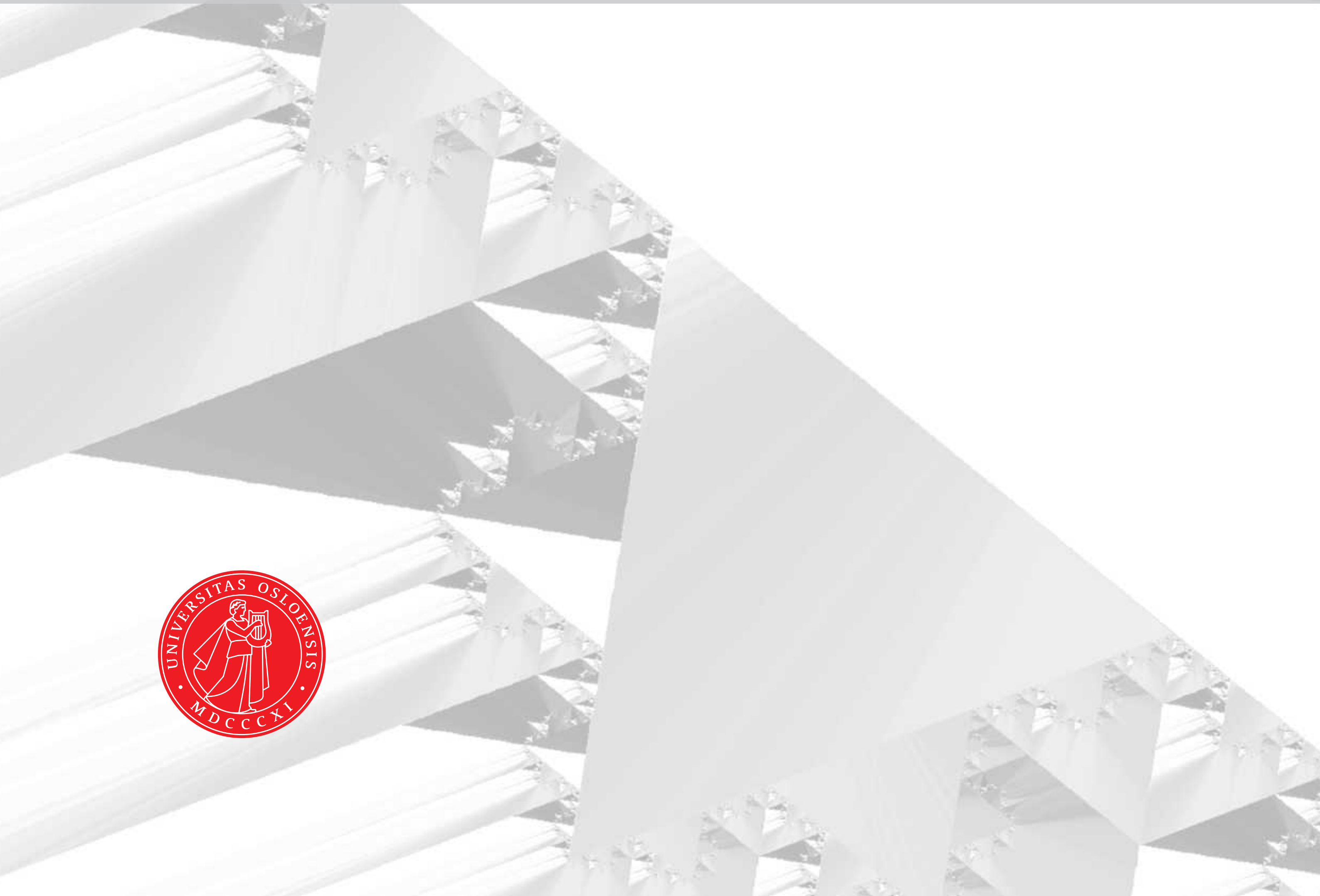
UiO **Department of Physics**
University of Oslo

Development of a Low-Cost Potentiostat with Cyclic Voltammetry and Amperometry Techniques Implemented

A Prototype Platform for Medical Applications using a Programmable System on Chip (PSoC)

Olav Bjerke
Thesis submitted for the degree of Master in Physics,
60 credits, Autumn 2020



Abstract

The Oslo Bioimpedance and Medical Technology Group at the Department of Physics (UiO) and the Department of Clinical and Biomedical Engineering (OUS) are involved in an EU-project named Training4CRM. The purpose of the project is to address gaps in Cell-based Regenerative Medicine (CRM) to treat neurodegenerative disorders, among others, Parkinson's disease. A potentiostat is needed to detect and characterize Dopamine in the project.

This thesis investigates the feasibility of making a prototype potentiostat on the PSoC 5LP by Cypress Semiconductors for Training4CRM. The device is small, portable, low-cost, and has extensive amounts of documentation. The firmware and software code needed to set up and control the potentiostat is provided and explained throughout the thesis.

The developed potentiostat was tested and verified as functioning, with some flaws considering noise. The noise issue was corrected and documented in the discussion, but the device lacks testing after the correction. The device was compared with another potentiostat developed on the same platform.

Contents

List of Figures

Acknowledgement

I would like to express my deepest gratitude to my supervisor Ørjan Grøttem Martinsen. You gave me the exciting opportunity to research and write about medical electronics, and you have given me lots of support throughout the work of this thesis.

A big thanks to my co-supervisors Christin Schuelke and André Cunha. They have both joined in on good discussions and taught me a lot each time. I would like to express extra gratitude to Christin Schuelke for taking some of her personal time to support me in the lab during measurements.

During my time at the University of Oslo, I have been a member of Realistforeningen, and I still am writing this. The association has taken a massive amount of my time during my degrees, but I do not regret it at all. This is also where I met my fiancee, and I would never undo that. Thank you for all the fun Realistforeningen.

I have met way too many people during my time at the university to show my gratitude to each one of them, but some of you that have made a special friendship with me. So to you, I would like to give my sincerest gratitude.

To my mom, my dad, and my sister, I would like to express how much it has meant for me that you have supported me through all of these years. There have been some bumps along the way, but you have always given me some motivational words to keep me going. This has meant the world to me. Thank you!

And finally, to my fiancee Jeanette. You have lifted me up, kept me strong, and inspired me to never give up. We have started a life together, and our family is growing as I write this ("Hi, Junior!") with Penny along, always happy to see us. To all of you, I love you. Thank you!

Olav Bjerke, October 2020, Algarheim

Chapter 1

Introduction

1.1 Background and Motivation

The Oslo Bioimpedance and Medical Technology Group at the Department of Physics, UiO, and the Department of Clinical and Biomedical Engineering, OUS, are involved in an EU-project named Training4CRM. The purpose of the project is to address gaps in Cell-based Regenerative Medicine (CRM) to treat neurodegenerative disorders, among others, Parkinson's disease. In the research on Parkinson's disease, the amount of dopamine in the brain is less than normal. Training4CRM is planning to develop a device that can measure the amount of dopamine and generate the needed amount with optogenetically modified human stem cells, and in the end, implement such a device in the human brain.

Dopamine is an electroactive neurotransmitter that can be analyzed with electroanalytical techniques such as amperometry and cyclic voltammetry (Bucher and Wightman, 2015). Amperometry is a technique that can measure the amount of substance in an analyte (David, 2013). Cyclic voltammetry is usually used to analyze redox processes and obtain the stability of reaction products (Elgrishi et al., 2018). To conduct these types of electroanalytical techniques, a potentiostat is needed. A potentiostat is a device used in electrochemistry to study the relation between electricity and chemical solutions (Elgrishi et al., 2018). Electron-transfer from one element to another generates the electricity and is called a redox (oxidation-reduction) reaction. With a potentiostat, it is possible to analyze a redox reaction and gather information about its electrochemical properties.

Potentiostats are commonly desktop versions and are expensive (Dryden and Wheeler, 2015). Today, with inexpensive microcontrollers and other electronics, several potentiostats are small and inexpensive; some also have wireless data transfer. The DStat by Dryden and Wheeler (2015) is a potentiostat developed from scratch, meaning that the entire potentiostat is produced on a produced PCBA (Printed Circuit Board Assembly). Dryden and Wheeler designed the schematic with relatively affordable components and have shared their design as open-source. They document measurements that are comparable with commercial potentiostats. Another potentiostat developed by Ainla et al. (2018) named UWED is based on a microcontroller with an RFDUINO Bluetooth adapter attached to make the data transfer wireless. As the DStat, the UWED is comparable with a commercial potentiostat and is also relatively affordable. By comparable, it is meant as not as precise in measurement as commercial potentiostats, but with only small deviations.

The suggested platform in the Training4CRM project is PSoC5LP by Cypress Semiconductor. PSoC 5LP is a versatile platform since it is a Programmable System on Chip (PSoC), which implies a microcontroller with configurable hardware on the platform. One of such a platform's benefits is that there is no need for external components since they are already integrated and configurable. Both the DStat and UWED are custom built and have to be produced for testing, versus a PSoC where all the configurations can be developed directly on a development board.

During the research, and after starting the development of a potentiostat on PSoC5LP, a new article by Lopin and Lopin (2018) was discovered. Lopin and Lopin (2018) developed a potentiostat on the PSoC5LP prototyping platform, where they documented promising results as for the DStat and UWED. Their work was open-source, with all configurations and software code available. Based on the recently published article by Lopin and Lopin (2018), it was decided during the work for this thesis to use their work as the base for a new potentiostat.

From the research on potentiostats, there are often documentation missing. As an example, the miniStat by Adams et al. (2019) lacks open-source software and firmware. However, they have documented how each component in the potentiostat is behaving, the purpose of each component, and how to use it. The problem occurs when there is a need to re-develop such an instrument. Here, Lopin and Lopin's work is standing out. Nevertheless, there are some difficulties with the documentation by Lopin and Lopin; the code (both software and firmware) is very complex and challenging to follow, and therefore also difficult to modify. To make sure the work in this thesis is easy to reproduce, all of the code will be well documented and explained throughout the thesis.

1.2 Goals

This thesis will research and develop an example of an inexpensive and small potentiostat as a prototype for the Training4CRM project. Its main purpose is to develop a potentiostat that can conduct experiments with the electroanalytical techniques cyclic voltammetry and amperometry. The hardware and software shall be well documented for future use. A simple Graphical User Interface (GUI) will be developed. As a starting point for the thesis, the firmware and software developed by Lopin and Lopin (2018) will be used. The software will be re-developed, and the system will be simplified for ease of use and the possibility to implement other functionalities in the Training4CRM project. Possible improvements of the potentiostat by Lopin and Lopin (2018) will also be researched.

This thesis seeks to answer:

1. Is PSoC by Cypress an appropriate platform for a potentiostat?
2. Is the work by Lopin and Lopin (2018) sufficient as a potentiostat?
3. If the work by Lopin and Lopin (2018) is sufficient for a potentiostat, are there any improvements that are possible to implement?

Chapter 2

Theoretical Background

2.1 Electrochemistry

This section is based on chapter 19 in "General chemistry : the essential concepts" (Chang, 2008).

Electrochemistry is a branch of chemistry that studies the relationship between electrical energy and chemical energy; or the relationship between electricity and an identifiable chemical change. The electrochemical process is called a redox (oxidation-reduction) reaction where electrons are transferred from one substance to another. A substance losing an electron is oxidized and is called the reducing agent, while the substance receiving an electron is reduced and is called the oxidizing agent. By applying this on figure 2.1, **A** loses an electron and becomes more positive, and is thus oxidized by **B**. The same approach can be used on **B**, where **B** becomes more negative and is thus reduced by **A**.

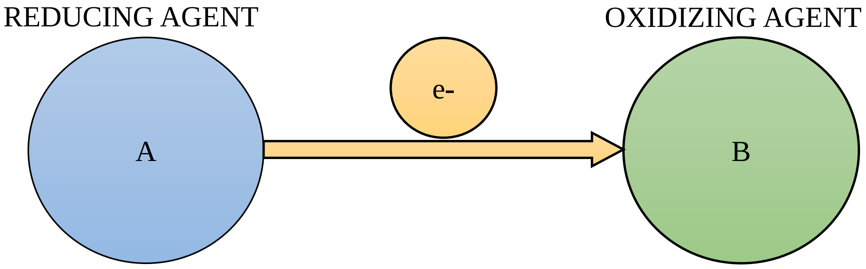


Figure 2.1: Illustration of the basic prinicple of a redox reaction. Adapted from Chang (2008).

By definition, a redox reaction involves both oxidation and reduction of a substance. The atoms' electrons will not freely move away from the atom as long as the positive nuclear charge (protons in the atom) and the surrounding electrons are in equilibrium. Thus, redox reactions involve both positively- and negatively charged ions. This reaction can be divided into two half-cell potentials in which one can study the electron transfer in a solution.

2.1.1 Half-Cell Potential

The *solution* mentioned in the previous section is more precisely an electrolyte. An electrolyte is a substance that contains ions free to move with a positive or negative charge. The ions act as charge carriers in the electrolyte, as electrons are charge carriers in metals. Current, the movement of charged particles per time, is divided between ionic current and electronic current. For this thesis, the ion movement of interest is electrophoresis; the movement of charged particles due to an exogenous electric field (Grimnes and Martinsen, 2015).

To measure the ionic current in an electrolyte, the current has to be transformed into an electric current. By inserting two electrodes in an electrolytic cell, a completed circuit establishes which will let the ionic current transform to the electric current flowing through the circuit. One electrode will be oxidized, and the other reduced. These two electrodes are both defined as half-cell potentials. The half-cell potentials of each electrode can summarize the overall potential of the electrolytic cell (Chang, 2008).

2.1.2 The Electrode

As mentioned in the previous section, each electrode works as a half-cell and is where the ionic current transforms into the electric current. This section will describe what happens at the electrodes and about different types of electrodes. The information in this section is found in chapter 7 of "Bioimpedance and bioelectricity" by Grimnes and Martinsen (2015).

By itself, an electrode is just a conducting material. With two electrodes, a circuit is closed, and there can be conductivity in the electrolytic cell. The electrode is said to be polarized when electrons are flowing through it. One of the basic phenomena at a polarized electrode is called the electric double layer. This layer is relatively thin and is the boundary between the electric conductor (the electrode) and the ionic conductor (the electrolyte). The ions in a bulk electrolyte are free to move except at the electrode. At the electrode, bonds establish caused by the charge distribution in the double layer. Since the electrodes consist of a conducting material with atoms in strict bonds, the double layer occurs in the electrolyte. The electric double layer is where the transfer of electrons happens, from ionic current to electric current.

In a bulk electrolyte with polarized electrodes, ions flow toward an electrode with opposite polarity. The electric double layer will occur nearly instantly, but the ionic current will reach a peak. The current peak is due to another phenomenon that takes place in an electrolyte called diffusion. Diffusion is the tendency for components in a solution to flow from higher concentration to lower concentration. The phenomena occur due to random motions in the solution, related to Brownian motion, and is described by Fick's law. As the concentration of the solution at the electrode grows, the diffusion layer grows. The concentration in the diffusion layer decreases exponentially the further away from the electrode the ions are. At some point, the ions will no longer be affected by the electrode, which gives the current peak and a decrease of ionic current towards the electrode. The consequence of Fick's law is essential to understand what happens in the electrolyte during measurements.

2.1.3 The Nernst Equation

This section is based on chapter 19 in "General chemistry: the essential concepts" (Chang, 2008) and chapter 7.6.2 in "Bioimpedance and bioelectricity basics" (Grimnes and Martinsen, 2015).

An essential tool to understand the output of experiments in an electrolytic cell is the Nernst equation (see equation 2.1). The equation gives the relationship between the half-cell reduction potential and the electrode potential, temperature, and chemical concentration.

$$E = E_0 + \frac{RT}{nF} \cdot ln\frac{(Ox)}{(Red)} \tag{2.1}$$

In the Nernst equation (2.1), E is the reduction half-cell potential, E_0 is the standard reduction half-cell potential in equilibrium, RT is the universal gas constant multiplied by the environmental temperature (in Kelvin), nF is the number of electrons transferred in the reaction multiplied by the Faraday constant, and $\frac{Ox}{Red}$ is the relative activities of the oxidized and reduced analyte in the system. $\frac{Ox}{Red}$ is equivalent to the concentration of the reducing- and oxidizing agent, and can be expressed as follows (see equation 2.2):

$$E = E_0 + \frac{RT}{nF} \cdot ln \frac{(C_{Ox})}{(C_{Red})} \tag{2.2}$$

In room temperature, the factor $\frac{RT}{nF}$ can be replaced with approximately 61 mV which simplifies the equation to (see equation 2.3):

$$E = E_0 + 0.061 \cdot log \frac{(C_{Ox})}{(C_{Red})} \tag{2.3}$$

Note that the logarithm is changed to base 10 instead of e. The purpose of simplifying the Nernst equation is to visualize that the potential in the electrolytic cell can be simplified to:

- E_0, the potential in equilibrium
- $\frac{RT}{nF}$, a factor dependent on temperature
- $\frac{(C_{Ox})}{(C_{Red})}$, the ratio of the oxidizing- and reducing agent

In other words, the Nernst equation estimates the potential of the reduction half-cell by knowing the concentration of the electrolytes in the electrolytic cell. The equation prerequisites that the reaction in the electrolytic cell is reversible. A reversible reaction is when reactants and products can react and return the reactants; a reaction where no bi-products occur.

2.2 Potentiostat

This section is based on an introduction to potentiostats by Gamry Instruments (2020). It will explain what a potentiostat is, what it does, and why it is a versatile instrument to conduct electrochemistry measurements.

A potentiostat is an electronic hardware device that controls the voltage difference between two electrodes; the working electrode and the reference electrode. An ionic current is flowing from a third electrode, the counter electrode, to the working electrode. All of the electrodes are in contact with the electrolytic cell. The primary usage of a potentiostat is to measure the current flow from the counter- to the working electrode while controlling the potential in the electrolytic cell. Sensing the voltage difference between the reference electrode and the working electrode achieves regulation of the voltage in the cell by adjusting the current from the counter electrode.

2.2.1 The Three-Electrode System

This section is based on chapter 7.10.2 in "Bioimpedance and bioelectricity basics" by Grimnes and Martinsen (2015).

Electroanalytical experiments need at least two half-cells (two electrodes) to be achievable. An ionic current will flow through the cell between the electrodes by applying a voltage over the electrolytic cell. By utilizing only two electrodes, the problem is that the applied voltage will be dependent on the current due to Ohm's law: $U = R \cdot I$. Hence, there will be difficulties controlling the applied voltage.

A common method to avoid this problem is to utilize three electrodes. The general principle is that the current will flow between the counter electrode and the working electrode (see figure 2.2), while the reference electrode senses the voltage over the cell without any current flowing through the electrode (high input impedance).

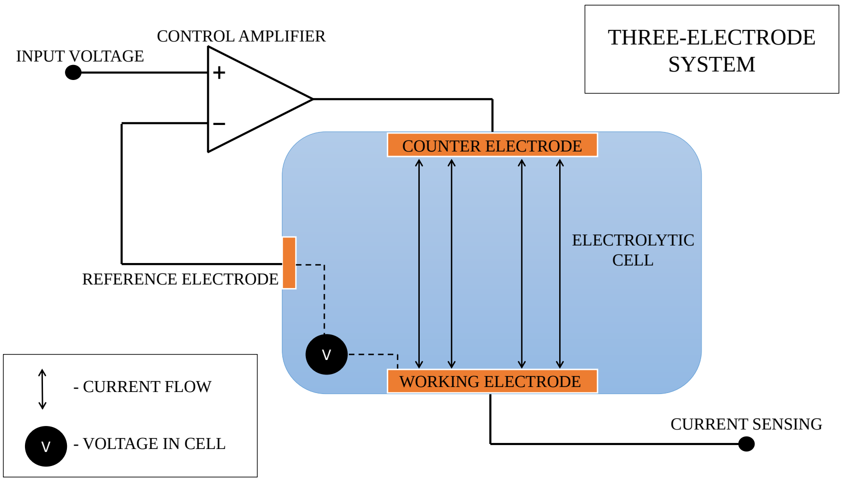


Figure 2.2: A simplified schematic of a three-electrode system. Adapted from Gamry Instruments (2020)

By sensing the voltage, the reference electrode feeds the measured voltage back to a control amplifier. The control amplifier has two inputs; on the positive input, the desired voltage is applied; on the negative input, the reference electrode's feedback is applied. The output of the control amplifier is the difference between the two inputs. In conclusion, the three-electrode configuration can regulate the voltage because of the reference electrode versus a two-electrode system.

For the electrodes to serve their purpose, some choices concerning the electrode geometry and material are needed. The material should be an inert material like inert metals (e.g., gold or platinum) or inert carbon materials (e.g., glassy carbon). A reference electrode should have a constant electrochemical potential when no current is flowing through it. Ag/AgCl is a common choice of material.

Section 2.2.1.1, 2.2.1.2 and 2.2.1.3 describe the different electrodes in the three-electrode system, and are all based on a document made by Gamry Instruments (Gamry Instruments, 2020).

2.2.1.1 Working Electrode

The working electrode is the electrode where the current is measured and where the electrochemical reaction occurs (as described in section 2.1.2). Its purpose is to transfer charge to and from the analyte.

2.2.1.2 Reference Electrode

The reference electrodes' purpose is to sense the potential in the cell. The electrode has to have very high impedance so that an infinitesimal amount of current is flowing through it. The input of the control amplifier obtains the high impedance, which ideally has infinite input impedance. The electrode should have a known half-cell potential and should not be affected by reactions occurring in the cell.

2.2.1.3 Counter Electrode

The counter electrodes' purpose is to complete the circuit. When current flows through the working electrode, the voltage difference between the working electrode and the reference electrode changes. The potentiostat will instantly regulate that change in a regulation loop by pumping an equal amount of current back into the electrolytic cell through the counter electrode. Due to this regulation, the potentiostat controls the voltage in the cell.

2.2.2 Proof of Regulation

To fully understand the regulation in the circuit, a proof will be provided. The proof is adapted from Umar et al. (2018).

The three-electrode system is visualized in figure 2.3 as an equivalent circuit. V_{in} is the desired applied voltage, V_{out} is the output voltage from the control amplifier, V_r is the reference voltage, Z_1 and Z_2 is a voltage divider for the reference electrode, and the working electrode has ground (return path) as the common reference for the entire system.

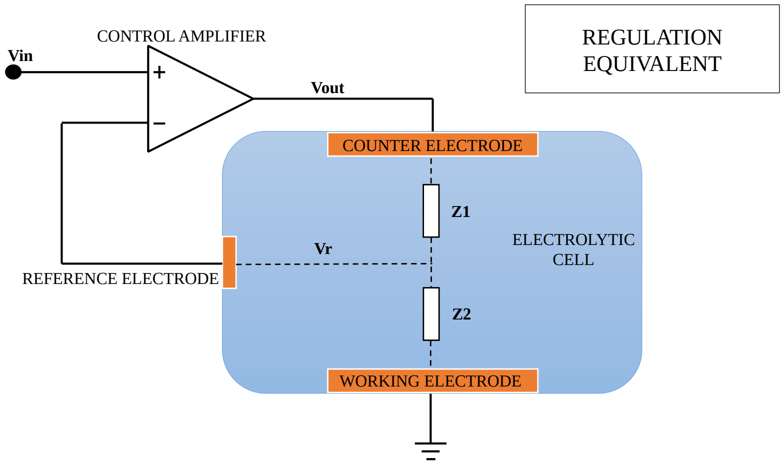


Figure 2.3: Equivalent circuit to a three-electrode system. Adapted from Umar et al. (2018).

The current flowing through the electrolytic cell originates at the counter electrode and flows through Z_1 and Z_2 to the working electrode and ground. As mentioned earlier, there is ideally zero current flowing through the reference electrode.

V_r can be rewritten as in equation 2.4 due to the voltage divider:

$$V_r = \frac{Z_2}{Z_1 + Z_2} \cdot V_{out} \tag{2.4}$$

The control amplifier has an amplification (denoted A), and as mentioned earlier, the output of the control amplifier is the difference between the positive and negative input. This sums up to equation 2.5:

$$V_{out} = A \cdot (V_{in} - V_r) \tag{2.5}$$

By combining equation 2.4 and 2.5, and denoting $\beta = \frac{Z_2}{Z_1+Z_2}$, it gives equation 2.6:

$$V_r = \beta \cdot A(V_{in} - V_r) \rightarrow \frac{V_r}{V_{in}} = \frac{1}{1 + \frac{1}{A\beta}} \tag{2.6}$$

Equation 2.6 is a proof that the relation between V_r and V_{in} is only dependent on the maximum amplification of the control amplifier (A) and the series of impedance in the cell. The relation is as follows:

$$A\beta \gg 1 \Longrightarrow \frac{V_r}{V_{in}} \rightarrow 1 \Longrightarrow V_r = V_{in} \tag{2.7}$$

Equation 2.7 shows that the control amplifier will keep the voltage between the reference electrode and the working electrode close to the applied voltage. This equation is the essence of how the potentiostat operates.

2.2.3 Schematic and Components

The three-electrode system described in section 2.2.1 is the crucial part of a potentiostat. To build the system, there is a need for a device generating the desired applied voltage, a device that can measure the current flowing through the working electrode, and a device to read out the measured current. A simplified schematic has been made in figure 2.4 to give an overview of the components usually used in the development of a potentiostat. This section will describe the functionality of each of the components.

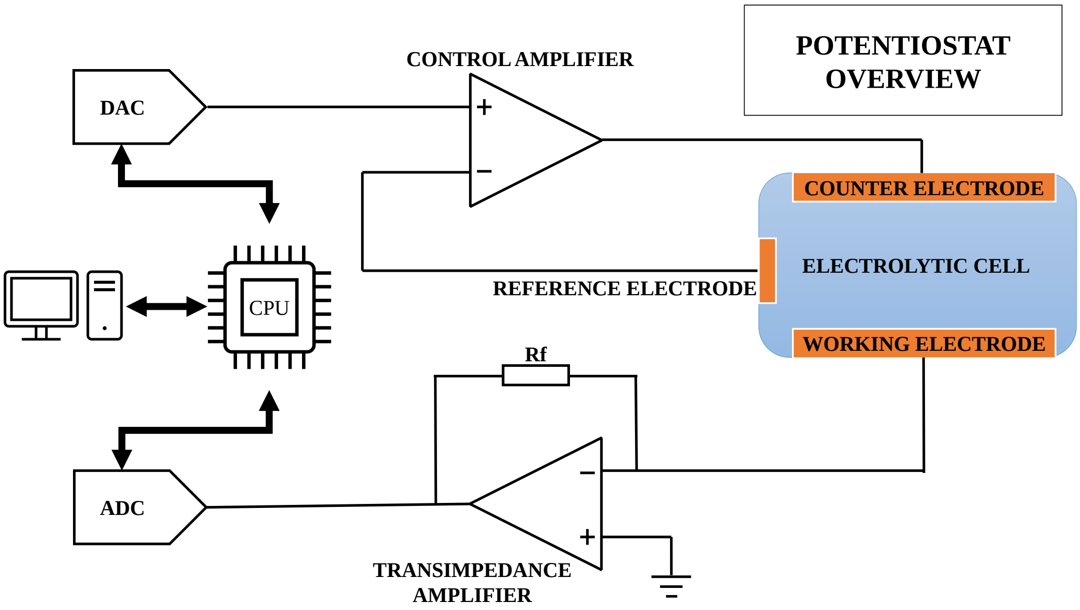


Figure 2.4: A simplifies schematic of a potentiostat.

2.2.3.1 Digital to Analog Converter

To generate a desired voltage, a DAC (Digital to Analog Converter) is often a preferred component (Gamry Instruments, 2020). The component has a limited voltage range where the total range is divided by the number of bits, which is the resolution of the DAC (Cypress Semiconductors and Infineon, 2020a). As an example, a DAC with a voltage range of 1 V and 8-bits will have a resolution calculated in the following way: $\Delta V = \frac{1V}{2^8} = \frac{1V}{256} \approx 3.9mV$. This voltage step is often called the LSB (Least Significant Bit). Since the component is digital, it can not generate every possible voltage between 0 V and 1 V. The DAC can generate every possible voltage step on the form $N \cdot \Delta V$ where the maximum value of $N = 256 - 1$ (see figure 2.5). A large variety of resolutions and voltage range for DACs exists (Digikey and Mouser are examples of places to see the variety).

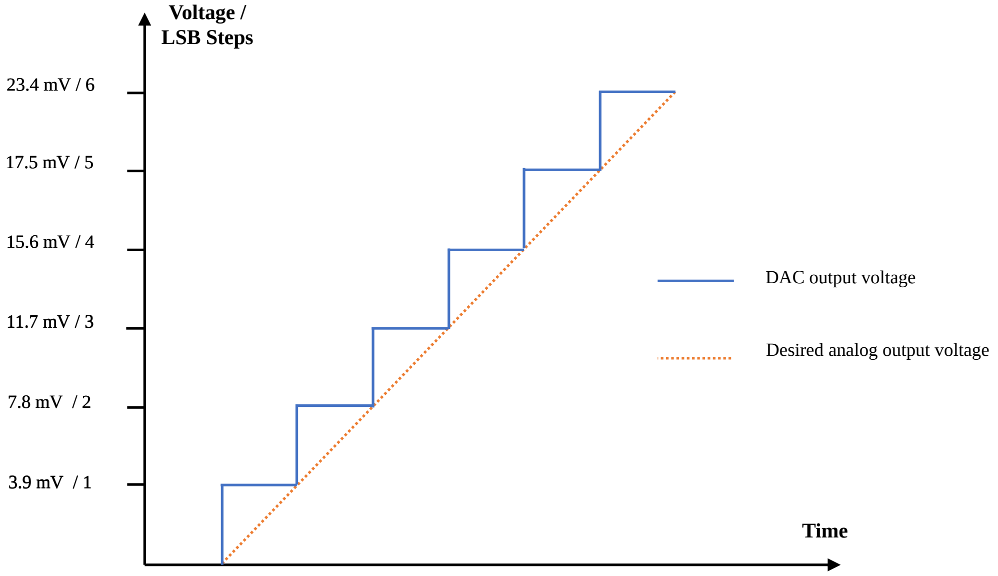


Figure 2.5: Difference between the output voltage from an 8-bit DAC and an analog signal.

2.2.3.2 Operational Amplifier (Control Amplifier)

The control amplifier was described in section 2.2.2, but the list below has some of the most important aspects for an ideal operational amplifier (OPAMP) listed (Scherz and Monk, 2016):

- The open-loop voltage gain is infinite, meaning that the OPAMP has infinite amplification.
- The inputs (positive and negative) have very high input impedance (ideally infinite).
- The output has very low output impedance (ideally zero).
- The inputs draw zero current.
- The general formula is $V_{out} = A_0(V_+ - V_-)$, where V_{out} is the output voltage, A_0 is the amplification, V_+ and V_- are respectively the positive and negative inputs.

