer to with the name **Track**. Track is a custom class used to encapsulate the content filter together with other useful visualization variables, necessary for the signal draw. Not only, the library defines also a class called **GroupedTrack** used to draw together a set of tracks.
Soundtrack did not implement all the requested features for EMGView8 software but was an excellent starting point. In EMGView8 the track drawn in screen are encapsulated in a grouped track to let them feel like they're part of the same thing because a set on single Track class are draw in a separate way and introduce a different managing. In our case the tracks are similar, they belong to the same recording module and I want to manage them as if they are the same object. I don't need to change the vertical scale or any other graphical parameters separately for each track otherwise will be complex to distinguish the voltage scale and a surgeon can confuse about the receiving data. Compacting them together create more cohesion.
I want to mention that they're still different Track class we can modify separately and with each distinct parameters but the EMGView8 doesn't need to loop for each of them but just use the GroupedTrack when it make changes.

Track class

The Track class is our basic element for the visualization of a EMG channel. It encapsulates the NerveanaContentFilter and define a set of functions for the draw on screen of the data. There are two important variables that have been added at the class for EMGView8 use: **isGhost** and **dataGhost**.
The isGhost variable is used to determine if the track is the usual EMG channel or the response to a stimulation. All the channels are drawn during the real-time visualization but when the operator is stimulating the soundtrack need to draw also the response. Checking the event flag the software can determine if the stimulation occurs and, in that case, fills the dataGhost array, that represent the stimuli response for the specific channel. When dataGhost is not empty and the stimuli occur the draw function are enabled to draw also the dataGhost in red color to light up that those data are from a response to a stimuli. In detail the data about the response are unpacked differently from the original because EMGView8 needs to draw the data with a fixed zoom scale. The result is shown in the following image:

Figure 4.10: Ghost Track

In a second the hardware sends four packets but when the user has to look at the ghost tracks he/she needs to have a close view to the form and the delay time of the highest peak. Two modification have been done to allow this: first, we plot only a packet at time. Each ghost track is in fact just 0,25 seconds, so the value received are stretched to the entire screen and the response is evident. And second an additional temporal bar are drawn to the bottom of the soundtrack with a different time scale. On the right side a white rectangle is drawn for each track displaying three important numbers:

- NPI value: calculated from the hardware module and contained in a packet.
- Delay: the delta time from the start of the stimuli and the time of the peak, calculated in software.
- Voltage peak: the value in microvolt of the response peak.

Managing the active channels

At the very beginning of this project all the EMG channels were plotted on the main screen but at some point the requirement changed, asking for plotting only the active channels. The hardware send a packet containing data about all eight channels so, the managing of this feature it had to be done on software. Fortunately it sends instead a byte that specify the active channels.
To do that an index variable was defined to help the algorithm in shifting inside

the packet to the correct byte position. As seen in section 4.1.7 the *GetDataMatrix* function made use of a variable called *lastK* that, multiplied by 2000, represent the byte position for the active channel to unpack. The manage of *lastK* variable is done in **FindNextIndex** method and works as follow.

```
1  private void FindNextIndex()
2  {
3      int k;
4      flagFound = false;
5      while (flagFound == false)
6      {
7          for (k = lastK + 1; k < 8; k++)
8          {
9              if (ActiveChannelsArray[k] == 1)
10             {
11                 lastK = k;
12                 flagFound = true;
13                 return;
14             }
15         }
16         lastK = -1;
17     }
18 }
```

Let's suppose only three sensors are connected: 1, 3 and 5. The hardware sends all eight channels but the software must discriminate the unused position and take only values about channel 1, channel 3 and channel 5. In *GetDataMatrix* the byte position shifts based on *lastK* variable, that is set inside the *FindNextIndex* function, called before the unpacking. *FindNextIndex* use a support array of eight integer to know if the channel is active or not. The integer array is called *ActiveChannelsArray* and can contain two value:

- 0 : if the channel is not active
- 1 : if the channel is active

In our example situation the array values are:

10101000

Starting from the last active channel, the function loop inside the array looking for the next "1" and update the variable's value. The same approach has been applied for the NPIs and Impedance values of the active channels bu to not overlap another function were created: *FindNextNPIIndex*.

