UNIVERSITY OF TWENTE

BIOMEDICAL SIGNALS & SYSTEMS (BSS) RESEARCH GROUP

Initial development of a Python-based Graphical User Interface for a Medical measurement system for Nociceptive (mal)Function.

Biomedical Engineering Bachelor Thesis

M.E.L. Janssen

Bachelor-assignment committee:
Prof. Dr. Ir. J.R. Buitenweg (Committee Chair)
Ir. F. Muijzer (Daily Supervisor)
Dr. J.C. Alers (External Member)

26-04-2024

Abstract

An average of around 20% of the adult population in western countries experiences chronic pain as a result of nociceptive thresholds not being reversed after disease, inflammation or injury. Earlier detection of chronic pain would be beneficial for treatment of this condition.

The Nociceptive and Somatosensory Processing (NSP) team, within the research-group Biomedical Signals and Systems (BSS) at University of Twente, has developed the NDT-EP method for observing the properties of nociceptive processing, by determining the nociceptive detection threshold (NDT) and evoked potentials (EP). The NDT-EP method has shown very promising results and therefore asks for larger clinical studies and applications. When the NDT-EP method would be implemented for larger clinical studies and applications, other people than researchers of BSS will have to operate the system. These non-expert users do not (necessarily) have knowledge about nociceptive processing and do not have a technical background. To properly operate the current Labview-based system, a technical background is needed. In addition, the current set-up is difficult to properly document following the Medical Device Regulations (MDR). It is therefore decided to switch from the current Labview-based system to a new Python-based Graphical User Interface (GUI). Hence, the design-assignment of this research reads as follows: **Initial development of a Python-based Graphical User Interface (GUI) for a Medical measurement system for Nociceptive (mal)Function.**

To realise an initial prototype, first a theoretical framework is designed. To create this, new implications are investigated. Switching from a Labview-based system to a Python-based system, allows for implementing object-oriented programming, by using classes, and concurrent programming, by using asyncio (library). A GUI is integrated in the Labview-based system, but not in Python. However, a Python-based GUI can be created by using Tkinter (module), which lets the developer build a GUI with several widgets and windows. The GUI designed in this research is created to allow non-expert users to conduct experiments on patients, using the NDT-EP method. A risk analysis is performed of the use of the software for the new system to demonstrate that software of a medical device entails new kinds of risks and to show how the MDR can be applied to such software.
The final design of this research provides a framework and a first prototype for a Python-based GUI, that allows non-expert users to conduct experiments on patients, using the NDT-EP method. As this research is conducted in the early stage of development, there are still several points that should be improved in further development, regarding the implementation of additional elements, risk analysis and testing and the issues that arise when combining a GUI with asynchronous concurrent programming.

Nederlandse vertaling
Ongeveer 20% van de volwassen bevolking in westerse landen heeft last van chronische pijn als gevolg van nociceptieve drempels die niet worden hersteld na ziekte, ontsteking of letsel. Vroegtijdige detectie van chronische pijn kan zorgen voor betere behandeling van deze aandoening.

Het Nociceptive and Somatosensory Processing (NSP) team, binnen de onderzoeksgroep Biomedical Signals and Systems (BSS) aan de Universiteit Twente, heeft de NDT-EP methode ontwikkeld voor het observeren van de eigenschappen van nociceptieve verwerking, door het bepalen van de nociceptieve detectiedrempel (NDT) en opgewekte potentialen (EP). De NDT-EP methode heeft veelbelovende resultaten laten zien en vraagt daarom om grotere klinische studies en toepassingen.
Wanneer de NDT-EP methode geïmplementeerd zou worden voor grotere klinische studies en toepassingen, zullen andere mensen dan onderzoekers van BSS het systeem moeten kunnen bedienen. Deze niet-deskundige gebruikers beschikken niet (altijd) over kennis van nociceptieve verwerking en hebben geen technische achtergrond. Om het huidige Labview-systeem goed te kunnen bedienen, is een technische achtergrond nodig. Daarnaast is het moeilijk om het huidige systeem goed te documenteren volgens de Medical Device Regulations (MDR). Er is daarom besloten om van het huidige Labview-systeem over te stappen op een nieuw systeem: een Grafical User Interface (GUI) in Python. De ontwerpopdracht van dit onderzoek luidt daarom als volgt: **Initiële ontwikkeling van een Graphical User Interface (GUI) in Python voor een Medisch meetsysteem voor Nociceptieve (mal)Functie.**

Om een eerste prototype te kunnen realiseren, wordt eerst een theoretisch framework ontworpen. Om dit te creëren worden de nieuwe implicaties onderzocht. Door over te stappen van een Labview-systeem naar een Python-systeem kan er object georiënteerd geprogrammeerd worden door classes te gebruiken en daarnaast kan er concurrent programming worden toegepast door asyncio (library) te gebruiken. In het huidige Labview-systeem zit een GUI geïntegreerd, maar in Python niet. Een GUI in Python kan echter worden gemaakt met behulp van Tkinter (module). De ontwikkelaar kan een GUI bouwen met verschillende widgets en vensters. De GUI die in dit onderzoek is ontworpen, is gemaakt om niet-deskundige gebruikers experimenten te laten uitvoeren op patiënten, volgens de NDT-EP methode. Er wordt een risicoanalyse uitgevoerd voor het gebruik van de software voor het nieuwe systeem om aan te tonen dat software van een medisch apparaat nieuwe soorten risico's met zich meebrengt en om te laten zien hoe de MDR op dergelijke software kan worden toegepast.
Het uiteindelijke ontwerp van dit onderzoek biedt een framework en een eerste prototype voor een GUI in Python, waarmee niet-deskundige gebruikers experimenten kunnen uitvoeren op patiënten, volgens de NDT-EP methode. Aangezien dit onderzoek in een vroeg stadium van de ontwikkeling is uitgevoerd, zijn er nog verschillende punten die verbeterd moeten worden in de verdere ontwikkeling, zijnde de implementatie van extra elementen, een volledige risicoanalyse en de problemen die zich voordoen bij het combineren van een GUI met asynchronous concurrent programming.

Abbreviations

Table 1: List of abbreviations.

IASP	International Association for the Study of Pain
NSP team	Nociceptive and Somatosensory Processing team
BSS research-group	Biomedical Signals and System research-group
NDT	Nociceptive Detection Threshold
EP	Evoked Potentials
MDR	Medical Device Regulations
GUI	Graphical User Interface
EEG	Electroencephalography
MTT	Multiple Threshold Tracking
EU	European Union
OOP	Object-oriented Programming
PoC	Probability of Occurrence
CwO	Consequence when Occurring

Contents

Chapter 1

Introduction

In western countries, an average of around 20% of the adult population experiences chronic pain [1, 2]. The prevalence of chronic pain increases with age [3]. We speak of chronic pain when someone experiences pain lasting more than 3-6 months [4].
Three types of pain can be distinguished: nociceptive, neuropathic and nociplastic [5]. This research focuses on nociceptive (mal)function. Nociceptive pain is described by the International Association for the Study of Pain (IASP) as: "Pain that arises from actual or threatened damage to non-neural tissue and is due to the activation of nociceptors" [6]. Nociceptors are sensory neurons that are activated by noxious stimuli when these stimuli surpass the (nociceptive) threshold, if so a sensation is felt which humans associate with the concept "pain" [7, 8, 9]. These sensory nerve fibers are located in the (epi)dermis [7]. When suffering from disease, inflammation or injury, nociceptive thresholds can change, which causes a higher sensitivity to these noxious stimuli [10]. Chronic pain can arise after a patient has recovered from disease, inflammation or injury, but the changes of these nociceptive thresholds are not being reversed and the central nervous system (CNS) remains disturbed, which results in a long-term enhanced sensitivity to noxious stimuli and thus chronic pain. [11, 12, 13].
Although 1 out of every 5 adults experiences chronic pain, once a patient is diagnosed with it, there are still limited (effective) treatment options available [5]. Various studies suggest that, earlier detection of (the factors causing) chronic pain would be beneficial to prevent further development of the pain, by decreasing factors that might eventually aggravate the pain [5, 14]. These factors include: demographic-factors (e.g. age and gender), lifestyle- and behaviour-factors (e.g. smoking, alcohol and nutrition) and clinical-factors (e.g. weight, sleeping disorders) [15].

The Nociceptive and Somatosensory Processing (NSP) team, within the research-group Biomedical Signals and Systems (BSS) at University of Twente, has developed a method for observing the properties of nociceptive processing. The method utilizes conscious detection of electrocutaneous stimulation of nociceptive nerve fibers (Nociceptive Detection Threshold, NDT) in combination with objective neurophysiological brain responses (Evoked Potentials, EP), called the NDT-EP method [16]. Varying several parameters, such as amplitude or number of pulses, will result in different thresholds being detected. The NDT-EP method has already shown very promising results and has now reached a point that it asks for larger clinical studies and applications.

When this method would be implemented in larger clinical studies and applications, other people than the researchers from BSS would have to work with the measurement system, including people without a technical background (non-expert users). The current system to measure nociceptive malfunction is Labview-based. Labview is a graphical programming environment with an integrated user interface and it can be used for dataflow programming. A technical background (including programming) is necessary to properly use Labview, and thus the current measurement system. In addition, the current set-up is difficult to properly document following the Medical Device Regulations (MDR).

The design-assignment of this research therefore reads as follows:

Initial development of a Python-based Graphical User Interface (GUI) for a Medical measurement system for Nociceptive (mal)Function.

Using a Python-based GUI instead of Labview, will allow non-expert users (with little training) to use the measurement system for the NDT-EP method. The GUI will navigate the user through the experiment and lets the user fill in desired inputs to execute a specific experiment, without the need to change any scripts or files. Therefore, knowledge of the software (and programming) will no longer be a requirement for the user to properly operate the system. Besides that, having a well structured Python-based software system will allow the researchers to more easily change and improve the measurement system.
In this research, a framework will be designed to accomplish the design-assignment as stated above and a first Python prototype will be realised to show that the realised theoretical design is a functional architecture for the new system. Several Python modules and libraries will be used for designing the framework, being: Tkinter (for creating a GUI), (abstract) classes (to allow for functional decomposition) and asyncio (for asynchronous concurrent programming). A simple risk analysis, according to the MDR, will be performed to show that software for a medical device entails new risks, different from the ones of the medical device. Lastly, risk control will be conducted to give recommendations on how to control the new identified risks.

Chapter 2

Background

2.1 NDT-EP Method

The NDT-EP method is developed by researchers from the Biomedical Signals & Systems (BSS) research-group at the University of Twente and measures the nociceptive detection thresholds (NDT) simultaneously with evoked potentials (EP) to get a better understanding of the nociceptive system [16].
The NDT are measured with a medical device; the NociTRACK AmbuStim. This stimulator sends out the commands and details regarding the pulses for electrocutaneous stimulation of certain nociceptive fibers. An IES-5 electrode, which has 5 needles closely placed together, is placed on top of the hand of the patient and produces the electrocutaneous pulses of the stimuli. The medical device contains a button, which the patient presses until a stimulus is felt and then the patient releases the button. The NDT are eventually determined by giving these stimuli consequently and based on if the previous stimulus was felt by the patient or not, the amplitude of the next stimuli sent will come closer and closer to the actual NDT. This can be seen in Figure 2.1. [17]

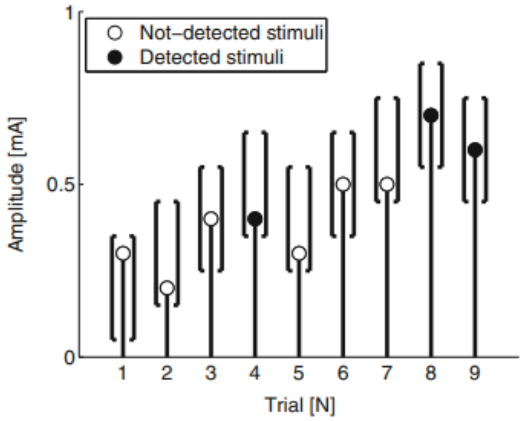


Figure 2.1: Representation of how the amplitude of the next stimulus is determined, based on whether the previous stimulus was felt or not. If a stimulus is indicated as not-detected, the set from which the next pulse is randomly chosen will be higher, if the pulse was indicated as detected, the set will be lower. This way, eventually the NDT will be reached. [18]

The EP are measured using electroencephalography (EEG) with an EEG-cap (number of channels can differ). The complete set-up can be seen in Figure 2.2[1]. Currently, the indication if the stimulus was felt or not by the patient by pressing and not-pressing a button is quite subjective, as patients can lose focus. Measuring EP with EEG is implemented to eventually switch to a more objective measure for indicating whether the stimulus was felt by the patient or not, by looking at the EEG signals. Besides,

[1]The device to measure EEG-signals represented in this figure is outdated.

if EEG signals were used and thus the indication of whether the pulse was felt or not is more objective, the stimuli to reach the NDT can be determined more precise, resulting in fewer stimuli being needed to reach the NDT and thus the experiment execution becoming faster.