It is important to clarify that the list above are notes for an ideal OPAMP. For real OPAMPs, there are limitations to all the notes above. However, for many purposes, the real OPAMPs behave very similarly to the notes above, with only small deviations in the result.

2.2.3.3 Transimpedance Amplifier

A transimpedance amplifier, also called a current-to-voltage converter, has the purpose of converting current to voltage (Scherz and Monk, 2016). One of the purposes of a potentiostat is to measure the current flowing through the working electrode, and the transimpedance amplifier is one alternative to achieve that.

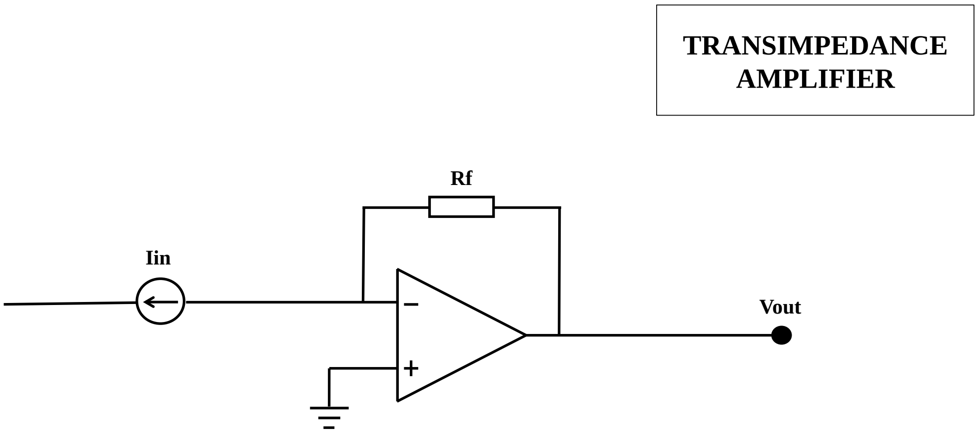


Figure 2.6: Schematic of a transimpedance amplifier, also called a current-to-voltage converter (Scherz and Monk, 2016).

In figure 2.6, a schematic of a transimpedance amplifier has been provided. From the theory of operational amplifiers provided in section 2.2.3.2, the current flowing from I_{in} can only flow through R_f since the negative input of the OPAMP draws no current. By Ohm's law, the output voltage has to be $V_{out} = R_f \cdot I_{in}$.

When the output voltage is sampled the current can be calculated by reversing

Ohm's law like this (equation 2.8):

$$I_{in} = \frac{V_{out}}{R_f} \tag{2.8}$$

2.2.3.4 Analog to Digital Converter

The component used to sample the measured current, or more precisely the current transformed to voltage, is an ADC (Analog to Digital Converter). ADCs work similarly as a DAC, but instead of transforming a digital signal to an analog signal, the ADC transforms an analog signal to a digital signal. There are several methods to achieve this, e.g., successive approximation (SAR) ADC and Delta-Sigma ADC (Kester, 2005).

As Kester (2005) describes, the SAR ADC utilizes a comparator that measures the difference between the signal and an internal reference voltage. It will then feed the measurement, whether the signal is higher or lower than the reference, to a logic block that will set a new reference. By doing this several times, the reference that is closest to the signal will be the correct bit. The signal is then converted from analog to digital. A SAR ADC is very appropriate for high-speed acquisition designs and is a common choice for ADCs. However, if the goal is to acquire high precision measurements at moderate speeds, the Delta-Sigma ADC is often considered a better choice.

Baker (2011) gives a detailed description of how a Delta-Sigma ADC operates. A brief, adapted version will be provided here. Instead of comparing the signal with a reference voltage for each bit as the SAR ADC, the Delta-Sigma ADC transforms the signal into the frequency domain with a Delta-Sigma modulator. The signal's lower frequencies will be pushed up to higher frequencies by oversampling the signal, increasing the signal-to-noise ratio. Digital filtering is then applied to remove noise, and then downsampling of the signal occurs as a counterpart to the oversampling. The benefit of the Delta-Sigma ADC is the higher precision and larger signal-to-noise ratio than the SAR ADC. The downside of a Delta-Sigma ADC is that each measurement takes more time to achieve (depending on resolution) than for the SAR ADC.

2.2.3.5 Microcontroller

The last essential component for a potentiostat is the device controlling all of the other components, the microcontroller or equivalent. In the most basic way, a microcontroller can be explained as a computer on a chip, as described in Scherz and Monk (2016). The book further explains that a microcontroller usually contain a processing unit, memory units, communication ports, ADC, DAC, etc. Its functionality is to control ports and components in an integrated circuit (IC). The device is usually configurable by a programming language.

The microcontrollers' manufacturers often make evaluation/development boards for users to experiment with and verify if the controller is suitable for their project. Arduino is one of the more popular firms for people curious about playing with electronics.

There are several types of devices available for users to use for their projects (Scherz and Monk, 2016). Some are application-specific integrated circuits (ASIC) and is only usable for its intended purpose. The microcontroller is very versatile concerning its application area, and can be configured to almost anything within its maximum electrical ratings. The system on chip (SoC) is one step more advanced than the microcontroller since it can not only configure the electronics within the microcontroller, but also configure the hardware surrounding the microcontroller. This makes the SoC a good alternative for projects where external components are unwanted.

2.3 PSoC-Stat: A single chip open source potentiostat by Lopin and Lopin (2018)

A good guideline for this project was mentioned in the introduction, the work documented in the article by Lopin and Lopin (2018). Their work used the same platform as this project to make a potentiostat. This section will be based on that article and enhance the essential aspects of their work. Also, the aspects of their work where improvements may be feasible will be highlighted.

Lopin and Lopin (2018) developed their potentiostat to demonstrate that a potentiostat can be developed on a programmable system on chip (PSoC), where they highlight the benefit of a system where no external components are needed in the design. They document their work well and conclude their work as a successful potentiostat with some limitations. The limitations they highlight are that the potentiostats' precision is limited to the components inside the PSoC. Selecting each component in the design specifically to the necessary limitations for high precision will make the potentiostat even more comparable to a commercial potentiostat. However, that is not possible with a PSoC since all the components are integrated within the platform. An ASIC will have to be developed to account for that issue or an SoC with "better" components. With that said, the potentiostat has a high precision if the correct filtering after each measurement is accomplished. The noise picked up by electromagnetic radiation in the potentiostat, as the 50 Hz in the power net, is filtered out by a moving average of a least two samples. With all the implementations made by Lopin and Lopin (2018), this potentiostat is solid work and a feasible start for this thesis, but the potentiostat has more functionalities than needed for this thesis. The rest of this section will involve possible modifications for the work of this thesis.

For this thesis's scope, there is a point in developing a potentiostat with the least amount of extra functionalities. This is due to other measurement techniques that might be implemented on the PSoC in the Training4CRM project. Therefore, the only techniques needed are the amperometry and the cyclic voltammetry.

The potentiostat by Lopin and Lopin (2018) is designed with the optional two-electrode configuration. This will not be implemented in the potentiostat for this thesis with the arguments described in section 2.2.1.

Lopin and Lopin (2018) made a graphical user interface (GUI) and software which are very impressive, but the complexity of their work makes it very difficult to follow. Therefore, all of the software and GUI will be re-developed with an extensive effort to make it re-producible.

Their design for amperometry and cyclic voltammetry is functional but cannot continuously transfer the data to the personal computer (PC). This implies that the PSoC memory might eventually be filled up, which will lead to an error in the system. Continuous data transfer is an improvement that will be researched.

To make the cyclic voltammetry triangle shape, Lopin and Lopin (2018) used a look-up table (LUT). This is also an element that uses memory on the PSoC. Research for an improvement where the cyclic voltammetry's triangle shape is made during the experiments will be researched.

Chapter 3

Material

This section will describe the materials needed for the potentiostat designed in this thesis and the materials needed for the potentiostat by Lopin and Lopin (2018). In addition, the electrodes will be presented.

3.1 Embedded Platform

The chosen platform for this project is the PSoC 5LP, mostly due to the intention of using that platform in the Training4CRM project, but also due to its abilities presented in table 3.1. PSoC 5LP is lacking the wireless communication compared to the other versions of PSoC, but is, as table 3.1 displays, considered a better choice for high precision measurements due to its ADCs, number of DACs, number of universal digital blocks (UDB), and a suitable number of general purpose input/outputs (GPIO) (Cypress Semiconductors, 2020b).

PSoC 5LP comes in two different versions of development kits:

- CY8CKIT-059 PSoC 5LP prototyping kit (Cypress Semiconductors, 2020e) - Cost: 143 NOK (www.digikey.com, October 14th 2020)
- CY8CKIT-050 PSoC PSoC 5LP development kit (Cypress Semiconductors, 2020e) - Cost: 894 NOK (www.digikey.com, October 14th 2020)

The prototyping kit is simpler and smaller than the development kit, meaning it has fewer peripherals and opportunities, but with the same processor and DAC/ADC. Whereas the development kit has a breadboard implemented for simple hardware configurations, an LCD display, and all the peripherals available. Lopin and Lopin (2018) used the prototyping kit for their potentiostat. For this thesis, the development kit will be used due to its breadboard and LCD, making the development easier when it comes to testing throughout the development process.

PSoC Family			
	PSoC 4	**PSoC 5LP**	**PSoC 6**
CPU	ARM Cortex-M0	ARM Cortex-M3	ARM Cortex-M4 ARM Cortex-M0+
Flash / SRAM	256 kB / 32 kB	256 kB / 63 kB	2048 kB / 512 kB
GPIO	98	72	104
Bluetooth	Yes	No	Yes
DAC	2 x DAC (8-bit)	4 x DAC (8-bit)	1 x DAC (12-bit)
ADC	1 x SAR ADC (12-bit)	1 x Delta Sigma ADC (8 to 20-bit) 2 x SAR ADC (12-bit)	1 x SAR ADC (12-bit)
Digital blocks	8	24	12

Table 3.1: Overview of the different PSoC microcontrollers: PSoC 4 (Cypress Semiconductors, 2020c), PSoC 5LP (Cypress Semiconductors, 2020b) and PSoC 6 (Cypress Semiconductors, 2020a).

PSoC is a Programmable System on Chip, which is what makes the circuit very suitable for development. It consists of programmable routing and configurable analog and digital blocks that are interconnected with the CPU (Central Processing Unit) sub-system (see figure 3.1). As the illustration presents, the top-level consist of all the ports and programmable routing. This is one of the strengths of an SoC versus a microcontroller (briefly explained in section 2.2.3.5). The mid-layer is divided into two parts: the digital block and the analog block. By separating analog and digital signals, there is a smaller risk of having interference between them. In order to fully achieve this, the two blocks have isolated return paths from each other. The bottom layer involves digital processing through the CPU, as well as the communication peripherals.

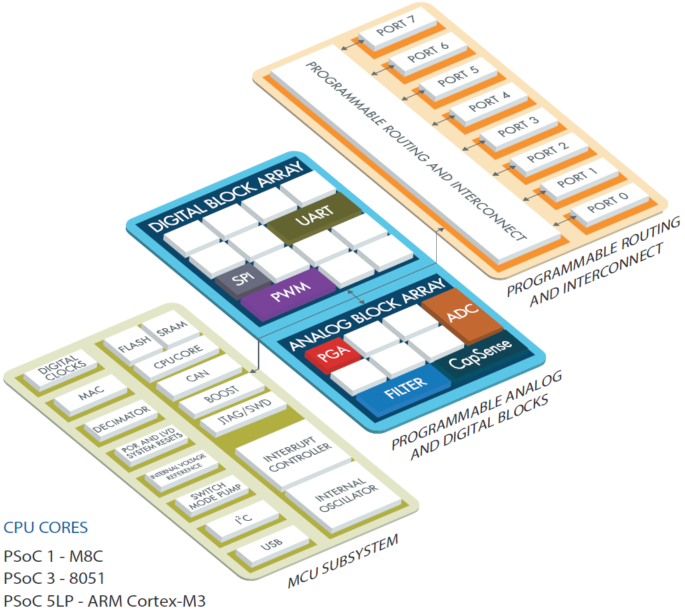


Figure 3.1: Illustration of the PSoCs build-up and sub-system (Cypress Semiconductors, 2020b).

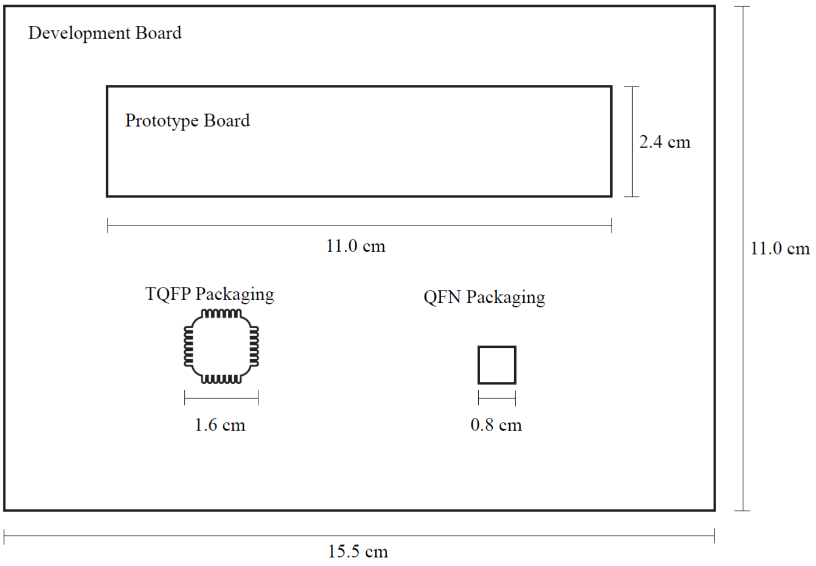


Figure 3.2: Illustration of the size differences between the PSoC5 LP development board, prototyping board, TQFP packaging and QFN packaging. The illustration is made by Ruud (2019).

It should be noted that the use of development kits are not the end of the line concerning the size of the potentiostat. Training4CRM is planning to make a device that can be implemented in the human brain, and, of course, a development kit is too large. It is possible to buy the actual SoC on the development board and implement it on a custom made PCB. The IC comes in different packages visualized in figure 3.2.

3.2 Electrodes

The carbon electrode chip used in this project is the same as Cunha et al. (2019) used for their bioimpedance measurements. They were provided by Technical University of Denmark (Hassan et al., 2017), and consist of a circular pyrolytic carbon working electrode with an area of 12.5 mm^2, surrounded by a carbon counter electrode with an area of 25.2 mm^2 and a gold reference electrode with an area of 0.8 mm^2 (see figure 3.3). To isolate the electrodes from each other, a passivation layer of SU-8 is used (see figure 3.3, right picture, marked C).

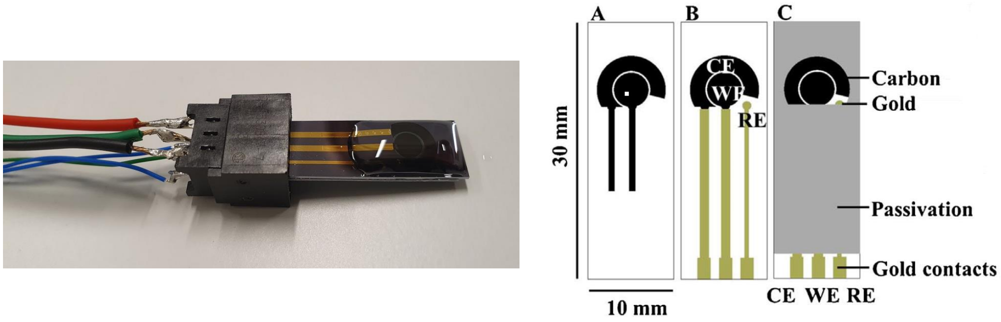


Figure 3.3: Too the left, a close-up of the carbon electrode chip used in this thesis is displayed. Here the electrode chip is mounted in a chip holder with wires attached, and a solution covering the electrodes. On the right side, a cross section of the electrode chip is displayed (Hassan et al., 2017).

This electrode system is suitable for cyclic voltammetry and amperometry measurements (Hassan et al., 2017).

Chapter 4

Method

This section will explain the use of the potentiostat. Firstly, it will explain the electroanalytical techniques implemented in the potentiostat for this thesis, and then explain the setup and use of the potentiostat.

4.1 Electroanalytical Techniques

Electroanalytical techniques are the methods used to perform measurements in the electrolytic cell. Several techniques are possible with a potentiostat, but this thesis will focus on cyclic voltammetry and amperometry, as explained in the introduction.

4.1.1 Cyclic Voltammetry

This section is based on a review article by Elgrishi et al. (2018) where they have given a practical approach for the use of cyclic voltammetry.

Cyclic voltammetry is a technique used to investigate the reduction and oxidation processes in an electrolytic cell. In order to conduct such an experiment, a cycling voltage is applied to the cell. The voltage is ramped up linearly from a starting voltage to a maximum, then ramped linearly down to a minimum and back up to the starting voltage (see figure 4.1). This will be referred to as one cycle or one period (the cycle may also be reversed). The increase rate of the slope is known as the scan rate, $\upsilon = \frac{dV}{dt}$, and is one of the most essential parameters for cyclic voltammetry. While the voltage is cycling over the electrolytic cell, an ionic current will flow through the working electrode. This current will be measured and is equivalent to the ionic current flow in the electrolytic cell. A cyclic voltammogram is an appropriate visualization for cyclic voltammetry, where it displays the voltage and current in the same plot.

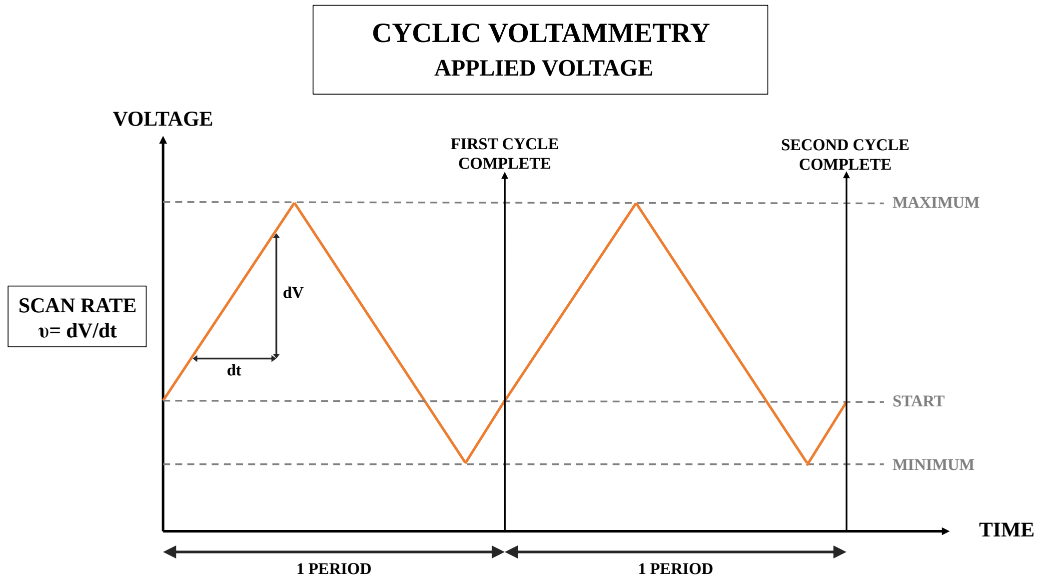


Figure 4.1: An example of the applied voltage for a cyclic voltammetry experiment. Here two cycles are displayed. The starting voltage is deliberately chosen at another position than the minimum voltage, they are often the same. The scan rate is also visualized in the figure.

The potentiostat is the device used to perform the experiment. By utilizing the three-electrode system, the given voltage will be kept at a known level (see description in section 2.2.1). An ADC in the potentiostat will measure the current flowing through the working electrode by utilizing the transimpedance amplifier, converting the current to voltage. Since the ADC samples at a known time (controlled by clocks in the instrument) and the DAC sets the voltage in the rate of the scan rate, the relationship between current and voltage is known. This will be further explained in the next section.

4.1.1.1 Cyclic Voltammogram

The cyclic voltammogram displays the relation between the current and the voltage in an electrolytic cell. Elgrishi et al. (2018) used a popular example with ferrocene, which is a reversible electrochemical solution. This implies that the voltammogram peaks have the same amplitude (as in figure 4.2). Reversible solutions were also mentioned in section 2.1.3, where it was noted that the Nernst equation prerequisites that the solution is reversible. This will be an important aspect of this section.

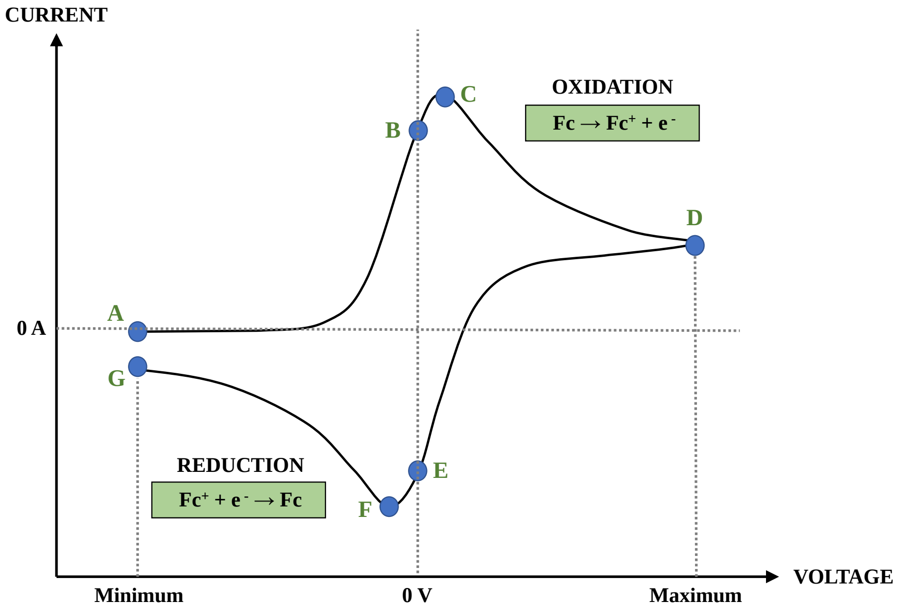


Figure 4.2: The process of cyclic voltammetry displayed in a cyclic voltammogram. Adapted from Elgrishi et al. (2018).

The cyclic voltammogram will be explained with figure 4.2 as reference. First, it is important to understand that A represents the start of the cyclic voltammetry and that G represents the end of one period. These two dots will also represent the starting- and minimum voltage for the cyclic voltammetry. D represents the maximum voltage. From A to D is where oxidation of the chemical ferrocene occurs; it looses electrons. The interval D to G represent reduction of ferrocene; it gains electrons. At the peaks, C and F, diffusion is the limiting factor of the reaction due to Fick's law (mentioned in section 2.1.2). All the Fc in close proximity to electrode surface is oxidized at peak C, and the diffusion has a slower rate to transport more Fc to the electrode surface than the rate of oxidation. Therefore, the current will decrease from C to D. At D, the voltage scan is reversed and will decrease, causing a decreased current until F. Here an opposite reaction occurs; Fc^+ is reduced, and the diffusion has a slower rate to transport more Fc^+ to the electrode surface than the rate of reduction. This will lead to an increase in current between peak F and G. B and E is where the concentration of oxidized- and reduced molecules are equal at the electrode surface, and is known as the halfway potential between the two peaks (C and F). By the Nernst equation, this potential gives a straight forward approach to find the standard half-cell potential in equilibrium (E_0) and is often used to calibrate the device for the electrodes.

The Nernst equation is a powerful tool to predict the cell's chemical reactions, but as with the halfway potential, the Nernst equation can also be utilized to give E_0. The cyclic voltammetry is a powerful technique to characterize chemical solutions, and the voltammogram is the product of such an experiment. For an irreversible chemical, e.g., ascorbic acid, there will be no occurrence of reduction. This implies that there are developed bi-products which no longer are electroactive. The next cycle applied to the solution will then have a peak C at a lower current level than the first cycle. After a while, further cycles will have zero ionic current flow due to only bi-products left in the solution.

4.1.1.2 Scan Rate

The scan rate in the cyclic voltammetry is one of the most important parameters for this type of experiment. This parameter sets the rate of the voltage change for the potentiostat. An increased scan rate influences how large the diffusion layer at the electrode will be. Hence, an increased scan rate will give higher peaks in the voltammogram, while a decreased scan rate will have smaller peaks. The peak height will change linearly to the square root of the scan rate.

4.1.2 Amperometry

Amperometry is an electroanalytical technique where the applied voltage is kept constant throughout the experiment (Bucher and Wightman, 2015). Current flowing through the working electrode is measured per time, and the quantity of the electroactive substance can be calculated by Cottrells law (Adeloju, 2005). As Adeloju (2005) explains, Cottrell's law gives a relation between the current flowing through the working electrode and the concentration of a substance in the electrolytic cell. This technique is very powerful when the substance that is measured has known electrochemical properties as dopamine (Bucher and Wightman, 2015). This is why this technique is implemented in the potentiostat since the Training4CRM project are developing an instrument detecting and measuring dopamine in the human brain.

As in cyclic voltammetry, the potentiostat will control the applied voltage during the amperometric measurements due to the three-electrode system. The only parameter to control in the potentiostat to perform amperometry is the applied voltage level. The DAC and the three-electrode system will keep that voltage steady while the ADC measures the current-to-voltage converted values. If the reaction in the electrolytic cell is quick, the ADC should have a more rapid sampling. Otherwise, the sampling rate can be kept at a moderate level.

4.2 Potentiostat

In this section, the instructions for the use of the potentiostat will be presented. It will involve the graphical user interface, electrodes' preparations before conducting an electrochemical technique, and how to set up the instrument.

4.2.1 Instrument Setup

The device setup is divided between the hardware setup and electrode connections, and the firmware setup.

4.2.1.1 Hardware setup

The PSoC5LP has to be configured in order to use it for electroanalytical measurements. Figure 4.3 has an overview of the device provided by Cypress Semiconductors (2020d) in the development kit start-up guide. In order to provide swift feedback from the device during measurement, an LCD display has been used and shall be mounted on the device (see figure 4.3, "Character LCD Interface").

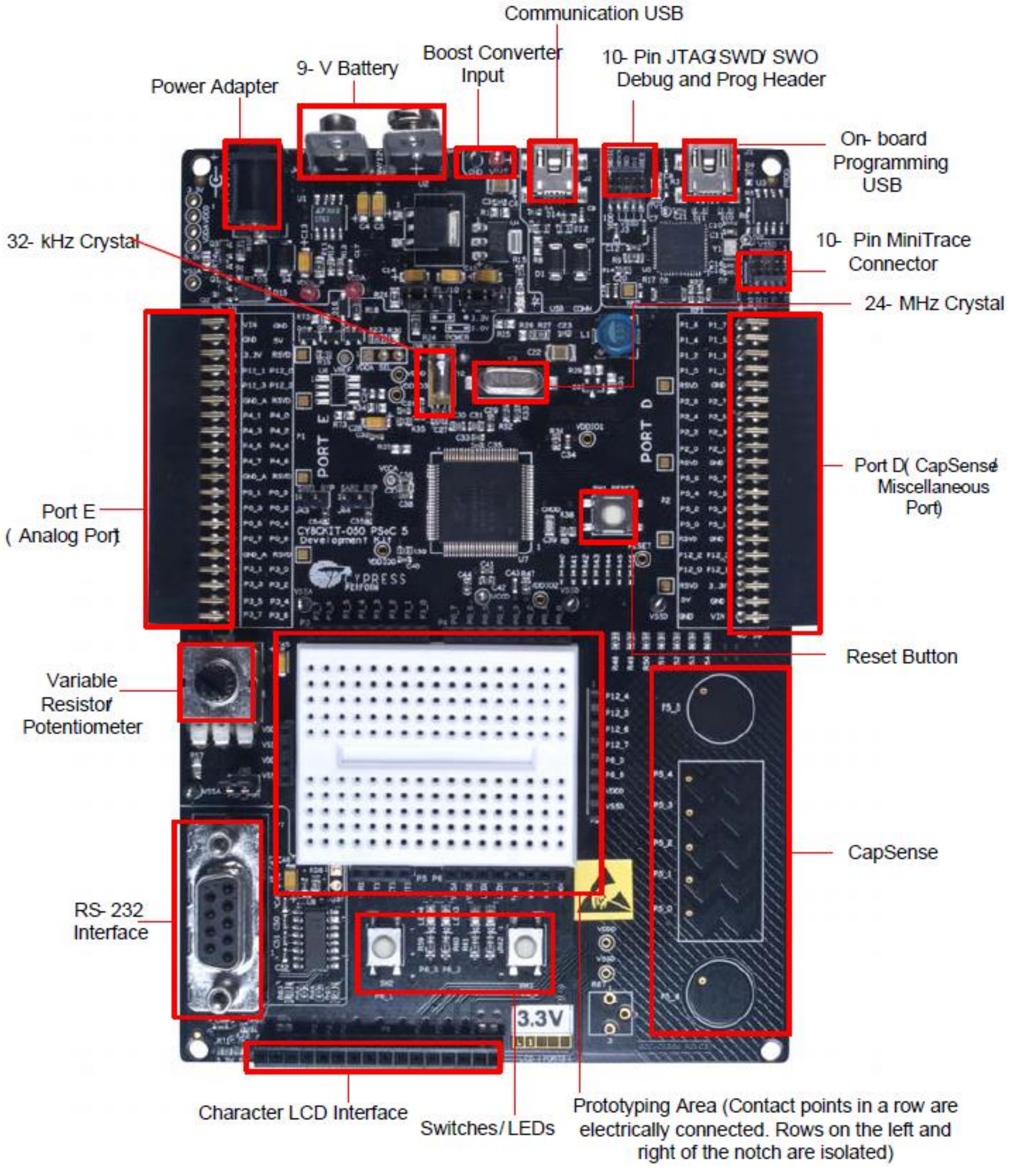


Figure 4.3: Overview of the PSoC5LP development kit. Image is taken from the development kit start-up guide (Cypress Semiconductors, 2020d).

Figure 4.4 has three narrow images from figure 4.3. Image **A** in figure 4.4 points to where the connections of the potentiostats electrodes and analog ground shall be connected. The counter electrode (CE) shall be mounted to $P3\_7$, the reference electrode (RE) shall be connected to $P3\_2$, the working electrode shall be connected to $P0\_0$, and the analog ground shall be connected to $VSSA$. Since there is a need for an external capacitor of $0.1\mu F$ for the dithering DAC, the capacitor shall be mounted between *VSSA* and *P3_7*. If desired, an LED can be connected in order to see when an experiment is running. The LED has to be connected by adding a strap wire between *P6_0* and *LED1*. Another additional option is to mount a capacitor to ground between the transimpedance amplifier and the ADC to reduce noise. This capacitor has to be connected between *P0_3* and *VSSA*.