High Stimulation warning message

During a real-time examination the operator could require to increase the stimulation level over safe value. This can be done only if the preference selected for the exam allow to set stimulation over 5 mA, that is acceptable from a clinical point of view but needs to be explicitly confirmed from the user. For this purpose comes up the needs to create a custom *Messagebox* with non-blocking functionality that ask the operator to confirm his/her choice. The problem of the usual messagebox is that it appears on top on any other windows and freeze their functionalities until a choice, usually Yes/No, is selected. During a real-time operation this is a problem because it could block the main thread in charge to unpack the data and let the buffer fill to the limit. The consequence is that after a choice, when the message disappears, the main form is stucked and fails to resume the communication. To avoid this problem the following code has been written.

```
1  if (counter_high_stim >= 10 && !highstimFlag)
2  {
3      counter_high_stim = 0;
4      if (ledKnobStimulation.milliAmpereValue > 5 && !highstimFlag)
5      {
6          frmHighStimAlert frm = new frmHighStimAlert(this);
7          frm.TopMost = true;
8          Invoke(new MethodInvoker(delegate () { HideAll(); }));
9          frm.ShowDialog();
10
11         if (frm._result == true)
12         {
13             counter_high_stim = 0;
14             highstimFlag = true;
15             ledKnobStimulation.LedColor = Color.Red;
16             ledKnobStimulation.ValueColor = Color.Red;
17         }else{
18             counter_high_stim = 0;
19             Invoke(new MethodInvoker(delegate () { UpdateStimTo5mA();
       }));
20             highstimFlag = false;
21         }
22         Invoke(new MethodInvoker(delegate () { ShowAll();
23             checkBoxPelvic.Visible = true;
24             checkboxPedicleScrew.Visible = true;}));
25     }
26 }else{
27     counter_high_stim++;
28 }
29 if (highstimFlag && ledKnobStimulation.milliAmpereValue < 5)
30 {
```

```
31          highstimFlag = false;
32          ledKnobStimulation.LedColor = Color.Green;
33          ledKnobStimulation.ValueColor = Color.White;
34          checkboxPedicleScrew.BackColor = SystemColors.Control;
35          checkBoxPelvic.BackColor = SystemColors.Control;
36      }
```

frmHighStimAlert is a custom form defined with two simple buttons to let the user accept the consequences and no other way to escape it. The method is simple: it asks for confirmation and if this has not been given, it reverts and set back the stimulation to 5 mA. The new window is showed on top of main form to which all controls contained are hidden using **HideAll** function, invoked on another thread. To emphasize the risk, if the user choose to increase the stimulation the virtual knob color change to red.
The *highstimFlag* variable is used to set the high stimulation condition because the user can always change the stimulation manually back under 5 mA but the EMGView8 software must ask for confirmation each time the stimulation is over the threshold. The *counter_high_stim* variable helps in detecting the condition.

Figure 4.11: High stim alert window

Main form in examination mode

During offline examination the main form appears a bit different, part of the action panel is cleared from unusable button while other appears. Another new custom control is used to display information and in general the new sections are:

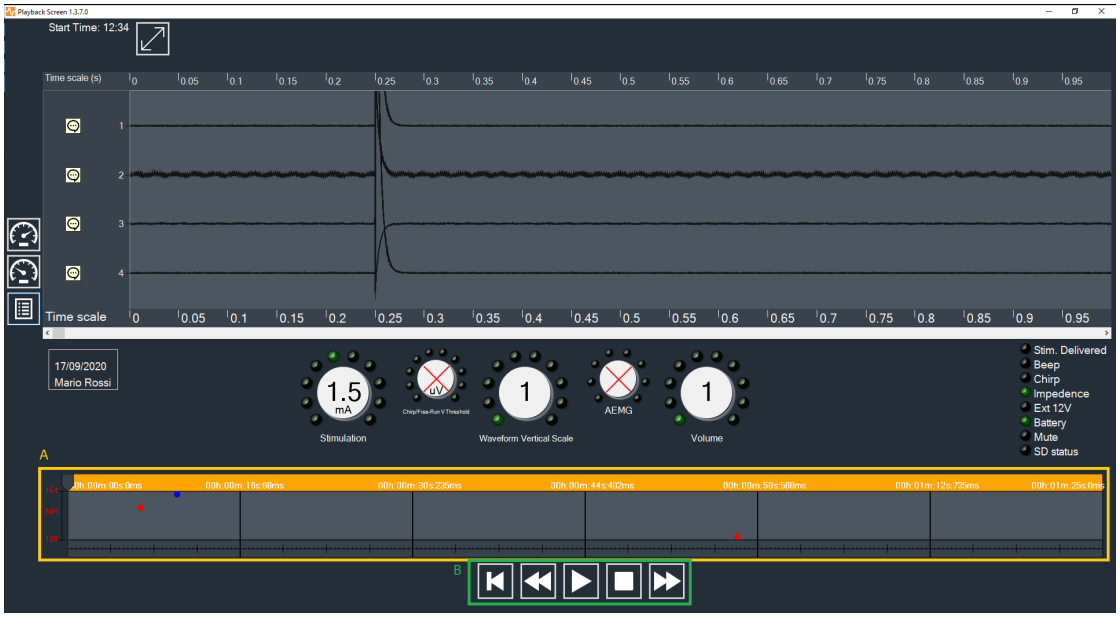


Figure 4.12: Offline review main form

A **Temporal scrollbar**: a new custom control, drawn as a ruler to illuminate the time section. On the left size the NPI scale is shown, starting from 100 which is the lower value to activate the NPI event up to the maximum NPI event recorded for the exam. A set of red and blu dots are drawn in the middle part and refer to specific even or odd channel. The 4.13 will explain this feature. Moving cursor over the dots open a tooltip and display the NPI value. There's also a small slider that interact with the soundtrack and allow to move along the signal duration.

B **Media Player Box**: a box where some button are displayed in order to slide the exam like a sound track. Action permitted are: Play, Stop, Fast forward, Slow forward and Go at Begin.

Led bulbs in right Led Panel as well as the knobs in Control Panel change value according to the control value at that specific time moment. All information are stored in file at specific position of each packet as described in section 4.6 and this is used to load a set of track from the file in which each track belong to a specific information. Once the software have these tracks, the EMG channel tracks are loaded in soundtrack for examination while the others are hidden and used just to update the control position. For the led bulbs the data are not stored in dedicated tracks but loaded in different double array to be easily retrieved. To do so, two custom event handler are defined: **ledBulbUpdated** and **knobUpdated**, together with two custom event arguments:

- **knobChangedEventArgs**: used to relate the virtual knobs with the soundtrack. While the soundtrack horizontal bar is moving the event arguments are filled with the appropriate information and the knobUpdated event is raised. The main form capture the event, unpack it and update the virtual knobs values.

- **ledBulbChangedEventArgs**: used to relate the led bulbs with the soundtrack. While moving the horizontal bar, as in previous case, the event arguments are filled with appropriate attributes and the ledBulbUpdated event is raised. The main form capture the event, unpack it and update the led bulbs values.

The two custom event arguments are defined as follows, with a value for each bulb or knob:

```
public class knobChangedEventArgs : EventArgs
{
    public int gain { get; set; }
    public int stimulation { get; set; }
    public int chirp { get; set; }
    public int aemg { get; set; }
    public int volume { get; set; }
}
public class ledBulbChangedEventArgs : EventArgs
{
    public int ledStim { get; set; }
    public int ledBeep { get; set; }
    public int ledChirp { get; set; }
    public int ledImpedance { get; set; }
    public int ledExt12V { get; set; }
    public int ledBattery { get; set; }
    public int ledMute { get; set; }
    public int ledSD { get; set; }
}
```

With the following functions the arguments are filled and then the event is raised. In *UpdateKnob* software takes the values inside the content filter of the knob tracks, at the specified time, and it fill the event's arguments. Once the event is captured events argument can be unpacked ad used as value for the knob.

```
public event EventHandler knobUpdated;
private void UpdateKnob()
{
    knobChangedEventArgs args = new knobChangedEventArgs();
```

```
5      args.gain = (int)knobTrack[1].ContentFilter[Convert.ToInt32(
       StartTime)];
6      args.stimulation = (int)knobTrack[0].ContentFilter[Convert.
       ToInt32(StartTime)];
7      args.chirp = (int)knobTrack[4].ContentFilter[Convert.ToInt32(
       StartTime)];
8      args.aemg = (int)knobTrack[3].ContentFilter[Convert.ToInt32(
       StartTime)];
9      args.volume = (int)knobTrack[2].ContentFilter[Convert.ToInt32(
       StartTime)];
10     this.knobUpdated(this, args);
11 }
```