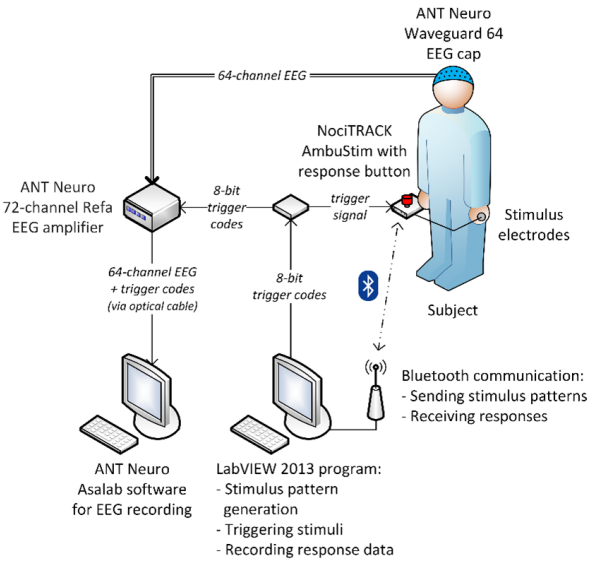


Figure 2.2: Graphical representation of the experiment set-up of the NDT-EP method, using an EEG cap, IES-5 electrode and a NociTRACK AmbuStim [17].

Prior to execution of the experiment, a so-called familiarisation is done, during which the approximate threshold (estimate of the NDT) of the patient is determined. This value serves as input for the initial stimuli in the experiment. By varying values of several parameters of a stimulus and based on whether the previous stimulus was felt by the patient or not, different thresholds (NDT) can be detected, this is the so-called Multiple Threshold Tracking (MTT) procedure. [17]

2.2 Medical Device Regulations (MDR)

The Medical Device Regulations (MDR) contain regulations for the design-process and production of medical devices, which have been drawn up by the European Parliament of the European Union (EU) [19]. The MDR also provides guidelines for the definition of what is considered a medical device and what rules and checks have to be carried out to ensure the medical device is safe for the patient, as well as the user (operator) of the system.

The software designed in this research is developed to eventually replace the Labview-based software, which is currently being used to control a medical device; the NociTRACK AmbuStim (intended-use of current system). This is considered a medical device, because the intended-use complies with the definition as given in the MDR: "An appliance intended by the manufacturer to be used, alone or in combination, for human beings for the investigation of a physiological or pathological process or state." [19]. The intended-use of the NociTRACK AmbuStim is stated as follows: "The AmbuStim device is intended for transcutaneous electrical stimulation of nerve fibers in the skin and in peripheral nerves of human subjects participating in scientific research and/or patients in clinical practice. Applications include electrical stimulation for clinical testing of neural functionality." [20].
The new system must have the same intended-use as the current system, so it can replace the current system. Therefore the intended use of (the software for) the new system is stated as follows:

***Intended-use** = The new system is intended to allow non-expert users to control[2] a NociTRACK AmbuStim stimulator (medical device) for electrocutaneous stimulation of nerve fibers and to guide these users correctly through the experiment flow to safely execute protocols to non-invasively measure (with an IES-5 electrode and EEG-cap) and determine the nociceptive detection threshold (NDT) and evoked potentials (EP) of patients, to get a better understanding of the nociceptive system.*

According to the MDR, the software that controls a medical device (NociTRACK AmbuStim) is considered part of the medical device and therefore has the same intended-use and thus the corresponding software is also covered in the MDR. However, for this research it is decided to split the intended-use of the medical device and the intended-use of the software, because this research is focused on the software for the medical device and the MDR involved with the software specifically. The intended-use of the software for the medical device is as stated above.
The MDR states that a risk analysis of the medical device, including the software, must be performed to establish that it is safe to use. Splitting the intended-use of the medical device and the intended-use of the corresponding software allows for a less extensive risk analysis, since only the risk analysis for the software has yet to be performed as the risk analysis for the medical device is already completed [20].

As this research is conducted in the early stage of development of a new system, a (simple) risk analysis is performed to demonstrate that software of a medical device probably entails new kinds of risks in addition to the already established ones from the medical device on its own and to show how the MDR can be applied to such software. At a further stage in the development, this risk analysis has to be extended to meet the regulations on risk analysis as stated in the MDR art. 82 [19].
A risk analysis can be performed in general for all types of users and patients, but the outcome of a risk analysis for a medical device, depends a great deal on what type of user is going the operate the device (or in this case software) and the patient. For this research only a small risk analysis will be performed and therefore it is chosen to limit the risk analysis and explicitly perform the risk analysis for a certain user and patient.
The user in this research is defined as a non-expert user, who should be able to operate the system with little training (such as a brief explanation of how to use the system to execute prescribed protocols). A non-expert user is someone who does not necessarily have knowledge about nociceptive processing and does not necessarily have a technical background, but is trained to correctly carry out protocols following the NDT-EP method (e.g. a lab-assistant). For the remaining of this research, the following definition is what the word (non-expert) user will refer to:

***User** = non-expert user; someone who is trained to carry out protocols following the NDT-EP method, but does not necessarily have knowledge on nociceptive processing and also does not necessarily have a technical background.*

The subject in this research is defined as a adult human patient, not within a specific age-range. Since only a very basic risk analysis will be performed for this research, it is chosen to exclude the risks imposed by particular vulnerabilities due to disease from the risk analysis and only include general imposed risks of a patient (and because whether risks imposed by particular vulnerabilities are present or not is to a large extent depended on the decision of a doctor to use this device for a specific patient or not). For the remaining of this research, the following definition is what the word subject will refer to:

***Subject** = adult human patient; without taking particular vulnerabilities due to disease in to consideration.*

Another very important point to address, because it affects the outcome of the risk analysis, is the fact that the new software is designed with only application in scientific-research in mind (for now), not in any way to diagnose or for treatment/monitoring of a patient.

[2] By controlling the medical device it is meant that the user has the ability to determine via a GUI when and which actions are being performed and that the user can deliver the desired input for the experiment and also access the output of the experiment.

2.3 Labview (current system)

The current measurement system is Labview-based. Labview is a graphical programming environment. It is used for data flow programming and the set-up consists of two different windows, a user interface window (Figure 2.3) and a block-diagram window. By constructing something in one window, it will automatically also be present in the other window. The Labview software has a set of all basic programming operations, such as loops, arrays etc. However, it is becoming more difficult to adapt this system to keep up with the current developments of the NDT-EP method at BSS.

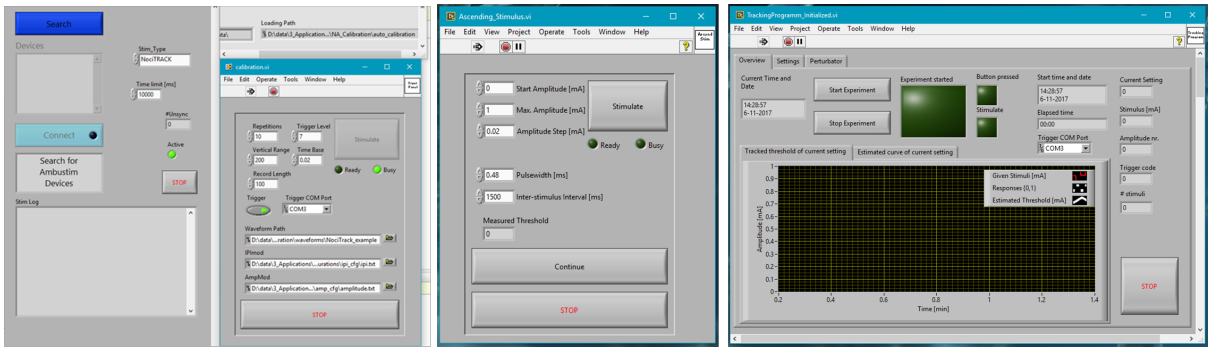


Figure 2.3: Several windows of the user interface for the NDT-EP method in the current software system (Labview) [17].

Because of its graphical programming nature in combination with the data flow programming, scripts for more complex systems tend to become very complicated and unclear quick. Besides that, programming in Labview is hardly taught to students anymore, so not a lot of people know how to use it nowadays. Up until now, this current approach has been sufficient for the research. However, the current promising results from the NDT-EP method at BSS ask for another approach that is more easily usable by a wider range of users, so it can be implemented in larger clinical studies and applications.

2.4 GUI in Python (new system)

When implemented in larger clinical studies and applications, the system is going to be used by non-expert users to conduct experiments on patients (following the NDT-EP method) and software developers should be able to make changes and adaptations to the system more easily than currently is the case with the Labview-based system. Taking this into consideration and also that Labview is hardly taught to students anymore, it is decided to switch to a new type of measurement system, namely Python-based[3]. A (Python-based) GUI for a measurement system allows non-expert users with little to no training to correctly execute a protocol and it allows researchers to more easily adapt and improve current protocols unlike the data flow programming which is used in Labview. Besides that, Python is currently one of the most commonly taught programming language (to students) and therefore usable by a wider range of users.

2.4.1 Implications of the New System

Switching to a Python-based system entails several implications. These new implications have been investigated and can be used for an initial design of an architecture for the new system. More information on different (technical) implications is given below.

[3]Since the assignment stated to design a new system in Python, the possibility of implementing another language has not been investigated.

Structured Programming
To achieve a structured software system, object-oriented programming (OOP) can be implemented. One of the advantages of using OOP is that a "blueprint" piece of script can be written and a script for different parts can be built from this without the need to write a separate script for each part. This way, if changes are made to e.g. a protocol, not every script has to be changed separately. In Python it is possible to implement OOP by using classes to allow for more structured coding and independently changing different parts of a protocol.
A downside is that programming in this way takes a lot of time before a first prototype can be realised (and tested), because all the classes have to work together smoothly before the software functions properly. In addition, this way of programming asks for proper maintenance for the same reason.

Graphical User Interface
Python offers the possibility to implement a GUI. Several modules/libraries are available for building a Python-based GUI: PyQT5, Tkinter, PySide 2 (Qt), Kivy and wxPython (many more are available). These modules/libraries can all be used to create GUI's.
Each module/library contains its own widgets and is used differently. Because they all differ in usage, they each have different advantages and disadvantages. In addition, some modules/libraries require a paid license, while others are free of charge.

Concurrent programming
Python contains modules/libraries for implementation of concurrent programming for smoother code execution. Usually, different tasks in a script are executed sequentially, whereas in concurrent programming, tasks are actually executed simultaneously. Concurrent programming can be achieved by implementing multi-threading programming or asynchronous-programming (Figure 2.4). In multi-threading programming, multiple tasks run simultaneously, while asynchronous-programming only gives the illusion that this is the case, when in fact these several tasks run alternately.

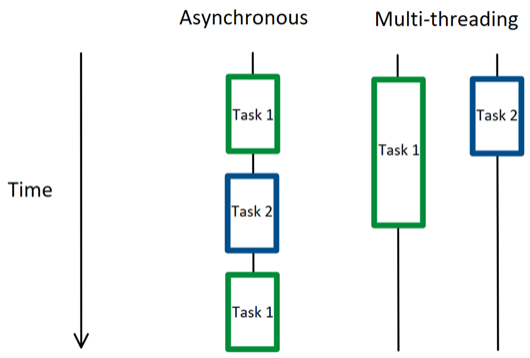


Figure 2.4: Visual representation of two forms of concurrent programming: asynchronous programming and multi-threaded programming.

Figure 2.4 shows the difference between the two types of concurrent programming: asynchronous programming and multi-threading programming.
With asynchronous programming, tasks are executed sequentially by executing tasks alternately for a given time or when another task is blocked, which gives the illusion that tasks run simultaneously, when in fact they do not (Figure 2.4). Programming in this way speeds up the execution time, because tasks do not "wait" to run for the other tasks to finish, but they run in between. However, when one task is being executed, all other tasks are hold on pause, which can be a major drawback when using this type of concurrent programming. This might raise complications when combining a GUI with asynchronous programming, because the GUI has to constantly run to allow interactions from the user with the GUI.

With multi-threading programming, tasks are actually executed simultaneously as opposed to asynchronous programming (Figure 2.4). This way, the problem that arises with asynchronous programming (tasks completely stopping execution while "waiting" for other tasks to finish) is not an issue. However, programming in this way takes up much more RAM (random access memory), which can be a disadvantage. This is because the system becomes slower when the RAM is full, which will prolong the execution time. Therefore multi-threading is mainly used for larger programming projects.

2.4.2 Design-choices Considering the Implications of the New System

Taking the implications of the new system into consideration, design-choices were made on what tools to use for a new system. Reasoning for why certain design-choices were made for a new system will be explained below.