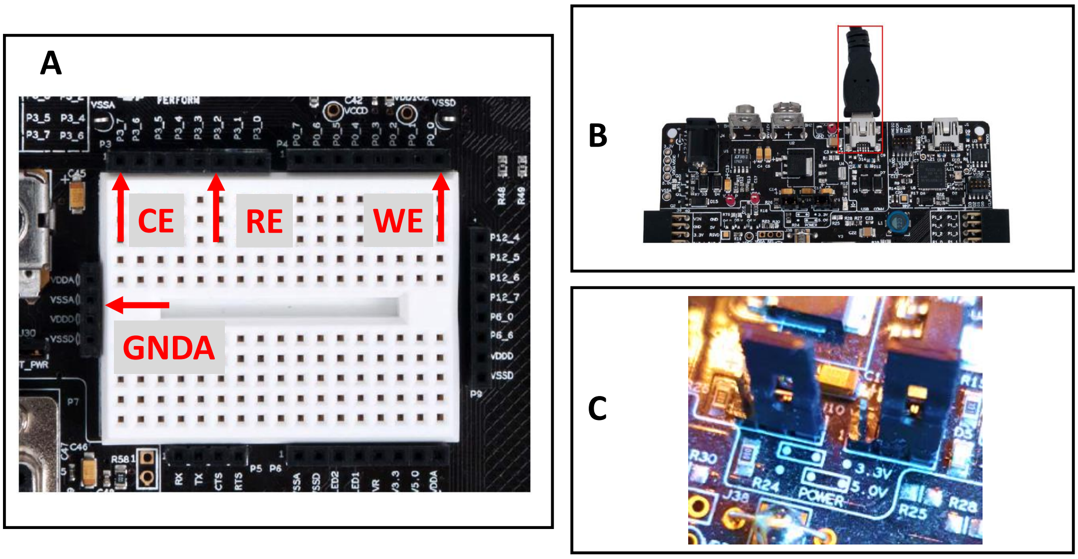


Figure 4.4: Overview of the PSoC5LP development kit. **A**: The arrows points to where the electrodes and analog ground shall be connected. **B**: Connection for communication with computer and power to the device. **C**: Correct placement of jumpers. Image is taken from the development kit start-up guide (Cypress Semiconductors, 2020d).

The image marked **B** in figure 4.4 shows where to connect the USB cable powering the device and data transfer between the computer and the device. There is another connection to the right of the one marked in the image. That connection is meant for programming only and shall not be used when conducting an experiment.

The image marked **C** in figure 4.4 displays how the jumpers shall be connected. This is important since a wrong connection will make the device malfunction since it only can provide 3.3 V maximum voltage instead of 5 V maximum voltage. The consequence of a wrong connection is that the potentiostat will have a limited voltage range.

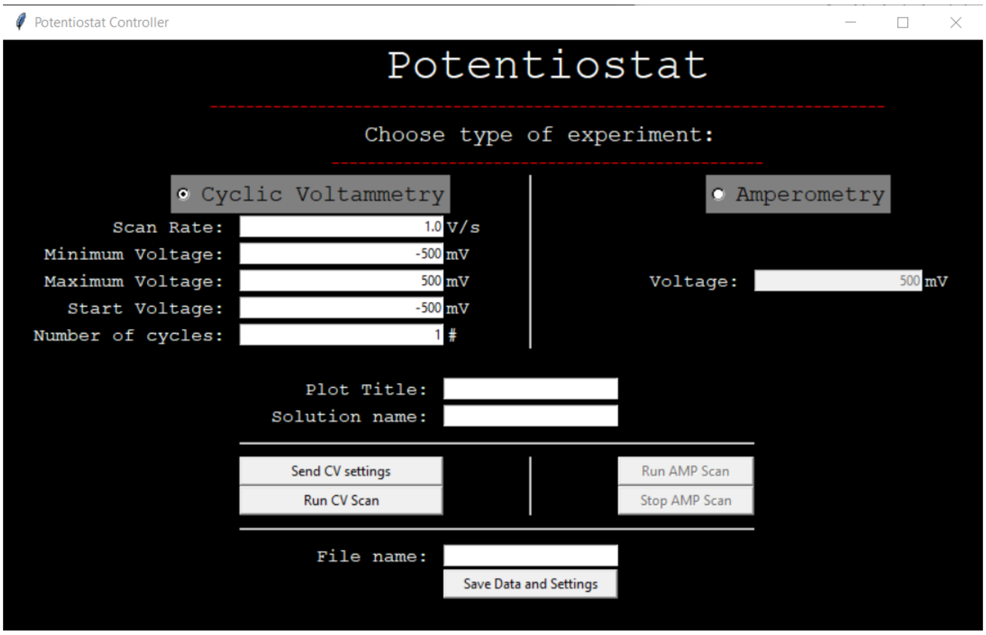
4.2.1.2 Firmware, Software, and Driver Setup

In order to operate the device, the following have to be done:

1. Obtain the PSoC5LP development kit (CY8CKIT-050) or the PSoC5LP prototyping kit (CY8KIT-059). The development kit is preferred due to the pin-out for this thesis. However, it is possible to perform some simple configurations to transfer the functionalities over to the prototyping kit.
2. Download the PSoC Creator (Cypress Semiconductors, 2020f). This is a program developed by Cypress Semiconductors specifically to configure their product's firmware.
3. Load in the project files attached in the appendix into PSoC Creator and program the device. The .hex - and .c - files have to be included.
4. Install the necessary USB drivers. This can be accomplished by downloading a free software from Zadig (Zadig, 2020).
5. Select "List all devices" in Zadig, select the "Potentiostat" device, select libusb-win32, and install the driver.
6. Install Python 3 with the packages Numpy, Matplotlib, TKinter, time, and Pandas.
7. Acquire all the python scripts provided in the appendix into the same folder on a computer.
8. Attach all the needed wires, components, and connections on the device.
9. Run GUI.py in a terminal on the computer.
10. Your device is now ready for measurements.

4.2.2 Graphical User Interface

The graphical user interface (GUI) for this thesis is developed by the author and is a simple method to communicate with the device. Figure 4.5 is a snapshot of the GUI named "Potentiostat Controller", and this section will give instructions on how to use it.



Scan Rate: 1.0 V/s
Minimum Voltage: -500 mV
Maximum Voltage: 500 mV
Start Voltage: -500 mV
Number of cycles: 1 #
Voltage: 500 mV
Plot Title:
Solution name:
File name:

Figure 4.5: Graphical user interface for the potentiostat.

4.2.2.1 Cyclic Voltammetry

To conduct a cyclic voltammetry experiment, the following configurations have to be set in "Potentiostat Controller":

1. Select the bullet option "Cyclic Voltammetry".
2. Set desired "Scan Rate", "Minimum Voltage", "Maximum Voltage", "Start Voltage" and "Number of cycles".
3. "Plot Title" is an option if it is desired to give the plot a name.
4. "Solution name" is an option if it is desired to give the plot a legend name. The operator of the device is free to choose whether it should be the name of the solution or other information desired for the legend.
5. "Send CV settings" is a button to be pressed when all the settings above are provided. All the settings will be transferred to the potentiostat.
6. When the settings are transferred to the potentiostat, the experiment may begin by pressing "Run CV Scan". A red lighted LED will be lit (if configured as explained in section 4.2.1.1), and the LCD on the potentiostat will inform the operator that an experiment is running.

7. The duration of the experiment varies with the settings transferred to the potentiostat. When the experiment is done, a plot will pop up in a new window. This plot can be saved directly as an image to the computer by the GUI provided by the matplotlib package.

4.2.2.2 Amperometry

To conduct an amperometry experiment, the following configurations have to be set in "Potentiostat Controller":

1. Select the bullet option "Amperometry".
2. Set desired applied voltage in "Voltage".
3. "Plot Title" is an option if it is desired to give the plot a name.
4. "Solution name" is an option if it is desired to give the plot a legend name. The operator of the device is free to choose whether it should be the name of the solution or other information desired for the legend.
5. "Run AMP Scan" is a button to be pressed when all the settings above are provided. The button will start the amperometry experiment, and this information will also be provided by the LCD display of the potentiostat.
6. The operator can stop the measurements by pressing the button "Stop AMP Scan". Data are transferred continuously to the computer, so the only limitation is the amount of free data memory on the computers RAM.
7. When the experiment is done, a plot will pop up in a new window. This plot can be saved directly as an image to the computer by the GUI provided by the matplotlib package.

4.2.2.3 Saving of Data

When either a cyclic voltammetry or amperometry experiment is finished and the plot has popped up, the operator can fill in the "File name" to give the saved data a name. The files will be saved locally (the same folder as the python files are stored) as a .csv-file (comma separated file).

4.2.3 Electrode Preparation

Before an experiment, it is advised to prepare the electrodes. This is accomplished by conducting an oxygen plasma treatment. The plasma treatment will clean the surface of the electrodes to remove contamination. The plasma treatment will also make the electrodes more hydrophilic, increasing the electrode's wettability. This improves the redox system's response, e.g., higher peaks in the cyclic voltammogram than a hydrophobic electrode (Yagi et al., 1999).

Chapter 5

Instrument Design and Development

This chapter will describe how the potentiostat was developed. It contains a section with an overview of the entire potentiostat, a section describing the hardware design, and a section describing the software development.

5.1 System Overview

The platform used for the potentiostat is a PSoC5LP by Cypress Semiconductors, which communicates with a computer through a USB interface (see figure 5.1 for system overview). There are two methods implemented in the device: cyclic voltammetry and amperometry. To provide for the applied voltage in the electrolytic cell, a DAC with a resolution of 12-bits and a voltage span of 4.080 V is used. The resolution of the DAC corresponds to 1 mV per bit with full voltage span utilized. An integrated transimpedance amplifier is used as a current-to-voltage converter that is connected to a Delta-Sigma ADC. As for Lopin and Lopin (2018), the precision of the ADC is configured to 12-bits with a voltage span from -2.032 V to 2.032 V. Since PSoC5LP does not provide for negative voltages, a virtual ground is constructed with an 8-bit DAC that holds a voltage at 2.032 V. This virtual ground sets the reference voltage for the transimpedance amplifier and the ADC. A timer is utilized to configure when the DAC sets a new voltage, triggered with an interrupt. At half of the timer period, an interrupt for the ADC is triggered, and the ADC samples and stores the sampled value as one signed 16-bit value. The 16-bit value is then transferred to the USB interface, which transfers the data to the computer. The transfer occurs for each measurement of the ADC.

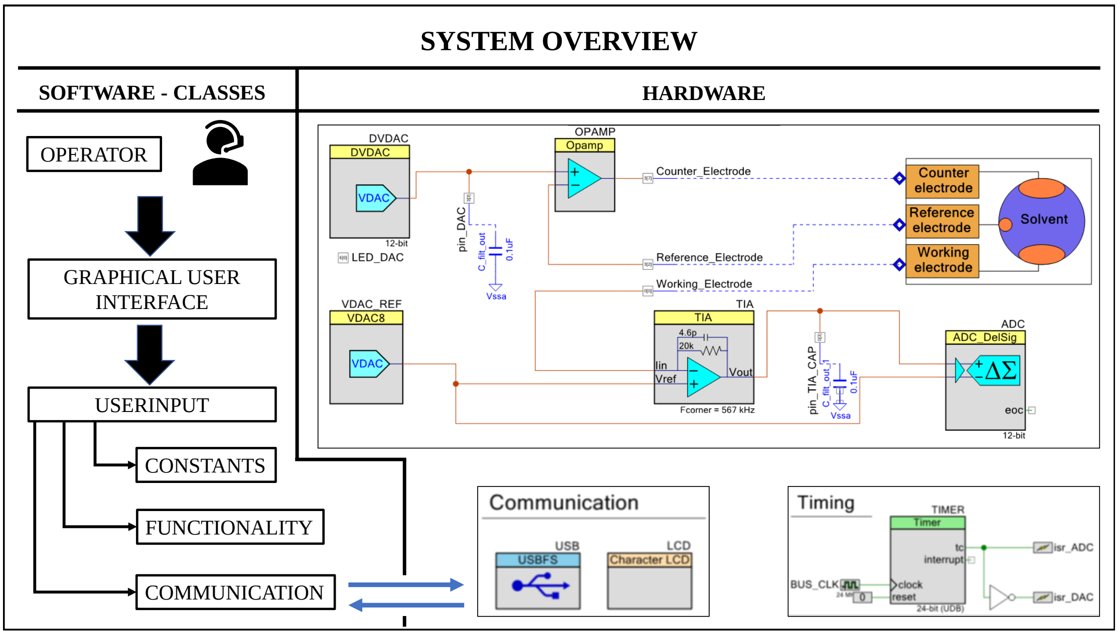


Figure 5.1: The figure illustrates a system overview for the potentiostat, both software and hardware. On the software side of the overview, the blocks' names refer to the names of the Python classes used. The hardware block is the schematic from PSoC Creator.

As figure 5.1 illustrates, the graphical user interface, on the software side of the overview, controls the entire potentiostat. An operator gives commands in the graphical user interface, communicating with a Python class named "Userinput". The "Userinput" class is connected to three other Python classes: "Constants", "Functionality" and "Communication". Together they store the values inserted by the operator, convert the values into a format understandable for the potentiostat, and communicate with the potentiostat through USB. After a scan has been completed, the potentiostat will have transferred all the measurements to the computer. The Python classes will do the necessary calculations for the operator to see plots of either a voltammogram or a current-vs-time plot, and give the option to save the measured data locally on the operator's computer.

5.2 Potentiostat - Hardware

This section will give an overview of the hardware of the potentiostat and how it operates.

5.2.1 Documentation

The PSoC5LP has extensive amounts of datasheets. PSoC Creator has the option to export a compressed datasheet of the potentiostat, where only the components utilized in the PSoC5LP are explained, and all configurations are documented (see chapter 8.5).

Throughout the hardware section there will be referred to different components on the device, and each of the components have their own datasheet. Instead of referring to the datasheet for every statement made, a list of the most important datasheets are listed below. This implies that it will be taken for granted that e.g. information about the Full Speed USB is documented in the reference provided for that component in the list below:

- Dithered Voltage Digital to Analog Converter (Cypress Semiconductors and Infineon, 2020d)
- 8-Bit Voltage Digital to Analog Converter (Cypress Semiconductors and Infineon, 2020a)
- Operational Amplifier (Cypress Semiconductors and Infineon, 2020g)
- Delta Sigma Analog to Digital Converter (Cypress Semiconductors and Infineon, 2020c)
- Trans-Impedance Amplifier (Cypress Semiconductors and Infineon, 2020i)
- Timer (Cypress Semiconductors and Infineon, 2020h)
- Interrupt (Cypress Semiconductors and Infineon, 2020f)
- Full Speed USB (Cypress Semiconductors and Infineon, 2020e)
- Character LCD (Cypress Semiconductors and Infineon, 2020b)

5.2.2 Schematic Overview

Figure 5.2 provides the schematic for the entire potentiostat. All the components are integrated into the PSoC except for the connections marked in dotted blue; these are external connections to the potentiostat. The schematic will be a reference throughout the hardware chapter.

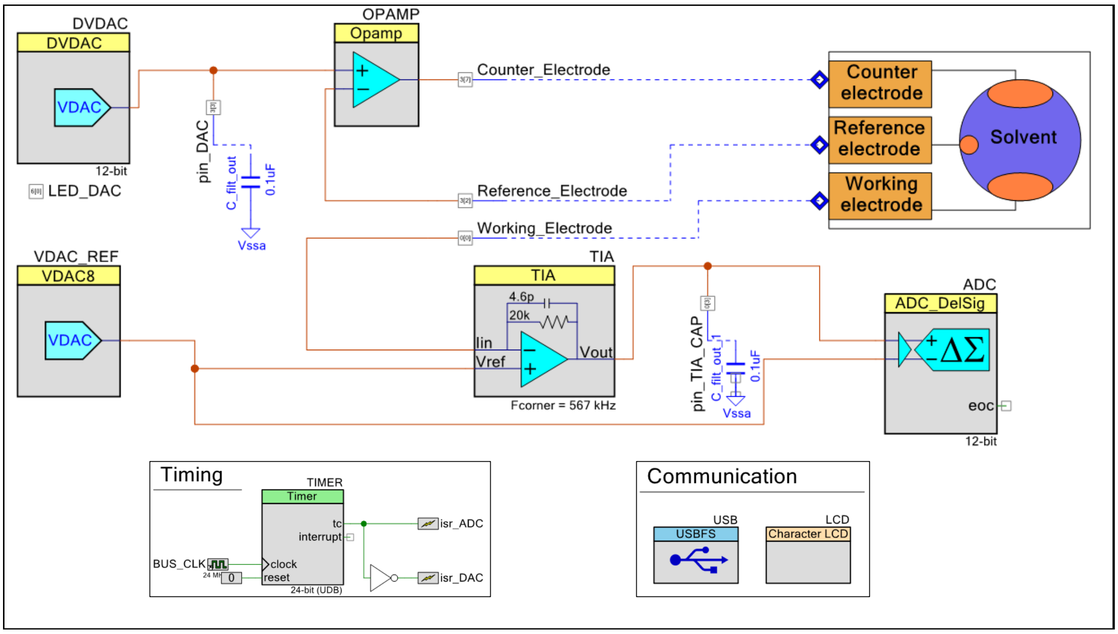


Figure 5.2: A block diagram / schematic of the potentiostat.

5.2.3 Applied Voltage

A 12-bit dithering DAC (DVDAC) generates the applied voltage for the control amplifier. The DVDAC is an 8-bit DAC, but the dithering switches the output voltage high and low systematically, which generates an average output with a 12-bit resolution. If the switching frequency is relatively high, the switching will not be noticeable on the output. An external capacitor added to the DVDAC's output smooths out the signal. As a result of the DVDAC's dithering, the output voltage is of 12-bit resolution generated with an 8-bit DAC.

The capacitor value on the output of the DAC needs to be calculated. Fortunately, PSoC Creator does the calculation if the following are provided: voltage range, resolution, and switching frequency. The potentiostat's chosen settings are the highest resolution at 12-bit, with a full voltage range of 0 V - 4.080 V and the maximum switching frequency of 250 kHz. By implementing these settings, an external capacitor of 100 μF is needed on the output of the DVDAC.

The output of the DVDAC is connected to the control amplifier where the regulation in the potentiostat happens. As explained in section 2.2.1, the difference between the applied voltage from the DVDAC and the reference electrode potential generates the necessary current at the counter electrode. The control amplifier is a standard, low powered, operational amplifier, with allocated output pins on the PCB. By utilizing the recommended pins for the control amplifier, unnecessary routing length is avoided within the PSoC, which will lead to reduced noise on the board.

5.2.4 Current Measurement

The current measured in the potentiostat flows through the working electrode. As the current has to flow through the transimpedance amplifier (TIA), the only current path available is through the integrated resistor in the TIA feedback loop. The resistor is set to 20kΩ with a parallel feedback capacitor of 4.6 pF to reduce the bandwidth to 567 kHz. Unfortunately, the TIA's integrated feedback resistor has low accuracy of -25% to +35%. This can be adjusted for in the ADC by adjusting the offset and gain offset of the current path. Another option is to use external resistors with high precision, but this is not utilized for this potentiostat to reduce the amount of necessary external components.

There is implemented an option to reduce the noise of the input of the ADC with a parallel external capacitor. This implementation is not critical for the potentiostat to operate, but it will work as a low-pass filter and reduce high frequency transients.

A virtual ground reference has been added to the design to operate with negative voltages in the electrolytic cell since the PSoC5LP does not produce negative voltages. The electrolytic cell will not see the analog ground, but only the virtual ground since the TIA and ADC have the virtual ground as its reference voltage. The reference voltage is set to 2.032 V.

The ADC samples the current flowing through the electrodes with help from the TIA that converts the current into a voltage that the ADC can measure. As for the DAC, the ADC has a resolution of 12-bit with a voltage span of 4.096 V, which implies a voltage resolution of 1 mV. Due to the virtual ground, the ADC can measure from -2.048 V to +2.048 V. The Delta-Sigma ADC has a conversion rate of 30000 samples per second, as for Lopin and Lopin (2018). This should be an appropriate conversion rate since the Delta-Sigma ADC depends on having oversampling of the signal for it to work. This potentiostat's highest expected frequency is 1 kHz (maximum for the triangular signal during cyclic voltammetry), where 30 kHz sampling frequency should be well beyond the minimum.

5.2.5 Timing

In order to control the scan rate of the potentiostat, an integrated, configurable timer is utilized. The timer is set to have a resolution of 24-bits with a clock input of 24 MHz. This gives the timer limitations with a minimum period of 83.333 ns and a maximum period of 699.051 ms, with a precision of 15 ns. The timer counts clock pulses from the 24 MHz clock and enables its "tc"-pin when the configured number of counts is achieved. This "tc"-pin is connected to two interrupts; one for the DAC and one for the ADC. The interrupt for the ADC will occur at a rising edge from the timer, while the interrupt for the DAC is inverse; it will enable the DAC interrupt at a falling edge. This implies that the ADC will measure the current flowing through the working electrode one half period after the DAC has set a new voltage during a cyclic voltammetry experiment.

5.2.6 Communication and Display

The communication interface chosen for the potentiostat is the same as for Lopin and Lopin (2018), Full Speed Universal Serial Bus (USBFS). This communication interface has lots of possible configurations and options. Since the work by Lopin and Lopin (2018) already were functioning, the same configurations were used for this potentiostat. There are three out of eight endpoints utilized:

- **EP0** - control endpoint for the interface to communicate with a computer
- **EP1** - endpoint to transfer data from the potentiostat to the computer
- **EP2** - endpoint to transfer commands from the computer to the potentiostat

EP1 has a maximum package size to send of 64 bytes with a maximum rate to send of 1 bulk package every 1 ms. This endpoint is configured to transfer the data collected by the potentiostat to the computer. **EP2** has a maximum package size to transfer of 32 bytes with a maximum rate to send of 1 package every 10 ms. This endpoint is configured to receive commands from the computer, and is therefore an interrupt endpoint while **EP2** is a bulk endpoint.

In addition to the USBFS, the potentiostat has an LCD display mounted to the PSoC5LP. This display is mostly used for development but is a versatile configuration where information may be displayed during experiments.

5.3 Potentiostat - Firmware

This section describes the firmware of the potentiostat. Each component in the potentiostat has its own application guide in its datasheet. A brief explanation of how the applications are utilized will be explained. In addition, the cyclic voltammetry and amperometry firmware will be explained. All firmware code can be found in chapter 8.3.

5.3.1 Overview

Figure 5.3 visualize how the information flow of the potentiostat is working together. The "main.c" file is where the input from the computer is enabled. All commands from the computer are in the form of a capital letter followed by initialization values for either amperometry or cyclic voltammetry. Each capital letter corresponds to its own functionality, checked for each iteration of the main loop. If a capital letter is detected, the functionality of the function will start.

There are six possible inputs for the potentiostat:

- *CV_TIMER* - Sets timer period for the timer component
- *CV_NO_CYCLES* - Sets the number of cycles for a cyclic voltammetry scan. The number is stored as a variable, but is used in the DAC interrupt routine.
- *CV_DEFINE_RANGE* Sets minimum-, maximum- and start- voltage for a cyclic voltammetry scan. The values are stored as variables, and are used in the DAC interrupt routine.
- *CV_RUN* - Enables a cyclic voltammetry scan. This will enable all of the necessary components for a scan and initialize the necessary variables.
- *AMP_RUN* - Enables an amperometry scan. This will enable all of the necessary components for a scan and initialize the necessary variables.

- *AMP_STOP* - Disables an amperometry scan. This will disable all of the operating components and send the final measured data through the USB interface.

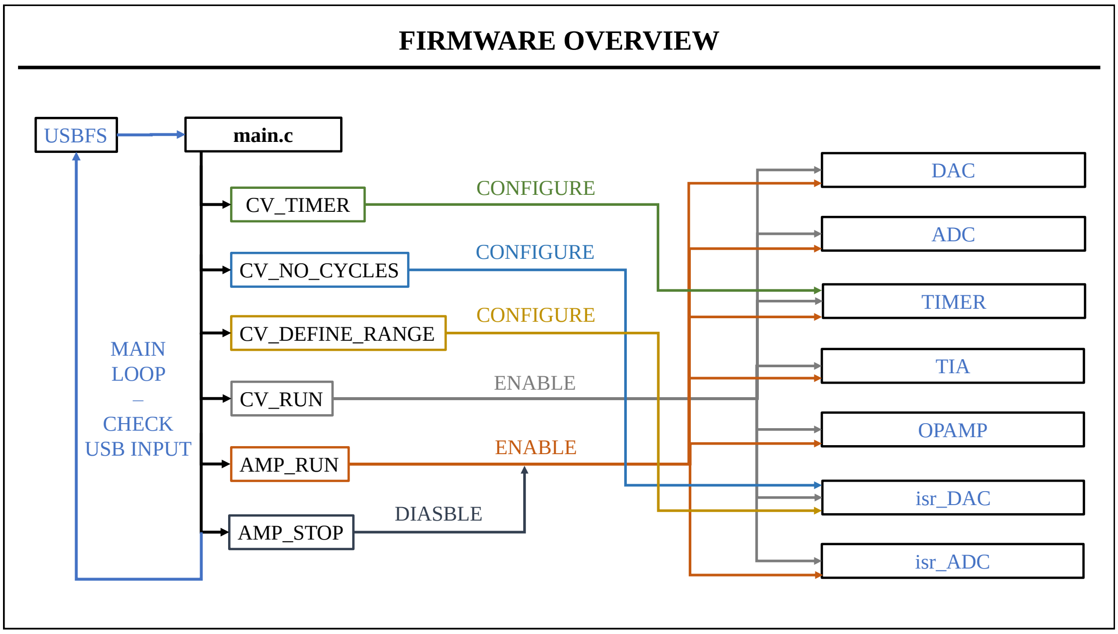


Figure 5.3: An overview of the firmware of the potentiostat. "main.c" has a main loop that checks the input of the USB interface for each iteration. If there is an input, one of the colored blocks will initialize. The initialization involves a configuration of a component (black boxes), an enable signal for one or several components, or a disable signal for one or several components.

5.3.2 Communication During Scans

One of the major differences from the work by Lopin and Lopin (2018) is that all data transfer during electroanalytical scans is continuous. This configuration change was done in order to run scans for an increased duration. Another benefit was that the potentiostat's memory never would be filled up since the measured data would be overwritten after a data transfer had finished.

By implementing that configuration change, a limitation of the potentiostat occurred. With bulk transfer as the USB transfer type, the data transfer rate is limited to 1 kHz.

The USBFS can only send data of UINT8 (8-bit unsigned integer), while measured data are INT16 (16-bit signed integer). This implies that all measured data are converted into two UINT8 instead of one INT16 and then converted back on the computer. The conversion is a function implemented in the "usb_protocol.c" (see chapter 8.3.1.3).

5.3.3 Cyclic Voltammetry

To run a cyclic voltammetry scan, the following will be initialized:

1. Set the scan rate of the scan by transferring the period of the timer through the USB interface. This is done by utilizing the function "CV_TIMER". The function will write to the timer component what the period is with a command documented in the application user guide of the component.

2. Set the number of cycles by utilizing the function "CV_NO_CYCLES". The number of cycles will be stored as a global variable used by the DAC interrupt routine.

3. Set the minimum-, maximum- and start- voltage by utilizing the function "CV_DEFINE_RANGE". The function will store the values as global variables used by the DAC interrupt routine. In addition, the function does a calculation to check whether the next voltage after the start voltage should be higher or lower than the start voltage. This is to initialize a variable (UP and DOWN) used in the DAC interrupt routine.

4. The potentiostat is now ready to begin the cyclic voltammetry scan. This is done by utilizing the command "CV_RUN". This function will first initialize variables used in the DAC interrupt routine, then enable all the hardware through the function "helper_HardwareWakeup()", then set the start voltage for the DVDAC and let it stabilize for 70 ms. When all hardware is ready, the ADC will start its conversion, and the first measurement will be done and transferred directly to the computer. The rest of the cycle will begin right after this by enabling the DAC and the ADC's interrupt routines.

The following will describe how the cyclic voltammetry scan sets new voltages for the DVDAC and how the potentiostat knows when the scan is complete. This is visualized in the code snippet below.

When the timer component enables the DAC interrupt, the "dacInterrupt" is enabled. The first thing that happens is that the interrupt releases the interrupt from the timer component by the "ReadStatusRegister()" function. It will then check the "index_value", which is a value that sets the voltage to the DVDAC, whether the next value should be iterated higher or lower than the previous value. This is where the "UP" and "DOWN" variables are configured with a TRUE/-FALSE statement for a higher or lower value. Another routine will, after that, check if one entire cycle is complete. If the number of cycles has reached the maximum number of cycles for the scan, the hardware components and firmware configuration will be set to sleep (disabled). If not, another IF-test will check if the "index_value" for the next iteration should start increasing or decreasing by configuring the "UP" and "DOWN" variables. Finally, the "index_value" is sent to the DVDAC that sets the next voltage in the scan.

```
CY_ISR(dacInterrupt) {
    TIMER_ReadStatusRegister();             // Release
   dacInterrupt
    /* Define next voltage value */
    if (direction == UP) { index_value += step_size; }
    else { index_value -= step_size; }

    /* Check if one cyclus is done */
    if (index_value == start_value) {           // One cycle
   completed
        cycles_index += 1;                      // Iterate cycle
   index
        if (cycles_index == number_of_cycles) { // CV complete
            isr_ADC_Disable();                  // Disable ADC
   interrupt
            isr_DAC_Disable();                  // Disable ADC
   interrupt
            helper_HardwareSleep();             // Set hardware to
    sleep mode
            data_usb16 = 49152;                 // Determintaion
   value for ADC_array
            USB_Export_Data(data_usb16);        // Transfer last
   array
            helper_LCD_write0("CV DONE");       // Write to LCD
            helper_LCD_clear1();                // Clear line two
   of LCD
            LED_DAC_Write(0);                   // LED_DAC off
        }
    }

    /* Check if direction should change */
    if (index_value >= max_value) {
        direction = DOWN;
    }
    if (index_value <= min_value) {
```

```
        direction = UP;
    }

    /* Set next value to DAC*/
    DVDAC_SetValue(index_value);
}
```

5.3.4 Amperometry

An amperometry scan is easier implemented than the cyclic voltammetry scan. The user does not have the option to set the ADC sampling rate; this is preconfigured in the potentiostat and is set to an ADC measurement every 25th ms. An operator only needs to send the desired voltage level for the amperometry to start. The firmware is already configured to start all of the necessary hardware components and give the DVDAC time to stabilize. After that, every 25th ms, data is transferred to the computer in the form of double UINT8.