In *UpdateLedBulb* software takes the values contained in its dedicated array and use it to fill the event arguments. The arrays are contained in the Soundtrack and filled during the file opening, in particular the tracks are loaded and all the data inside the content filter are put into the arrays for an easy access. To access a led bulb value at the specified moment the time is used as an index but first multiplied by four because of the sampling rate. Some more checks are done to insert information about ledChirp, ledBeep and LedStim because they're related to the value in the eventFlag. On the other side there's the function that unpack these arguments. This function is a bit more complex because it need to associate the value retrieved to the color of the led that could have four different value: green, orange, red and black together with its blinking behaviour.

```
1  public event EventHandler ledBulbUpdated;
2  public double[] eventFlagArray;
3  public double[] ledBulbSD;
4  public double[] ledBulbMute;
5  public double[] ledBulbExt12V;
6  public double[] ledBulbBattery;
7  private void UpdateLedBulb()
8  {
9      ledBulbChangedEventArgs args = new ledBulbChangedEventArgs();
10     int eventFlag = args.ledMute = (int)eventFlagArray[Convert.
       ToInt32(StartTime) * 4];
11     switch (eventFlag)
12     {
13         case 0:
14             args.ledChirp = 0;
15             args.ledBeep = 0;
16             args.ledStim = 0;
17             break;
18         case 1:
19             args.ledChirp = 0;
20             args.ledBeep = 0;
```

```
21              args.ledStim = 0;
22              break;
23          case 2:
24              args.ledChirp = 0;
25              args.ledBeep = 0;
26              if(flashingLed == 0)
27              {
28                  flashingLed = 1;
29                  args.ledStim = 1;
30              }else{
31                  flashingLed = 0;
32                  args.ledStim = 0;
33              }
34              break;
35          case 4:
36              args.ledChirp = 0;
37              if(flashingLed == 0)
38              {
39                  flashingLed = 1;
40                  args.ledBeep = 1;
41              }else{
42                  flashingLed = 0;
43                  args.ledBeep = 0;
44              }
45              args.ledStim = 0;
46              break;
47          case 8:
48              if (flashingLed == 0)
49              {
50                  flashingLed = 1;
51                  args.ledChirp = 1;
52              }else{
53                  flashingLed = 0;
54                  args.ledChirp = 0;
55              }
56              args.ledBeep = 0;
57              args.ledStim = 0;
58              break;
59      }
60      args.ledSD = (int)ledBulbSD[Convert.ToInt32(StartTime*4)];
61      args.ledMute = (int)ledBulbMute[Convert.ToInt32(StartTime*4)];
62      args.ledExt12V = (int)ledBulbExt12V[Convert.ToInt32(StartTime*4)
        ];
63      args.ledBattery = (int)ledBulbBattery[Convert.ToInt32(StartTime
        *4)];
64      this.ledBulbUpdated(this, args);
65  }
```

4.3.3 Analysis of ExamDetail form

The ExamDetail form is what appears when the surgery operation is completed and the user ask to save the data. In this window operator can insert the patient personal data (first name and last name) or insert the title he/she prefer for the examination, he/she can delete or modify the comments that has been inserted during the exam, look at the screenshot taken or produce the PDF report. On the top part of the window a graph is shown, displaying all events that has been captured in real-time and that will be explained in next few lines.