Object-oriented Programming with Classes and Objects

An important tool for the new measurement system, is the use of classes and objects in Python (OOP). Using OOP, allows for more convenient and more structured programming. Therefore it was chosen to use classes and objects as the foundation for the software of the new system.
Classes serve as "blueprints" for objects, so for different experiments only an object's attributes have to be changed and not the overall code. The different parts of the experiment, can be made into different classes and several protocols can be different objects of the main controller of the measurement device, allowing to more easily make adaptations to different parts of an experiment and implementing new types of experiments. The use of OOP prevents that the whole system has to be revised when changes to only a part of the system are made.
Besides this, the classes can be combined with the module/library for creating the GUI nicely, to create several different windows for the user to navigate through. Easy navigation for the user is also a requirement, making classes an important tool for the design of the new system. [21]

Abstract Classes
For the controller class of the new system, it was decided to use Abstract Classes. An abstract class serves as a "blueprint" for multiple classes that contain several, but not all, of the same variables and methods. Using this type of class allows for an even less extensive code for the software of the new system. [22]

Tkinter (themed)

Another important part of the new system is the GUI. It was chosen to use the Python (themed) Tkinter module, because this module is considered the standard Python interface to the GUI toolkit [23] and because it contains all elements for an initial development of GUI (for a measurement system for nociceptive (mal)function). It is also relatively easy to use, because of its set-up and, as it is considered the standard Python interface to the GUI toolkit, it is very accessible and relatively many resources about using Tkinter are available, making it beginner-friendly to use. In addition, a paid license is needed for some GUI modules/libraries and Tkinter is completely free of charge.
The Tkinter module provides the most important components needed for designing a straight-forward GUI, that is easy to use for non-expert users. The Tkinter module consists of several classes, each for a different type of widget. Widgets are constructed in Python using this module, by making an object of the desired widget-class. The most important widgets in the module are: labels, entry-widgets, text-widgets, (different types of) buttons, (selection-)menus, canvasses (to display images) and (different types of) windows and frames. By using Python's Tkinter module and its wide variety of widgets to choose from, a GUI that meets the requirements can be realized.
A themed version of the module is available that gives access to more modern designs of the widgets within the general Tkinter module, which allows for a more aesthetically pleasing look of the GUI. [23]

Concurrent programming with asyncio

It was chosen to implement the asyncio library for concurrent programming, because this module is used in the already existing script for the controller of the NociTRACK AmbuStim, which will be implemented in the new software design. This way, any potential errors in further development, caused by combining the asyncio library functions combined with some other library for concurrent programming, are prevented.
As explained in detail in Chapter 2.4.1, concurrent programming with asyncio gives the illusion that multiple tasks run parallel, when in fact they run alternately (asynchronous programming). Each task runs for an assigned amount of time or when nothing else runs, which results in smoother code execution. Asyncio is a library that provides functions needed for asynchronous coding. Implementing these functions correctly for several tasks within the final design, might improve the usability of the measurement system, since it can reduce running time. Especially when it comes to larger pieces of code, this can be beneficial to use. [24]
For now, asynchronous programming is only implemented in the controller-script of the NociTRACK AmbuStim. There is no direct need to implement this already into the rest of the new system, but it is worth looking into as it can maybe be used to constantly update graphs or connection statuses in the GUI, without the need of using unnecessary high amounts of RAM. So, concurrent programming will not yet be implemented in the initial development of a new system, but recommendations to use this module for several parts of the new system will be made.

2.4.3 Requirements

The new design has to meet several requirements (based on the good features of the current system and the desired features for the new system) to eventually replace the current measurement system. These requirements entail certain implementations. Both are listed and explained below.

- **The new system must be able to control the NociTRACK AmbuStim and determine the NDT.**
  The new system is going to replace the current system used to control the NociTRACK AmbuStim for implementing the NDT-EP method and therefore must have the same intended use.
  Implementation: The new system must be able to control the NociTRACK AmbuStim for sending out different stimuli to an electrode. Therefore, the new system should be able to sent corresponding commands to the stimulator and determine the NDT. Additionally, the new system should eventually also be able to implement the MTT procedure to determine the stimuli that are being sent out then.

- **The new system must meet the general safety & performance regulations as stated in the MDR art. 82.**
  As explained in the previous, the software of the new system is part of a medical device making it that also the same rules apply for the software system.
  Implementation: A risk analysis of the design for the software architecture of the new system should be carried out. As this is an initial development of a software for a new system (and for the sake of time) it was decided that for now, only a simple risk analysis will be carried out, to test if the (theoretically) designed software does not cause any safety issues and is safe for the patient and the user of the system. However, for further development also the other regulations as stated in the MDR should be taken into consideration.

- **The new system must be compatible with the current analysing methods.**
  Since all current analysing methods at BSS are based on .ini type files (config-files), the output-files of the new system have to be of the same type. This allows for analysing with already established analysing methods and avoids the need to change the existing analysing protocols.
  Implementation: The output files of the new system have to be of the type .ini.

- **The new system must be compatible with the file-type of the current input files.**
  Since all current input files at BSS are of the type .ini (config-files), the new system must be able to use these files as input for the system. This prevents the need for redesigning the current input files for the already existing protocols.
  Implementation: The new system has to be compatible with .ini files as input.

- **The software of the system has to be organised in such way that each part can be independently changed.**
  One of the main reasons to switch to a new measurement system was because in the current Labview-based system it was rather difficult to make adaptions and/or improvements to the already existing protocols. To eventually replace the current system, the new systems has to provide a solution to make it possible that each part can be independently changed.
  Implementation: The new system has to be designed in several, blueprint-like, components, that each represent a different part of the experiment, to allow for adaptations and improvements to different parts independently.

- **The GUI must be usable by non-expert users.**
  The GUI has to be designed in a way that the user only needs to fill in or select the information needed to execute the desired experiment. It also has to be straightforward for the user in which order the software-parts of the system have to be completed.
  Implementation: The GUI in the new system should consist of widgets and windows that are straight-forward and comprehensible for non-expert users.

- **The functional components[4] of the current system have to be present in the new system (adaptations allowed).**
  For several parts of the current system, there is no need to change them. These include[5]:

  - A slider in the familiarisation phase for determining the approximate threshold.
  - A log-file that stores updates on the experiment.
  - The components of the bluetooth connection window (e.g. a sync status light).
  - A graph showing the stimuli and indicating whether they were felt by the patient.
  - Different windows for different stages of the experiment (e.g. a different window for device configuration, familiarisation, EEG-graphs, experiment execution etc.).

  If these components meet the requirements of the new system, there is no need to change them. However, it might be needed to make some minor adjustments to better fit the design of the new system.
  Implementation: A list of functional features of the current system, should also be incorporated into the design of the GUI of the new system.

In this research, an initial framework (class-diagram) of software for a new system will be designed that meets all the requirements above. Small bits of this framework will be programmed (Proof of Principle), to proof that it is a functional architecture for developing a GUI (and corresponding software) for a new system for measuring nociceptive (mal)function.

[4] Component = widget (to indicate or control something in the experiment) or functionality (e.g. different windows, log-file)

[5] The given list is not complete, this is just an own interpretation of the functional components of the current system. For further development, it should be investigated with the current users which components of the current system are considered functional components and should be preserved in the design of the new system.

Chapter 3

Design

In the following chapter the final design will be discussed. The new system is designed to meet the requirements from Chapter 2.4.3 and the tools discussed previously were used to realise this.
Before designing the software architecture of the new system, a flowchart of the current system[1] (Labview-based) was made to get a better understanding of what some of the key features of the current system are and which ones could be improved or replaced in the new system (Python-based).

3.1 Current system (Labview)

The current system uses Labview to build and improve protocols and run experiments, using the integrated user interface. The flowchart in Appendix 7.1 shows an overview of the current system with its interactions indicated by arrows. From left to right, it shows the user that interacts with the GUI, which interacts with the software (Labview unless otherwise specified (indicated by "(Matlab)")). The software interacts with the NociTRACK AmbuStim (and indirectly the patient), EEG-cap (and indirectly the patient) and the different types of data(-files), represented by different blocks in the flowchart. The solid arrows represent a sent-message and the dotted arrows represent a return-message. The colors in the system indicate different parts of the experiment. These different parts will be explained further in the following. A simplified version of Appendix 7.1 can be seen in Figure 3.1.

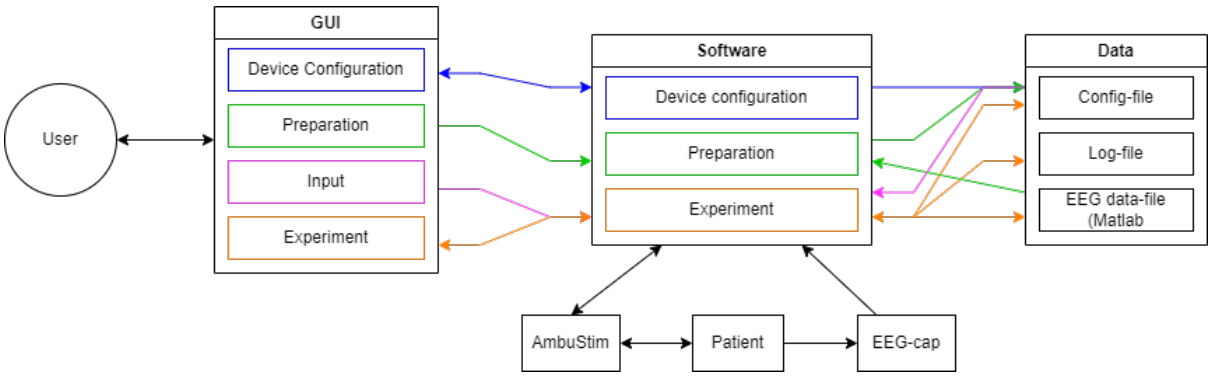


Figure 3.1: Simplified Flowchart of the extended version in Appendix 7.1. The simplified version shows only the main components of the system and the arrows indicate an interaction between two components.

[1]The flowchart of the current system is not representative for the actual software of the current system as it was merely created as an own interpretation of how the system works.

3.1.1 Device Configuration

In Device Configuration, the user can search for the available Bluetooth devices and select the desired device. Once selected, the user interface will show updates about the connection with the device (sync-status) to the user. In this part the selected device will also be calibrated to correct for the minor deviation in the pulse. The corrected value is loaded in the config-file.

3.1.2 Preparation

In Preparation, the user can prepare the equipment and software for use. The impedance of all the channels on the EEG-cap can be checked. This information will be pulled from the EEG data-file where the data from the EEG-cap is stored. Familiarisation can be executed to determine the approximate threshold of the patient. The approximate threshold will be stored in the config-file.

3.1.3 Experiment and Input

In Experiment, the user can give the desired input for executing the experiment and the user also has control over the execution of the experiment with different buttons to start, pause and quit the protocol. In addition, the user interface shows a graph with the threshold and the stimuli that are sent out and whether they were felt or not by the patient. It also shows a graph with the EEG-signal of all channels (in Matlab).
For this part the software also interacts with the NociTRACK AmbuStim and the EEG-cap. The software ensures that the NociTRACK AmbuStim sends out pulses to the patient; the NociTRACK AmbuStim returns whether the button is pressed or not by the patient. All information regarding if the pulse was felt by the patient or not is stored in the config-file. The software stores the EEG-data collected by the EEG-cap in a EEG data-file. A log-file tracks updates on the experiment execution. Both the config-file and the EEG data-file are used by the software to show graphs in the user interface. The log-file can also be accessed.

3.2 New system (Python)

The foundation of the new system is Python and its module Tkinter and classes. The Class Diagram in Appendix 7.2 shows the complete overview of the design for the system architecture for the new system with its interactions indicated by arrows. The "+" indicates variables ("self-variable") and functions that can be accessed from other classes, the "-" indicates variables that are only accessible in its own class. Figure 3.2 shows a simplified version of the Class Diagram in Appendix 7.2.
The new system has some elements from the old system as well as some new, improved components. The foundation of the new system's software architecture are (abstract) classes. The controller classes («abstract» ControllerAmbuStim()) and the log class (LoggerSaveFiles()) are not classes from the Tkinter module and do not have their own window. Therefore, these classes can not be accessed from the graphical user interface (GUI), but they do function as input for several other classes. Each part of an experiment has its own class and with that its own window with several input options in the GUI for the user. All classes will be explained in the following.

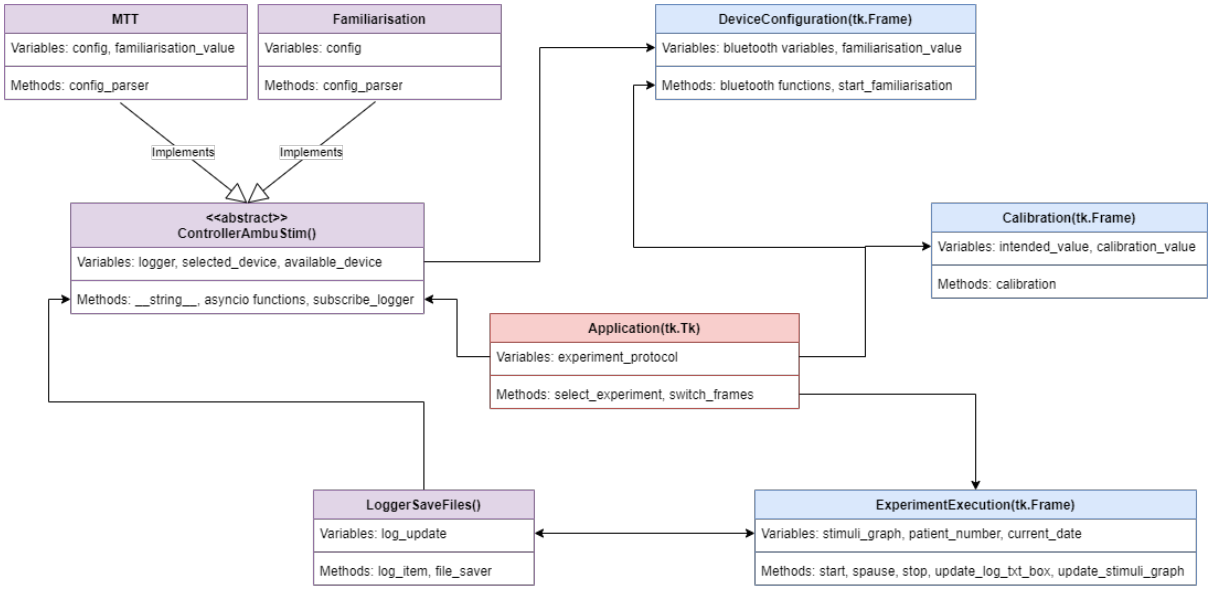


Figure 3.2: Simplified Class Diagram of the extended version in Appendix 7.2. The simplified version shows only the classes with the most important functions and variables and the arrows indicate a connection between two classes.