Another function is implemented for the operator to stop the amperometry scan. This command will shut all of the hardware components off (disable them), and the potentiostat is ready for a new scan.

5.4 Potentiostat - Software

This section will introduce how the software is implemented and how the different classes of the software communicate with each other. The software is deliberately written in Python to make the potentiostat available for everyone since Python is open-source and free of charge. Python 3 is the version used for this thesis. All of the software code can be found in chapter 8.4.

For all of the code produced in this thesis, there has been an effort to make the code as simple as possible and document each function in the scripts. Some of the code will be described in the thesis, but the rest have documentation in the scripts found in the appendix.

Most of the code has an output to the terminal as a confirmation that a command has been conducted. Several tests in the software will catch an error and write an error report to the terminal. There are boundaries for the user inputs so that the user does not send settings to the potentiostat out of bounds.

5.4.1 Software Overview

Figure 5.4 gives an overview of how the software classes and functions work together. Everything marked with green color are functions used within amperometry, everything marked with blue color are functions used within cyclic voltammetry, and everything marked with black color are functions used both by cyclic voltammetry and amperometry, or a general function.

These classes will have their own sub-section within this chapter, where its functionalities are described.

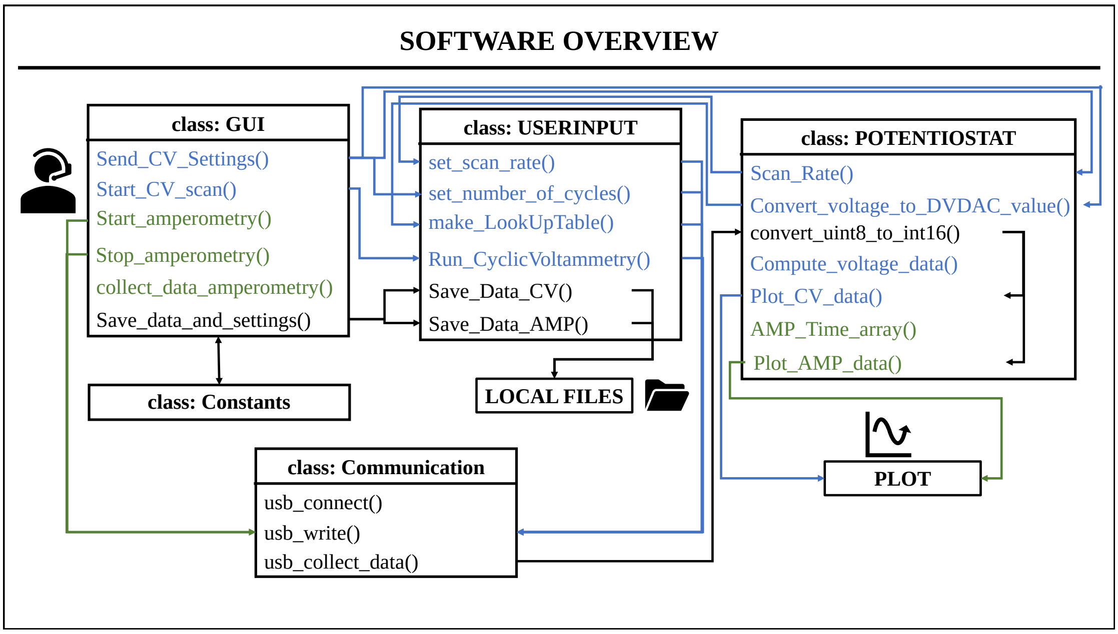


Figure 5.4: An overview of the software of the potentiostat. Blue corresponds to cyclic voltammetry, green corresponds to amperometry, black corresponds to both cyclic voltammetry and amperometry functions or general settings.

5.4.2 Communication

The communication interface between the potentiostat and the computer is USB. Lopin and Lopin (2018) used PyUSB for their potentiostat, and the same is used for this potentiostat. PyUSB is an open-source package used for communication over USB with Python. There exist other options, but PyUSB is well documented online with well described forums as well.

The setup of a USB interface with Python is quite similar to other setups. This will not be further described here, but the code is available in the appendix.

There are three functions within the communication class that are of importance:

- *usb_connect()* - is the function that establishes the connection with the potentiostat and initializes the input- and output descriptors used to send and receive data. The configurations done in the firmware of the potentiostat matches the descriptors in this function.

- *usb_write()* - is the function that transfers commands from the computer to the potentiostat. The commands sent are in the form of strings. This is a configuration constructed by PyUSB, which means that the potentiostat receives it as strings. The potentiostat has implemented functions in the firmware that converts from string to the wanted type.

- *usb_collect data()* - is the function that receives data from the potentiostat. This is configured to operate continuously. The potentiostat transfers a double UINT8, which is converted back to INT16 in another class, and then this function is ready to receive new data straight afterward.

5.4.3 Constants

The "Constants" class is developed to make the software as general as possible. All the constant parameters are stored here. In addition, all of the messages written to the terminal are stored here. When a scan has finished, the data from the measurements will be stored in this class to ensure that all the classes have it available. This has been an issue throughout software development since some of the classes can not communicate. The reason is that one class will try to import another, and then that class will try to import the other class, and then an eternal loop will be established. Python notices this and will write an error message and terminate the scripts running. The "Constants" class is developed to come around this issue and is doing its intended job.

Another reason for the development of the class is to make it easier to change the constant parameters. E.g., a change of the resolution of the ADC should be inserted in this class.

5.4.4 Graphical User Interface

The GUI of the potentiostat is constructed in this class. Lopin and Lopin (2018) used the same package to do this, "tkinter". This is also an open-source package with good documentation. However, it is difficult to get an overview of the code since the graphics are written as code and coordinates. This is also the reason why several classes have been established; to get a better overview of the entire software code of the potentiostat.

There will not be a walk-through of this code, but a general description is provided. All of the user inputs are caught by a variable where it is stored or used. Each variable is then inserted into their given function, bringing the settings from the user one step closer to sending it to the potentiostat. All of the code is sent to another class for further processing, with one exception; the amperometry settings.

The amperometry code is sent directly from this class to the potentiostat due to issues with the termination of the amperometry scan. The "tkinter" window has an update function so that the operator can press the stop button. For this to happen, there is a concise time window for the Python script to process the information it has been given. A solution to the problem was to insert the amperometry settings within the GUI class, and this solution is working.

5.4.5 Userinput

The "Userinput" class is where most user inputs are processed before they are sent to the potentiostat. The most important functions are listed below:

- *set_Scan_rate()* - acquire the scan rate from the user and converts it into the period of the timer in the potentiostat. This is done by first sending it to the functionality class that converts it and sends the converted value back. The value is then zero padded (to make sure the potentiostat can interpret the value) and transferred to the communication class that sends the potentiostat's command.

- *set_number_of_cycles()* - acquire the number of cycles from the user, zero pads it, and send the command to the potentiostat.

- *make_LookUpTable()* - acquire the minimum-, maximum- and start- value of the user. The values are sent to the functionality class to convert the voltages to ensure the virtual ground is accounted for. The values are then zero padded and sent to the potentiostat.

- *run_CyclicVoltammetry()* - sends a command to the potentiostat that the cyclic voltammetry shall begin. The function then starts a loop to receive data continuously from the potentiostat. This loop will only terminate when the potentiostat has ended its scan and transferred a determination value to the computer. All data are converted into current, by utilizing Ohms law, with the received voltage and the known 20 kΩ as the resistance. To reduce noise, a filtering function is applied to all of the received data (5% moving average); a post scan low pass filtering. The applied voltages from the scan are re-constructed with a function, and the unfiltered data is plotted.

- *Save_Data_CV()* - the data is stored in the same folder as the scripts are stored. The user has the option to give the file a name, which is inserted in this function. The applied voltage data, measured current data, and the filtered measured current data are stored in a CSV-file. To order the data in columns, a package in Python called Pandas is used.

- *Save_Data_AMP()* - the data is stored in the same folder as the scripts are stored. The user has the option to give the file a name, which is inserted in this function. The time data and the measured current data are stored in a CSV-file. To order the data in columns, a package in Python called Pandas is used.

5.4.6 Functionality

The "Functionality" class (or the "Potentiostat" class, a difference between filename and class name) is where most of the calculations of the user inputs are done before they are sent to the potentiostat. The most important functions are listed below:

- *Scan_Rate()* - is the function that converts the scan rate from Volts/seconds into the period of the timer in the potentiostat. There are three variables necessary to calculate the period: the step size of the DAC, the clock frequency for the timer, and the scan rate. The period is calculated as follows: $P = (step \cdot clock/scanRate) - 1$, where the step size is 1 mV (since the voltage span is 4.080 V with 12-bits resolution), the clock frequency is 48 MHz (firmware configured), and the scan rate is inputted by the user. One is subtracted from the calculation, as described by the datasheet of the timer. The period value is then exported to the timer.

- *Convert_voltage_to_DVDAC_value()* - is the function that takes into account the virtual ground (2.032 V) from the given minimum- and maximum voltages given by the user. This implies that the DC-level of the applied voltage will be increased with the analog ground as a reference.

- *convert_uint8_to_int16()* - is the function that converts the two UINT8 values to INT16 after the computer have received the data.
- *Plot_CV_data()* - is the function that plots the measured current versus the applied voltage and makes a voltammogram. This is accomplished by utilizing the "matplotlib" package.
- *Plot_AMP_data()* - is the function that plots the measured current versus the time for the amperometry scan. This is accomplished by utilizing the "matplotlib" package.

Chapter 6

Results

This chapter will present the results from measurements conducted with the potentiostat developed in this thesis. The potentiostat made by Lopin and Lopin (2018) will be used as a reference.

Cyclic voltammetry with Ferri-/Ferrocyanide has been preformed, and results were obtained. After a few measurements, the electrodes were damaged. Unfortunately, there was not enough time to perform more experiments after acquiring new electrodes. The measurements that were obtained were only from cyclic voltammetry at one scan rate, and non from amperometry.

As a start of this thesis, the potentiostat developed by Lopin and Lopin (2018) was tested. The amperometry measurements on dopamine will be provided as the only results from amperometry.

6.1 Cyclic voltammetry

This section presents the measurements conducted with cyclic voltammetry. Table 6.1 has all the settings for the potentiostats listed. These settings will be used for all the measurements obtained unless otherwise are informed for each measurement.

In the following voltammograms, "Ref" refers to the reference potentiostat by Lopin and Lopin (2018), "Raw" refers to the potentiostat developed in this thesis, and "Average" refers to a moving average of 5% of the "Raw" data. The "Average" data is added to reduce noise and works as a low-pass filter.

In the voltammograms, there are provided additional information such as the scan rate, the number of cycles (only one cycle for the reference potentiostat since the number of cycles is not an option for that potentiostat), and a close-up area of the origin. The close-up is added for easier visualization of the noise from the measurements.

Before the measurements were conducted, the electrodes had oxygen plasma treatment to remove contamination on the electrodes' surface. It should be noted that the plasma treatment was only applied before the start of the first measurement and not in between each measurement. This implies that the electrodes became more hydrophobic for each measurement conducted. The potentiostat measurement and the reference potentiostat measurement were done consecutively to minimize the risk of contamination and change of the electrodes' wettability. After each measurement, the electrodes were cleaned with destilled water. Fine paper was used to dry off the water after cleaning.

Settings for Cyclic Voltammetry		
Scan rate:	50 mV/s	
Minimum voltage:	-500 mV	
Maximum voltage:	500 mV	
Starting voltage:	-500 mV	
Number of cycles:	**Reference potentiostat:**	always 1 cycle
	Potentiostat:	presented for each measurement

Table 6.1: Settings of the potentiostats for cyclic voltammetry measurements.

6.1.1 Ferri-/Ferrocyanide 1mM

All the following measurements have a solution of 1 mM Ferri-/Ferrocyanide on the electrodes. PBS was mixed with the Ferri-/Ferrocyanide as a buffer.

In addition to the voltammograms of the raw data, average data, and the reference data, additional plots with "Raw x factor" are provided. These measurements are added due to a wrong configuration in the potentiostat and will be further explained in the discussion.

6.1.1.1 Measurement - 1 Cycle - Scan Rate 50 mV/s

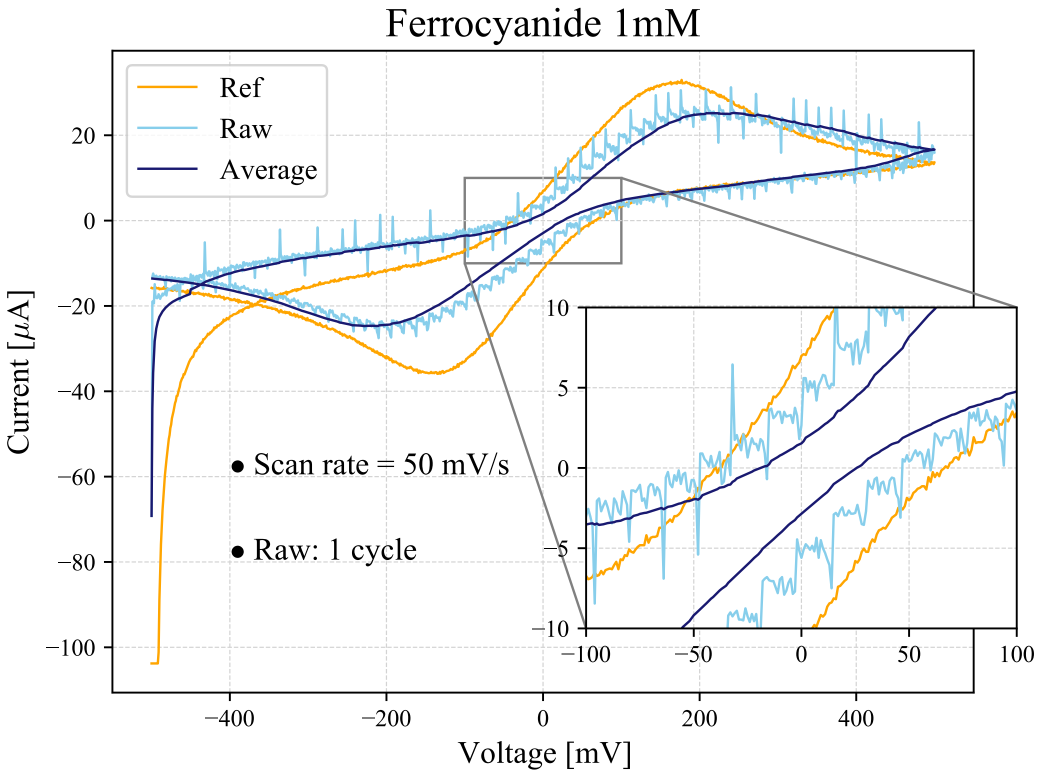


Figure 6.1: Cyclic voltammogram of 1 mM Ferri-/Ferrocyanide with a scan rate of 50 mV/s, 1 cycle. *Ref* is the potentiostat by Lopin and Lopin (2018), *Raw* is the measurements from the potentiostat from this thesis, *Average* is a moving average of 5% of the *raw* data.

6.1.1.2 Measurement Corrected - 1 Cycle - Scan Rate 50 mV/s

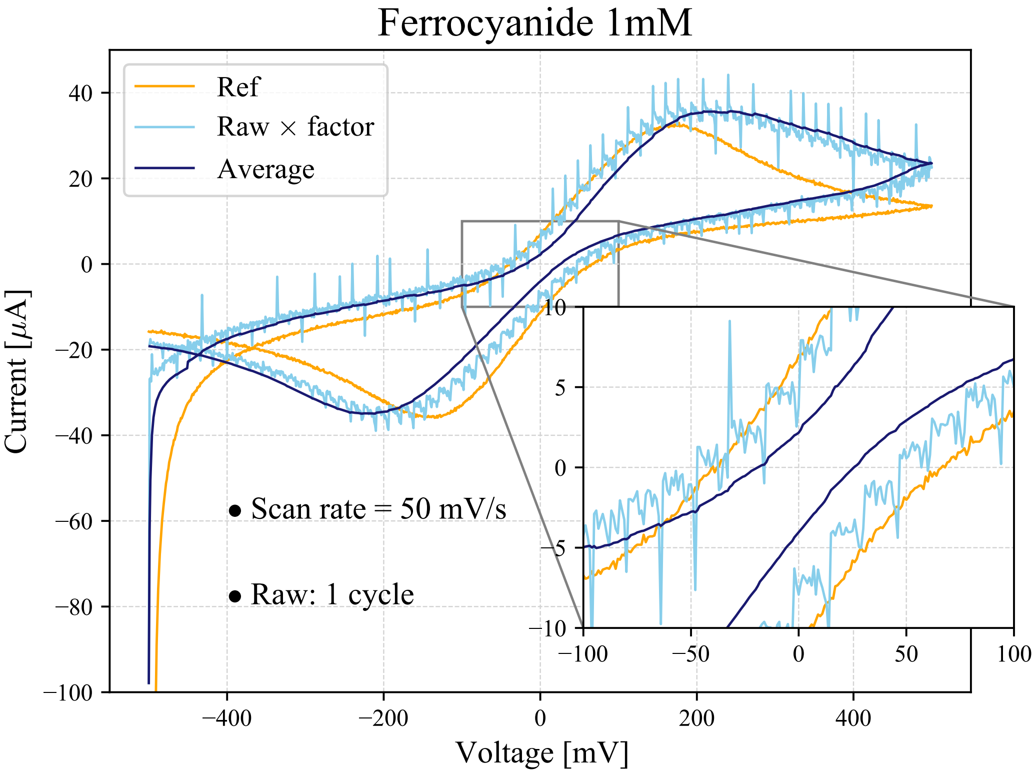


Figure 6.2: Cyclic voltammogram of 1 mM Ferri-/Ferrocyanide with a scan rate of 50 mV/s, 1 cycle. *Ref* is the potentiostat by Lopin and Lopin (2018), *Raw x factor* is the corrected measurements with the potentiostat from this thesis, *Average* is a moving average of 5% of the *raw* data.

6.1.1.3 Measurement - 5 Cycles - Scan Rate 50 mV/s

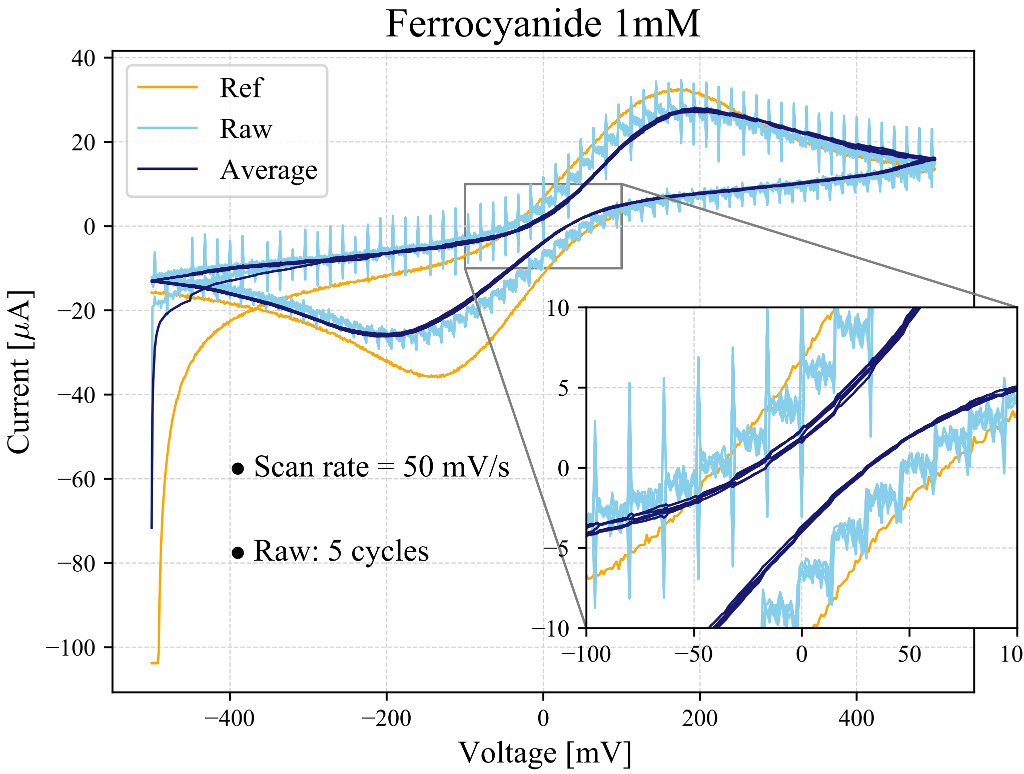


Figure 6.3: Cyclic voltammogram of 1 mM Ferri-/Ferrocyanide with a scan rate of 50 mV/s.5 cycles for the potentiostat in this thesis, 1 cycle for the reference. *Ref* is the potentiostat by Lopin and Lopin (2018), *Raw* is the measurements from the potentiostat from this thesis, *Average* is a moving average of 5% of the *raw* data.

6.1.1.4 Measurement Corrected - 5 Cycles - Scan Rate 50 mV/s

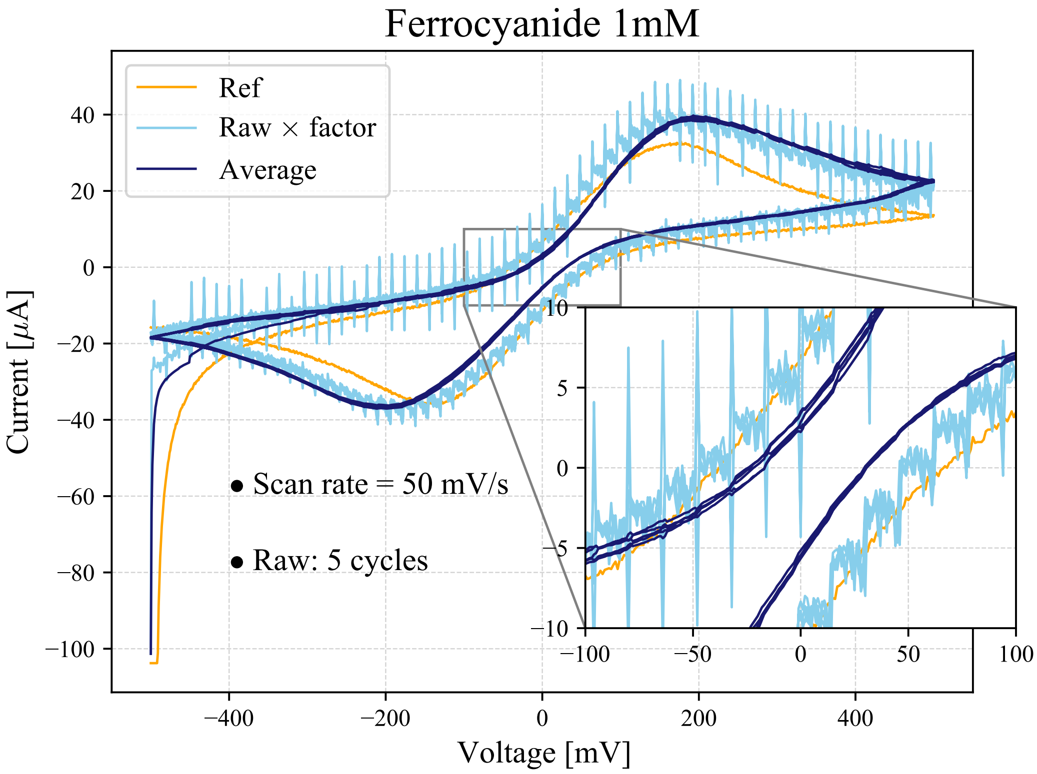


Figure 6.4: Cyclic voltammogram of 1 mM Ferri-/Ferrocyanide with a scan rate of 50 mV/s. 5 cycles for the potentiostat in this thesis, 1 cycle for the reference. *Ref* is the potentiostat by Lopin and Lopin (2018), *Raw x factor* is the corrected measurements with the potentiostat from this thesis, *Average* is a moving average of 5% of the *raw* data.

6.2 Amperometry

The results from amperometry are obtained with the potentiostat by Lopin and Lopin (2018). These measurements were obtained early in the process of this thesis before the potentiostat developed in this thesis was ready for measurements.

Figure 6.5 displays the result from an amperometry experiment. Initially, the electrodes only contained a PBS buffer. The solution applied was 1 mM dopamine, where approximately 20 μL was added to the electrode with a pipette every 8th second for 62 seconds. At the beginning of the measurements, the working electrode had 16 seconds to stabilize. The applied voltage for the experiment was 350 mV.

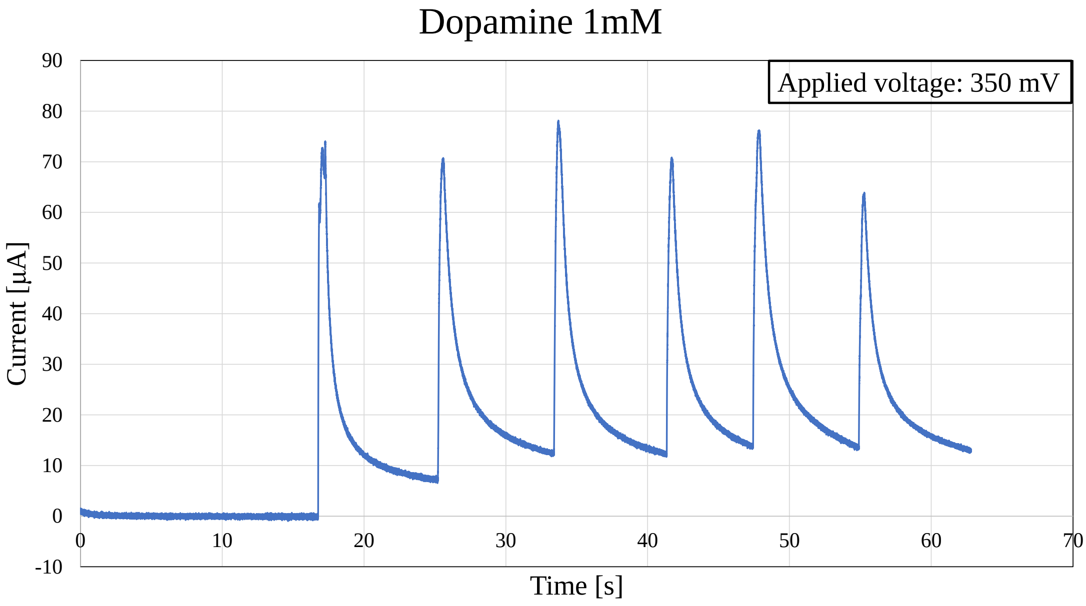


Figure 6.5: Amperometry measurement of 1 mM dopamine. 20 μL were applied every 8th second for 62 seconds. 350 mV of applied voltage were provided by the potentiostat.

Chapter 7

Discussion

This section will discuss the results presented in chapter 6 and discuss how this potentiostat behaves compared to the potentiostat developed by Lopin and Lopin (2018). The potentiostat developed in this thesis will be denoted "the Potentiostat", while the potentiostat developed by Lopin and Lopin (2018) will be denoted "the Reference Potentiostat" throughout this chapter.

7.1 Results - Cyclic Voltammetry

As mentioned in chapter 6, the electrodes used in the experimental setup for measurements were damaged and were unusable after only a few measurements with cyclic voltammetry. Therefore, there are limited data obtained from the potentiostats during measurements. Nevertheless, there are some observations from the results that will be discussed in the following subsections:

1. It is very noticeable that the Potentiostat had issues with noise. In the plots from cyclic voltammetry (figure 6.1 and 6.3), the Reference Potentiostat had a significantly more stable response compared to the Potentiostat.

2. In figure 6.1 and 6.3, the Potentiostat and the Reference Potentiostat have different shapes; their peaks are at different current levels, the derivative of their slopes are different and their peaks are at different voltages levels. In the close-up plots, it is visualized that the potentiostats have approximately even distance from the origin (apart from the Potentiostats rolling mean plot).

7.1.1 Noise

The source of the noise in the Potentiostat was for a long time a mystery during the work of this thesis, but after the measurements presented in chapter 6 were obtained, a probable source of the noise was discovered. In chapter 5.2.3, the dithering of the DVDAC was explained. A capacitor mounted on the output of the DVDAC was inserted to low-pass filter the dithering switching noise. This functionality was tested (before measurements were conducted on an electrolytic cell) by measuring the output of the DVDAC directly with an oscilloscope. The output behaved as expected, with no significant noise observed on the oscilloscope. When the control amplifier was inserted into the schematic, the error occurred; the capacitor was placed directly on the output pin for the working electrode (see figure 7.1). The consequence of the error was that the control amplifier subtracted its inputs before low-pass filtering the dithering of the DVDAC.

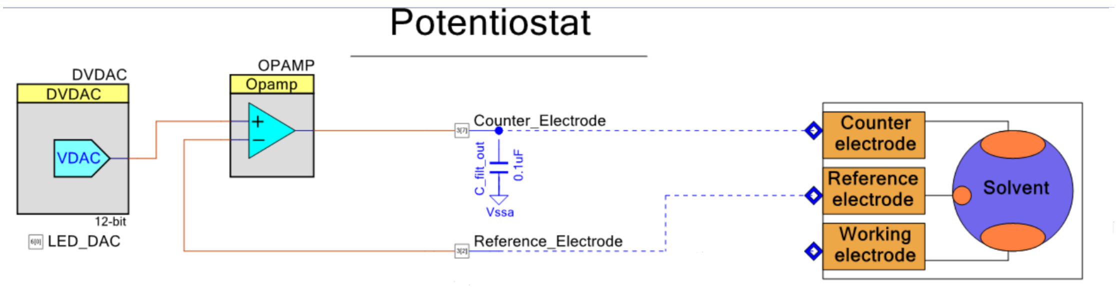


Figure 7.1: Snippet of the schematic of the potentiostat. The output capacitor of the DVDAC is wrongly placed causing switching noise on the working electrode.

To test if the possible noise source had been found, the Potentiostat ran a cyclic voltammetry experiment without any electrodes connected. Then, the voltage over the working electrode's output and the analog ground was measured with an oscilloscope. Figure 7.2 is an image of the result from the measurement with the capacitor for the DVDAC misplaced in the schematic. The image shows instability and an average frequency at 56 Hz. This frequency has not yet been mentioned, but is close to the measured main frequency of the noise in figure 6.1 and 6.3 at 62.5 Hz. This implies that the output of the control amplifier generates an unstable, low frequency.

Figure 7.3 is an image of the result from the measurement with the capacitor for the DVDAC correctly placed in the schematic. The result is a stable signal implying that the modification should impact the stability of the Potentiostat. Unfortunately, there was no time to test the impact of this modification on an electrolytic cell. The modification is presented in chapter 5 to make sure the future use of this work can be reproduced correctly.