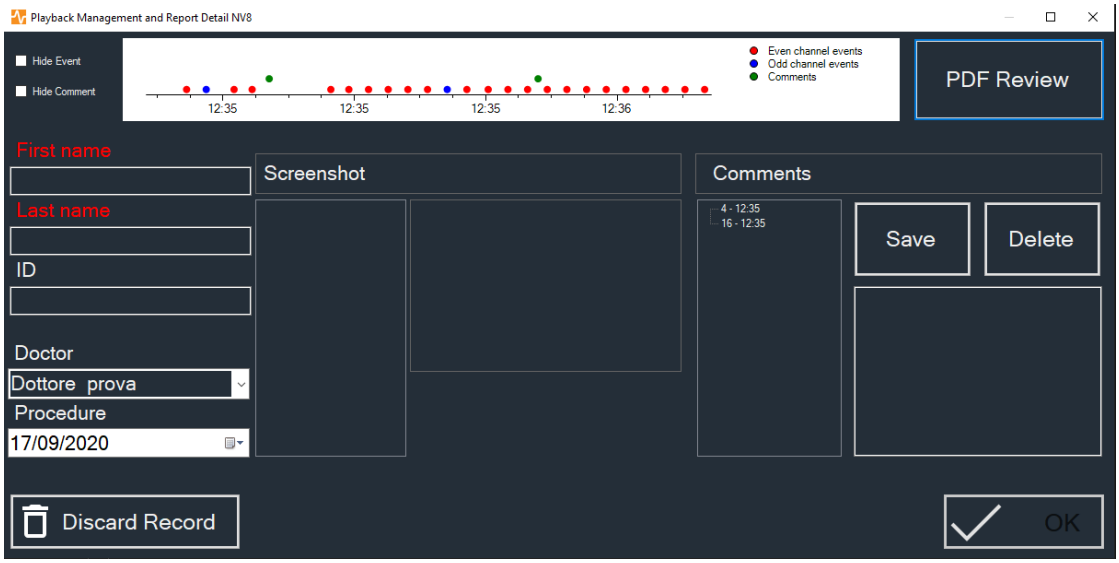


Figure 4.13: ExamDetail form

Events

I take advantage of this window to explains a feature that has been asked from customers during the design of this project: the need of detecting particular events. This need is not related to a real-time behaviour from the software but in order to have a wide view of the exam when the PDF Report in produced. The are three type of event:

- a comment: at any moment, as show in Action Panel, the operator can click on a button and insert a text. The comment is saved in a list and retrieved by frmExamDetail at the end of the exam. Let's suppose the operator wants to take notes of a particular moment: "Preparing the stimulator", "Applying the sensor" and so on.

- channel NPI over 100:

– odd channel

– even channel

In both case the check is the same, the software must detect when an NPI value of a channel is over 100, that is the value calculate from Neurovision after hundreds of test to know that the response to the stimuli is significant.

A specific class was define to encapsulate all useful information and all these three event are stored and plotted in a time graph to illuminate the moment when something happened. A surgery operation can last a long time and can be confusing to look for the right spot.

4.3.4 Analysis of SelectDoctor form

SelectDoctor window is used to choose a preference to start the exam with. On the left side a listview displays all doctors that are registered into database and to each of them on the right side another listview show the preferences. Many buttons allow the user to insert a new doctor, modify or delete an existing one or to import and export a doctor structure. In this last case software read or write an XML file filled with doctor information and preferences. Other buttons are instead related to the preference setup, the set of hardware configuration, and allow to create a new one, modify or delete an existing one or to clone a preference of a doctor into another doctor. The card drawing of the two listview is custom, while the doctor card is very simple, the preference card want to give an overview of the most important parameters set inside it. To do so three circles are drawn to represent the three main important knobs while other important background parameters are written.