3.2.1 «abstract» ControllerAmbuStim()

The ControllerAmbuStim() class functions as the controller for the NociTRACK AmbuStim device. This class is an abstract class. Different experiments (MTT and familiarisation) are sub-classes of this abstract class. This class holds all the information needed to execute the experiment.
It has a variable for all the available devices and it has a function to (dis)connect a selected device. It also has a function (*__string__*), which at the start prints to the log-file the selected device and the current controller (from which experiment). It also contains a dictionary with values for all unique parameters of an experiment from a config-file. It contains a set of four asyncio functions that, combined with the parameter values from the config file, control the pulses that are being send out (and when) by the NociTRACK AmbuStim. Besides that, this class also has a function to subscribe a logger (creating an object of the log-class) and in all the functions that execute a task, it is integrated that a log-update is sent out as a string.

3.2.2 LoggerSaveFiles()

The LoggerSaveFiles() is a class, which object logs all updates (strings printed by functions) from the ControllerAmbuStim() class to a log-file. It contains a function to enable displaying these string print statements (saved to log_update variable) in the GUI (*log_item*) in the frame from the ExperimentExecution(tk.Frame) class. It also holds a function to save all data at the end of an experiment to the desired location (*file_saver*).

3.2.3 Application(tk.Tk)

The Application(tk.Tk) class takes a class from the Tkinter module as input and its object functions as the root-window of the application. This window will contain the first frame the user sees when the application is opened. This class contains an entry widget for the user to choose which experiment they would like to perform. When a certain experiment is selected, a function will make an object from the corresponding ControllerAmbuStim() sub-class (*select_experiment*) and save this object to the variable experiment_protocol of its own object (controller_root), so it is easily accessible from other classes. This is done in the root-frame of the GUI, so that the variables of this object (*select_experiment*) can

be accessed and changed in the other frames. This class also contains the function to switch between different frames (*switch_frames*) when a button is clicked. From this frame, the only option is to go to the DeviceConfiguration(tk.Frame) frame by clicking the "Next"-button.

3.2.4 DeviceConfiguration(tk.Frame)

The DeviceConfiguration(tk.Frame) class takes the frame class from the Tkinter module as input and its object takes the frame as well as the object of the Application(tk.Tk) class as its input (parent_frm, controller_root). The class contains several functions necessary to ensure that the selected experiment can be executed smoothly later on.
First of all we have a set of three function regarding the Bluetooth connection. *bluetooth_search()* can be initiated by clicking the "Search"-button in the GUI. This function gets the available devices from the object of the ControllerAmbuStim() class (controller_root.experiment_protocol) and displays the available devices in the GUI. The user can then enter the desired device and by clicking the "Connect"-button, the function *bluetooth_connect* will connect the selected device by changing the associated variable in the ControllerAmbuStim() object. It also contains a function that displays the status of the connection status in a text-box in the GUI (*bluetooth_connection_status*).
Besides those functions, there is also a button that starts the familiarisation by initiating the function *start_familiarisation*. This function creates an object of the familiarisation sub-class of the ControllerAmbuStim() abstract class and executes its protocol. After the familiarisation is completed, it returns the familiarisation_value to the experiment_protocol object of the ControllerAmbuStim() abstract class from the root_controller object.
Lastly, there are two buttons to switch to a different frame by using the function *switching_frames* from the controller_root object. The "Next"-button navigates to the Calibration(tk.Frame) frame and the "Back"-button navigates to the Application(tk.Tk) root-frame.

3.2.5 Calibration(tk.Frame)

The Calibration(tk.Frame) class takes the frame class from the Tkinter module as input and its object takes the frame as well as the object of the Application(tk.Tk) class as its input (parent_frm, controller_root).
By clicking the "Calibration"-button, the function *calibration* is initiated. This function measures the exact value of the pulse being sent out, compares this to the input value (intended_value) and calculates the error. The determined calibration_value is being returned to the experiment_protocol object of the ControllerAmbuStim() abstract class from the root_controller object.
This class also contains a "Next"-button and a "Back"-button to switch to a different frame by using the function *switching_frames* from the controller_root object. The "Next"-button navigates to the ExperimentExecution(tk.Frame) frame and the "Back"-button navigates to the DeviceConfiguration(tk.Frame) frame.

3.2.6 ExperimentExecution(tk.Frame)

The ExperimentExecution(tk.Frame) class takes the frame class from the Tkinter module as input and its object takes the frame as well as the object of the Application(tk.Tk) class as its input (parent_frm, controller_root).
This class contains a set of functions to stop, pause or quit the experiment (*start*, *pause*, *stop*). These functions are initiated when the corresponding buttons in the GUI are pressed by the user.
Furthermore, this class contains two entry widgets for both the patient number and the date. These entries are the input for the function *update_log_txt_box*. This functions constantly displays updates in a text-box in the GUI from the LoggerSaveFiles() object (logger) from the object of the controller of the experiment from the ControllerAmbuStim() sub-class (controller_root.experiment_protocol).
This class also contains a function that displays in a graph in the GUI the stimuli (including whether they

were felt by the patient or not) and the threshold (*update_stimuli_graph*). The info needed to display this, is pulled from the object of the subclass of ControllerAmbuStim() (controller_root.experiment_protocol). Lastly, this class also contains a "Next"-button and a "Back"-button to switch to a different frame by using the function *switching_frames* from the controller_root object. The "Next"-button navigates back to the Application(tk.Tk) root-frame (end of the experiment) and the "Back"-button navigates to the Calibration(tk.Frame) frame.

Chapter 4

Prototype Realisation

To demonstrate that the theoretical design discussed in Chapter 3.2 (class-diagram in Appendix 7.2) will work when implemented, a small part of the architecture is turned in to a script for Proof of Principle, which can be found in Appendix 7.3.2. The GUI starts with a root-window where the user can enter the name of the config-file of an experiment and a patient number. By using several buttons, the user can easily navigate through the different windows. In the ExperimentExecution window the user can start an experiment by clicking the corresponding button. It was chosen to program these parts, because switching windows, starting an experiment and the possibility to provide input as user are key-features of the theoretical design realised in this research.
Appendix 7.3.1 contains an already existing controller script for the NociTRACK AmbuStim (Original script by Prof. Dr. ir. J.R. Buitenweg (BSS, University of Twente), adapted by M.E.L. Janssen (BMT-B student, University of Twente)), which was used for this initial prototype realisation (Proof of Principle).

4.1 Graphical User Interface

A few screenshots of the GUI the user will see on the computer screen, will be shown and explained below.
In Figure 4.1 the start-window of the application can be seen. The user will see the root-window with the root-frame. In the root-frame there is an entry-widget for entering the name of the config-file of the desired experiment and there is an entry-widget for entering the patient number. At the bottom of the frame there is a button to save the entered settings and a button to proceed to the next page, that will be enabled once the settings are saved.

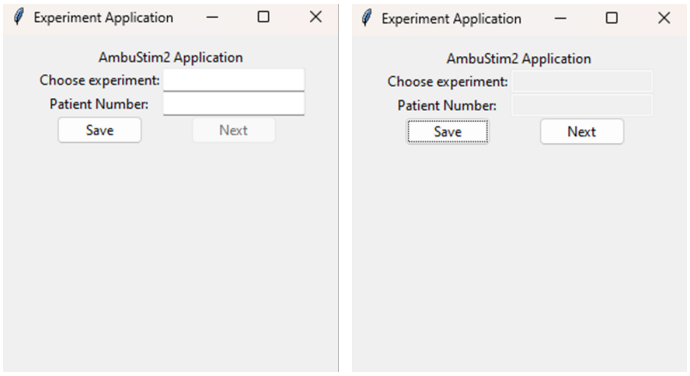


Figure 4.1: Window of the GUI the user will see when the application is opened. There are two entry widgets: one to choose the experiment and the other to enter the patient number. After the input-values have been saved by the corresponding button, the entry-widgets become grey and the user can no longer change the input and the "Next"-button becomes available.

After the "Next"-button is clicked, the user can proceed through three different windows: DeviceConfiguration, Calibration and ExperimentExecution as can be seen in Figure 4.2. In the last window, the user can start an experiment with the chosen config-file by clicking the "Start"-button. The application can be closed by clicking the "Quit"-button.

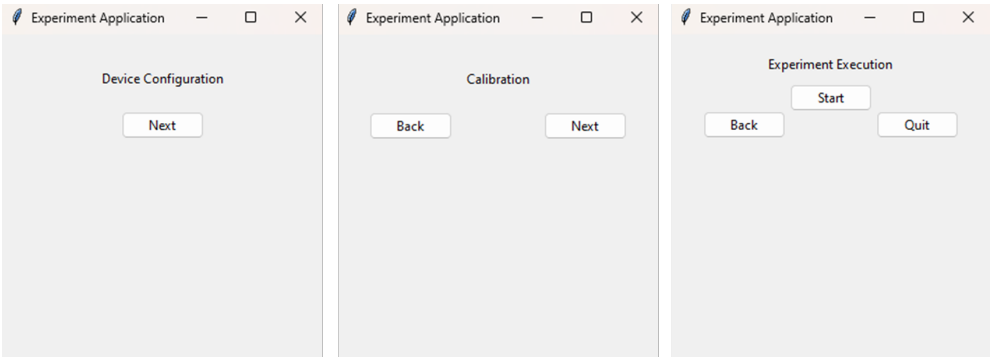


Figure 4.2: The three different windows the user will see by clicking the corresponding buttons. The last window also contains a button to start an experiment with the selected config-file from the first window.

Once the application is closed, the user can access a log-file from the same folder the application is in. The log-file has a title with the patient number and date/time in it. The log-file of an experiment without a NociTRACK AmbuStim connected can be seen below in Figure 4.3.

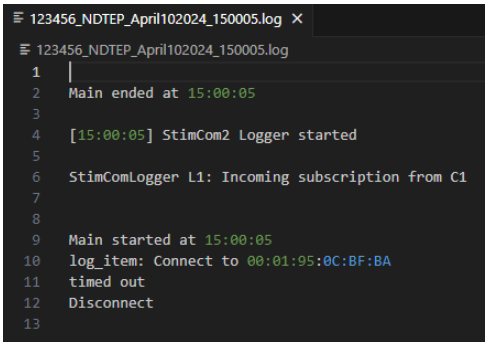


Figure 4.3: Log-file of an experiment without a NociTRACK AmbuStim connected.

4.2 Methods of Classes

A few important methods (functions of classes) of the Proof of Principle script for the GUI are explained and shortly discussed below. The full script can be found in Appendix 8.3.2.

4.2.1 Method 1: save_experiment()

The method *save_experiment* is a method from the Root(tk.Tk) class and takes the input from the entry widgets in the main window and uses a config parser module (cp) to retrieve the right information from the config-file and load this information into the dictionary of the controller object.

```
    def save_experiment(self, main_frm, input_var1, input_var2):      # Only works for single threshold determination(familiarisation)
        print(self.experiment_protocol.Config.config)    # Check for developers to see if the correct dict is used

        self.main_entry_output = input_var1.get()
        self.patient_entry = input_var2.get()

```

```
7         config = cp.ConfigParser()
8         config.read(self.main_entry_output)
9
10        settings_list = []
11        config_dict = {}
12        for key in config['Threshold 1']:                # Creating a dict with
    al settings and corresponding values
13            settings_list.append(key)
14
15            for setting in settings_list:
16                item = config['Threshold 1'][setting]
17
18                if item == '':                           # To make sure the
    values are of the correct type
19                    value_list_stripped = item
20                else:
21                    try:
22                        value_list_stripped = int(item)
23                    except: # if not an integer --> makes it a float
24                        value_list_stripped = float(item)
25
26                config_dict[setting] = value_list_stripped      # Dict with key
     = setting-name and value = setting-value
27
28        nop = config_dict['nop']    # variable (int) for Number of Pulses (NoP)
29
30        # Putting values of config.ini file into config_dict of controller
    object
31        self.experiment_protocol.Config.config["V"]["Value"] = [config_dict['
    selectionmethod']]
32        self.experiment_protocol.Config.config["I"]["Value"] = [(config_dict['
    ipi'])/(0.035) for x in range(nop)]
33        self.experiment_protocol.Config.config["P"]["Value"] = [(config_dict['
    channel']) for x in range(nop)]
34        self.experiment_protocol.Config.config["A"]["Value"] = [1 for x in
    range(nop)]
35        self.experiment_protocol.Config.config["a"]["Value"] = [0 for x in
    range(nop)]
36        self.experiment_protocol.Config.config["W"]["Value"] = [(config_dict['
    pw'])/(0.035) for x in range(nop)]
37        self.experiment_protocol.Config.config["w"]["Value"] = [0 for x in
    range(nop)]
38        self.experiment_protocol.Config.config["C"]["Value"] = [1 for x in
    range(nop)]
39        self.experiment_protocol.Config.config["M"]["Value"] = [1,1]
40        self.experiment_protocol.Config.config["R"]["Value"] = [0,0,0]
41
42        print(self.experiment_protocol.Config.config)           # Check for
    developers to see if the correct dict is used
43
44        input_var1 = ttk.Entry(main_frm, state="disabled")       # Overlays the
     entry widget and disables it after chosen experiment is saved
45        input_var1.grid(row=1, column=1)
46        input_var2 = ttk.Entry(main_frm, state="disabled")       # Overlays the
     entry widget and disables it after chosen experiment is saved
47        input_var2.grid(row=2, column=1)
48        main_btn = ttk.Button(main_frm, text="Next", command=lambda: self.
    show_frame(DeviceConfiguration))  # Enables "next"-button
49        main_btn.grid(row=3, column=1)
50
51        return
```

Listing 4.1: Function to get the config-file from the entry-widget and implement the parameter values in the dictionary of the ControllerAmbuStim() object.