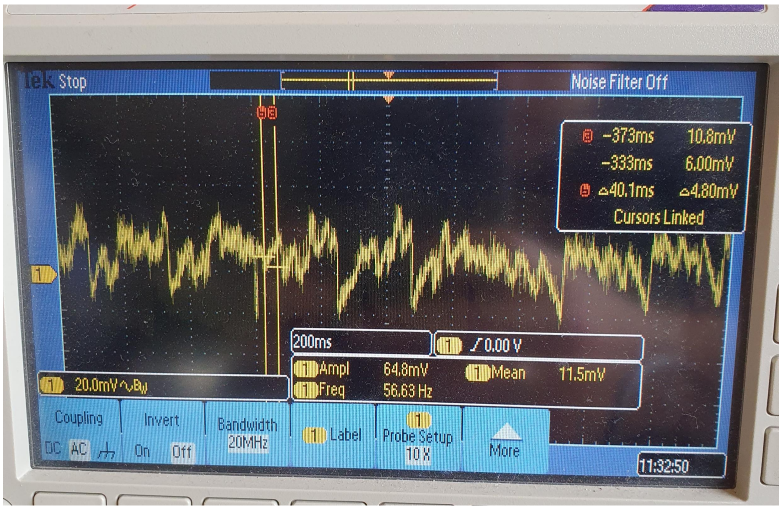


Figure 7.2: Picture of oscilloscope during an AC analysis of the counter electrode vs. analog ground. A 100 nF capacitor is mounted directly to the counter electrode output of the potentiostat.

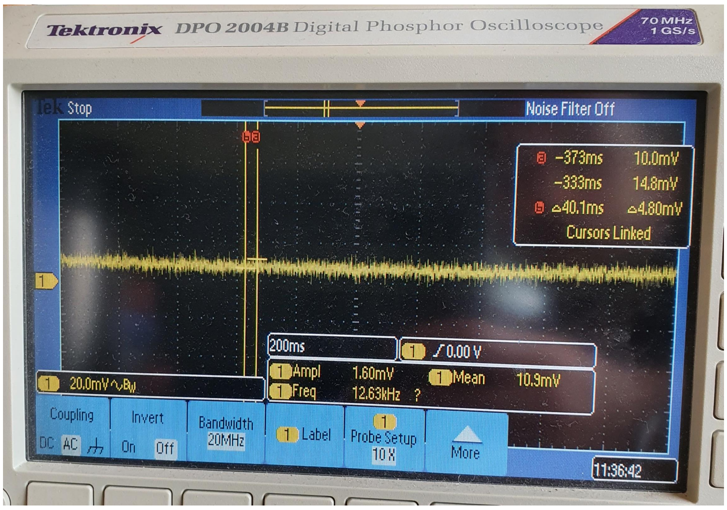


Figure 7.3: Picture of oscilloscope during an AC analysis of the counter electrode vs. analog ground. A 100 nF capacitor is mounted between the DAC and the control amplifier vs analog ground.

7.1.2 Voltammogram Shape

The voltammograms in figure 6.1 and 6.3 shows that the current peaks are at different levels for the potentiostats, that the derivatives of the slopes are different, and that the current peaks are at different voltage levels. Since both of the potentiostats are based on the same platform and have implemented the same functionality, this result was unexpected. Again, an error in the Potentiostat was found after these results were gathered. The Python function that calculates the period of the timer component was implemented incorrectly. The impact of this error is an applied scan rate that deviates from what the operator inserts in the GUI.

The correct calculation for period of the timer with the scan rate as input, is what was provided in chapter 5.4.6:

$$P = \frac{step \cdot clock}{scanRate} - 1 \tag{7.1}$$

where P is the period sent to the Potentiostat, *step* is the minimum voltage step of the DVDAC (1 mV), *clock* is the clock frequency for the timer component, *scanRate* is the scan rate provided by the operator and the subtracted one is an implementation instructed by the datasheet of the timer component.

The results in figure 6.1 and 6.3 had the following calculated period for the timer component:

$$P = \frac{step \cdot clock}{2 \cdot scanRate} - 1 \tag{7.2}$$

The division by two makes the scan rate twice of its intended rate, and was unfortunately not discovered in advance of the measurements. A doubling of the scan rate have direct impact on the current peaks in the voltammogram (as explained in chapter 4.1.1.2) as follows:

$$peak \propto \sqrt{scanRate} \tag{7.3}$$

Because of this discovery, the current data of the plots in figure 6.1 and 6.3 were all multiplied by $\sqrt{2}$. Since the current peaks in the voltammogram should behave linearly with the scan rate as formula 7.3 states, the voltammogram of the Potentiostat should be more similar to the voltammogram of the Reference Potentiostat. The results are plotted in figure 6.2 and 6.4.

Figure 6.2 shows an improvement in the level of the current peak. The shape of the voltammogram for the Potentiostat is also improved overall except for at positive voltages. Figure 6.4 has some of the same results but overshoots the current peak of the Reference Potentiostat. The results imply that the scan rate formula

7.1 should have been the implementation used while conducting measurements. As an additional test, the scan rate was measured with an oscilloscope in retrospect, which verified this finding.

The last observation mentioned in the introduction to the discussion was that the voltammogram peaks had different voltage levels for the two potentiostats. One reason for this might be that the Potentiostat does not have a calibration routine. Lopin and Lopin (2018) have a calibration routine for their ADC that can adjust for gain- and offset- error. As a consequence of not calibrating the Potentiostat, the offset errors noted in the datasheets of the PSoC5LP will influence the behavior of the signals. This can further lead to differences in measured currents for the potentiostats.

7.2 Comparison of the Potentiostats

Firstly it should be noted that the Reference Potentiostat is not commercial. This implies that to verify the accuracy of the Potentiostat, the results in the article by Lopin and Lopin (2018) have to be examined. As they write in their article, the OPAMPs in the PSoC5LP have a noise of $45nV/\sqrt{Hz}$ with an offset uncertainty of 12 mV. These uncertainties are also in the Potentiostat since it is based on the same platform, which means that the Potentiostat's uncertainties are approximately 12 mV as well.

One of the improvements added to the Potentiostat compared to the Reference Potentiostat was that it could transfer data continuously while conducting scans. This functionality worked as expected. With this functionality, it is theoretically possible to conduct measurements for as long as the operator desires. There were, however, some communication issues with the USB interface. Sporadically, the communication between the Potentiostat and the computer stopped, and the GUI and Potentiostat had to be rebooted. This has not been experienced with the Reference Potentiostat, but others at the Department of Physics (University of Oslo) have had the same experience. A probable cause for this problem is that the driver installed with Zadig is unstable. Other drivers have been tested by the university and were functional. Lopin and Lopin (2018) utilized the same driver as was used for the Potentiostat, but they had extensive amounts of tests and error corrections in their software. These error corrections were not implemented on the Potentiostat software due to the complexity of the code.

Another implementation for the Potentiostat compared to the Reference Potentiostat was the possibility to set the number of cycles for cyclic voltammetry. This implementation was successful. The Reference Potentiostat can start a new scan right after a scan has been completed, but there has to be a human operator to start the new scan. The time it takes to start a new scan might be too long for a cyclic voltammetry experiment.

Figure 6.5 shows how the Reference Potentiostat behaved during an amperometric scan with dopamine. The plot illustrates that the device can detect that dopamine has been applied to the electrodes. It is expected that the Potentiostat would behave similarly to the Reference Potentiostat, but this is not verified by testing on an electrolytic cell. There were conducted more experiments with the Reference Potentiostat than provided in the result chapter. Cyclic voltammetry performed on ascorbic acid, dopamine and Ferri-/Ferrocyanide were conducted, in addition to amperometry on dopamine. These results verified that the Reference Potentiostat gave similar results as in the article by Lopin and Lopin (2018).

The Reference Potentiostat can choose the sensing resistor in the TIA both from the integrated resistors and as an externally connected resistor. This functionality was not implemented on the Potentiostat, where the only resistor value available at the moment is 20 kΩ. The resistor value is an important variable for the potentiostat to adjust the possible current range to measure. The only reason for the Potentiostat's lack of that option was that the functionality was not needed during testing without an electrolytic cell.

Chapter 8

Conclusions and Further Work

8.1 Conclusion

A potentiostat has been developed as a prototype for The Oslo Bioimpedance and Medical Technology Group at the Department of Physics (UiO) and the Department of Clinical and Biomedical Engineering (OUS). They are involved in an EU-project named Training4CRM. The purpose of the project is to address gaps in Cell-based Regenerative Medicine (CRM) to treat neurodegenerative disorders, among others, Parkinson's disease. A potentiostat is needed to detect and characterize dopamine in the project.

The prototype potentiostat can conduct cyclic voltammetry and amperometry experiments. The device was developed on a PSoC 5LP development kit with the possibility to conduct experiments with a voltage range of -2.032 V to +2.032 V, a scan rate of maximum 1 V/s, and a current sense limited to $\pm 101 \mu A$. The device communicates with a computer via a USB interface. During measurements, data is transferred continuously from the device to a computer. A Python program has been developed to control the potentiostat, receive data from it, and plot the measured data.

The work by Lopin and Lopin (2018) was the basis for the development of the potentiostat developed in this thesis. The potentiostat by Lopin and Lopin (2018) has been successfully reproduced and tested for functionality in cyclic voltammetry and amperometry experiments on Ferri-/Ferrocyanide and ascorbic acid. The results are similar to the results presented in their article. Their work was extensive, with too many measurement techniques implemented for the scope of this project. Some of their source code was re-used, but all of the software had to be re-developed. Their potentiostat had room for improvement. Continuous data transfer was implemented to conduct as many cycles in a cyclic voltammetry scan as feasible (not fill up the platform's internal memory). There has been a focus on writing good documentation for all the code and writing code to give a better overview of the system. This was lacking in the work by Lopin and Lopin (2018), which lead to the re-development of their work instead of re-use.

The results obtained with the potentiostat have flaws due to errors in firmware and software. These errors have been corrected after the results were provided, but the device has not been tested on an electrolytic cell with the corrections. The results were compared with the potentiostat by Lopin and Lopin (2018). Their potentiostat was also developed on the PSoC 5LP, which implies that the platform is feasible as a potentiostat. There are, however, limitations to the platform since it is a system on chip where all components are integrated into the platform. This means that the accuracy of measurements can not be better than the precision of the integrated components.

8.2 Further Work

The potentiostat needs improvements for it to work in the Training4CRM project. First of all, the device must be tested to verify that the new corrections/implementations are functioning. Then it is important to implement a calibration routine to make sure the measurements are correct. There should be a possibility of adjusting the resistor in TIA, either by changing to another integrated resistor or by connecting a resistor externally. The resolution of the ADC can be improved by setting the resolution to 20-bits instead of 12-bits. This depends on the needed resolution and the needed sampling rate, since an ADC with better resolution also needs more time to convert.

When the potentiostat is functioning properly, it would be feasible to make the data transfer wireless by, e.g., a Bluetooth module. For this to be possible, there should be a serial interface utilized instead of USB, e.g., UART. If there is a need for better efficiency of the CPU in the PSoC 5LP, it is possible to utilize its direct memory access (DMA). This will make sure processes bypass the CPU and let the CPU do other jobs simultaneously.

As a final product, the PSoC 5LP should be designed on a PCB to minimize the area used for the brain implant in the Training4CRM project. At that point, power consumption could be an issue, so all the firmware codes should be reviewed to improve efficiency.

References

Adams, S. D., Doeven, E. H., Quayle, K., and Kouzani, A. Z. (2019). MiniStat: Development and Evaluation of a Mini-Potentiostat for Electrochemical Measurements. *IEEE Access*, 7:31903–31912. Conference Name: IEEE Access.

Adeloju, S. B. (2005). AMPEROMETRY. In Worsfold, P., Townshend, A., and Poole, C., editors, *Encyclopedia of Analytical Science (Second Edition)*, pages 70–79. Elsevier, Oxford.

Ainla, A., Mousavi, M. P. S., Tsaloglou, M.-N., Redston, J., Bell, J. G., Fernández-Abedul, M. T., and Whitesides, G. M. (2018). Open-Source Potentiostat for Wireless Electrochemical Detection with Smartphones. *Analytical Chemistry*, 90(10):6240–6246. Publisher: American Chemical Society.

Baker, B. (2011). How delta-sigma ADCs work, Part 1. *Analog Applications*, 7.

Bucher, E. S. and Wightman, R. M. (2015). Electrochemical Analysis of Neurotransmitters. *Annual review of analytical chemistry (Palo Alto, Calif.)*, 8:239–261.

Chang, R. (2008). *General chemistry : the essential concepts*. McGraw-Hill, 5th ed. edition.

Cunha, A. B., Schuelke, C., Heiskanen, A., Asif, A., Hassan, Y., Keller, S. S., Kalvøy, H., Martínez-Serrano, A., Emnéus, J., and Martinsen, G. (2019). Bioimpedance measurements on human neural stem cells as a benchmark for the development of smart mobile biomedical applications. *EasyChair*, 991.

Cypress Semiconductors (2020a). 32-bit Arm® Cortex®-M0 PSoC® 4. Retrieved from https://www.cypress.com/products/32-bit-arm-cortex-m0-psoc-4, 28.09.2020.

Cypress Semiconductors (2020b). 32-bit Arm® Cortex®-M3 PSoC® 5LP. Retrieved from https://www.cypress.com/products/32-bit-arm-cortex-m3-psoc-5lp, 28.09.2020.

Cypress Semiconductors (2020c). 32-bit Arm® Cortex®-M4 Cortex-M0+ PSoC® 6. Retrieved from https://www.cypress.com/products/32-bit-arm-cortex-m4-cortex-m0-psoc-6, 28.09.2020.

Cypress Semiconductors (2020d). CY8CKIT-050 PSoC® 5LP Development Kit. Retrieved from https://www.cypress.com/documentation/development-kitsboards/cy8ckit-050-psoc-5lp-development-kit, 28.09.2020.

Cypress Semiconductors (2020e). CY8CKIT-059 PSoC® 5LP Prototyping Kit With Onboard Programmer and Debugger. Retrieved from https://www.cypress.com/documentation/development-kitsboards/cy8ckit-059-psoc-5lp-prototyping-kit-onboard-programmer-and, 28.09.2020.

Cypress Semiconductors (2020f). PSoC® Creator™ Integrated Design Environment (IDE). Retrieved from https://www.cypress.com/products/psoc-creator-integrated-design-environment-ide, 04.10.2020.

Cypress Semiconductors and Infineon (2020a). 8-Bit Voltage Digital to Analog Converter (VDAC8). Retrieved from https://www.cypress.com/documentation/component-datasheets/8-bit-voltage-digital-analog-converter-vdac8, 26.09.2020.

Cypress Semiconductors and Infineon (2020b). Character LCD (CharLCD). Retrieved from https://www.cypress.com/documentation/component-datasheets/character-lcd-charlcd, 07.10.2020.

Cypress Semiconductors and Infineon (2020c). Delta Sigma Analog to Digital Converter (ADC_delsig). Retrieved from https://www.cypress.com/documentation/component-datasheets/delta-sigma-analog-digital-converter-adcdelsig, 07.10.2020.

Cypress Semiconductors and Infineon (2020d). Dithered Voltage Digital to Analog Converter (DVDAC). Retrieved from https://www.cypress.com/documentation/component-datasheets/dithered-voltage-digital-analog-converter-dvdac, 07.10.2020.

Cypress Semiconductors and Infineon (2020e). Full Speed USB (USBFS). Retrieved from https://www.cypress.com/documentation/component-datasheets/full-speed-usb-usbfs, 07.10.2020.

Cypress Semiconductors and Infineon (2020f). Interrupt. Retrieved from https://www.cypress.com/interrupt, 07.10.2020.

Cypress Semiconductors and Infineon (2020g). Operational Amplifier (Opamp). Retrieved from https://www.cypress.com/documentation/component-datasheets/operational-amplifier-opamp, 07.10.2020.

Cypress Semiconductors and Infineon (2020h). Timer. Retrieved from https://www.cypress.com/documentation/component-datasheets/timer, 07.10.2020.

Cypress Semiconductors and Infineon (2020i). Trans-Impedance Amplifier (TIA). Retrieved from https://www.cypress.com/documentation/component-datasheets/trans-impedance-amplifier-tia, 07.10.2020.

David, H. (2013). 11.4: Voltammetric Methods. Retrieved from https://chem.libretexts.org/Under_Construction/Purgatory/Book%3A_ Analytical_Chemistry_2.0_(Harvey)/11_ Electrochemical_Methods/11.4%3A_ Voltammetric_Methods, 21.09.2020.

Dryden, M. D. M. and Wheeler, A. R. (2015). DStat: A Versatile, Open-Source Potentiostat for Electroanalysis and Integration. *PLOS ONE*, 10(10):e0140349. Publisher: Public Library of Science.

Elgrishi, N., Rountree, K. J., McCarthy, B. D., Rountree, E. S., Eisenhart, T. T., and Dempsey, J. L. (2018). A Practical Beginner's Guide to Cyclic Voltammetry. *Journal of Chemical Education*, 95(2):197–206. Publisher: American Chemical Society.

Gamry Instruments (2020). Potentiostat/Galvanostat Electrochemical Instrument Basics. Retrieved from https://www.gamry.com/application-notes/instrumentation/potentiostat-fundamentals/, 21.09.2020.

Grimnes, S. and Martinsen, G. (2015). *Bioimpedance and Bioelectricity Basics, 3rd Edition.* Academic Press, 3 edition.

Hassan, Y. M., Caviglia, C., Hemanth, S., Mackenzie, D. M. A., Alstrøm, T. S., Petersen, D. H., and Keller, S. S. (2017). High temperature SU-8 pyrolysis for fabrication of carbon electrodes. *Journal of Analytical and Applied Pyrolysis*, 125:91–99.

Kester, W. (2005). Which ADC Architecture Is Right for Your Application? | Analog Devices. Retrieved fromhttps://www.analog.com/en/analog-dialogue/articles/the-right-adc-architecture.html, 27.09.2020.

Lopin, P. and Lopin, K. V. (2018). PSoC-Stat: A single chip open source potentiostat based on a Programmable System on a Chip. *PLOS ONE*, 13(7):e0201353.

Ruud, S. K. (2019). Embedded Development of a Wireless SoC Instrument for Electrical Impedance Spectroscopy on Cells. Master's thesis, University of Oslo.

Scherz, P. and Monk, S. (2016). *Practical Electronics for Inventors, Fourth Edition, 4th Edition*. McGraw-Hill Education TAB, 4 edition.

Umar, S. N. H., Bakar, E. A., Kamaruddin, N. M., and Uchiyama, N. (2018). A Low Cost Potentiostat Device For Monitoring Aqueous Solution. *MATEC Web of Conferences*, 217:04001. Publisher: EDP Sciences.

Yagi, I., Notsu, H., Kondo, T., Tryk, D. A., and Fujishima, A. (1999). Electrochemical selectivity for redox systems at oxygen-terminated diamond electrodes. *Journal of Electroanalytical Chemistry*, 473(1):173–178.

Zadig (2020). Zadig - USB driver installation made easy. Retrieved from https://zadig.akeo.ie/, 04.10.2020.

Appendix

8.3 Firmware

8.3.1 Source Code (.c-files)

8.3.1.1 main.c

```
/* ========================================
 * File name: main.c
 * Version: A8
 *
 * Description:
 * Main code for the controller. All functionality will be controlled
 * from the main loop. Pointers will access functions placed in
 * other scripts.
 *
 * Progress:
 * -------------------------------------------------
 * |       ISSUE                * STATUS * TESTED  |
 * -------------------------------------------------
 * |Communication with computer *   OK   *   YES   |
 * |DAC setup                   *   OK   *   YES   |
 * |DAC timing                  *   OK   *   YES   |
 * |Enabling functions          *   OK   *   YES   |
 * |ADC                         *   OK   *   YES   |
 * |REFERENCE DAC               *   OK   *   YES   |
 * |TRANSFER DATA               *   OK   *   YES   |
 * |ADC Timing                  *   OK   *   YES   |
 * |TIA                         *   OK   *   YES   |
 * |OPAMP                       *   OK   *   YES   |
 * |Cyclic Voltammetry          *   OK   *   YES   |
 * |Amperometry                 *   OK   *   YES   |
 * |Code cleanup                *   OK   *   YES   |
 * -------------------------------------------------
 *
 * ISSUE:
```

```
 * Measurements misbehaving
 * USB communication not working as expected
 *
 * Copyright Univeristy of Oslo, 2020
 * ========================================
*/
/*  Project Files  */
#include "project.h"
#include "general_functions.h"
#include "globals.h"
#include "usb_protocol.h"


/* Declaration of variables */
uint8 Input_Flag = FALSE;                  // True if EP2 has changed.
uint8 OUT_Data_Buffer[MAX_NUM_BYTES];      // Buffer USB.

CY_ISR(dacInterrupt) {
    TIMER_ReadStatusRegister();             // Release dacInterrupt
    /* Define next voltage value */
    if (direction == UP) { index_value += step_size; }
    else { index_value -= step_size; }

    /* Check if one cyclus is done */
    if (index_value == start_value) {          // One cycle completed
        cycles_index += 1;                     // Iterate cycle index
        if (cycles_index == number_of_cycles) { // CV complete
            isr_ADC_Disable();                 // Disable ADC interrupt
            isr_DAC_Disable();                 // Disable ADC interrupt
            helper_HardwareSleep();            // Set hardware to sleep mode
            data_usb16 = 49152;                // Determintaion value for ADC_array
            USB_Export_Data(data_usb16);       // Transfer last array
            helper_LCD_write0("CV DONE");      // Write to LCD
            helper_LCD_clear1();               // Clear line two of LCD
            LED_DAC_Write(0);                  // LED_DAC off
        }
    }

    /* Check if direction should change */
```

```
    if (index_value >= max_value) {
        direction = DOWN;
    }
    if (index_value <= min_value) {
        direction = UP;
    }

    /* Set next value to DAC*/
    DVDAC_SetValue(index_value);
}

CY_ISR(adcInterrupt) {
    TIMER_ReadStatusRegister();     // Release adcInterrupt
    data_usb16 = ADC_GetResult16(); // Fetch adc measurement in
   data_usb16
    USB_Export_Data(data_usb16);    // Export the data
}


int main(void){
    CyGlobalIntEnable;                      // Enable global
   interrupts.

    /* Initialize hardware and interrupts */
    isr_DAC_StartEx(dacInterrupt);          // Setup interrupt
    isr_DAC_Disable();                      // Disable interrupt
    isr_ADC_StartEx(adcInterrupt);          // Setup interrupt
    isr_ADC_Disable();                      // Disable interrupt
    helper_HardwareSetup();                 // Setup HW

    USB_Start(0, USB_DWR_VDDD_OPERATION);   // Start the USB
   peripherals.

    while(!USB_GetConfiguration());         // Wait until USB is
   configured.
    USB_EnableOutEP(OUT_ENDPOINT);          // Enable out endpoint
    (EP2).

    for(;;) {
        USB_Config_Change();                // Check if
   configuration has changed

        /* Check if host has tranferred commands to device. If yes
   : Input_Flag = True. */
        if (Input_Flag == FALSE) { Input_Flag = USB_CheckInput(
   OUT_Data_Buffer); }

        /* Input_Flag == TRUE   -> Switch statement checks input
   for functionalities below. */
```

```
        /* Input_Flag == FALSE  -> Skip switch statement. Loop. */
        if (Input_Flag == TRUE) {
            switch (OUT_Data_Buffer[0]) {
                case CV_TIMER:
                    // User input: C xxxxxxxx
                    counter = helper_Convert2Dec32(&
   OUT_Data_Buffer[2], 8);
                    TIMER_WritePeriod(counter);
                    break;

                case CV_NO_CYCLES:
                    // User input: N xx
                    number_of_cycles = helper_Convert2Dec8(&
   OUT_Data_Buffer[2],2);
                    break;

                case CV_DEFINE_RANGE:
                    // User input: L xxxx xxxx xxxx
                    min_value   = helper_Convert2Dec16(&
   OUT_Data_Buffer[2],4);
                    max_value   = helper_Convert2Dec16(&
   OUT_Data_Buffer[7],4);
                    start_value = helper_Convert2Dec16(&
   OUT_Data_Buffer[12],4);

                    // Set direction of next step in sweep
                    // Direction = DOWN     IF  start_value ==
   max_value
                    // Direction = UP       IF  else
                    if      (start_value == min_value) {
   direction_initial = UP;}
                    else if (start_value == max_value) {
   direction_initial = DOWN;}
                    else    {direction_initial = UP;}

                    helper_LCD_write0("Data uploaded.");    //
   Write to display
                    helper_LCD_write1("Ready for CV.");     //
   Write to display
                    break;

                case CV_RUN:
                    // User input: R
                    index_value = start_value;              // Set
    initial value
                    buffer_index = 0;                       // Set
    buffer_count to initial count
                    cycles_index = 0;                       // Set
    cycle_index to initial state
```

```
                    channel = 1;                            // Set
    initial channel
                    step_size = 1;                          // Set
    step size
                    direction = direction_initial;          // Set
    start direction as initial direction
                    helper_HardwareWakeup();                //
   Wakeup hardware
                    DVDAC_SetValue(index_value);            // Set
    initial dac value
                    CyDelay(70);                            //
   Delay for DVDAC to stabilize
                    data_usb16 = ADC_GetResult16();         //
   Save first ADC measurement in ADC_array
                    USB_Export_Data(data_usb16);            //
   Send first value to USB
                    isr_ADC_Enable();                       //
   Enable ADC interrupt
                    isr_DAC_Enable();                       //
   Enable DAC interrupt
                    LED_DAC_Write(1);                       // LED
    indicating CV is running
                    helper_LCD_write0("CV start. Cycles:"); //
   Write to display
                    helper_LCD_format1(number_of_cycles);   //
   Write no of cycles on line two
                    break;

                case AMP_RUN:
                    // User input: A
                    LCD_ClearDisplay();
                    helper_LCD_write0("Amperometry");
                    helper_LCD_write1("is running");
                    amp_voltage = helper_Convert2Dec16(&
   OUT_Data_Buffer[2],4);
                    TIMER_WritePeriod(600000);   // 25 ms period
                    helper_HardwareWakeup();
                    DVDAC_SetValue(amp_voltage);
                    isr_ADC_Enable();
                    break;

                case AMP_STOP:
                    isr_ADC_Disable();
                    helper_HardwareSleep();
                    LCD_ClearDisplay();
                    helper_LCD_write0("Amperometry");
                    helper_LCD_write1("has ended");
                    break;
            }                           // End of switch statement
```

```
            OUT_Data_Buffer[0] = '0';   // Clear data buffer, ready for new loop.
            Input_Flag = FALSE;         // Set flag to False, ready for new loop.
        }
    }
}
/* [] END OF FILE */
```

8.3.1.2 general_functions.c

```c
/* ========================================
 * File name: general_functions.c
 *
 * Description:
 * Functions to assist main.c.
 * Involves functions to edit formats and to display on LCD.
 *
 * Copyright Univeristy of Oslo, 2020
 * ========================================
*/
#include "general_functions.h"


/* *****************************************************************************
* Function Name: helper_HardwareSetup
*******************************************************************************
*
* Summary:
*    Setup all the hardware needed for an experiment.  This will start all the hardware
*    and then put them to sleep so they can be awoke for an experiment.
*
****************************************************************************** */
void helper_HardwareSetup(void) {
    LCD_Start();                            // Start LCD
    helper_LCD_write0("Potentiostat: A8");  // Start message
    helper_LCD_write1("Created by: OBJ");   // Created by message
    DVDAC_Start();                          // Initialize DVDAC
    DVDAC_Sleep();                          // DVDAC sleep
    OPAMP_Start();                          // Start OPAMP for DAC
    OPAMP_Sleep();                          // OPAMP sleep
    TIA_Start();                            // TIA start
    TIA_Sleep();                            // TIA sleep
    VDAC_REF_Start();                       // VDAC_REF start
    VDAC_REF_Sleep();                       // VDAC_REF sleep
    ADC_Start();                            // ADC start
    ADC_Sleep();                            // ADC sleep
    TIMER_Start();                          // TIMER start
    TIMER_Sleep();                          // TIMER sleep
    LED_DAC_Write(0);                       // LED off
}

```

```
/*
   ***************************************************************************

* Function Name: helper_HardwareWakeup
******************************************************************************

*
* Summary:
*    Wakes up all the desired hardware.
*
******************************************************************************
   */
void helper_HardwareWakeup(void) {
    DVDAC_Wakeup();                         // Wakeup DVDAC
    OPAMP_Wakeup();                         // Wakeup OPAMP
    TIA_Wakeup();                           // Wakeup TIA
    VDAC_REF_Wakeup();                      // Wakeup VDAC_REF
    ADC_Wakeup();                           // Wakeup ADC
    ADC_StartConvert();                     // Start ADC
   conversion
    TIMER_Wakeup();                         // Wakeup TIMER
}

/*
   ***************************************************************************

* Function Name: helper_HardwareSleep
******************************************************************************

*
* Summary:
*    Sets all hardware to sleep mode.
*
******************************************************************************
   */
void helper_HardwareSleep(void) {
    TIMER_Sleep();                          // Sleep TIMER
    DVDAC_Sleep();                          // Sleep DVDAC
    OPAMP_Sleep();                          // Sleep OPAMP
    TIA_Sleep();                            // Sleep TIA
    VDAC_REF_Sleep();                       // Sleep VDAC_REF
    ADC_StopConvert();                      // Stop ADC conversion
    ADC_Sleep();                            // Sleep ADC
}

/*
   ***************************************************************************

* Function Name: helper_LCD_write
```

```
********************************************************************************

*
* Summary:
*     Function to print message to the LCD.
*     Purpose is to save space in main.c
*
*******************************************************************************
    */
// Write text in the first row of LCD
void helper_LCD_write0(char message[]) {
    helper_LCD_clear0();
    LCD_Position(0u,0u);
    LCD_PrintString(message);
}

// Write text in the second row of LCD
void helper_LCD_write1(char message[]) {
    helper_LCD_clear1();
    LCD_Position(1u,0u);
    LCD_PrintString(message);
}

// Write number in the first row of LCD
void helper_LCD_format0(uint16 message) {
    helper_LCD_clear0();
    char a[32];
    LCD_Position(0,3);
    sprintf(a,"%4u",message);
    LCD_PrintString(a);
}

// Write number in the second row of LCD
void helper_LCD_format1(uint16 message) {
    helper_LCD_clear1();
    char b[32];
    LCD_Position(1,3);
    sprintf(b,"%4u",message);
    LCD_PrintString(b);
}

// Clear the first row of LCD
void helper_LCD_clear0(void) {
    LCD_Position(0,0);
    LCD_PrintString("                ");
}

// Clear the second row of LCD
void helper_LCD_clear1(void) {
```

```
    LCD_Position(1,0);
    LCD_PrintString("                ");
}

/* ***************************************************************************
* Function Name: helper_Convert2Dec
*******************************************************************************
*
* Summary:
*     Takes in an array of numbers and length, returns the number as
*     a number not an array of text.
*
******************************************************************************* */
uint32 helper_Convert2Dec32(uint8 array[], uint8 len){
    uint32 num = 0;
    for (int i = 0; i < len; i++){
        num = num * 10 + (array[i] - '0');
    }
    return num;
}
uint16 helper_Convert2Dec16(uint8 array[], uint8 len){
    uint16 num = 0;
    for (int i = 0; i < len; i++){
        num = num * 10 + (array[i] - '0');
    }
    return num;
}
uint8 helper_Convert2Dec8(uint8 array[], uint8 len){
    uint8 num = 0;
    for (int i = 0; i < len; i++){
        num = num * 10 + (array[i] - '0');
    }
    return num;
}
/* ***************************************************************************
* Function Name: helper_Convert16to8
*******************************************************************************
*
* Summary:
*    Takes in a UINT16 and converts it to double UINT8.
*    The convertion is on the form low to high. Least significant
```

```
   first and then most significant.
*
******************************************************************************
   */
void helper_Convert16to8(uint16 value){
    data_usb8[0] = (uint8) value;
    data_usb8[1] = (uint8)(value >> 8);
}
/* [] END OF FILE */
```