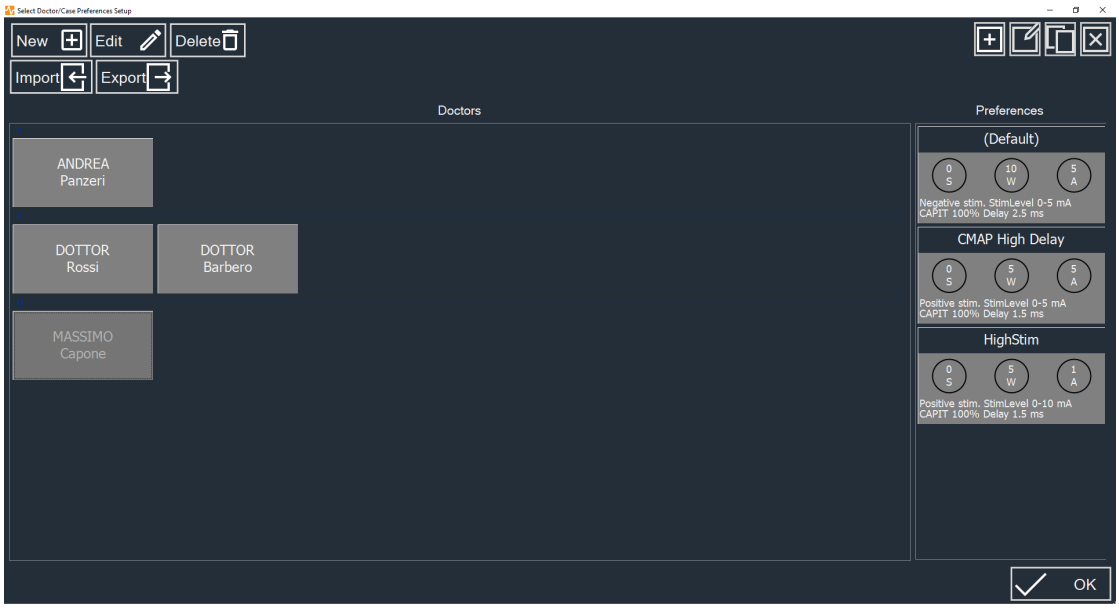


Figure 4.14: SelectDoctor form

4.3.5 Privacy

From the very beginning of this project customer asked to worry about privacy related to all data belonging to the patients. In EMGView8 most of the data belongs to the exam itself and the EMG channels, that can be opened effectively only using the LoaderNerveana, that knows the protocol and how data are stored and is able to load the EMG signal and plot the data. Anyone can look inside the computer hidden directories and find an exam but also with the file is almost impossible to interpret it without EMGView8. Anyway to leave no track at all and don't allow to associate the directory to the patient all the files stored and all the directories created are created with random name string. In addition it was taken into consideration to let the user access the data and the preference configuration through the software only with the use of a simple password. The password chosen is a four digit password, saved inside the NerveanaRegistry and encrypted with SHA256. This means that if someone access patient data it's intentional and punishable by law. All other considerations that include the database are described in section 4.4. The form used to ask the password is the following:

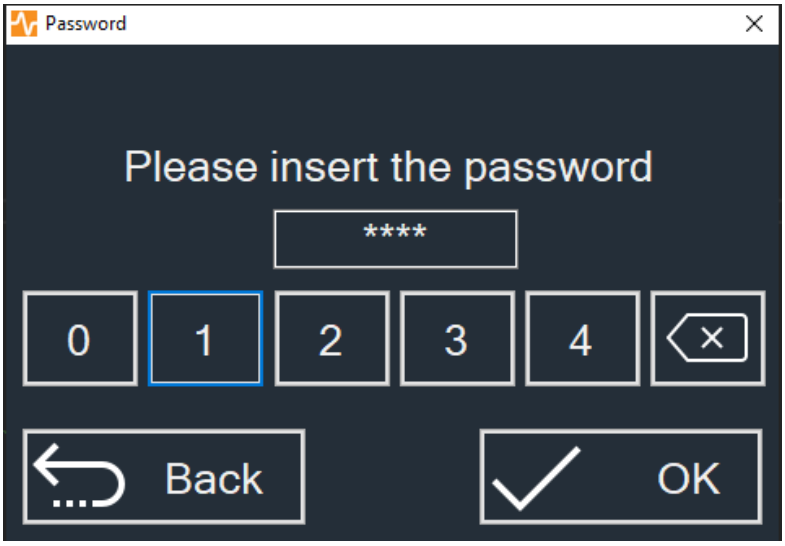


Figure 4.15: Password form

4.4 Database and NerveanaRegistry

This is the last subsection and during this section are gonna be explained the choices made for the database implementation. A database is essential for the use of the EMGView8 software, as we have seen many information needs to be stored on the local machine and in section 3.3.3 has been defined three important element to take care of: Surgeon, Exam and Preference. Why the patient was not considered? Essentially for privacy reason. During the saving of the exam data of patient are not mandatory but they can be optionally added together with a comment. Those three cited above components have led the design of the database tables and all the attributes definitions. To access the database the library *System.Data.SQLite* was used to create and execute SQL commands. To read the data has been made use of *SQLiteDataReader* class together with *DataTable* structure.

Let's exploit each main element.

Surgeon table identifies the man in charge of using the hardware during the surgery operation. The software do not care about the correctness of surgeon's data but use it as an identifier. When the operator select for an **Exam** or a **Preset** he/she want to understand who it belongs to. It's clear that the Surgeon is our key element for the database design and needs to be linked with other two tables. How to identify the Doctor? Neurovision Medical Products did not set any constraint and showed itself to be inclined to privacy care. During the surgery operation is not always the surgeon itself to use the software but a technical operator so the table was defined as simple as possible, just considering first name and last name.

Exam is the second table considered and have to link together information about the patient, the surgeon and the stored files. For this purpose it was decided to add a foreign key to the doctor id and create the exam table with four others important attributes:

- First name and last name of the patient
- Comments about the exam itself
- Name of the folder that contains the recorded files

Information about the patient are optional and can be inserted before the data are saved. The name of the folder is probably the most important attributes of this table and is used to load the data during the offline visualization. The path to the main folder is stored inside the registry and is read from the EMGView8 software. Then the folder's name is combined to create real path to the desired exam. This is completely transparent to the user because the software itself generate a random string name for the directory and a random string for each file to store. Those name must be stored in database to know which files open to access an exam.