4.2.2 Method 2: show_frames()

The method *show_frame* is a method from the Root(tk.Tk) class and enables to switch between different frames by using the Tkinter function *.raise()* to allow the user to easily navigate through different windows.

```
def show_frame(self, frame_class):
        frame = self.frames[frame_class]
        frame.tkraise()
```

Listing 4.2: Function to switch between frames.

4.2.3 Method 3: start_measurement()

The method *start_measurement* is a method from the ExperimentExecution(tk.Frame) class and lets the user run the desired experiment (config-file), by clicking the corresponding button. This button starts an asyncio-tasks routine.

```
    def start_measurement(self, root_controller_object, root_controller):   # Function that starts a measurement
        datetime_info = datetime.datetime.now()
        date_str = datetime_info.strftime("%B%d%Y")
        time_str = datetime_info.strftime("%H%M%S")

        old_std_out = sys.stdout                                             # To have the command output be put into a log-file
        log_file = open(f"{root_controller.patient_entry}_NDTEP_{date_str}_{time_str}.log","w")
        sys.stdout = log_file

        async def main():   # Async function is used for the actual experiment execution
            print(f"\n\nMain started at {time.strftime('%X')}")
            C1.connect(StimCom2DeviceBTaddress["NociTRACK-017"])
            await asyncio.sleep(3)

            await asyncio.sleep(3)
            C1.disconnect()
            await asyncio.sleep(3)

        print(f"\nMain ended at {time.strftime('%X')}")

        C1=root_controller_object   # Input for running the controller is the StimCom2Controller object made in the root class
        L1=StimCom2Logger("L1")
        C1.subscribe_logger(L1)

        asyncio.run(main())

        sys.stdout = old_std_out
        log_file.close()

        print("Measurement is finished.")
        return
```

Listing 4.3: Function to start the experiment execution.

Chapter 5

MDR and Risk Management

5.1 Risk Management

The design process, and eventually the use of the software, poses risks. The MDR states that a risk analysis has to be performed to identify all potential risks of the software and how to control or either minimize these risks. As this research is conducted in the early stage of development of a new system, only a simple risk analysis is performed. The risk analysis is divided into two parts: risk assessment and risk control. Also, the intended use, the user and the patient need to be clearly indicated.
Since the software of the new system is part of a medical device (NociTRACK AmbuStim), also the risks associated with the medical device have to be taken into consideration. A risk analysis according to ISO14971 has already been performed for the NociTRACK AmbuStim, so no more attention will be paid to this [20].

5.1.1 Intended use

The software is intended to only be deployed in clinical testing, by non-expert users on patients as defined in Chapter 2.2 and the intended-use is defined as stated in Chapter 2.2 as well.

5.1.2 Risk Assessment

The risk assessment is carried out to identify and evaluate the potential risks. First, the potential risks of the software design will be identified. Afterwards, the potential risks will be analysed by determining the probability of occurrence (PoC) and evaluating the consequences when occurring (CwO). The PoC and CwO are quantified by a classification number. The identified and quantified potential risks are put into a risk evaluation matrix. This diagram gives an overview of the risks relative to each other and the gradient colours are a measure of severity of the risks. Lastly, the potential risks will be prioritised based on the risk analysis.

The significance of the classification numbers is as follows:
Classification number:
1 = impossible that it will happen/harmless consequence.
2 = improbable that it will happen/negligible consequence.
3 = remote that it will happen/marginal consequence.
4 = it will happen occasionally/severe consequence.
5 = probable that it will happen/critical consequence.
6 = it will happen frequently/catastrophic consequence.

Risk Identification and Analysis
The following risks were identified based on the theoretical design of the architecture for the new sys-

tem. To determine this, it was looked at the different parts of the system and actions that the user can perform, through the GUI, that may carry risks.

- R_1: **User can press the buttons in the wrong order.**
  R_1 can result in incorrect values for experiment parameters (e.g. incorrect approximate threshold), which can cause the NociTRACK AmbuStim to sent out unwanted/unexpected stimuli. The user will probably not do this on purpose, but a wrong button can be accidentally pressed occasionally, therefor PoC = 4. Unwanted/unexpected stimuli will probably not have severe consequences but might be unpleasant and temporarily painful for the patient and thus CwO = 3.

- R_2: **User manually fills out incorrect values for experiment parameters in the ExperimentExecution(tk.Frame) window (e.g. too high amplitude).**
  R_2 can cause the NociTRACK AmbuStim to sent out wrong/dangerous stimuli. The user will probably not do this on purpose, but a wrong value can be accidentally filled in occasionally, therefor PoC = 4. However, entering dangerously high values for parameters can have severe consequences, so CwO = 4.

- R_3: **The Application(tk.Tk) class can fail.**
  If R_3 occurs, the user is no longer able to navigate to a different window. Also the ControllerAmbuStim() class object is no longer accessible, so the experiment can not be executed anymore and no stimuli can be sent out. As the script for the system is tested before hand, the chances of the script running an error are almost negligible, however not completely as the system can always shut-down by an unidentified error since hardly any software is bug-free, therefor PoC = 2. When occurring, no experiment can be executed and thus the consequences will be harmless, CwO = 1.

- R_4: **A part of the application (GUI) can fail during the execution of an experiment.**
  If R_4 occurs, the software system no longer works correctly, but when an experiment execution is in process, wrong parameter values can be sent to the NociTRACK AmbuStim or "Stop"-buttons are not properly functioning anymore, which can cause dangerous/unwanted stimulation and might have severe consequences, therefor CwO = 4. However, it is improbable that system will shut down during an experiment, because the scripts need to be properly tested beforehand, so PoC = 2.

- R_5: **User can not directly access a "Stop"-button in several windows.**
  The user has to navigate to the ExperimentExecution(tk.Frame) window first, when R_5 is occurring. If a patient gets undesired pulses, these will not stop until the user has navigated to a window with a "Stop"-button and clicked it, these stimuli will not be severe but can cause (temporary) marginal injuries, therefor CwO = 3. It is very likely that a patient requests the user to stop the experiment, so it will probably happen occasionally, PoC = 4.

- R_6: **User can input variables of the wrong type (e.g. string-type variable instead of an integer-type variable) for certain entry widgets.**
  R_6 will result in the code not working (properly). If the user has had training the chances of entering a wrong data-type in a entry widget is small, but not completely unlikely therefor PoC = 3. As the script will run an error when occurring and the system will shut down, the consequences will be harmless, CwO = 1.

- R_7: **User can enter values that are too high or low for the NociTRACK AmbuStim to process.**
  R_7 will result in the device sending out its maximum or minimum achievable value. If the user has had training, the chances of entering values that are too high or too low for the device to process, are improbable, because one will most probably choose another medical device if the purpose is to use higher or lower values for stimulation than the NociTRACK AmbuStim can process, therefore PoC = 2. As the NociTRACK AmbuStim can simply not sent out values that exceed the maximum value or not reach the minimum value, the consequences will probably be harmless, so CwO = 1.

Table 5.1 below shows an overview of the risk identification and analysis. The first column indicates which risk the values belong to (R_#). In the second and the third column the potential risks are quantified by a classification number for both the PoC and the CwO. The fourth columns shows the Risk Priority Number (RPN = probability * consequences).

Table 5.1: Identification and analysis of potential risks. Each risk is graded for probability of it occurring (PoC) and the consequence if so (CwO). The significance of the classification numbers are as explained before.

$R_{\#}$:	PoC:	CwO:	RPN:
R_1	4	3	12
R_2	4	4	16
R_3	2	1	2
R_4	2	4	8
R_5	4	3	12
R_6	3	1	3
R_7	2	1	2

Risk Evaluation Matrix
The risks are put into a risk evaluation matrix (Figure 5.1) to get an overview of how the identified potential risks compare to each other in terms of PoC and CwO. The gradient colour is added to the diagram, to visualize the level of severity of the risks (relative to each other). The darker shade of blue indicates the highest severity and the lightest shade of blue indicates the lowest severity.

Figure 5.1: Risk evaluation matrix with the identified and quantified potential risks in their corresponding positions according to the PoC and CwO.

This risk evaluation matrix clearly indicates that potential risks R_2, R_1 and R_5 are the most dangerous potential risks and should have the highest urgency for risk control. This outcome is in line with the risk prioritization according to the RPN, since these risks have the highest RPN as well.

Risk Prioritization
Below a list can be found of the prioritization of the potential risks according to their RPN. First place is highest RPN, last place is lowest RPN.
1. Risk R_2
2. Risk R_1 & Risk R_5
3. Risk R_4
4. Risk R_6
5. Risk R_7 & Risk R_3

5.1.3 Risk Control

The risk control is carried out to determine whether the risks can be prevented by changing the design, by adding new, protective components to the design or by providing information on safety. The risk

control will be discussed for each risk, from high RPN to low RPN.

Risk R_2
Risk R_2 has the highest RPN, meaning it is the most urgent risk to control. If a user fills out incorrect values for experiment parameters, the NociTRACK AmbuStim can sent out wrong or dangerous stimuli, which can have critical consequences.
This risk can be prevented by making changes to the design by integrating maximum allowed values in the script for the specific entry-widgets that can cause dangers. This risk can be completed neglected then, because the script will raise an error if someone enters a value higher than the maximum allowed value. Provided this is added correctly, the addition of these maximum allowed values to the system will not entail new risks.

Risk R_1
The RPN of Risk R_1 is rather high and therefor also has a high priority to be solved. If the user presses certain buttons in a wrong order, some crucial steps in the experiment process can be skipped. This would mean that certain values are not changed in the config-file. It is still unknown what the precise effect is of this phenomenon.
To prevent any risks, protective measures can be added to the design. A solution would be that the buttons should only be enabled if the required previous steps are executed correctly. Adding this to the design will not entail new risks.

Risk R_5
Risk R_5 has a quite high occurrence, because it is likely that a patient gets uncomfortable whilst experiencing nociceptive stimulation. However, there is only a "Stop"-button available in the last window, which means that the user would first have to navigate to that screen before the user is able to stop the application. In the meantime, the patient is receiving unwanted stimuli.
This risk can be easily prevented by making a change to the design, namely integrating a "Stop"-button in EACH window. Adding extra buttons to the remaining windows will not entail new safety risks. However, if an experiment is stopped at a random moment during execution it might raise errors in the data, which might entail new kinds risks, regarding the data and the data-analysis. The user should be informed of these potential risks prior to experiment execution.

Risk R_4
Risk R_4 states that it is possible, however improbable, that the application fails at some point during execution. When this happens, the user no longer has control over the experiment, which can cause dangerous/unwanted stimulation.
Although it is improbable that the application fails, it is important to perform risk control, because it can cause dangerous situations. Unfortunately, there is no possibility to prevent this risk by changing the design or adding new, protective components. A solution might be to let the device automatically shut down when this happens, but then changes to the device should be made. However, it is possible to inform the user and the patient prior to experiment execution about what to do if such situation occurs. In addition to that, a component should be added to the software that shows in the GUI if something like this occurs. Implementing safety information prior to experiment execution and a safety warning, should not entail new risks. However, it is possible that the user does not read the safety information or does not understand it.
Besides, if this happens, it would mean there is an error in the software which has to be fixed. Fixing a error in the software would be very time-consuming.

Risk R_6
Even though, the consequences of Risk R_6 are harmless, one would still want to prevent this, because this would result in unnecessary loss of time and maybe wrong type of data.
This risk can be prevented by making changes to the design, by integrating a "type-check" for each entry-widget. The risk would then only raise an error during the experiment, so the user is able to imme-

diately correct the mistake. This would prevent the need to execute the experiment again if one would find out a mistake was made in the output-data after the experiment is done. Adding these maximum allowed values to the system will not entail new risks.

Risk R_7
If a user fills in values that are too high or too low for the NociTRACK AmbuStim to process, this would be harmless for the patient (or unpleasant at most), since the stimuli that will be sent out then, have the maximum or minimum achievable value of the device. However, this would probably also raise an error, which might causes loss of data and/or time.
This risk can be prevented by making changes to the design, by integrating maximum allowed values in the script for the specific entry-widgets. This is also a way of risk control for Risk R_2, so these two solutions can be integrated nicely for an optimal design. Adding these maximum allowed values to the system will not entail new risks.

Risk R_3
If the Application(tk.Tk) class fails, the user can no longer navigate through the windows, but also the ControllerAmbuStim() class object is no longer accessible from other classes. Because that object can no longer be accessed, the NociTRACK AmbuStim also is not able to sent out stimuli, so the consequence of this risk is harmless.
Despite the risk being harmless if occurring, the same goes for this risk as for Risk R_6 and Risk R_7, if this would happen, the experiment has to be executed again, which means loss of time. There is no possibility to prevent this risk by changing the design or adding new, protective components. However, it is possible to inform the user and the patient prior to experiment execution about what to do if such situation occurs. Implementing safety information prior to experiment execution should not entail new risks.
Besides, if this happens, it would mean there is an error in the software which has to be fixed. Fixing a error in the software would be very time-consuming.