8.3.1.3 usb_protocol.c

```c
/* ========================================
 * File Name: usb_protocols.c
 *
 * Description:
 *  Source code for the protocols used by the USB.
 *
 * Copyright University of Oslo, 2019
 * ========================================
*/

#include <project.h>
#include "usb_protocol.h"
#include "stdio.h"
#include "stdlib.h"

/* ***************************************************************************
* Function Name: USB_CheckInput
******************************************************************************
*
* Summary:
*  Check if any incoming USB data and store it to the input buffer
*
* Parameters:
*  uint8 buffer: array where the data is stored
*
* Return:
*  true (1) if data has been inputed or false (0) if no data
*
* Global variables:
*  OUT_ENDPOINT:  EP2
*
****************************************************************************** */

uint8 USB_CheckInput(uint8 buffer[]) {

    if(USB_GetEPState(OUT_ENDPOINT) == USB_OUT_BUFFER_FULL) {
        uint8 OUT_COUNT = USB_GetEPCount(OUT_ENDPOINT);     // There is data coming in, get the number of bytes.
        USB_ReadOutEP(OUT_ENDPOINT, buffer, OUT_COUNT);     // Read the OUT endpoint and store data in OUT_COUNT.
        USB_EnableOutEP(OUT_ENDPOINT);                      // Re-enable OUT endpoint.
        return TRUE;
```

```
    }

    return FALSE;
}

/* ******************************************************************************
* Function Name: USB_Export_Data
*************************************************************************************
*
* Summary:
*  Take a buffer as input and export it, the number of bytes to send is the second argument.
*
* Parameters:
*  uint16 array: array of data to export
*  uint16 size: the number of bytes to send in the array
*
* Return:
*  None
*
* Global variables:
*  MAX_BUFFER_SIZE:  the number of bytes the USB EP1 device can transfer
*
************************************************************************************* */

void USB_Export_Data(uint16 value) {
    data_usb8[0] = (uint8) value;
    data_usb8[1] = (uint8)(value >> 8);
    while(USB_GetEPState(IN_ENDPOINT) != USB_IN_BUFFER_EMPTY);  // Wait until EP1 is empty

    if(USB_GetEPState(IN_ENDPOINT) == USB_IN_BUFFER_EMPTY){
        USB_LoadInEP(IN_ENDPOINT, data_usb8, 2);
        USB_EnableOutEP(OUT_ENDPOINT);
    }
}

/* ******************************************************************************
* Function Name: USB_Config_Change
*************************************************************************************

*
```

```
* Summary:
*   If configurations is changed, reenable the OUT endpoint.
*   Wait for the configuration.
*   Re-enable out endpoint
*
* Parameters:
*   None
*
* Return:
*   None
*
* Global variables:
*   OUT_ENDPOINT:  out endpoint number
*
****************************************************************************
    */

void USB_Config_Change() {
    if (USB_IsConfigurationChanged()) {
        while(!USB_GetConfiguration()) {}
        USB_EnableOutEP(OUT_ENDPOINT);
    }
}
/* [] END OF FILE */
```

8.3.2 Header Code (.h-files)

8.3.2.1 globals.h

```c
/* ========================================
 * File name: cv_functions.h
 *
 * Description:
 * User input functionality is defined here.
 *
 * Copyright Univeristy of Oslo, 2020
 * ========================================
*/


#if !defined(GLOBALS)
#define GLOBALS

#include "cytypes.h"
/************************************
*        USB INPUT OPTIONS
************************************/
#define CV_TIMER            'C'
#define CV_NO_CYCLES        'N'
#define CV_DEFINE_RANGE     'L'
#define CV_RUN              'R'
#define START_DAC           'D'
#define VALUE_DAC           'V'
#define USB_TEST            'T'
#define AMP_RUN             'A'
#define AMP_STOP            'S'

/************************************
*        Global Variables
************************************/
#define MAX_BUFFER_SIZE         64
#define CHANNEL_MAX             300

/************************************
*        ADC -> USB VARIABLES
************************************/
uint16  channel;                                        // initializer for adc channels for storage
uint16  data_usb16;                                     // array for ADC values UINT16 with four channels
uint8   data_usb8[ MAX_BUFFER_SIZE ];                   // array for ADC values converted to double UINT8

/************************************
```

```
*         CV VARIABLES
*************************************/
uint16  buffer_index;           // indexing for adc usb transfer
uint8   number_of_cycles;       // number of cycles for cyclic
   voltammetry
uint8   cycles_index;           // index for number of cycles in
   cuyclic voltammetry
uint16  step_size;              // incremental step size
uint16  index_value;            // Counter for dac values
uint8   direction;              // Direction for dac values (next
   value up or down)
uint8   direction_initial;      // Direction for dac values stored
    in this variable
uint16  start_value;            // Initial value for CV
uint16  min_value;              // Minimum value for CV
uint16  max_value;              // Maximum value for CV
uint32  counter;                // Value to set correct timing for
    the TIMER
#define UP      1
#define DOWN    0

/*************************************
*         AMP VARIABLES
*************************************/
uint16 amp_voltage;         // Amperometry voltage

#endif
/* [] END OF FILE */
```

8.3.2.2 general_functions.h

```
/* ========================================
 * File name: general_functions.h
 *
 * Description:
 * Functions to assist main.c.
 * The variables are defined in this header.
 *
 * Copyright Univeristy of Oslo, 2020
 * ========================================
*/
#if !defined(GENERAL_FUNCTIONS_H)
#define GENERAL_FUNCTIONS_H

/*  Project Files  */
#include <project.h>
#include "globals.h"

/*  Standard C Files  */
#include "stdio.h"
#include "cytypes.h"

/*************************************
*         Function Prototypes
*************************************/

uint8 helper_Convert2Dec8(uint8 array[], uint8 len);
uint16 helper_Convert2Dec16(uint8 array[], uint8 len);
uint32 helper_Convert2Dec32(uint8 array[], uint8 len);
void helper_Convert16to8(uint16 value);
void helper_HardwareSetup(void);
void helper_HardwareWakeup(void);
void helper_HardwareSleep(void);
void helper_LCD_write0(char message[]);
void helper_LCD_write1(char message[]);
void helper_LCD_format0(uint16 message);
void helper_LCD_format1(uint16 message);
void helper_LCD_clear0(void);
void helper_LCD_clear1(void);

#endif
/* [] END OF FILE */
```

8.3.2.3 usb_protocol.h

```
/* ========================================
 * File name: usb_protocal.h
 *
 * Description:
 *  Contains function prototypes and constants for the
 *  USB protocals.
 *
 ********************************************
 * Copyright University of Oslo, 2019
 * ========================================
*/
#if !defined(USB_PROTOCOL_H)
#define USB_PROTOCOL_H

#include <project.h>
#include "general_functions.h"

/**************************************
*      Constants
**************************************/

#define IN_ENDPOINT         1           // Endpoint for transfer to host.
#define OUT_ENDPOINT        2           // Endpoint for transfer from host.
#define MAX_BUFFER_SIZE     64          // Maximum output to host package size.
#define MAX_NUM_BYTES       512         // Maximum size of USB buffer.
#define FALSE               0           // Define boolean statement False.
#define TRUE                (!FALSE)    // Define boolean statement True.

/***************************************
*        Function Prototypes
***************************************/
uint8 USB_CheckInput(uint8 buffer[]);
void USB_Export_Data(uint16 value);
void USB_Config_Change();

#endif
/* [] END OF FILE */
```

8.4 Software

8.4.0.1 Potentiostat_GUI.py

```python
import tkinter as tk
from tkinter import ttk as ttk

import Potentiostat_communication
import Potentiostat_functionality
import Potentiostat_Constants
import Potentiostat_userinput

comm = Potentiostat_communication.Communication()
dev, ep_out, ep_in = comm.usb_connect(comm.vendor_id, comm.product_id)

class Potentiostat_GUI(tk.Frame):
  """
  Potentiostat_GUI makes the interface from the Potentiostat commands.
  The GUI is made using Tkinter.
  """
  def __init__(self, master=None):
    """
    The layout is made in this function.
    All buttons have a corresponding function underneath this class that calls for other classes.
    """
    #### Importing classes to variables ####
    self.con = Potentiostat_Constants.Constants()
    self.func = Potentiostat_functionality.Potentiostat()
    self.user = Potentiostat_userinput.UserInput()
    self.comm = Potentiostat_communication.Communication()  # Path to communication class

    #### Storage of data arrays for amperometry ####
    self.time_data_store    =   None
    self.current_data_store   =   []

    tk.Frame.__init__(self, master)         # Main frame

    self.master.title("Potentiostat Controller")  # Set name to window
    self.master.configure(background = "black")   # Set background color
    self.master.geometry("870x525")         # Set size of window

    self.after_id = None              # Determination for GUI
```

```

 ### Make title ###
 tk.Label(self.master, text = "Potentiostat", fg = "white",  bg
 = "black", width = 40,
         font=("Courier", 30)).grid(rowspan = 2, columnspan =
10)
 tk.Label(self.master, text = "
---------------------------------------------------------------------------
",
       fg = "red", bg = "black", font=("Courier", 10)).grid(
rowspan = 2, columnspan = 10)

 ### Make design ###
 tk.Label(self.master, text = "Choose type of experiment: ", fg
 = "white", bg = "black",
               font=("Courier", 15)).grid(rowspan = 1,
columnspan = 10)
 tk.Label(self.master, text = "
------------------------------------------------",
       fg = "red", bg = "black", font=("Courier", 10)).grid(
rowspan = 2, columnspan = 10)
 tk.ttk.Separator(self.master, orient="vertical").grid(row =
11, column =2, rowspan=11, sticky='ns')
 tk.Label(self.master, text = "Scan Rate: ", fg = "white", bg =
 "black",             # Scan rate text
               font=("Courier", 12)).grid(row = 17, column = 0,
 sticky = "e")
 tk.Label(self.master, text = "V/s", fg = "white", bg = "black"
,                 # Scan rate unit
               font=("Courier", 12)).grid(row = 17, column = 2,
 sticky = "w")
 tk.Label(self.master, text = "Minimum Voltage: ", fg = "white"
, bg = "black",         # Min voltage text
               font=("Courier", 12)).grid(row = 18, column = 0,
 sticky = "e")
 tk.Label(self.master, text = "mV", fg = "white", bg = "black",
                  # Min voltage unit
               font=("Courier", 12)).grid(row = 18, column = 2,
 sticky = "w")
 tk.Label(self.master, text = "Maximum Voltage: ", fg = "white"
, bg = "black",         # Max voltage text
               font=("Courier", 12)).grid(row = 19, column = 0,
 sticky = "e")
 tk.Label(self.master, text = "mV", fg = "white", bg = "black",
                  # Max voltage unit
               font=("Courier", 12)).grid(row = 19, column = 2,
 sticky = "w")
 tk.Label(self.master, text = "Voltage: ", fg = "white", bg = "
black",             # Amperometry voltage text
```

```
                        font=("Courier", 12)).grid(row = 19, column = 3, sticky = "e")
    tk.Label(self.master, text = "mV", fg = "white", bg = "black",                   # Amperometry voltage unit
                        font=("Courier", 12)).grid(row = 19, column = 5, sticky = "w")
    tk.Label(self.master, text = "Start Voltage: ", fg = "white", bg = "black",             # Start voltage text
                        font=("Courier", 12)).grid(row = 20, column = 0, sticky = "e")
    tk.Label(self.master, text = "mV", fg = "white", bg = "black",                   # Start voltage unit
                        font=("Courier", 12)).grid(row = 20, column = 2, sticky = "w")
    tk.Label(self.master, text = "Number of cycles: ", fg = "white", bg = "black",            # Number of cycles text
                        font=("Courier", 12)).grid(row = 21, column = 0, sticky = "e")
    tk.Label(self.master, text = "#", fg = "white", bg = "black",                   # Start voltage unit
                        font=("Courier", 12)).grid(row = 21, column = 2, sticky = "w")
    tk.Label(self.master, text = "", fg = "white", bg = "black",                    # Horizontal space
                        font=("Courier", 12)).grid(row = 22, column = 0, sticky = "e")
    tk.Label(self.master, text = "", fg = "white", bg = "black",                    # Horizontal space
                        font=("Courier", 12)).grid(row = 25, column = 0, sticky = "e")
    tk.Label(self.master, text = "Plot Title: ", fg = "white", bg = "black",              # Plot title text
                        font=("Courier", 12)).grid(row = 24, column = 1, sticky = "e")
    tk.Label(self.master, text = "Solution name: ", fg = "white", bg = "black",             # Plot legend text
                        font=("Courier", 12)).grid(row = 25, column = 1, sticky = "e")
    tk.ttk.Separator(self.master, orient="horizontal").grid(row = 27, column = 1, columnspan=3, sticky=’ew’)
    tk.Label(self.master, text = "", fg = "white", bg = "black",                    # Horizontal space
                        font=("Courier", 12)).grid(row = 27, column = 0, sticky = "e")
    tk.Label(self.master, text = "", fg = "white", bg = "black",                    # Horizontal space
                        font=("Courier", 12)).grid(row = 29, column = 0, sticky = "e")
    tk.Label(self.master, text = "", fg = "white", bg = "black",
```

```
                   # Horizontal space
                  font=("Courier", 12)).grid(row = 30, column = 0,
    sticky = "e")
    tk.Label(self.master, text = "", fg = "white", bg = "black",
                   # Horizontal space
                  font=("Courier", 12)).grid(row = 29, column = 0,
    sticky = "e")
    tk.ttk.Separator(self.master, orient="vertical").grid(row =
   28, column =2, rowspan=2, sticky='ns')
    tk.ttk.Separator(self.master, orient="horizontal").grid(row =
   30, column = 1, columnspan=3, sticky='ew')
    tk.Label(self.master, text = "File name: ", fg = "white", bg =
    "black",            # File name text
                  font=("Courier", 12)).grid(row = 33, column = 1,
    sticky = "e")


    #### Choice of experiment radiobutton####
    # Disables the experiment that is not choosen.
    # Calls for Disable_CV or Disable_AMP functions
    self.choose_experiment = tk.IntVar()    # Choose experiment
   variable

    self.CV_button  = tk.Radiobutton(self.master, text = "Cyclic
   Voltammetry", font=("Courier", 15), bg = "grey",
              variable = self.choose_experiment, value = 1,
              command = self.Disable_AMP).grid(row = 13, column =
   0, columnspan = 3)
    self.AMP_button = tk.Radiobutton(self.master, text = "
   Amperometry", font=("Courier", 15), bg = "grey",
              variable = self.choose_experiment, value = 2,
              command = self.Disable_CV).grid(row = 13, column =
   3, columnspan = 3)

    #### Cyclic Voltammetry User Interface ####
    # Scan rate
    self.Scan_rate      =   tk.Entry(self.master, justify = "right
   ")
    self.Scan_rate.grid(row = 17, column = 1, sticky = "e,w")
    self.Scan_rate.insert(0, self.con.scan_rate)

    # Min voltage
    self.Min_voltage    = tk.Entry(self.master, justify = "right")
    self.Min_voltage.grid(row = 18, column = 1, sticky = "e,w")
    self.Min_voltage.insert(0, self.con.min_voltage)

    # Max voltage
    self.Max_voltage    = tk.Entry(self.master, justify = "right")
    self.Max_voltage.grid(row = 19, column = 1, sticky = "e,w")
```

```
 self.Max_voltage.insert(0, self.con.max_voltage)

 # Start voltage
 self.Start_voltage    = tk.Entry(self.master, justify = "right")
 self.Start_voltage.grid(row = 20, column = 1, sticky = "e,w")
 self.Start_voltage.insert(0, self.con.start_voltage)

 # Number of cycles
 self.Number_of_cycles     = tk.Entry(self.master, justify = "right")
 self.Number_of_cycles.grid(row = 21, column = 1, sticky = "e,w")
 self.Number_of_cycles.insert(0, self.con.number_of_cycles)



 # Send CV settings
 self.send_cv_settings   = tk.Button(self.master, text = "Send CV settings", command = self.Send_CV_Settings)
 self.send_cv_settings.grid(row = 28, column = 1, columnspan = 1, sticky = "e,w")

 # Plot CV title name
 self.plot_title     =   tk.Entry(self.master)
 self.plot_title.grid(row = 24, column = 2, sticky = "e,w")

 # Plot CV legend name
 self.plot_legend    =   tk.Entry(self.master)
 self.plot_legend.grid(row = 25, column = 2, sticky = "e,w")

 # Run CV
 self.run_CV_scan    = tk.Button(self.master, text = "Run CV Scan", command = self.Start_CV_scan)
 self.run_CV_scan.grid(row = 29, column = 1, columnspan = 1, sticky = "e,w")

 # File name
 self.file_name      =   tk.Entry(self.master)
 self.file_name.grid(row = 33, column = 2, sticky = "e,w")

 # Save data
 self.save_data      = tk.Button(self.master, text = "Save Data and Settings", command=self.Save_data_and_settings)
 self.save_data.grid(row = 34, column = 2, columnspan = 1, sticky = "e,w")


 #### Amperometry ####
```

```
        # Run AMP
        self.run_AMP_scan   = tk.Button(self.master, text = "Run AMP Scan", command=self.Start_amperometry)   # Run AMP button
        self.run_AMP_scan.grid(row = 28, column = 3,  columnspan = 1, sticky = "e,w")

        # Stop AMP
        self.stop_AMP_scan    = tk.Button(self.master, text = "Stop AMP Scan", command=self.Stop_amperometry)   # Run AMP button
        self.stop_AMP_scan.grid(row = 29, column = 3, columnspan = 1, sticky = "e,w")


        # Amperometry voltage
        self.amp_voltage    =   tk.Entry(self.master, justify = "right")
        self.amp_voltage.grid(row = 19, column = 4, sticky = "e,w")
        self.amp_voltage.insert(0, self.con.amp_voltage)



        #####################

        self.choose_experiment.set(1)       # Initial state is CV
        self.Disable_AMP()                # Disable amperometry options

    def Disable_AMP(self):
        """
        If Cyclic Voltammetry is choosen, all entrys for Amperometry will be disabled.
        """
        self.run_AMP_scan["state"] = "disable"
        self.stop_AMP_scan["state"] = "disable"
        self.amp_voltage["state"] = "disable"
        self.run_CV_scan["state"] = "normal"
        self.send_cv_settings["state"] = "normal"
        self.Start_voltage["state"] = "normal"
        self.Max_voltage["state"] = "normal"
        self.Min_voltage["state"] = "normal"
        self.Scan_rate["state"] = "normal"

    def Disable_CV(self):
        """
        If Amperometry is choosen, all entrys for Cyclic Voltammetry will be disabled.
        """
        self.run_CV_scan["state"] = "disable"
        self.send_cv_settings["state"] = "disable"
        self.Start_voltage["state"] = "disable"
```

```
    self.Max_voltage["state"] = "disable"
    self.Min_voltage["state"] = "disable"
    self.Scan_rate["state"] = "disable"
    self.run_AMP_scan["state"] = "normal"
    self.amp_voltage["state"] = "normal"
    self.stop_AMP_scan["state"] = "normal"


  def Send_CV_Settings(self):
    """
    Functions collects all settings from user and sends them to the potentiostat.
    """
    #### Collect all settings for Cyclic Voltammetry ####
    _scan_rate = float(self.Scan_rate.get())
    _min_voltage = int(self.Min_voltage.get())
    _max_voltage = int(self.Max_voltage.get())
    _start_voltage = int(self.Start_voltage.get())
    _number_of_cycles = int(self.Number_of_cycles.get())

    #### Send scan rate if value is within range ####
    if (self.con.min_scan_rate <= _scan_rate) and (self.con.max_scan_rate >= _scan_rate):
      self.user.set_Scan_rate(_scan_rate)
    else:
      self.con.scan_rate_out_of_range()
      return

    #### Send min, max, start, number of cycles to LUT ####
    # Min voltage
    if (_min_voltage <= self.con.min_voltage_limit) or (_min_voltage >= self.con.max_voltage_limit):
      self.con.min_voltage_out_of_range()
      return
    # Max voltage
    elif (_max_voltage <= self.con.min_voltage_limit) or (_max_voltage >= self.con.max_voltage_limit):
      self.con.max_voltage_out_of_range()
      return
    # Start voltage
    elif (_start_voltage < _min_voltage) or (_start_voltage > _max_voltage):
      self.con.start_voltage_out_of_range()
    # Send values to potentiostat
    else:
      self.user.make_LookUpTable(_min_voltage, _max_voltage, _start_voltage)

    #### Send number of cycles ####
```

```
    if (_number_of_cycles >= self.con.min_number_of_cycles) and (_number_of_cycles <= self.con.max_number_of_cycles):
      self.user.set_number_of_cycles(_number_of_cycles)
    else:
      self.con.number_of_cycles_out_of_range()
      return

    #### Print to command window what happens ####
    self.con.settings_sent()

  def Start_CV_scan(self):
    """
    Function start cyclic voltammetry with given settings.
    """
    #### Store plot settings in variables ####
    self.user.plot_title_store = str(self.plot_title.get())
    self.user.plot_legend_store = str(self.plot_legend.get())

    #### Print to command window what happens ####
    self.user.con.plot_title_message();
    self.con.plot_legend_message();
    self.con.start_CV_message()

    #### Start cyclic voltammetry ####
    self.user.run_CyclicVoltammetry()

    #### Print to command window what happens ####
    self.con.end_CV_message()

  def Start_amperometry(self):
    """
    Function start amperometry with given settings.
    """
    #### Store plot settings in variables ####
    self.user.plot_title_store = str(self.plot_title.get())
    self.user.plot_legend_store = str(self.plot_legend.get())

    #### Print to command window what happens ####
    self.user.con.plot_title_message();
    self.con.plot_legend_message();
    self.con.start_AMP_message()

    #### Start amperometry ####
    send_amp_voltage  = str(self.func.Convert_voltage_to_DVDAC_value(self.con.amp_voltage)).zfill(4)

    #### Potentiostat command to start amperometry ####
    amp_voltage_formatted = "A {}".format(send_amp_voltage)
    self.comm.usb_write(amp_voltage_formatted)
```

```

    #self.current_data_store = [] # Empty stored data
    self.collect_data_amperometry()

  def Stop_amperometry(self):
    """
    Function stops amperometry
    """
    #### Stop cyclic voltammetry ####
    if self.after_id:
      self.master.after_cancel(self.after_id)
      self.after_id = None

    self.comm.usb_write("S")

    #### Print to command window what happens ####
    self.con.stop_AMP_message()

    ### Generate time array ###
    self.time_data_store = self.func.AMP_Time_array(self.
   current_data_store)

    #### Plot data ####
    self.func.Plot_AMP_data(self.current_data_store, self.
   time_data_store, self.user.plot_title_store, self.user.
   plot_legend_store)

  def collect_data_amperometry(self):
    #### Collect data ####
    data_raw = []

    data = self.comm.usb_collect_data()
    data_raw.extend(data)

    data_int16 = self.func.convert_uint8_to_int16_AMP(data)
    current_value = (-1*data_int16 / 20000)
    self.current_data_store.append(current_value)

    self.after_id = self.master.after(20, self.
   collect_data_amperometry)

  def Save_data_and_settings(self):
    self.user.filename_store = str(self.file_name.get())  # Store
   filename in variable

    if self.choose_experiment.get() == 1:          # Check if CV is
    the data to be saved
      self.user.Save_Data_CV()
    else:                               # Check if AMP is the data to be
```

```
    saved
      self.user.Save_Data_AMP(self.current_data_store, self.
   time_data_store)

    #### Print to command window what happens ####
    self.con.save_data_message()



if __name__ == '__main__':
  app = Potentiostat_GUI()
  app.mainloop()
```

8.4.0.2 Potentiostat_userinput.py

```python
import time
import pandas as pd
import matplotlib.pyplot as plt
import Potentiostat_communication
import Potentiostat_functionality
import Potentiostat_Constants
import Potentiostat_GUI

class UserInput:
  """
  Class that collects all userinputs and sends them to the potentiostat
  """

  def __init__(self):
    #### Importing classes to variables ####
    self.func = Potentiostat_functionality.Potentiostat() # Path to functionality class
    self.con  = Potentiostat_Constants.Constants()
    self.comm = Potentiostat_communication.Communication()  # Path to communication class
    self.dev, self.ep_out, self.ep_in = self.comm.usb_connect(self.comm.vendor_id, self.comm.product_id) # Communication variables

    #### Storage for users set values. Used for later saving. ####
    self.scan_rate_store    =   None
    self.min_voltage_store    =   None
    self.max_voltage_store    =   None
    self.start_voltage_store  =   None
    self.current_range_store  =   None
    self.number_of_cycles_store =   None

    #### Storage for users plot settings ####
    self.filename_store     =   "CV"
    self.plot_title_store   =   " "
    self.plot_legend_store    =   "Data"

    #### Storage of data arrays ####
    self.voltage_data_store   =   None
    self.current_data_store   =   None
    self.moving_average_store   =   None

  def set_Scan_rate(self, scan_rate):
    """
    Function recieves users scan_rate and convert it to formatted text, and sends it to the potentiostat.
```

```
        :param scan_rate: users scan rate [V/s]
        """
        send_scan_rate = self.func.Scan_Rate(scan_rate)     # Converts scan rate to appropriate clock timing
        scan_rate_formatted = "C {}".format(send_scan_rate)   # Potentiostat command for scan rate: "C xxxxxxxx"
        self.comm.usb_write(scan_rate_formatted)        # Send command
        time.sleep(0.3)                          # Wait 0.1 s for messages to be recieved and handled
        self.scan_rate_store = scan_rate            # Storing scan rate variable

        #### Print to command window what happens ####
        print(self.con.divider)
        print("Scan rate input: {} V/s".format(scan_rate))
        print("Command sent: {}".format(scan_rate_formatted))


    def set_number_of_cycles(self, number_of_cycles):
        """
        Function that sends the number of cycles to the potentiostat.
        :param number_of_cycles: integer of cycles
        """
        #### Zeropad to two digits ####
        send_number_of_cycles = str(number_of_cycles).zfill(2)

        #### Potentiostat command for number of cycles: "N xx"
        number_of_cycles_formatted = "N {0}".format(send_number_of_cycles)
        self.comm.usb_write(number_of_cycles_formatted)       # Send command
        time.sleep(0.3)                            # Wait 0.2 s for messages to be recieved and handled

        #### Store variable for later settings save function ####
        self.number_of_cycles_store   =   number_of_cycles

        #### Print to command window what happens ####
        print(self.con.divider)
        print("Number of cycles input: {}".format(number_of_cycles))
        print("Command sent: {}".format(number_of_cycles_formatted))

    def make_LookUpTable(self, min_voltage, max_voltage, start_voltage):
        """
        Function recieves min, max and start voltage, convert it to formatted text and sends it to the potentiostat.
        The potentistat will then make a lookup table from the values.
        :param min_voltage: minimum voltage
```

```
    :param max_voltage: maximum voltage
    :param start_voltage: start voltage
    """
    #### Convert to DVDAC values and zeropad to 4 digits each ####
    send_min_voltage  = str(self.func.
   Convert_voltage_to_DVDAC_value(min_voltage)).zfill(4)
    send_max_voltage  = str(self.func.
   Convert_voltage_to_DVDAC_value(max_voltage)).zfill(4)
    send_start_voltage  = str(self.func.
   Convert_voltage_to_DVDAC_value(start_voltage)).zfill(4)

    #### Potentiostat command for LUT: "L xxxx yyyy zzzz" [min,
   max, start]
    lut_values_formatted = "L {0} {1} {2}".format(send_min_voltage
   , send_max_voltage, send_start_voltage)
    self.comm.usb_write(lut_values_formatted)       # Send command
    time.sleep(0.3)                           # Wait 0.2 s for messages
   to be recieved and handled

    #### Store variable for later settings save function ####
    self.min_voltage_store    =   min_voltage
    self.max_voltage_store    =   max_voltage
    self.start_voltage_store  =   start_voltage

    #### Print to command window what happens ####
    print(self.con.divider)
    print("Minimum voltage input: {} mV".format(min_voltage))
    print("Maximum voltage input: {} mV".format(max_voltage))
    print("Start voltage input: {} mV".format(start_voltage))
    print("Command sent: {}".format(lut_values_formatted))

  def run_CyclicVoltammetry(self):
    """
    Function starts cyclic voltammetry and collects data in a list
    and plot the data.
    """

    #### Potentiostat command to start cyclic voltammetry ####
    self.comm.usb_write("R")

    #### Collect data ####
    data_raw = []
    formatted_data = []
    CV_RUN = True
    while CV_RUN:
      data = self.comm.usb_collect_data()
      data_raw.extend(data)

      status, data_int16 = self.func.convert_uint8_to_int16(data)
```

```
      formatted_data.extend(data_int16)
      CV_RUN = status

    #### Convert voltage to current ####
    # Current values are inverted, therefore *-1
    # Divide by 20k by Ohms law. Resistance in TIA is set to 20k Ohm.
    current_data = []
    for elements in formatted_data:
      current = (-1*elements / 20000)
      current_data.append(current)

    #### Moving average ####
    # Rolling mean at 5% of the total number of steps
    N = int(abs(self.max_voltage_store - self.min_voltage_store)*0.05)

    # Moving average of data
    moving_average = []
    average = 0

    for i in range(N):
      average += current_data[i]
      moving_average.append(average / (i+1))

    for i in range(N, len(current_data)):
      average += -current_data[i-N] + current_data[i]
      moving_average.append(average/N)

    #### Generate voltage data for plotting ####
    voltage_data = self.func.Compute_voltage_data(self.min_voltage_store, self.max_voltage_store, self.start_voltage_store, self.number_of_cycles_store)

    #### Store data in class ####
    self.voltage_data_store   = voltage_data
    self.current_data_store   = current_data
    self.moving_average_store   = moving_average

    #### Plot data ####
    self.func.Plot_CV_data(voltage_data, current_data, moving_average, self.min_voltage_store,
      self.max_voltage_store, self.plot_title_store, self.plot_legend_store)

  def Save_Data_CV(self):
    d = {'Voltage[mV]': self.voltage_data_store, 'Current Raw [mA]': self.current_data_store, 'Current Movinger Average[mA]': self.moving_average_store}
```

```

        df = pd.DataFrame(d)

        df.to_csv("{0}_{1}V-s_{2}cycles.csv".format(self.filename_store, self.scan_rate_store, self.number_of_cycles_store), index=False)

    def Save_Data_AMP(self, current_data, time_data):
        d = {'Time[s]': time_data, 'Current Raw [mA]': current_data}

        df = pd.DataFrame(d)

        df.to_csv("Amperometry_{0}.csv".format(self.filename_store), index=False)
```