Presets table is our last table. For each surgeon a set of preference need to be stored and he/she can create or modify them as he/she prefer based on the hardware settings needed. Among the three tables this is the one with more fields because it stores all the variables that can be loaded inside the hardware main module during the operation startup, the main and the background parameters explained in section 3.10. The table need also to be linked to the doctor table because each preference belongs to a specific surgeon and to him/her only

In figure 4.16 are shown the three tables.

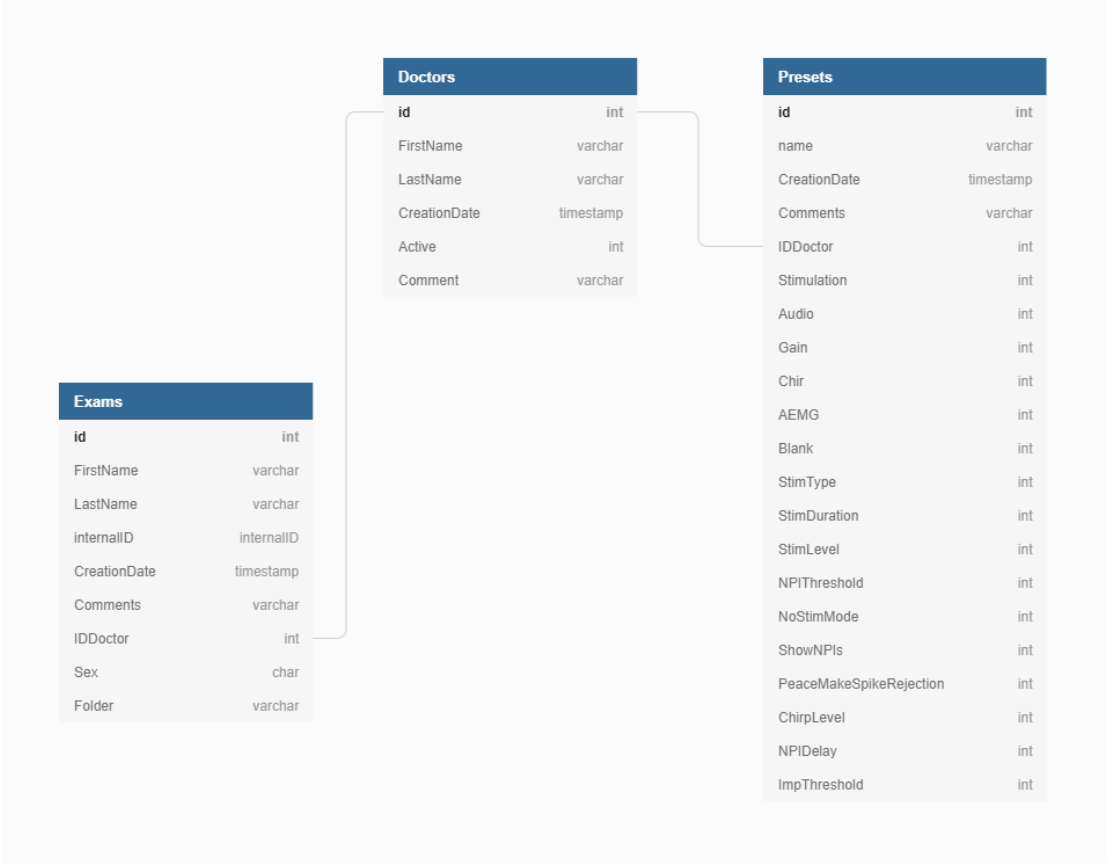


Figure 4.16: Database tables

On software side some important variables need to be stored and accessed frequently and for this purpose a dedicated class has been defined: **NerveanaRegistry**. This class define three custom classes:

- RegistryBooleanValue
- RegistriStringValue
- RegistryIntValue

These three classes contain a variable associated to a registry key. NerveanaRegistry allows to save on local machine registry a list of important variable and access them quickly. Variable saved are related to preference default values, folder paths, brush size and colors, font size, password, selected theme and so on.