Chapter 6

Discussion

The aim of this research is to develop an initial design for a Python-based GUI for a medical measurement system for nociceptive (mal)function. This new system should allow non-expert users to conduct experiments correctly following the NDT-EP method and it should also allow developers to more easily make adaptations and improvements to the software. This research indicates that using Tkinter for a GUI, asyncio for asynchronous concurrent programming and classes for OOP, make this possible. However, developing software in this way takes a lot of time and asks for proper maintenance, in order to prevent (unforeseen) problems in the future. It has also not been investigated yet if the realised prototype is actually usable by the target-audience (non-expert users). For further development, the target audience should be involved to realise the user-friendly GUI as envisioned.
Besides this general remark, some more specific points have emerged during this research. These points will be appointed and discussed below.

6.1 Implementation of (Additional) Elements

Most of the requirements for the new system and functional components of the current system, as stated in Chapter 2.4.3, are implemented in the new system. However, as this research covers an initial development of a new system, several recommendations can be made regarding requirements not met and desired features not yet implemented.

6.1.1 EEG-signals and Impedance

The current system lets the user measure EEG-signals and check the impedance of the channels of the EEG-cap to ensure proper experiment preparation. These components were not yet implemented in the design proposed in this research. Implementing these components would require a separate controller class for the device used to measure the EEG-signals. Because of time-limitations, it was chosen to leave this out of the scope of this assignment. However, given the set-up of the proposed design, one can later add a controller for measuring EEG, and thus impedance, to the new system. This added class would be of the same structure as the ControllerAmbuStim() class and would also be integrated into the whole set-up the same way.

6.1.2 MTT

For the first prototype of this initial design, only a config-file to detect a single threshold was implemented. Introducing the MTT procedure might raise some difficulties because the current controller of the NociTRACK AmbuStim contains only one dictionary to store values of parameters for a single threshold from the chosen config-file.
Abstract classes were implemented in the design for the new system for the ControllerAmbuStim() class to make implementation of other experiments, including the MTT procedure, possible. However, to

actually implement the MTT procedure (or other experiments), some changes to the current Controller-AmbuStim() class are required. By using an abstract class (as in the design of this research), it might be possible to design the abstract class in such way that multiple config-dictionaries (within one bigger dictionary) or additional settings can be created and invoked to allow for performing the MTT procedure or other experiments that require multiple thresholds or additional settings.

6.1.3 Config-file

During this research it was achieved in the prototype to get a log-file as output. However, an important requirement is that also an output file was of type .ini (config-file) and this was not yet implemented in the current prototype. Yet the design of the architecture of the new system is such that this should be possible.
All the data needed for creating the config-file as output is already present in the current output of the new system (this can be seen from the log-file that was already managed to implement in the design). By this, it can be concluded that it should be possible to reorganise this information into a config-file as well as a log-file. The config-dictionary should then update after each stimulus, if so it is be possible to extract the data from this dictionary and comparse it back to the config-file.
Besides, implementation of a config-file as output should be done in such way that the data is stored continuously, to prevent loosing data if the system might crashes.

6.2 Asyncio combined with a GUI (tkinter)

After the prototype was realised, a unforeseen problem arose. Once the "Start"-button was clicked, the asyncio function runs the experiment by executing all asyncio tasks in a specific routine. However, while executing the asyncio tasks, the GUI completely freezes and no buttons can be clicked or other interactions with the GUI are possible.
This raises a major issue, as it would first of all mean that the GUI can not show real-time data, like graphs, bluetooth status etc. when asyncio tasks are executed, which would mean that several (design-) requirements are not met. In addition and even bigger problem, safety-wise, is that this would mean that if the patient would get unwanted/unexpected/dangerous stimuli while the GUI is frozen, the user is not in any way able to stop the stimulation, which can have serious implications for the patient. It can be concluded from this, that the issue must be solved in further development for ensuring a safe measurement system.
There are several solutions available to solve this problem. First of all, an asyncio function can be created that updates the GUI and its components, by conveniently using the asyncio function *.sleep()*. An asyncio task can then be added that within an interval performs the function that temporary pauses the current asyncio tasks with *.sleep()* and meanwhile executes the other tasks (GUI). [24]
Another solution that it worth looking into, is the use of multi-threading programming instead of asynchronous programming for concurrent programming. Multi-threading differs from asynchronous programming as it actually runs multiple threads side by side instead of only giving the illusion, like asynchronous programming (by running specific tasks alternately for a specific amount of time). In Python the module *threading* is available to implement multi-threading. Using multi-threading correctly, will prevent the issue discussed before as the GUI can run separately from the other tasks. One downside, implementing multi-threading would increase the RAM needed to run the software and the software is most likely to become slower if the RAM is full. [25]

6.3 MDR and Risk analysis

6.3.1 Risk Analysis

As already stated in Chapter 2.2 Medical Device Regulations (MDR), for the design in this research, only a very basic risk analysis was performed, that is far from complete from a full risk analysis as stated in the MDR art. 82 and is only applicable for the device being used for scientific research (with a specific user and patient). This was due to time-limitations.
It was not yet determined what the level of acceptance is for the risk evaluation matrix. This should be done by an expert (e.g. a specialized doctor or researcher), to determine which potential risks are acceptable and which should be controlled to ensure safe use of the software. This was left out of the scope of this research, also because of time-limitations.
The risk analysis was merely performed to demonstrate that software entails a lot of risks, that are to a large extent different from the ones from the connected medical device as well as to demonstrate that the MDR also apply to software for medical devices and that is should therefore be implemented in further development.
Another aim was to show that not only the entailed risks are different, but also to address the biggest potential problems with the theoretical design, when it would be turned into a first complete prototype. The potential risks were therefore identified and analysed and recommendations are given on how to reduce these risks for further development of the system.

6.3.2 Testing

It is important to notice that this is an initial design of an architecture for software for a new system, therefore no testing was performed. The design is only theoretical, thus there is not yet a completely functional prototype that can be used for testing. In this research a small part of the architecture is programmed, with the sole purpose of showing that the theoretical design of a software architecture, as created in this research, can be turned into functional software for a new system.
Since one needs all components of the design completed to perform an experiment, testing for safety and function is not possible until the first complete prototype is realised. However, it might be possible to test separate parts of the experiment by using a mock-up.

6.3.3 Patients

The following strongly correlates with the previous. The subject for the risk analysis of this research is defined as a patient (human subject), without taking particular vulnerabilities, due to disease, into consideration. However, the aim of the research at BSS, regarding the NDT-EP method, is to eventually use this software for measurements on patient with these vulnerabilities as well. Using the system for clinical testing on any type of patient is not yet possible anyway, because the software (of the new system) has not been tested yet and also no complete risk analysis is performed, as mentioned before. When this is done, the system might eventually be used for all types of patients.

Chapter 7

Conclusion

The aim of this bachelor-assignment was to develop a first Python-based prototype of a GUI that can be used for laboratory experiments on human subjects for performing the NDT-EP method and also to explore the MDR guidelines for medical software development. The design assignment of this research is stated as follows:

Initial development of a Python-based Graphical User Interface for a Medical measurement system for Nociceptive (mal)Function.

The resulting design and design-choices were explained in this report and a first prototype was realised. The final design provides a framework and a first prototype for a Python-based GUI (Tkinter), that allows non-expert users to conduct experiments on patients, using the NDT-EP method. The GUI is straightforward for non-expert users to navigate through and the overall set-up by using classes allows for easy adaptations and additions by software engineers.
From this, it can be concluded that the final design realised with this bachelor-assignment provides a good foundation for further development of a new Python-based system for measuring nociceptive (mal)function to eventually replace the current Labview-based system.

As this research provides an initial development of a GUI for a measurement system and a first Python-based prototype, there are several points that should be improved in further development.
First of all, the device to measure EEG-signals should be implemented in the design and also there should be a widget in the GUI that lets users check the impedance of channels of an EEG-cap. The application of the MTT method and a config-file as output should be implemented into the design as well. There should also be performed a more extensive risk analysis in further development according to the MDR art. 82. In addition, the risk control measures, as proposed in Chapter 5.1.3, should be implemented in a next design. Lastly, the most important recommendation for further development and new prototypes is to solve the issue that arises when combining asynchronous concurrent programming (asyncio) and a GUI (Tkinter), to prevent the GUI from freezing and to avoid dangerous situations.

References

[1] H. Breivik, B. Collett, V. Ventafridda, R. Cohen, and D. Gallacher, "Survey of chronic pain in Europe: Prevalence, impact on daily life, and treatment," *Eur J Pain*, vol. 10, no. 4, pp. 287–333, 2006. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1016/j.ejpain.2005.06.009

[2] S. M. Rikard, A. E. Strahan, K. M. Schmit, and G. P. Guy, "Chronic Pain Among Adults - United States, 2019-2021," *MMWR. Morb Mortal Wkly Rep*, vol. 72, no. 15, pp. 379–385, 2023. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/37053114/

[3] A. Fayaz, P. Croft, R. M. Langford, L. J. Donaldson, and G. T. Jones, "Prevalence of chronic pain in the UK: a systematic review and meta-analysis of population studies," *BMJ Open*, vol. 6, no. 6, pp. 1–12, 2016. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/27324708/

[4] R. D. Treede, W. Rief, A. Barke, Q. Aziz, M. I. Bennett, R. Benoliel, M. Cohen, S. Evers, N. B. Finnerup, M. B. First, M. A. Giamberardino, S. Kaasa, E. Kosek, P. Lavand'homme, M. Nicholas, S. Perrot, J. Scholz, S. Schug, B. H. Smith, P. Svensson, J. W. Vlaeyen, and S. J. Wang, "A classification of chronic pain for ICD-11," *Pain*, vol. 156, no. 6, pp. 1003–1007, 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4450869/

[5] S. P. Cohen, L. Vase, and W. M. Hooten, "Chronic pain: an update on burden, best practices, and new advances," *Lancet*, vol. 397, no. 10289, pp. 2082–2097, 2021. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/34062143/

[6] "Terminology of Pain| International Association for the Study of Pain." [Online]. Available: https://www.iasp-pain.org/resources/terminology/#pain

[7] M. Fei Yam, Y. Chun Loh, C. Shan Tan, S. Khadijah Adam, N. Abdul Manan, and R. Basir, "General Pathways of Pain Sensation and the Major Neurotransmitters Involved in Pain Regulation," *Int J Mol Sci.*, vol. 19, no. 8, p. 2146, 2018.

[8] C. J. Woolf, "Review series introduction What is this thing called pain?" *J Clin Invest*, vol. 120, no. 11, pp. 3742–4, 2010. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/21041955/

[9] V. N. Nikolenko, E. M. Shelomentseva, M. M. Tsvetkova, E. I. Abdeeva, D. B. Giller, J. V. Babayeva, E. E. Achkasov, L. V. Gavryushova, and M. Y. Sinelnikov, "Nociceptors: Their Role in Body's Defenses, Tissue Specific Variations and Anatomical Update," *J Pain Res*, vol. 15, pp. 867–877, 2022.

[10] D. J. Clauw, M. Noyes Essex, V. Pitman, and K. D. Jones, "Reframing chronic pain as a disease, not a symptom: rationale and implications for pain management," *Postgrad Med.*, vol. 131, no. 3, pp. 185–198, 2019.

[11] K. A. Mifflin, B. J. Kerr, K. A. Mifflin, and Á. B. J. Kerr, "The transition from acute to chronic pain: understanding how different biological systems interact," *Can J Anaesth*, vol. 61, no. 2, pp. 112–22, 2014. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/24277113/

[12] D. B. Reichling and J. D. Levine, "Critical role of nociceptor plasticity in chronic pain," *Trends Neurosci2*, vol. 32, no. 12, pp. 611–8, 2009. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/19781793/

[13] D. E. Henry, A. E. Chiodo, and W. Yang, "Central nervous system reorganization in a variety of chronic pain states: A review," *PM R*, vol. 3, no. 12, pp. 1116–1125, dec 2011. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/22192321/

[14] T. Landmark, O. Dale, P. Romundstad, A. Woodhouse, S. Kaasa, and P. C. Borchgrevink, "Development and course of chronic pain over 4 years in the general population: The HUNT pain study," *Eur J*, vol. 22, no. 9, pp. 1606–1616, oct 2018.

[15] S. E. E. Mills, K. P. Nicolson, and B. H. Smith, "Chronic pain: a review of its epidemiology and associated factors in population-based studies," *Br J Anaesth*, vol. 123, no. 2, pp. 273–283, 2019. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/31079836/

[16] N. Jansen, R. Dollen, B. Van Den Berg, T. Berfelo, I. P. Krabbenbos, and J. R. Buitenweg, "Combined Evaluation of Nociceptive Detection Thresholds and Evoked Potentials during Conditioned Pain Modulation: A Feasibility Study," *Annu Int Conf IEEE Eng Med Biol Soc*, vol. 2021, pp. 1427–1430, 2021. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/34891553/

[17] B. van den Berg, "BSS-NSP-M002 MTT-EP Experiments," University of Twente, Enschede, pp. 1–17, 2017.

[18] R. J. Doll, P. H. Veltink, and J. R. Buitenweg, "Observation of time-dependent psychophysical functions and accounting for threshold drifts," *Atten Percept Psychophys*, vol. 77, no. 4, pp. 1440–7, 2015. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/25810158/

[19] Publicatieblad van de Europese Unie, "Medical Device Regulations (NL) - Verordening (EU) 2017/745 van het Europees parlement en de raad." 2017.

[20] NSP team of BSS University of Twente, "Risk Analysis AmbuStim ISO14971," University of Twente, Enschede, pp. 1–15, 2017.