8.4.0.3 Potentiostat_functionality.py

```
######## Potentiostat functionality ########
"""
Class to generate functions to the potentiostat
"""
import matplotlib.pyplot as plt
import numpy as np

import Potentiostat_Constants
import Potentiostat_userinput
import Potentiostat_functionality
con = Potentiostat_Constants.Constants()

class Potentiostat(object):
  """
  Class for all calculations to be sent to the potentiostat.
  """
  def __init__(self):
    self.clk_freq = con.clk_freq
    self.voltage_range = con.voltage_range
    self.dac_resolution = con.dac_resolution
    self.voltage_step = con.voltage_step


  def Scan_Rate(self, voltage_rate):
    """
    Function that converts scan rate [V/s] to number of clock counts in potentiostat.
    Thereafter it zero-padds the period to 8 digits for it to send to the potentiostat.
    :param voltage_rate: scan rate in V/s
    :return Period_padded: period in potentiostat with 8 digits
    """
    step_size = self.voltage_step                    # V/step
    #Period = int((step_size * self.clk_freq / voltage_rate) / 2) - 1 # number of clk pulses to count
    Period = int((step_size * self.clk_freq / voltage_rate)) - 1  # number of clk pulses to count
    print(con.divider)
    frequency = (self.clk_freq / Period) / 2
    print("Measurement frequency: {} Hz".format(frequency))
    Period_padded = str(Period).zfill(8)                   # pads with zero on left side, total of 8 digits
    return Period_padded

  def Convert_voltage_to_DVDAC_value(self, input_voltage):
    """
    Converts input voltage to a value recognized by the DVDAC.
```

```python
    :param input_voltage: voltage to be converted
    :return dac_voltage: converted voltage
    """
    dac_voltage = input_voltage + con.reference_voltage

    return dac_voltage

  def Number_of_steps(self, min_voltage_bit, max_voltage_bit, number_of_cycles):
    """
    Function to calculate number of steps the potentiostat will do for a complete scan.
    :param min_voltage_bit: minimum voltage value
    :param max_voltage_bit: maximum voltage value
    :param number_of_cycles: number of cycles
    :return number_of_steps: number of steps
    """
    number_of_steps = int(2*(abs(max_voltage_bit - min_voltage_bit) - 1) * number_of_cycles)  # -1 start_value (double count)
    return number_of_steps

  def convert_uint8_to_int16(self, uint8_data):
    """
    Converts data from double uint8 to int16.
    :param uint8_data: data set with uint8 values
    :return
    """
    not_found = True
    data_length = int(len(uint8_data) / 2)
    int16_array = [0] * data_length
    max_value = (2 ** 16) / 2
    for i in range(data_length):
      hold = uint8_data.pop(0) + uint8_data.pop(0) * 256
      if hold == con.determination_value:
        int16_array.pop(-1)
        not_found = False
        return not_found, int16_array
      if hold >= max_value:
        hold -= 2 * max_value
      int16_array[i] = hold

    return not_found, int16_array

  def convert_uint8_to_int16_AMP(self, uint8_data):
    """
    Converts data from double uint8 to int16.
    :param uint8_data: data set with uint8 values
    :return
    """
```

```
    hold = uint8_data.pop(0) + uint8_data.pop(0) * 256
    max_value = (2 ** 16) / 2
    if hold >= max_value:
        hold -= 2 * max_value
    return hold

  def Compute_voltage_data(self, min_value, max_value, start_value
   , number_of_cycles):
    """
    Generates the voltage data array for the CV-cycle.
    :param min_value: minimum voltage value
    :param max_value: maximum voltage value
    :param start_value: start voltage value
    :param number_of_cycles: number of cycles to run
    :return array of the voltage data
    """
    array = []

    # Defines UP and DOWN direction for the sweep
    if ( start_value == min_value ):
      direction_up = True

    elif ( start_value == max_value):
      direction_up = False

    else:
      direction_up = True

    array.append(start_value) # Sets initial voltage data value
    index_value = start_value # Index for iterating through the
   range
    cycles_index = 0      # Index for number of cycles

    while ( cycles_index <= number_of_cycles ):
      if ( direction_up == True ):
        index_value += 1
      else:
        index_value -= 1

      if ( index_value == start_value ):
        cycles_index += 1
        if ( cycles_index == number_of_cycles):
          return array

      if ( index_value >= max_value ):
        direction_up = False
      if ( index_value <= min_value ):
        direction_up = True

```

```python
      array.append(index_value)

  def Plot_CV_data(self, voltage, current, average, x_min, x_max, title, legend):
    """
    Function to plot the measured data.
    :param voltage: voltage data
    :param current: current data
    :param average: averag current data with 5% rolling average
    :param x_min: minimum voltage
    :param x_max: maximum voltage
    :param title: plot title
    :param legend: plot legend
    """
    user = Potentiostat_userinput.UserInput()

    #### Convert to uA ####
    current_data = []
    average_data = []
    for i in range(len(current)):
      current_data.append(current[i]*1000)
      average_data.append(average[i]*1000)

    #### Configure x-axis ####
    xmin = x_min - abs(x_min*0.15)
    xmax = x_max + abs(x_max*0.15)

    current_data_np = np.array(current_data)
    voltage_np = np.array(voltage)

    plt.ion()

    plt.figure()
    plt.suptitle("CV - {}".format(title))
    plt.title("Raw data")
    plt.xlim(xmin, xmax)
    plt.xlabel("Voltage [mV]")
    plt.ylabel("Current [$\mu$A]")
    plt.plot(voltage, current_data, label="{} - raw data".format(legend))
    plt.legend(loc="best")

    plt.show()

  def AMP_Time_array(self, current):
    """
    Generates an array of the time of the amperometric scan.
    :param voltage: current
    :return time array
```

```
    """
    total_time = len(current)*0.025        #25 ms per sample
    time_np = np.linspace(0, total_time, len(current))
    return time_np

  def Plot_AMP_data(self, current, time, title, legend):
    """
    Function to plot the measured data.
    :param current: current data
    :param time: time data
    :param title: plot title
    :param legend: plot legend
    """
    user = Potentiostat_userinput.UserInput()

    #### Convert to uA ####
    current_data = []
    for i in range(len(current)):
      current_data.append(current[i]*1000)

    current_data_np = np.array(current_data)

    plt.ion()

    #### Plot data ####
    plt.figure()
    plt.suptitle("AMP - {}".format(title))
    plt.title("Amperometry")
    plt.xlabel("Time [s]")
    plt.ylabel("Current [$\mu$A]")
    plt.plot(time, current_data, label="{} - raw data".format(legend))
    plt.legend(loc="best")

    plt.show()
```

8.4.0.4 Potentiostat_communication.py

```python
######## Potentiostat_communication ########
"""
Communication script to control the potentiostat.
"""

### Standard librabries ###

### Installed libraries ###
import usb.core
import usb.util
### Local files ###
import Potentiostat_Constants
con = Potentiostat_Constants.Constants()


class Communication(object):
  """
  Class that handles all communication with usb microcontroller.
  """
  def __init__(self, vendor_id=con.USB_VENDOR_ID, product_id=con.USB_PRODUCT_ID):
    self.vendor_id = vendor_id
    self.product_id = product_id
    self.found = False
    self.device, self.ep_out, self.ep_in = self.usb_connect(vendor_id, product_id)

  def usb_connect(self, vendor_id, product_id):
    """
    Attempt to connect with the PSoC device with a USBFS module.
    If the device is not found returns None.

    The pyUSB module is used. See documentation: https://pyusb.github.io/pyusb/.

    :param vendor_id: the USB vendor id, used to identify the proper device connected to the computer
    :param product_id: the USB product id
    :return: the device if found, None if not
    """
    try:
      dev = usb.core.find(idVendor=vendor_id, idProduct=product_id)
    except usb.core.NoBackendError:
      self.found = False
      return None, None, None

```

```
    if dev is None:
      self.found = False
      return None, None, None
    else:
      self.found = True

    dev.set_configuration()
    interface = dev.get_active_configuration()[(0, 0)]

    ep_out = usb.util.find_descriptor(interface,
                    custom_match= lambda e: \
                    usb.util.endpoint_direction(
                    e.bEndpointAddress) ==
                    usb.util.ENDPOINT_OUT)

    ep_in = usb.util.find_descriptor(interface,
                    custom_match= lambda e: \
                    usb.util.endpoint_direction(
                    e.bEndpointAddress) ==
                    usb.util.ENDPOINT_IN)
    assert ep_out is not None
    assert ep_in is not None

    return dev, ep_out, ep_in

  def usb_connection_test(self):
    self.usb_write(con.TEST_MESSAGE)
    self.usb_read()

  def usb_write(self, message):
    if len(message) > con.USB_OUT_BYTE_SIZE:
      print("ERROR: --- Message is too long. Maximum out byte size is {:d} ---".format(con.USB_OUT_BYTE_SIZE))
    else:
      self.ep_out.write(message)

  def usb_read(self, size=con.USB_IN_BYTE_SIZE, timeout=None):
    try:
      usb_input = self.ep_in.read(size, timeout)
    except Exception as error:
      print("ERROR: --- Failed to read. ---")
      print(self.ep_in.read(size, timeout))
    return usb_input

  def usb_collect_data(self):
    data_collect = self.usb_read(64,timeout=10000)
    return data_collect
```

8.4.0.5 Potentiostat_Constants.py

```python

class Constants:

  def __init__(self):
    # USB constants
    self.USB_OUT_BYTE_SIZE = 32
    self.USB_IN_BYTE_SIZE = 64
    self.USB_VENDOR_ID = 0x4B5
    self.USB_PRODUCT_ID = 0x81

    # DVDAC constants
    self.clk_freq = 24000000                      # Hz [24 MHz]
    self.dac_resolution = 4080                      # 12-bit DVDAC
    self.voltage_range = 4.080                      # V
    self.voltage_step = float(self.voltage_range / self.dac_resolution) # V/bit
    self.reference_voltage = 2032                   # mV

    # Cyclic Voltammetry settings
    self.min_voltage = -500
    self.max_voltage = 500
    self.start_voltage = -500
    self.scan_rate = 1.0
    self.number_of_cycles = 1
    self.current_range = [100, 70, 50, 25, 17, 8, 4, 2]   # uA
    self.test = 0
    self.determination_value = 49152

    # Amperometry settings
    self.amp_voltage = 500
    self.current_data_store = []
    self.amp_time_store = []

    #### Limitations on user inputs ####
    # Scan rate limitation
    self.min_scan_rate = self.voltage_step / 0.699051     # max period = 699 ms
    self.max_scan_rate = self.voltage_step / (83.33 * 10**(-9)) # min period = 83 ns

    # Voltage limitations
    self.min_voltage_limit  = -1 * self.reference_voltage   # mV
    self.max_voltage_limit  = self.reference_voltage      # mV

    # Number of cycles limitation
    self.min_number_of_cycles = 1                # cycles
    self.max_number_of_cycles = 99                 # cycles
```

```

    # Messages format
    self.divider = "
   ------------------------------------------------"
    self.error = "##################### ERROR
   #####################"

  def scan_rate_out_of_range(self):
    print(self.divider)
    print(self.error)
    print("SCAN RATE IS OUT OF RANGE.")
    print("Keep scan rate within: {0:.2f} mV/s and {1:.2f} V/s".
   format(self.min_scan_rate*1000, self.max_scan_rate))

  def min_voltage_out_of_range(self):
    print(self.divider)
    print(self.error)
    print("MINIMUM VOLTAGE IS OUT OF RANGE.")
    print("Keep minimum voltage within: {0} mV and {1} mV.".format
   (self.min_voltage_limit, self.max_voltage_limit))

  def max_voltage_out_of_range(self):
    print(self.divider)
    print(self.error)
    print("MAXIMUM VOLTAGE IS OUT OF RANGE.")
    print("Keep maximum voltage within: {0} mV and {1} mV.".format
   (self.min_voltage_limit, self.max_voltage_limit))

  def start_voltage_out_of_range(self):
    print(self.divider)
    print(self.error)
    print("START VOLTAGE IS OUT OF RANGE.")
    print("Keep start voltage within minimum and maximum voltage
   of your desire")

  def number_of_cycles_out_of_range(self):
    print(self.divider)
    print(self.error)
    print("NUMBER OF CYCLES IS OUT OF RANGE.")
    print("Keep number of cycles within: {0} and {1}.".format(self
   .min_number_of_cycles, self.max_number_of_cycles))

  def settings_sent(self):
    print(self.divider)
    print("Settings have been sent.")

  def start_CV_message(self):
    print(self.divider)
    print("Cyclic Voltammetry initialization has started.")
```

```

  def end_CV_message(self):
    print(self.divider)
    print("Cyclic Voltammetry is done.")

  def start_AMP_message(self):
    print(self.divider)
    print("Amperometry is running.")

  def stop_AMP_message(self):
    print(self.divider)
    print("Amperometry has ended.")

  def plot_title_message(self):
    print(self.divider)
    print("Plot title is stored.")

  def plot_legend_message(self):
    print(self.divider)
    print("Plot legend is stored.")

  def save_data_message(self):
    print(self.divider)
    print("Data is saved locally.")
```

8.5 Potentiostat Datasheet

The following document is generated by PSoC Creator and is a datasheet for all the configurations of the potentiostat.

PSoC® Creator™
Project Datasheet for Potentiostat_RevA8

Creation Time: 10/07/2020 16:41:24
User: DESKTOP-51KCO67\Reodor Felgen
Project: Potentiostat_RevA8
Tool: PSoC Creator 4.2

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
http://www.cypress.com

Copyright

Copyright © 2020 Cypress Semiconductor Corporation. All rights reserved. Any design information or characteristics specifically provided by our customer or other third party inputs contained in this document are not intended to be claimed under Cypress's copyright.

Trademarks

PSoC and CapSense are registered trademarks of Cypress Semiconductor Corporation. PSoC Creator is a trademark of Cypress Semiconductor Corporation. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Philips I2C Patent Rights

Purchase of I2C components from Cypress or one of its sublicensed Associated Companies conveys a license under the Philips I2C Patent Rights to use these components in an I2C system, provided that the system conforms to the I2C Standard Specification as defined by Philips. As from October 1st, 2006 Philips Semiconductors has a new trade name, NXP Semiconductors.

Disclaimer

CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. While reasonable precautions have been taken, Cypress assumes no responsibility for any errors that may appear in this document. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of a Cypress product in a life support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Flash Code Protection

Cypress products meet the specifications contained in their particular Cypress PSoC Datasheets. Cypress believes that its family of PSoC products is one of the most secure families of its kind on the market today, regardless of how they are used. There may be methods, unknown to Cypress, that can breach the code protection features. Any of these methods, to our knowledge, would be dishonest and possibly illegal. Neither Cypress nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as 'unbreakable.'

Cypress is willing to work with the customer who is concerned about the integrity of their code. Code protection is constantly evolving. We at Cypress are committed to continuously improving the code protection features of our products.

Table of Contents

1 Overview

The Cypress PSoC 5 is a family of 32-bit devices with the following characteristics:

- High-performance 32-bit ARM Cortex-M3 core with a nested vectored interrupt controller (NVIC) and a high-performance DMA controller
- Digital system that includes configurable Universal Digital Blocks (UDBs) and specific function peripherals, such as USB, I2C and SPI
- Analog subsystem that includes 20-bit Delta Sigma converters (ADC), SAR ADCs, 8-bit DACs that can be configured for 12-bit operation, comparators, op amps and configurable switched capacitor (SC) and continuous time (CT) blocks to create PGAs, TIAs, mixers, and more
- Several types of memory elements, including SRAM, flash, and EEPROM
- Programming and debug system through JTAG, serial wire debug (SWD), and single wire viewer (SWV)
- Flexible routing to all pins

Figure 1 shows the major components of a typical CY8C58LP series member PSoC 5LP device. For details on all the systems listed above, please refer to the PSoC 5LP Technical Reference Manual .

Figure 1. CY8C58LP Device Series Block Diagram

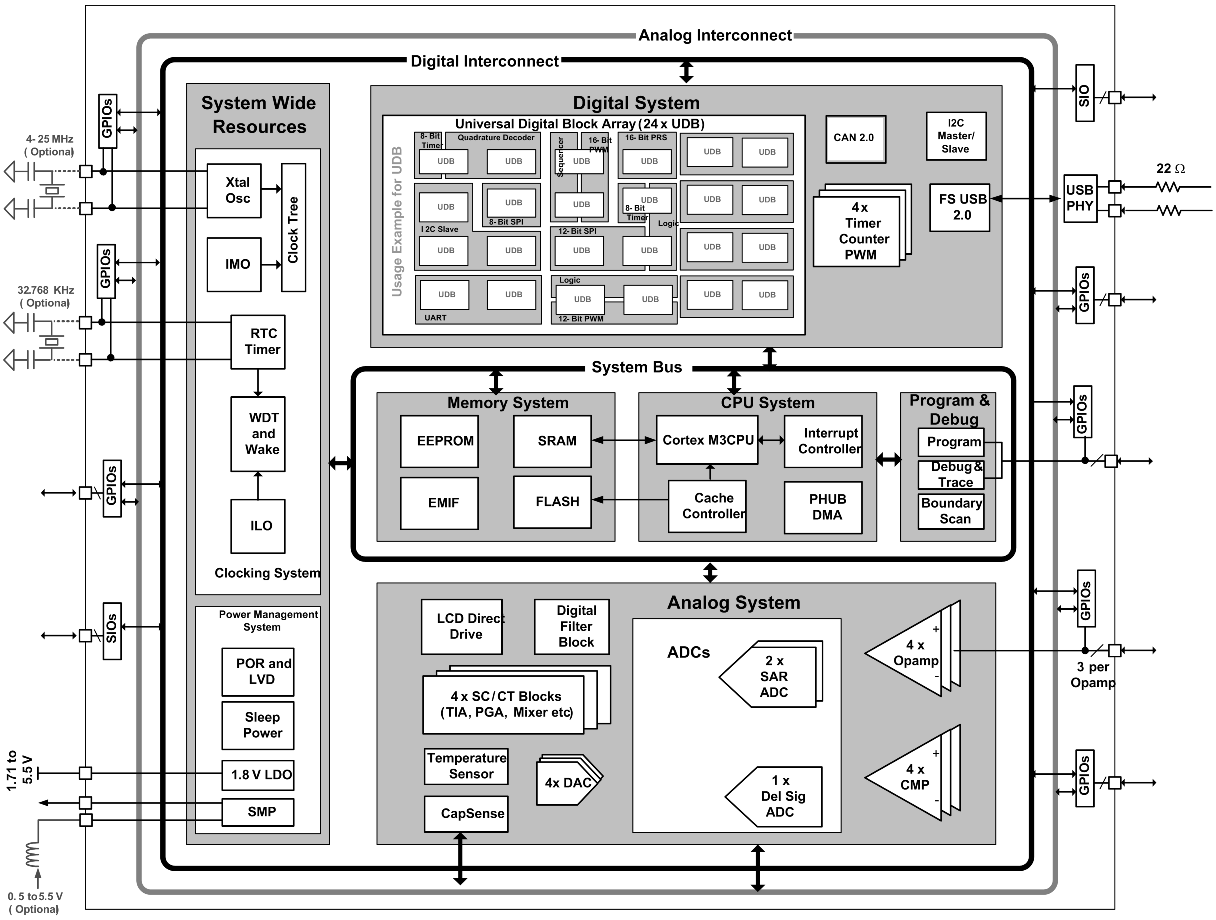


Table 1 lists the key characteristics of this device.

Table 1. Device Characteristics

Name	Value
Part Number	CY8C5868AXI-LP035
Package Name	100-TQFP
Family	PSoC 5LP
Series	CY8C58LP
Max CPU speed (MHz)	0
Flash size (kB)	256
SRAM size (kB)	64
EEPROM size (bytes)	2048
Vdd range (V)	1.71 to 5.5
Automotive qualified	No (Industrial Grade Only)
Temp range (Celsius)	-40 to 85
JTAG ID	0x2E123069

NOTE: The CPU speed noted above is the maximum available speed. The CPU is clocked by Bus Clock, listed in the System Clocks section below.

Table 2 lists the device resources that this design uses:

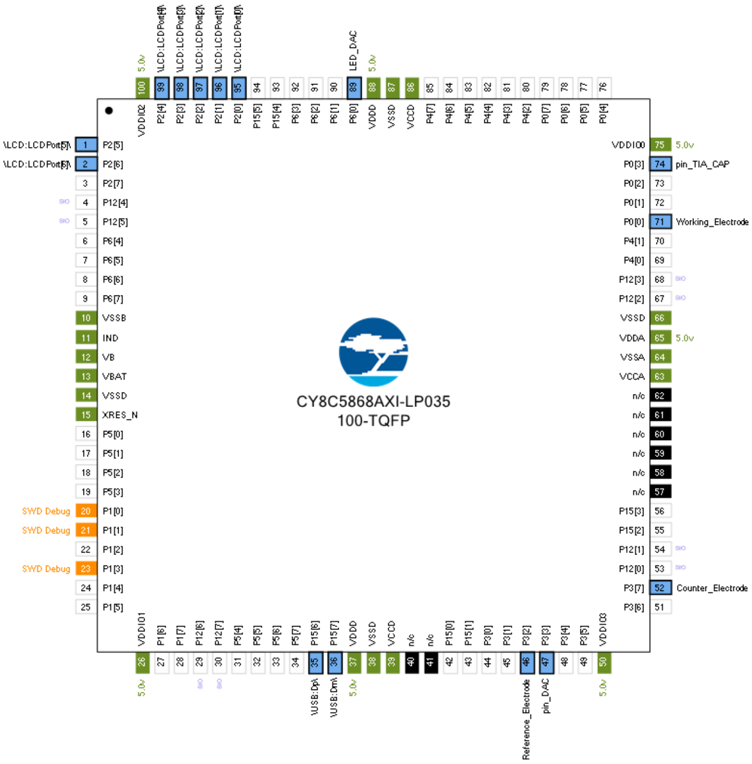
Table 2. Device Resources

Resource Type	Used	Free	Max	% Used
Digital Clocks	2	6	8	25.00 %
Analog Clocks	1	3	4	25.00 %
CapSense Buffers	0	2	2	0.00 %
Digital Filter Block	0	1	1	0.00 %
Interrupts	9	23	32	28.13 %
IO	18	54	72	25.00 %
Segment LCD	0	1	1	0.00 %
CAN 2.0b	0	1	1	0.00 %
I2C	0	1	1	0.00 %
USB	1	0	1	100.00 %
DMA Channels	1	23	24	4.17 %
Timer	0	4	4	0.00 %
UDB				
Macrocells	4	188	192	2.08 %
Unique P-terms	2	382	384	0.52 %
Total P-terms	3			
Datapath Cells	3	21	24	12.50 %
Status Cells	1	23	24	4.17 %
Statusl Registers	1			
Control Cells	1	23	24	4.17 %
Control Registers	1			
Opamp	1	3	4	25.00 %
Comparator	0	4	4	0.00 %
Delta-Sigma ADC	1	0	1	100.00 %
LPF	0	2	2	0.00 %
SAR ADC	0	2	2	0.00 %
Analog (SC/CT) Blocks	1	3	4	25.00 %
DAC				
VIDAC	2	2	4	50.00 %

2 Pins

Figure 2 shows the pin layout of this device.

Figure 2. Device Pin Layout



2.1 Hardware Pins

Table 3 contains information about the pins on this device in device pin order. (No connection ["n/c"] pins have been omitted.)

Table 3. Device Pins

Pin	Port	Name	Type	Drive Mode	Reset State
1	P2[5]	\LCD:LCDPort[5]\	Software In/Out	Strong drive	HiZ Analog Unb
2	P2[6]	\LCD:LCDPort[6]\	Software In/Out	Strong drive	HiZ Analog Unb
3	P2[7]	GPIO [unused]			HiZ Analog Unb
4	P12[4]	SIO [unused]			HiZ Analog Unb
5	P12[5]	SIO [unused]			HiZ Analog Unb
6	P6[4]	GPIO [unused]			HiZ Analog Unb
7	P6[5]	GPIO [unused]			HiZ Analog Unb
8	P6[6]	GPIO [unused]			HiZ Analog Unb
9	P6[7]	GPIO [unused]			HiZ Analog Unb
10	VSSB	VSSB	Dedicated		
11	IND	IND	Dedicated		
12	VB	VB	Dedicated		
13	VBAT	VBAT	Dedicated		
14	VSSD	VSSD	Power		
15	XRES_N	XRES_N	Dedicated		
16	P5[0]	GPIO [unused]			HiZ Analog Unb
17	P5[1]	GPIO [unused]			HiZ Analog Unb
18	P5[2]	GPIO [unused]			HiZ Analog Unb
19	P5[3]	GPIO [unused]			HiZ Analog Unb
20	P1[0]	Debug:SWD_IO	Reserved		
21	P1[1]	Debug:SWD_CK	Reserved		
22	P1[2]	GPIO [unused]			HiZ Analog Unb
23	P1[3]	Debug:SWV	Reserved		
24	P1[4]	GPIO [unused]			HiZ Analog Unb
25	P1[5]	GPIO [unused]			HiZ Analog Unb
26	VDDIO1	VDDIO1	Power		
27	P1[6]	GPIO [unused]			HiZ Analog Unb
28	P1[7]	GPIO [unused]			HiZ Analog Unb
29	P12[6]	SIO [unused]			HiZ Analog Unb
30	P12[7]	SIO [unused]			HiZ Analog Unb
31	P5[4]	GPIO [unused]			HiZ Analog Unb
32	P5[5]	GPIO [unused]			HiZ Analog Unb
33	P5[6]	GPIO [unused]			HiZ Analog Unb
34	P5[7]	GPIO [unused]			HiZ Analog Unb
35	P15[6]	\USB:Dp\	Analog	HiZ analog	HiZ Analog Unb
36	P15[7]	\USB:Dm\	Analog	HiZ analog	HiZ Analog Unb
37	VDDD	VDDD	Power		
38	VSSD	VSSD	Power		
39	VCCD	VCCD	Power		
42	P15[0]	GPIO [unused]			HiZ Analog Unb
43	P15[1]	GPIO [unused]			HiZ Analog Unb
44	P3[0]	GPIO [unused]			HiZ Analog Unb
45	P3[1]	GPIO [unused]			HiZ Analog Unb
46	P3[2]	Reference_Electrode	Analog	HiZ analog	HiZ Analog Unb

Pin	Port	Name	Type	Drive Mode	Reset State
47	P3[3]	pin_DAC	Analog	HiZ analog	HiZ Analog Unb
48	P3[4]	GPIO [unused]			HiZ Analog Unb
49	P3[5]	GPIO [unused]			HiZ Analog Unb
50	VDDIO3	VDDIO3	Power		
51	P3[6]	GPIO [unused]			HiZ Analog Unb
52	P3[7]	Counter_Electrode	Analog	HiZ analog	HiZ Analog Unb
53	P12[0]	SIO [unused]			HiZ Analog Unb
54	P12[1]	SIO [unused]			HiZ Analog Unb
55	P15[2]	GPIO [unused]			HiZ Analog Unb
56	P15[3]	GPIO [unused]			HiZ Analog Unb
63	VCCA	VCCA	Power		
64	VSSA	VSSA	Power		
65	VDDA	VDDA	Power		
66	VSSD	VSSD	Power		
67	P12[2]	SIO [unused]			HiZ Analog Unb
68	P12[3]	SIO [unused]			HiZ Analog Unb
69	P4[0]	GPIO [unused]			HiZ Analog Unb
70	P4[1]	GPIO [unused]			HiZ Analog Unb
71	P0[0]	Working_Electrode	Analog	HiZ analog	HiZ Analog Unb
72	P0[1]	GPIO [unused]			HiZ Analog Unb
73	P0[2]	GPIO [unused]			HiZ Analog Unb
74	P0[3]	pin_TIA_CAP	Analog	HiZ analog	HiZ Analog Unb
75	VDDIO0	VDDIO0	Power		
76	P0[4]	GPIO [unused]			HiZ Analog Unb
77	P0[5]	GPIO [unused]			HiZ Analog Unb
78	P0[6]	GPIO [unused]			HiZ Analog Unb
79	P0[7]	GPIO [unused]			HiZ Analog Unb
80	P4[2]	GPIO [unused]			HiZ Analog Unb
81	P4[3]	GPIO [unused]			HiZ Analog Unb
82	P4[4]	GPIO [unused]			HiZ Analog Unb
83	P4[5]	GPIO [unused]			HiZ Analog Unb
84	P4[6]	GPIO [unused]			HiZ Analog Unb
85	P4[7]	GPIO [unused]			HiZ Analog Unb
86	VCCD	VCCD	Power		
87	VSSD	VSSD	Power		
88	VDDD	VDDD	Power		
89	P6[0]	LED_DAC	Software In/Out	Strong drive	HiZ Analog Unb
90	P6[1]	GPIO [unused]			HiZ Analog Unb
91	P6[2]	GPIO [unused]			HiZ Analog Unb
92	P6[3]	GPIO [unused]			HiZ Analog Unb
93	P15[4]	GPIO [unused]			HiZ Analog Unb
94	P15[5]	GPIO [unused]			HiZ Analog Unb
95	P2[0]	\LCD:LCDPort[0]\	Software In/Out	Strong drive	HiZ Analog Unb
96	P2[1]	\LCD:LCDPort[1]\	Software In/Out	Strong drive	HiZ Analog Unb
97	P2[2]	\LCD:LCDPort[2]\	Software In/Out	Strong drive	HiZ Analog Unb
98	P2[3]	\LCD:LCDPort[3]\	Software In/Out	Strong drive	HiZ Analog Unb
99	P2[4]	\LCD:LCDPort[4]\	Software In/Out	Strong drive	HiZ Analog Unb
100	VDDIO2	VDDIO2	Power		

Abbreviations used in Table 3 have the following meanings:

- HiZ Analog Unb = Hi-Z Analog Unbuffered
- HiZ analog = High impedance analog

2.2 Hardware Ports

Table 4 contains information about the pins on this device in device port order. (No connection ["n/c"], power and dedicated pins have been omitted.)