Chapter 5

Conclusions

This project was the real test of fire to me, the first true approach to the real world and allows me to learn something new and very specific. When you study computer engineering you faces many topics, from programming to network and telecommunication, from algorithm to graphics or data manage and also encryption, communication protocol, web programming, server managing and so on without really knows where you will dedicate and learn the most. This was the first real project where you feel insecure in receiving a lot of new updates to code but also excited when you got answered and you feel the customers happy about how you have managed a situation.
Dedicating to this project made me put my hands into very different sections. Starting from the two libraries proprietary of OTBioelettronica, *Soundtrack* and *OTComm*, one related to the signals draw and the other to the hardware communication, transformed and included in the new project. Moreover, all the GUI parts and the creation of custom controls, out of the toolbox, or the dealing with the file interpretation. Without forgetting the database, the privacy section and the password management. I followed this project at 360° and under all points of view, I didn't even know how to program in C# when all of this started.
The software is not finished yet, all the system is in clinical test and probably so much new updates will come and I can guess some of them.

1. New hardware commands: the firmware, as well as the software, is in constant change and will be modified. If any new feature will be added on firmware it will probably need to be activated, this means that it require a new command to set it or, in alternative, it can be handled as a new background parameter to care about during startup. It will probably comes a new command to activate or deactivate a notch filter and to choose if filter 50 Hz or 60 Hz.

2. Offline ghost track visualization: some work still needs to be done for the offline examination, a feature that can be implemented could be the offline

visualization of the ghost track or in alternative, the NPI value while temporal scrollbar slides.

3. Software sound system: at the moment only the hardware play sound during the surgery operation but what if also the software emits alert?

4. Sound tones assignment: inside the hardware main module a set of sounds is defined and attached to particular event. A new features could be let the user choose how to relate sound to event.

Bibliography

[1] VandenBos and Gary R. «evoked potential (EP). APA dictionary of psychology (2nd ed.).» In: (2015), p. 390 (cit. on p. 4).

[2] L Dıaz Rodrıguez, M Varga, J K Wolter, and U Pliquett. «Impedance as guidance for electrode placement in intraoperative monitoring of nerve fibers». In: *Journal of Physics: Conference Series* 434 (Apr. 2013), p. 012025. DOI: 10.1088/1742-6596/434/1/012025. URL: https://doi.org/10.1088%2F1742-6596%2F434%2F1%2F012025 (cit. on p. 10).

[3] Saed Moradi, Esmaeel Maghsoudloo, and Reza Lotfi. «A new approach to design safe and reliable electrical stimulator». In: *International Journal of Biomedical Engineering and Technology* 15 (Sept. 2014), pp. 305–316. DOI: 10.1504/IJBET.2014.064820 (cit. on p. 21).

[4] Stefan Hansen Anna Karilainen and Jorg Muller. «Dry and Capacitive Electrodes for Long-Term ECG Monitoring.» In: (Nov. 2015) (cit. on p. 24).

Acknowledgements

Mi fa strano pensare di essere alla fine di questo percorso, come se un percorso dovesse per forza avere una fine. Probabilmente cambiamo strada senza nemmeno rendercene conto e ogni volta ci portiamo dietro qualcosa in più nel bagaglio. Se così è, allora sto per lasciare questo sentiero impervio per una strada, credo, meno dissestata. Ogni buona strada che si rispetti però ha salite e discese e spero che i passi percorsi fino a qui abbiano preparato le mie gambe a dovere.
Sicuramente non sarei qui senza alcune delle persone che hanno fatto qualche passo insieme a me. Prima di tutti ringrazio Paolo, per aver camminato le ultime miglia con me, per avermi condotto e spronato, probabilmente ho imparato più nel guardarti lavorare che in interi esami. Ringrazio i miei genitori, per i sacrifici che hanno fatto e per avermi permesso di intraprendere questa strada. Ringrazio Eugenio, senza il quale starei ancora cercando di passare Fisica II e tutti gli amici con cui ho condiviso ore in biblioteca e momenti felici per tirarsi su di morale dopo un esame non passato. Infine ringrazio Marta per essere stata la luce più forte, quando tutto intorno si faceva troppo buio.

Voglio finire queste pagine con un proverbio a cui ho ripensato più e più volte. Grazie al quale ho capito che va bene non farcela al primo colpo, che ci vuole tempo per imparare, che non importa se si sbaglia e che nessuno nasce pronto.

"Ciò che cresce lentamente mette radici profonde"