[21] Python Software Foundation, "Classes — Python 3.12.3 documentation," 2024. [Online]. Available: https://docs.python.org/3/tutorial/classes.html

[22] Python software Foundation, "abc — Abstract Base Classes — Python 3.12.3 documentation," 2024. [Online]. Available: https://docs.python.org/3/library/abc.html

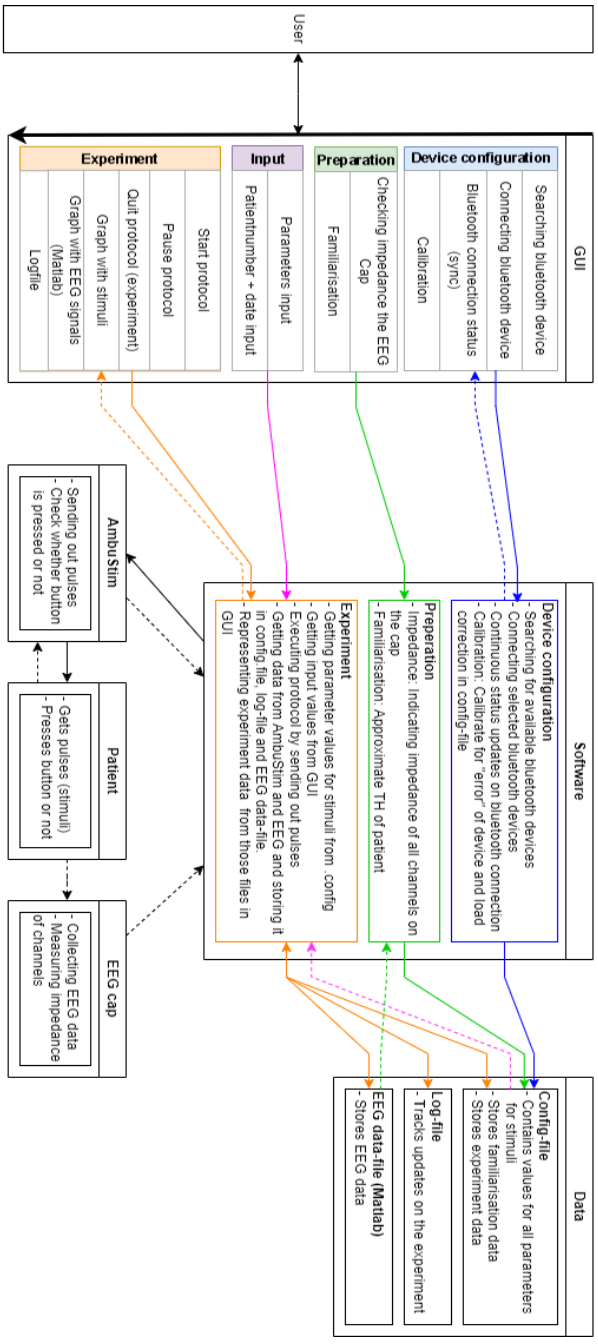
[23] Python Software Foundation, "tkinter — Python interface to Tcl/Tk — Python 3.12.3 documentation," 2024. [Online]. Available: https://docs.python.org/3/library/tkinter.html

[24] Python software Foundation, "asyncio — Asynchronous I/O — Python 3.12.3 documentation," 2024. [Online]. Available: https://docs.python.org/3/library/asyncio.html

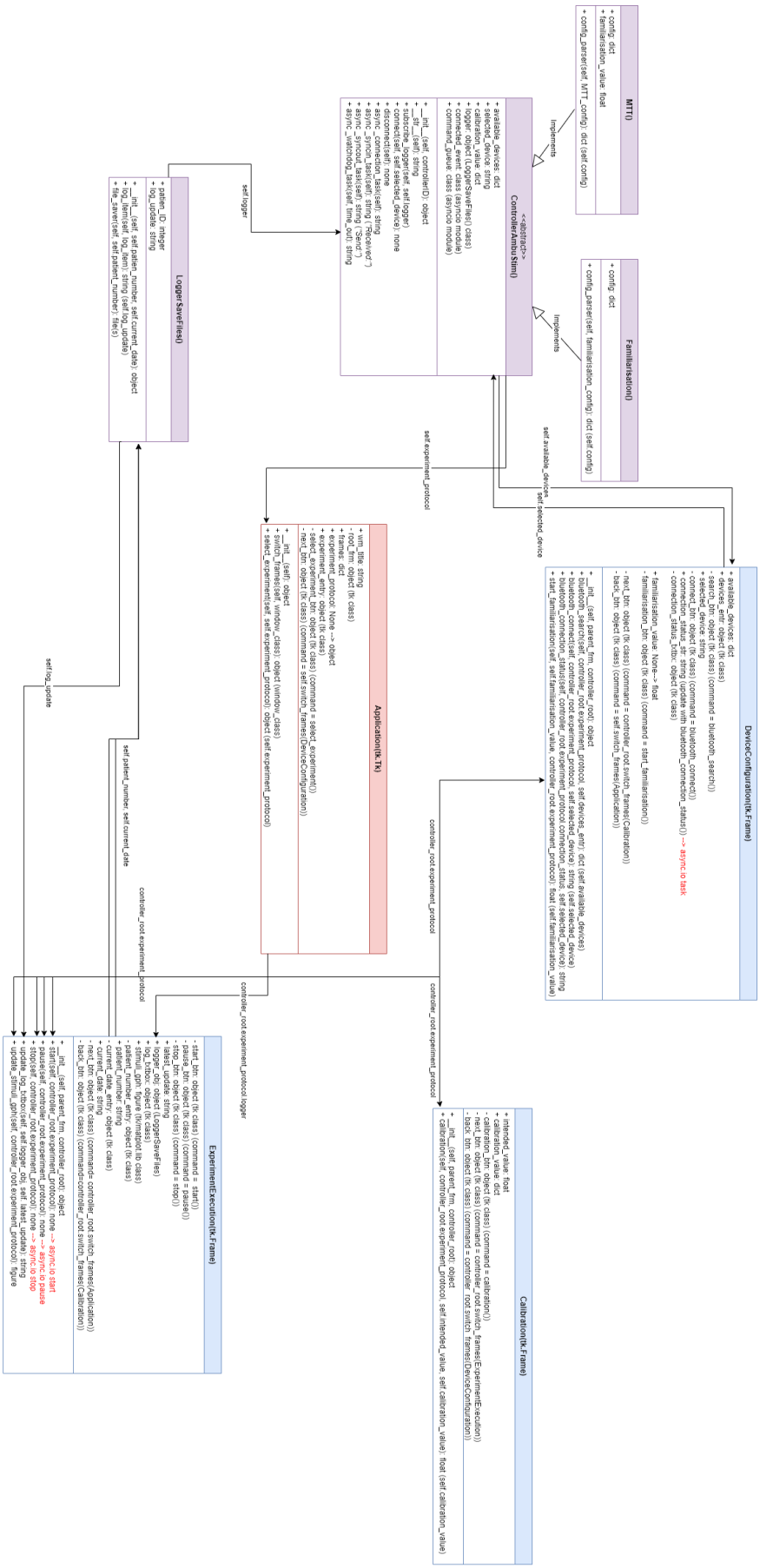
[25] Python Software Foundation, "threading — Thread-based parallelism — Python 3.12.3 documentation," 2024. [Online]. Available: https://docs.python.org/3/library/threading.html#module-threading

Appendix

7.1 Flowchart current system (Labview)



7.2 Class Diagram new system (Python)



7.3 Proof of Principle script

7.3.1 Current Controller/Logger Script

```
import socket
import asyncio
import time


# Dictionary with NociTRACK device and there corresponding mac-adres
StimCom2DeviceBTaddress = {"NociTRACK-017" : "00:01:95:0C:BF:BA",
                           "NociTRACK-025" : "00:01:95:0C:6A:A0",
                           "NociTRACK-028" : "00:01:95:0C:C2:5F",
                           "NociTRACK-029" : "00:01:95:0C:6A:9B"}


class StimCom2Config:

    def __init__(self):
        self.config = {                                             # self.config is dictionary with parameters and corresponding values for sending out stimuli
            "V": {"Value": [0,0,0], "Sync": 0, "Ready": 1},
            "F": {"Value": [0,0,0,0], "Sync": 0, "Ready": 1},       # Parameter is no longer used
            "I": {"Value": [0], "Sync": 0, "Ready": 1},
            "P": {"Value": [1], "Sync": 0, "Ready": 1},
            "A": {"Value": [0], "Sync": 0, "Ready": 1},
            "a": {"Value": [0], "Sync": 0, "Ready": 1},
            "W": {"Value": [3], "Sync": 0, "Ready": 1},             # define PW_MIN=3 as a StimCom2 constant!
            "w": {"Value": [3], "Sync": 0, "Ready": 1},
            "C": {"Value": [1,1,1], "Sync": 0, "Ready": 1},         # Pos and Neg phase of channel 1 are enabled -> should be changed in embedded code!
            "M": {"Value": [0,1], "Sync": 0, "Ready": 1},
            "S": {"Value": [0,0,0], "Sync": 1, "Ready": 1},         # Parameter is no longer used
            "Q": {"Value": [0,0,0,0], "Sync": 1, "Ready": 1},       # Parameter is no longer used
            "R": {"Value": [0,0,0], "Sync": 0, "Ready": 1},
            "D": {"Value": "", "Sync": 1, "Ready": 1}
            }
        self.reset_config=self.config

    def __str__(self):                                              # Printing string for device configuration
        str1 = "\n-----------------------------\nStimCom2 configuration:"
        for setting,status in self.config.items():
            str1 = str1 + f"\n   {setting}: "

            if isinstance(status.get("Value"),str):
                str1 = str1 + status.get("Value")
            else:
                for e in status.get("Value"):
                    str1 = str1 + f"{e}, "
        str1 = str1 + "\n-----------------------------\n"
        return str1

    def reset(self):
        print("\nResetting StimCom2 config\n")                       # Resetting to resit_config
        self.config=self.reset_config

```

```
51
52      def set_value(self,Setting,Value,Ready):                    # Function for
    changing values of setting of config dictionary
53          print("\n-----------------------------")
54          print(f"Setting {Setting}")
55          self.config[Setting]["Value"]=Value
56          self.config[Setting]["Sync"]=0
57          self.config[Setting]["Ready"]=Ready
58          print("-----------------------------")
59
60      def check(self):
61          pass
62
63  class StimCom2Logger:                                           # Class to
    create and update log-file
64      def __init__(self,LoggerID):
65          self.LoggerID=LoggerID
66          self.LoggingObject="None"
67          print(f"\n[{time.strftime('%X')}] StimCom2 Logger started")
68
69      def __str__(self):
70          str = f"\nStimCom2Logger ID: {self.LoggerID} "
71          str = str + f"\nLogging Object: {self.LoggingObject}"
72          return str
73
74      def subscribe(self, LoggingObject):
75          self.LoggingObject=LoggingObject
76          print(f"\nStimComLogger {self.LoggerID}: Incoming subscription from {
    self.LoggingObject}")
77
78      def log(self,log_item):
79          print(f"log_item: {log_item}")
80
81  class StimCom2Controller:
82      def __init__(self,ControllerID):
83          self.Config=StimCom2Config()
84          self.ControllerID=ControllerID
85          self.Logger="None"
86          self.DeviceID="None"
87          self.connected_event = asyncio.Event()                  # object:
    asyncio event can be used to notify multiple asyncio tasks that some event
    has happened.
88          self.command_queue = asyncio.Queue()                    # object:
    FIFO queue
89
90      def __str__(self):                                          # Printing
    Controller ID and Device ID
91          str = f"\nStimCom2Controller ID: {self.ControllerID} "
92          str = str + f"\nDevice ID: {self.DeviceID}"
93          str = str + self.Config.__str__()
94          return str
95
96      def subscribe_logger(self,Logger):                          # Object of
     Logger is made
97          self.Logger=Logger
98          self.Logger.subscribe(self.ControllerID)                # Function
    to initiate log
99
100
101     def connect(self,DeviceID):
102         # if connected_condition == false: initiate connection_task else error
    (or first disconnect?)
103         self.DeviceID=DeviceID
```

```
104         self.Logger.log(f"Connect to {self.DeviceID}")              # Function
    to log a log-item
105 
106         # setup connection socket (should succeed, therefore use try...)
107         try:
108             self.client = socket.socket(socket.AF_BLUETOOTH, socket.SOCK_STREAM
    , socket.BTPROTO_RFCOMM)
109             self.client.settimeout(5)
110             self.client.connect( (self.DeviceID, 1))
111         except Exception as e:
112              print(e)
113         else:
114             # set connection_event=true (only if socket setup is successful!)
115             self.connected_event.set()                                  # asyncio.
    Event().set() --> True notifies asyncio tasks that something has happened
116             self.c_task=asyncio.create_task(self._connection_task())
117 
118     def disconnect(self):
119         # set connection_condition=false
120         print("Disconnect")
121         self.connected_event.clear()                                    # asyncio.
    Event().clear() --> False notifies asyncio tasks that nothing has happened
    yet (no event (treshold/condition))
122 
123     async def _connection_task(self):
124 
125         print("Connection task started")
126 
127         # start syncin_task
128         self.si_task = asyncio.create_task(self._syncin_task())
129 
130         # start syncout_task
131         self.so_task = asyncio.create_task(self._syncout_task())
132 
133         # add sleep task to allow syncin and syncout task to become active
134         await asyncio.sleep(0)
135 
136         # monitor connection using keep-alive action until connection lost or
    terminated (by desync of R-command)
137         while self.connected_event.is_set():
138             await self.command_queue.join()
139             #self.Config.set_value("R",[0,0,0],1)
140             self.Config.config["R"]["Sync"]=0
141 
142             await asyncio.sleep(2)
143 
144         # stop syncin_task
145         # stop syncout_task
146 
147         # close socket
148         self.client.close()
149         print("Connection task ended")
150 
151     async def _syncin_task(self):
152         # while connection_condition=true: read data from socket, append to
    command_buffer, parse full commands to update StimCom2Config (use
    command_lock)
153         print("SyncIn task started")
154 
155 
156         #  define or clear the command_buffer
157         self.command_buffer= ""
158 
```

```
159         # adust timeout of socket for quick reading
160         self.client.settimeout(0.1)
161 
162         # repeat during connection
163         while self.connected_event.is_set():
164 
165             # read available incoming data
166             try:
167                 incoming_data = self.client.recv(1024)
168             except:
169                 pass
170             else:
171                 self.command_buffer = self.command_buffer + incoming_data.
    decode('utf-8')
172 
173             # check command_buffer for complete commands and if found, parse
    the first
174             if "\x00" in self.command_buffer:
175 
176                 # find end of first command
177                 i=self.command_buffer.find("\x00")
178 
179                 # extract and remove first command from buffer
180                 command = self.command_buffer[0:i]
181                 self.command_buffer = self.command_buffer[i+1:]
182 
183                 # parse command
184                 if command[0]=="!":
185                     print("\n++++++++++++++++++++++++\n  NotAck received!\n
    ++++++++++++++++++++++++")
186                     self.connected_event.clear()
187                     break
188                 elif isinstance(self.Config.config[command[0]]["Value"],str):
189                     self.Config.config[command[0]]["Value"]=command[2:]
190                 else:
191                     self.Config.config[command[0]]["Value"]=[int(x) for x in
    command[2:].split(",")]
192                     self.Config.config[command[0]]["Sync"]=self.Config.config[
    command[0]]["Sync"]+1
193 
194                 self.Logger.log(f"Received: {command}")
195 
196 
197             # add sleep task for coptimal concurrent scheduling (otherwise
    while loop will block other tasks)
198             await asyncio.sleep(0)
199         print("SyncIn task ended")
200 
201 
202     async def _syncout_task(self):
203         print("SyncOut task started")
204 
205         # while StimComConfig has unsynced items: get first unsynced setting,
    format command, submit command, wait for sync, if timeout connection lost...
     if desired reconnect (with S,0,0,0, synced)
206         while self.connected_event.is_set():
207 
208             # Collect unsynced settings in a queue
209             for setting,status in self.Config.config.items():
210                 if status["Sync"]==0 and isinstance(status["Value"],list):
211                     # generate command
212                     command=f"{setting},"+",".join(str(n) for n in status["
    Value"])
```

```
                    command=command+"\0"
                    ready=status['Ready']

                    # put in queue
                    self.command_queue.put_nowait( (command,ready) )

            while self.command_queue.qsize()>0:
                # get command from queue
                command,ready = self.command_queue.get_nowait()

                # send command
                self.client.send(command.encode('utf-8'))
                self.Logger.log(f"Send: {command}")

                # start watchdog
                wd_task = asyncio.create_task(self._watchdog_task(2))

                # wait for sync=ready, if sync=1 stop watchdog, if sync=ready report task_done()
                old_sync=0
                while old_sync!=ready:
                    new_sync=self.Config.config[command[0]]["Sync"]
                    if old_sync==0 and new_sync==1:
                        wd_task.cancel()
                        old_sync=1
                    if new_sync==ready:
                        self.command_queue.task_done()
                        old_sync=ready
                    await asyncio.sleep(0)
            # add sleep task for coptimal concurrent scheduling (otherwise while loop will block other tasks)
            await asyncio.sleep(0)
        print("SyncOut task ended")

    async def _watchdog_task(self,timeout):
        await asyncio.sleep(timeout)
        print("\n++++++++++++++++++++++\n    Watchdog Timeout!\n++++++++++++++++++++++")
        self.connected_event.clear()
```

Listing 7.1: Current Controller/Logger Script. Original script by Prof. Dr. ir. J.R. Buitenweg (BSS, University of Twente), adapted by M.E.L. Janssen (BMT-B student, University of Twente).

7.3.2 Proof of Principle Script GUI Application

```
import tkinter as tk
from tkinter import ttk

import configparser as cp
import sys
import datetime
import time

from controller_v2 import *

class Root(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self.wm_title("Experiment Application")     # title for root window
        self.resizable(True, True)                  # resizable: yes or no
        self.geometry("300x300")                    # start size of window

```

```
18          self.rowconfigure(0, weight=1)
19          self.columnconfigure(0, weight=1)
20
21          root_frm = tk.Frame(self)                    # Adding a root frame to
    the root window and "filling" it with widgets
22          root_frm.grid(row=0, column=0, sticky="n")
23
24          root_frm.rowconfigure(0, weight=1)
25          root_frm.columnconfigure(0, weight=1)
26
27          main_frm = tk.Frame(root_frm, width=600, height=600)
28          main_frm.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
29
30          main_lbl1 = ttk.Label(main_frm, text="AmbuStim2 Application")
31          main_lbl1.grid(row=0, column=0, columnspan=2)
32
33          main_lbl2 = ttk.Label(main_frm, text="Choose experiment:")
34          main_lbl2.grid(row=1, column=0)
35
36          main_entry = ttk.Entry(main_frm)
37          main_entry.grid(row=1, column=1)
38
39          main_lbl3 = ttk.Label(main_frm, text="Patient Number:")
40          main_lbl3.grid(row=2, column=0)
41
42          patient_entry = ttk.Entry(main_frm)
43          patient_entry.grid(row=2, column=1)
44
45          self.main_entry_output = None                # Name of chosen config
    file will be saved to this variable
46          self.patient_entry = None                    # Variable to save
    patientnumber to
47          self.experiment_protocol = StimCom2Controller("C1")     # Variable that
     contains the object of the controller
48
49          save_btn = ttk.Button(main_frm, text="Save", command=lambda: self.
    save_experiment(main_frm, main_entry, patient_entry))
50          save_btn.grid(row=3, column=0)
51
52          main_btn = ttk.Button(main_frm, text="Next", state="disabled", command=
    lambda: self.show_frame(DeviceConfiguration))
53          main_btn.grid(row=3, column=1)
54
55          self.frames = {}
56
57          for F in (DeviceConfiguration, Calibration, ExperimentExecution):   #
    Creating a dict for all Classes and their object for switch_frames function
58              frame = F(root_frm, self)
59              self.frames[F] = frame
60              frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
61
62          main_frm.tkraise()
63
64      def save_experiment(self, main_frm, input_var1, input_var2):     # Only
    works for single threshold determination(familiarisation)
65          print(self.experiment_protocol.Config.config)   # Check for developers
    to see if the correct dict is used
66
67          self.main_entry_output = input_var1.get()
68          self.patient_entry = input_var2.get()
69
70          config = cp.ConfigParser()
71          config.read(self.main_entry_output)
```

```
72
73          settings_list = []
74          config_dict = {}
75          for key in config['Threshold 1']:                # Creating a dict with
    al settings and corresponding values
76              settings_list.append(key)
77
78              for setting in settings_list:
79                  item = config['Threshold 1'][setting]
80
81                  if item == '':                           # To make sure the
    values are of the correct type
82                      value_list_stripped = item
83                  else:
84                      try:
85                          value_list_stripped = int(item)
86                      except: # if not an integer --> makes it a float
87                          value_list_stripped = float(item)
88
89                  config_dict[setting] = value_list_stripped      # Dict with key
     = setting-name and value = setting-value
90
91          nop = config_dict['nop']    # variable (int) for Number of Pulses (NoP)
92
93          # Putting values of config.ini file into config_dict of controller
    object
94          self.experiment_protocol.Config.config["V"]["Value"] = [config_dict['
    selectionmethod']]
95          self.experiment_protocol.Config.config["I"]["Value"] = [(config_dict['
    ipi'])/(0.035) for x in range(nop)]
96          self.experiment_protocol.Config.config["P"]["Value"] = [(config_dict['
    channel']) for x in range(nop)]
97          self.experiment_protocol.Config.config["A"]["Value"] = [1 for x in
    range(nop)]
98          self.experiment_protocol.Config.config["a"]["Value"] = [0 for x in
    range(nop)]
99          self.experiment_protocol.Config.config["W"]["Value"] = [(config_dict['
    pw'])/(0.035) for x in range(nop)]
100         self.experiment_protocol.Config.config["w"]["Value"] = [0 for x in
    range(nop)]
101         self.experiment_protocol.Config.config["C"]["Value"] = [1 for x in
    range(nop)]
102         self.experiment_protocol.Config.config["M"]["Value"] = [1,1]
103         self.experiment_protocol.Config.config["R"]["Value"] = [0,0,0]
104
105         print(self.experiment_protocol.Config.config)            # Check for
    developers to see if the correct dict is used
106
107         input_var1 = ttk.Entry(main_frm, state="disabled")       # Overlays the
     entry widget and disables it after chosen experiment is saved
108         input_var1.grid(row=1, column=1)
109         input_var2 = ttk.Entry(main_frm, state="disabled")       # Overlays the
     entry widget and disables it after chosen experiment is saved
110         input_var2.grid(row=2, column=1)
111         main_btn = ttk.Button(main_frm, text="Next", command=lambda: self.
    show_frame(DeviceConfiguration))  # Enables "next"-button
112         main_btn.grid(row=3, column=1)
113
114         return
115
116     def show_frame(self, frame_class):      # frame_class is de class of the
    frame that needs to be displayed after performing function
117         frame = self.frames[frame_class]
```

```
118         frame.tkraise()
119
120
121
122 class DeviceConfiguration(tk.Frame):        # Frame for DeviceConfiguration
123     def __init__(self, parent_frm, root_controller):
124         tk.Frame.__init__(self, parent_frm)
125
126         self.rowconfigure(0, weight=1)
127         self.columnconfigure(0, weight=1)
128
129         lbl_1 = ttk.Label(self, text="Device Configuration")
130         lbl_1.grid(row=0, column=0)
131
132         next_btn = ttk.Button(self, text="Next", command=lambda:
    root_controller.show_frame(Calibration))
133         next_btn.grid(row=1, column=0)
134
135 class Calibration(tk.Frame):                # Frame for Calibration
136     def __init__(self, parent_frm, root_controller):
137         tk.Frame.__init__(self, parent_frm)
138
139         self.rowconfigure(0, weight=1)
140         self.columnconfigure(0, weight=1)
141
142         lbl_1 = ttk.Label(self, text="Calibration")
143         lbl_1.grid(row=0, column=0, columnspan=2)
144
145         back_btn = ttk.Button(self, text="Back", command=lambda:
    root_controller.show_frame(DeviceConfiguration))
146         back_btn.grid(row=1, column=0, sticky='w')
147
148         next_btn = ttk.Button(self, text="Next", command=lambda:
    root_controller.show_frame(ExperimentExecution))
149         next_btn.grid(row=1, column=0, sticky='e')
150
151 class ExperimentExecution(tk.Frame):        # Frame for ExperimentExecution
152     def __init__(self, parent_frm, root_controller):
153         tk.Frame.__init__(self, parent_frm)
154
155         self.rowconfigure(0, weight=1)
156         self.columnconfigure(0, weight=1)
157
158         lbl_1 = ttk.Label(self, text="Experiment Execution")
159         lbl_1.grid(row=0, column=0, columnspan=2)
160
161         start_btn = ttk.Button(self, text="Start", command=lambda: self.
    start_measurement(root_controller.experiment_protocol, root_controller))
162         start_btn.grid(row=1, column=0, columnspan=2)
163
164         back_btn = ttk.Button(self, text="Back", command=lambda:
    root_controller.show_frame(Calibration))
165         back_btn.grid(row=2, column=0, sticky='w')
166
167         quit_btn = ttk.Button(self, text="Quit", command=lambda:
    root_controller.destroy())
168         quit_btn.grid(row=2, column=1, sticky='e')
169
170     def start_measurement(self, root_controller_object, root_controller):   #
    Function that starts a measurement
171         datetime_info = datetime.datetime.now()
172         date_str = datetime_info.strftime("%B%d%Y")
173         time_str = datetime_info.strftime("%H%M%S")
```

```
174
175         old_std_out = sys.stdout                                                    #
    To have the command output be put into a log-file
176         log_file = open(f"{root_controller.patient_entry}_NDTEP_{date_str}_{
    time_str}.log","w")
177         sys.stdout = log_file
178
179         async def main():   # Async function is used for the actual experiment
    execution
180             print(f"\n\nMain started at {time.strftime('%X')}")
181             C1.connect(StimCom2DeviceBTaddress["NociTRACK-017"])
182             await asyncio.sleep(3)
183
184             await asyncio.sleep(3)
185             C1.disconnect()
186             await asyncio.sleep(3)
187
188         print(f"\nMain ended at {time.strftime('%X')}")
189
190         C1=root_controller_object   # Input for running the controller is the
    StimCom2Controller object made in the root class
191         L1=StimCom2Logger("L1")
192         C1.subscribe_logger(L1)
193
194         asyncio.run(main())
195
196         sys.stdout = old_std_out
197         log_file.close()
198
199         print("Measurement is finished.")
200         return
201
202
203
204
205 if __name__ == "__main__":
206     testObj = Root()
207     testObj.mainloop()
```

Listing 7.2: Script of first Python-based prototype of a GUI Application. The GUI consists of a main window to enter the desired config.ini-file and the patient number. With several buttons it is possible to navigate through a total of 4 windows. In the last window (in combination with the Controller/Logger script from Appendix 7.3.1) it is possible to start an experiment by clicking the "Start"-button.