Table 4. Device Ports

Port	Pin	Name	Type	Drive Mode	Reset State
P0[0]	71	Working_Electrode	Analog	HiZ analog	HiZ Analog Unb
P0[1]	72	GPIO [unused]			HiZ Analog Unb
P0[2]	73	GPIO [unused]			HiZ Analog Unb
P0[3]	74	pin_TIA_CAP	Analog	HiZ analog	HiZ Analog Unb
P0[4]	76	GPIO [unused]			HiZ Analog Unb
P0[5]	77	GPIO [unused]			HiZ Analog Unb
P0[6]	78	GPIO [unused]			HiZ Analog Unb
P0[7]	79	GPIO [unused]			HiZ Analog Unb
P1[0]	20	Debug:SWD_IO	Reserved		
P1[1]	21	Debug:SWD_CK	Reserved		
P1[2]	22	GPIO [unused]			HiZ Analog Unb
P1[3]	23	Debug:SWV	Reserved		
P1[4]	24	GPIO [unused]			HiZ Analog Unb
P1[5]	25	GPIO [unused]			HiZ Analog Unb
P1[6]	27	GPIO [unused]			HiZ Analog Unb
P1[7]	28	GPIO [unused]			HiZ Analog Unb
P12[0]	53	SIO [unused]			HiZ Analog Unb
P12[1]	54	SIO [unused]			HiZ Analog Unb
P12[2]	67	SIO [unused]			HiZ Analog Unb
P12[3]	68	SIO [unused]			HiZ Analog Unb
P12[4]	4	SIO [unused]			HiZ Analog Unb
P12[5]	5	SIO [unused]			HiZ Analog Unb
P12[6]	29	SIO [unused]			HiZ Analog Unb
P12[7]	30	SIO [unused]			HiZ Analog Unb
P15[0]	42	GPIO [unused]			HiZ Analog Unb
P15[1]	43	GPIO [unused]			HiZ Analog Unb
P15[2]	55	GPIO [unused]			HiZ Analog Unb
P15[3]	56	GPIO [unused]			HiZ Analog Unb
P15[4]	93	GPIO [unused]			HiZ Analog Unb
P15[5]	94	GPIO [unused]			HiZ Analog Unb
P15[6]	35	\USB:Dp\	Analog	HiZ analog	HiZ Analog Unb
P15[7]	36	\USB:Dm\	Analog	HiZ analog	HiZ Analog Unb
P2[0]	95	\LCD:LCDPort[0]\	Software In/Out	Strong drive	HiZ Analog Unb
P2[1]	96	\LCD:LCDPort[1]\	Software In/Out	Strong drive	HiZ Analog Unb
P2[2]	97	\LCD:LCDPort[2]\	Software In/Out	Strong drive	HiZ Analog Unb
P2[3]	98	\LCD:LCDPort[3]\	Software In/Out	Strong drive	HiZ Analog Unb
P2[4]	99	\LCD:LCDPort[4]\	Software In/Out	Strong drive	HiZ Analog Unb
P2[5]	1	\LCD:LCDPort[5]\	Software In/Out	Strong drive	HiZ Analog Unb
P2[6]	2	\LCD:LCDPort[6]\	Software In/Out	Strong drive	HiZ Analog Unb

Port	Pin	Name	Type	Drive Mode	Reset State
P2[7]	3	GPIO [unused]			HiZ Analog Unb
P3[0]	44	GPIO [unused]			HiZ Analog Unb
P3[1]	45	GPIO [unused]			HiZ Analog Unb
P3[2]	46	Reference_Electrode	Analog	HiZ analog	HiZ Analog Unb
P3[3]	47	pin_DAC	Analog	HiZ analog	HiZ Analog Unb
P3[4]	48	GPIO [unused]			HiZ Analog Unb
P3[5]	49	GPIO [unused]			HiZ Analog Unb
P3[6]	51	GPIO [unused]			HiZ Analog Unb
P3[7]	52	Counter_Electrode	Analog	HiZ analog	HiZ Analog Unb
P4[0]	69	GPIO [unused]			HiZ Analog Unb
P4[1]	70	GPIO [unused]			HiZ Analog Unb
P4[2]	80	GPIO [unused]			HiZ Analog Unb
P4[3]	81	GPIO [unused]			HiZ Analog Unb
P4[4]	82	GPIO [unused]			HiZ Analog Unb
P4[5]	83	GPIO [unused]			HiZ Analog Unb
P4[6]	84	GPIO [unused]			HiZ Analog Unb
P4[7]	85	GPIO [unused]			HiZ Analog Unb
P5[0]	16	GPIO [unused]			HiZ Analog Unb
P5[1]	17	GPIO [unused]			HiZ Analog Unb
P5[2]	18	GPIO [unused]			HiZ Analog Unb
P5[3]	19	GPIO [unused]			HiZ Analog Unb
P5[4]	31	GPIO [unused]			HiZ Analog Unb
P5[5]	32	GPIO [unused]			HiZ Analog Unb
P5[6]	33	GPIO [unused]			HiZ Analog Unb
P5[7]	34	GPIO [unused]			HiZ Analog Unb
P6[0]	89	LED_DAC	Software In/Out	Strong drive	HiZ Analog Unb
P6[1]	90	GPIO [unused]			HiZ Analog Unb
P6[2]	91	GPIO [unused]			HiZ Analog Unb
P6[3]	92	GPIO [unused]			HiZ Analog Unb
P6[4]	6	GPIO [unused]			HiZ Analog Unb
P6[5]	7	GPIO [unused]			HiZ Analog Unb
P6[6]	8	GPIO [unused]			HiZ Analog Unb
P6[7]	9	GPIO [unused]			HiZ Analog Unb

Abbreviations used in Table 4 have the following meanings:

- HiZ analog = High impedance analog
- HiZ Analog Unb = Hi-Z Analog Unbuffered

2.3 Software Pins

Table 5 contains information about the software pins on this device in alphabetical order. (Only software-accessible pins are shown.)

Table 5. Software Pins

Name	Port	Type	Reset State
\LCD:LCDPort[0]\	P2[0]	Software In/Out	HiZ Analog Unb
\LCD:LCDPort[1]\	P2[1]	Software In/Out	HiZ Analog Unb
\LCD:LCDPort[2]\	P2[2]	Software In/Out	HiZ Analog Unb
\LCD:LCDPort[3]\	P2[3]	Software In/Out	HiZ Analog Unb
\LCD:LCDPort[4]\	P2[4]	Software In/Out	HiZ Analog Unb
\LCD:LCDPort[5]\	P2[5]	Software In/Out	HiZ Analog Unb
\LCD:LCDPort[6]\	P2[6]	Software In/Out	HiZ Analog Unb
\USB:Dm\	P15[7]	Analog	HiZ Analog Unb
\USB:Dp\	P15[6]	Analog	HiZ Analog Unb
Counter_Electrode	P3[7]	Analog	HiZ Analog Unb
Debug:SWD_CK	P1[1]	Reserved	
Debug:SWD_IO	P1[0]	Reserved	
Debug:SWV	P1[3]	Reserved	
GPIO [unused]	P6[1]		HiZ Analog Unb
GPIO [unused]	P2[7]		HiZ Analog Unb
GPIO [unused]	P6[6]		HiZ Analog Unb
GPIO [unused]	P3[5]		HiZ Analog Unb
GPIO [unused]	P6[4]		HiZ Analog Unb
GPIO [unused]	P15[3]		HiZ Analog Unb
GPIO [unused]	P6[5]		HiZ Analog Unb
GPIO [unused]	P3[6]		HiZ Analog Unb
GPIO [unused]	P15[2]		HiZ Analog Unb
GPIO [unused]	P4[2]		HiZ Analog Unb
GPIO [unused]	P0[2]		HiZ Analog Unb
GPIO [unused]	P0[1]		HiZ Analog Unb
GPIO [unused]	P0[6]		HiZ Analog Unb
GPIO [unused]	P0[5]		HiZ Analog Unb
GPIO [unused]	P0[4]		HiZ Analog Unb
GPIO [unused]	P4[3]		HiZ Analog Unb
GPIO [unused]	P4[5]		HiZ Analog Unb
GPIO [unused]	P4[6]		HiZ Analog Unb
GPIO [unused]	P4[7]		HiZ Analog Unb
GPIO [unused]	P4[1]		HiZ Analog Unb
GPIO [unused]	P4[0]		HiZ Analog Unb
GPIO [unused]	P4[4]	128	HiZ Analog Unb
GPIO [unused]	P5[7]		HiZ Analog Unb
GPIO [unused]	P5[6]		HiZ Analog Unb
GPIO [unused]	P5[5]		HiZ Analog Unb
GPIO [unused]	P15[5]		HiZ Analog Unb

Name	Port	Type	Reset State
GPIO [unused]	P5[2]		HiZ Analog Unb
GPIO [unused]	P5[3]		HiZ Analog Unb
GPIO [unused]	P1[5]		HiZ Analog Unb
GPIO [unused]	P1[4]		HiZ Analog Unb
GPIO [unused]	P1[2]		HiZ Analog Unb
GPIO [unused]	P5[4]		HiZ Analog Unb
GPIO [unused]	P1[7]		HiZ Analog Unb
GPIO [unused]	P1[6]		HiZ Analog Unb
GPIO [unused]	P15[4]		HiZ Analog Unb
GPIO [unused]	P3[1]		HiZ Analog Unb
GPIO [unused]	P5[0]		HiZ Analog Unb
GPIO [unused]	P6[2]		HiZ Analog Unb
GPIO [unused]	P3[4]		HiZ Analog Unb
GPIO [unused]	P0[7]		HiZ Analog Unb
GPIO [unused]	P3[0]		HiZ Analog Unb
GPIO [unused]	P6[7]		HiZ Analog Unb
GPIO [unused]	P6[3]		HiZ Analog Unb
GPIO [unused]	P5[1]		HiZ Analog Unb
GPIO [unused]	P15[1]		HiZ Analog Unb
GPIO [unused]	P15[0]		HiZ Analog Unb
LED_DAC	P6[0]	Software In/Out	HiZ Analog Unb
pin_DAC	P3[3]	Analog	HiZ Analog Unb
pin_TIA_CAP	P0[3]	Analog	HiZ Analog Unb
Reference_Electrode	P3[2]	Analog	HiZ Analog Unb
SIO [unused]	P12[6]		HiZ Analog Unb
SIO [unused]	P12[5]		HiZ Analog Unb
SIO [unused]	P12[4]		HiZ Analog Unb
SIO [unused]	P12[7]		HiZ Analog Unb
SIO [unused]	P12[0]		HiZ Analog Unb
SIO [unused]	P12[1]		HiZ Analog Unb
SIO [unused]	P12[2]		HiZ Analog Unb
SIO [unused]	P12[3]		HiZ Analog Unb
Working_Electrode	P0[0]	Analog	HiZ Analog Unb

Abbreviations used in Table 5 have the following meanings:

- HiZ Analog Unb = Hi-Z Analog Unbuffered

For more information on reading, writing and configuring pins, please refer to:

- Pins chapter in the System Reference Guide
  - CyPins API routines
- Programming Application Interface section in the cy_pins component datasheet

3 System Settings

3.1 System Configuration

Table 6. System Configuration Settings

Name	Value
Device Configuration Mode	Compressed
Enable Error Correcting Code (ECC)	False
Store Configuration Data in ECC Memory	True
Instruction Cache Enabled	True
Enable Fast IMO During Startup	True
Unused Bonded IO	Allow but warn
Heap Size (bytes)	0x80
Stack Size (bytes)	0x0800
Include CMSIS Core Peripheral Library Files	True

3.2 System Debug Settings

Table 7. System Debug Settings

Name	Value
Debug Select	SWD+SWV (serial wire debug and viewer)
Enable Device Protection	False
Embedded Trace (ETM)	False
Use Optional XRES	False

3.3 System Operating Conditions

Table 8. System Operating Conditions

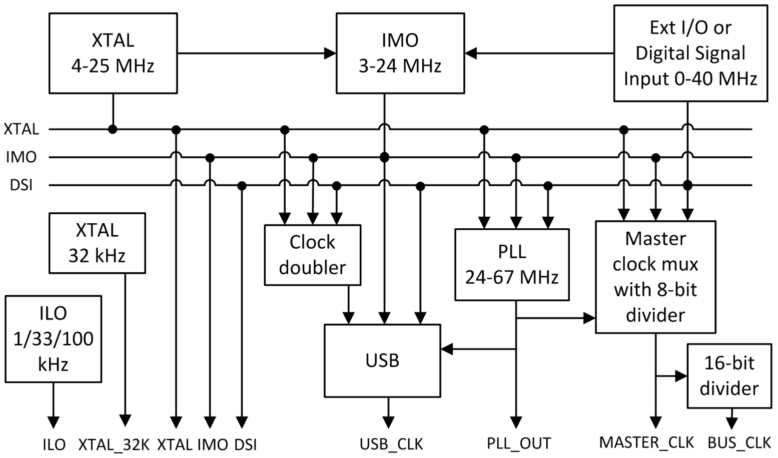
Name	Value
VDDA (V)	5.0
VDDD (V)	5.0
VDDIO0 (V)	5.0
VDDIO1 (V)	5.0
VDDIO2 (V)	5.0
VDDIO3 (V)	5.0
Variable VDDA	False
Temperature Range	-40C - 85/125C

4 Clocks

The clock system includes these clock resources:

- Four internal clock sources increase system integration:
  - 3 to 74.7 MHz Internal Main Oscillator (IMO) ±1% at 3 MHz
  - 1 kHz, 33 kHz, and 100 kHz Internal Low Speed Oscillator (ILO) outputs
  - 12 to 80 MHz clock doubler output, sourced from IMO, MHz External Crystal Oscillator (MHzECO), and Digital System Interconnect (DSI)
  - 24 to 80 MHz fractional Phase-Locked Loop (PLL) sourced from IMO, MHzECO, and DSI
- Clock generated using a DSI signal from an external I/O pin or other logic
- Two external clock sources provide high precision clocks:
  - 4 to 25 MHz External Crystal Oscillator (MHzECO)
  - 32.768 kHz External Crystal Oscillator (kHzECO) for Real Time Clock (RTC)
- Dedicated 16-bit divider for bus clock
- Eight individually sourced 16-bit clock dividers for the digital system peripherals
- Four individually sourced 16-bit clock dividers with skew for the analog system peripherals
- IMO has a USB mode that synchronizes to USB host traffic, requiring no external crystal for USB. (USB equipped parts only)

Figure 3. System Clock Configuration



4.1 System Clocks

Table 9 lists the system clocks used in this design.

Table 9. System Clocks

Name	Domain	Source	Desired Freq	Nominal Freq	Accuracy (%)	Start at Reset	Enabled
USB_CLK	DIGITAL	IMO	48 MHz	48 MHz	±0.25	False	True
IMO	DIGITAL		24 MHz	24 MHz	±0.25	True	True
MASTER_CLK	DIGITAL	PLL_OUT	? MHz	24 MHz	±0.25	True	True
BUS_CLK	DIGITAL	MASTER_CLK	? MHz	24 MHz	±0.25	True	True
PLL_OUT	DIGITAL	IMO	24 MHz	24 MHz	±0.25	True	True
ILO	DIGITAL		? MHz	100 kHz	-55,+100	True	True
XTAL 32kHz	DIGITAL		32.768 kHz	? MHz	±0	False	False
Digital Signal	DIGITAL		? MHz	? MHz	±0	False	False
XTAL	DIGITAL		24 MHz	? MHz	±0	False	False

4.2 Local and Design Wide Clocks

Local clocks drive individual analog and digital blocks. Design wide clocks are a user-defined optimization, where two or more analog or digital blocks that share a common clock profile (frequency, etc) can be driven from the same clock divider output source.

Figure 4. Local and Design Wide Clock Configuration

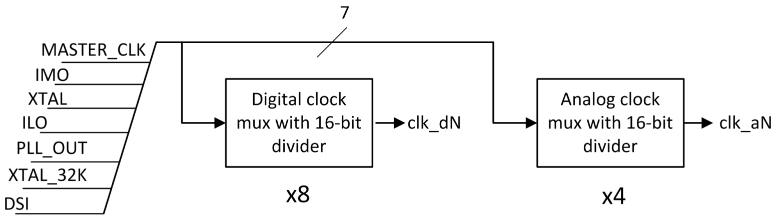


Table 10 lists the local clocks used in this design.

Table 10. Local Clocks

Name	Domain	Source	Desired Freq	Nominal Freq	Accuracy (%)	Start at Reset	Enabled
ADC_Ext_CP_-Clk	DIGITAL	MASTER_CLK	? MHz	24 MHz	±0.25	True	True
timer_clock	DIGITAL	BUS_CLK	? MHz	24 MHz	±0.25	True	True
DVDAC_BUS_-CLK	DIGITAL	BUS_CLK	? MHz	24 MHz	±0.25	True	True
ADC_theACLK	ANALOG	MASTER_CLK	960 kHz	960 kHz	±0.25	True	True
DVDAC_-IntClock	DIGITAL	MASTER_CLK	250 kHz	250 kHz	±0.25	True	True

For more information on clocking resources, please refer to:

- Clocking System chapter in the PSoC 5LP Technical Reference Manual
- Clocking chapter in the System Reference Guide
  - CyPLL API routines
  - CyIMO API routines

- CyILO API routines
- CyMaster API routines
- CyXTAL API routines

5 Interrupts and DMAs

5.1 Interrupts

This design contains the following interrupt components: (0 is the highest priority)

Table 11. Interrupts

Name	Intr Num	Vector	Priority
USB_ep_1	0	0	7
USB_ep_2	1	1	7
isr_ADC	2	2	2
isr_DAC	3	3	1
USB_dp_int	12	12	7
USB_arb_int	22	22	7
USB_bus_reset	23	23	7
USB_ep_0	24	24	7
ADC_IRQ	29	29	7

For more information on interrupts, please refer to:

- Interrupt Controller chapter in the PSoC 5LP Technical Reference Manual
- Interrupts chapter in the System Reference Guide
  - CyInt API routines and related registers
- Datasheet for cy_isr component

5.2 DMAs

This design contains the following DMA components: (0 is the highest priority)

Table 12. DMAs

Name	Priority	Channel Number
DVDAC_DMA	2	0

For more information on DMAs, please refer to:

- PHUB and DMAC chapter in the PSoC 5LP Technical Reference Manual
- DMA chapter in the System Reference Guide
  - DMA API routines and related registers
- Datasheet for cy_dma component

6 Flash Memory

PSoC 5LP devices offer a host of Flash protection options and device security features that you can leverage to meet the security and protection requirements of an application. These requirements range from protecting configuration settings or Flash data to locking the entire device from external access.

Table 13 lists the Flash protection settings for your design.

Table 13. Flash Protection Settings

Start Address	End Address	Protection Level
0x0	0x3FFFF	U - Unprotected

Flash memory is organized as rows with each row of flash having 256 bytes. Each flash row can be assigned one of four protection levels:

- U - Unprotected
- F - Factory Upgrade
- R - Field Upgrade
- W - Full Protection

For more information on Flash memory and protection, please refer to:

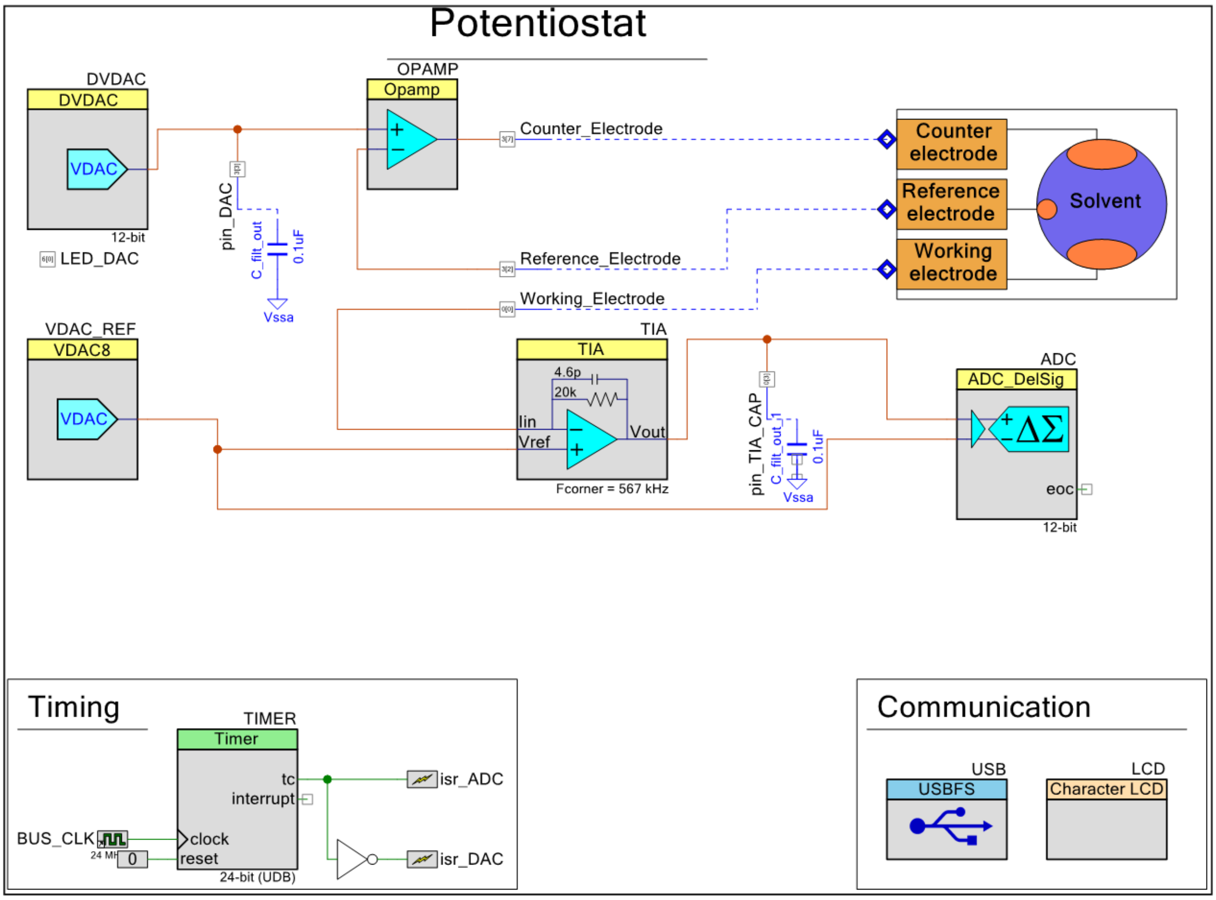
- Flash Protection chapter in the PSoC 5LP Technical Reference Manual
- Flash and EEPROM chapter in the System Reference Guide
    - CyWrite API routines
    - CyFlash API routines

7 Design Contents

This design's schematic content consists of the following schematic sheet:

7.1 Schematic Sheet: Potentiostat

Figure 5. Schematic Sheet: Potentiostat



This schematic sheet contains the following component instances:

- Instance ADC (type: ADC_DelSig_v3_30)
- Instance DVDAC (type: DVDAC_v2_10)
- Instance LCD (type: CharLCD_v2_20)
- Instance OPAMP (type: OpAmp_v1_90)
- Instance TIA (type: TIA_v2_0)
- Instance TIMER (type: Timer_v2_80)
- Instance USB (type: USBFS_v3_20)
- Instance VDAC_REF (type: VDAC8_v1_90)

8 Components

8.1 Component type: ADC_DelSig [v3.30]

8.1.1 Instance ADC

Description: Delta-Sigma ADC
Instance type: ADC_DelSig [v3.30]
Datasheet: online component datasheet for ADC_DelSig

Table 14. Component Parameters for ADC

Parameter Name	Value	Description
ADC_Alignment	Right	This parameter determines how the result is aligned in the 24 bit result word.
ADC_Alignment_Config2	Right	This parameter determines how the result is aligned in the 24 bit result word.
ADC_Alignment_Config3	Right	This parameter determines how the result is aligned in the 24 bit result word.
ADC_Alignment_Config4	Right	This parameter determines how the result is aligned in the 24 bit result word.
ADC_Charge_Pump_Clock	true	Low power charge pump clock selection
ADC_Clock	Internal	Parameter for selecting the ADC clock type.
ADC_Input_Mode	Differential	Differential or Single ended input mode
ADC_Input_Range	-Input +/- 2*Vref	Choose input operating mode that best supports the range of the signals being measured.
ADC_Input_Range_Config2	-Input +/- Vref	Choose input operating mode that best supports the range of the signals being measured.
ADC_Input_Range_Config3	-Input +/- Vref	Choose input operating mode that best supports the range of the signals being measured.
ADC_Input_Range_Config4	-Input +/- Vref	Choose input operating mode that best supports the range of the signals being measured.
ADC_Power	Medium Power	Sets power level of ADC.
ADC_Reference	Internal 1.024 Volts	Selects voltage reference source and configuration.
ADC_Reference_Config2	Internal 1.024 Volts	Selects voltage reference source and configuration.
ADC_Reference_Config3	Internal 1.024 Volts	Selects voltage reference source and configuration.
ADC_Reference_Config4	Internal 1.024 Volts	Selects voltage reference source and configuration.
ADC_Resolution	12	ADC Resolution in bits
ADC_Resolution_Config2	16	ADC Resolution in bits
ADC_Resolution_Config3	16	ADC Resolution in bits
ADC_Resolution_Config4	16	ADC Resolution in bits

Parameter Name	Value	Description
Clock_Frequency	64000	Determines the ADC clock frequency.
Comment_Config1	Cyclic voltammetry	Parameter which holds the user comment for the config1.
Comment_Config2	Second Config	Parameter which holds the user comment for the config2.
Comment_Config3	Third Config	Parameter which holds the user comment for the config3.
Comment_Config4	Fourth Config	Parameter which holds the user comment for the config4.
Config1_Name	CV	This parameter is used to create constants in the header file for config 1.
Config2_Name	CFG2	This parameter is used to create constants in the header file for config 2.
Config3_Name	CFG3	This parameter is used to create constants in the header file for config 3.
Config4_Name	CFG4	This parameter is used to create constants in the header file for config 4.
Configs	1	Number of active configurations
Conversion_Mode	2 - Continuous	ADC conversion mode
Conversion_Mode_Config2	2 - Continuous	ADC conversion mode
Conversion_Mode_Config3	2 - Continuous	ADC conversion mode
Conversion_Mode_Config4	2 - Continuous	ADC conversion mode
Enable_Vref_Vss	false	Determines whether or not to connect ADC's reference Vssa to AGL[6].
EnableModulatorInput	false	When this parameter is enabled, the modulator input terminal will be enabled on the symbol.
Input_Buffer_Gain	1	Gain of input amplifier
Input_Buffer_Gain_Config2	1	Gain of input amplifier
Input_Buffer_Gain_Config3	1	Gain of input amplifier
Input_Buffer_Gain_Config4	1	Gain of input amplifier
Input_Buffer_Mode	Level Shift	Buffer Mode type selection
Input_Buffer_Mode_Config2	Rail to Rail	Buffer Mode type selection
Input_Buffer_Mode_Config3	Rail to Rail	Buffer Mode type selection
Input_Buffer_Mode_Config4	Rail to Rail	Buffer Mode type selection
Ref_Voltage	1.024	Set reference voltage
Ref_Voltage_Config2	1.024	Set reference voltage
Ref_Voltage_Config3	1.024	Set reference voltage
Ref_Voltage_Config4	1.024	Set reference voltage
rm_int	false	Removes internal interrupt (IRQ)
Sample_Rate	30000	Sample Rate in Hz
Sample_Rate_Config2	10000	Sample Rate in Hz
Sample_Rate_Config3	10000	Sample Rate in Hz
Sample_Rate_Config4	10000	Sample Rate in Hz
Start_of_Conversion	Software	Continuous conversions or hardware controlled
User Comments		Instance-specific comments.

8.2 Component type: CharLCD [v2.20]

8.2.1 Instance LCD

Description: Character LCD Component
Instance type: CharLCD [v2.20]
Datasheet: [online component datasheet for CharLCD]

Table 15. Component Parameters for LCD

Parameter Name	Value	Description
ConversionRoutines	true	Defines if the conversion routines will be included in the project.
CustomCharacterSet	None	Defines the type of custom character set (User defined, Vertical or Horizontal bargraph). Based on the selection a look-up table with proper characters representation will be generated in the source code.
User Comments		Instance-specific comments.

8.3 Component type: DVDAC [v2.10]

8.3.1 Instance DVDAC

Description: 9 to 12 bit Dithered Voltage DAC
Instance type: DVDAC [v2.10]
Datasheet: [online component datasheet for DVDAC]

Table 16. Component Parameters for DVDAC

Parameter Name	Value	Description
DAC_Range	4 Volt	This parameter allows you to set one of the two voltage ranges. This option cannot be changed during runtime.
Initial_Value	2048	This parameter allows you to set the DVDAC voltage value. The maximum value will depend on the resolution selected. Refer to the DVDAC_SetValue() function description in this component datasheet.
InternalClock	true	This parameter allows you to configure the component's clock source: internal or external. This option cannot be changed during runtime.
InternalClockFreqHz	250000	When the clock source is configured to be internal, this parameter defines the frequency in Hz at which DMA is triggered. The parameter alsowrites the next value from the dithered array into the VDAC8 data register.

Parameter Name	Value	Description
Resolution	12 Bits	This parameter allows you to set the DVDAC resolution. The resolution cannot be changed during runtime.
User Comments		Instance-specific comments.

8.4 Component type: OpAmp [v1.90]

8.4.1 Instance OPAMP

Description: Opamp
Instance type: OpAmp [v1.90]
Datasheet: online component datasheet for OpAmp

Table 17. Component Parameters for OPAMP

Parameter Name	Value	Description
Mode	OpAmp	Selects between uncommitted op-amp or follower mode.
Power	Low Power	Selects the device power level.
User Comments		Instance-specific comments.

8.5 Component type: TIA [v2.0]

8.5.1 Instance TIA

Description: Trans-Impedance Amplifier
Instance type: TIA [v2.0]
Datasheet: online component datasheet for TIA

Table 18. Component Parameters for TIA

Parameter Name	Value	Description
Capacitive_Feedback	4.6 pF	Capacitive feedback for the TIA
Fcorner	567 kHz	Calculated -3dB frequency for the given feedback settings.
Power	Medium Power	Power setting for TIA
Resistive_Feedback	20k ohms	Nominal resistive feedback for the TIA
User Comments		Instance-specific comments.

8.6 Component type: Timer [v2.80]

8.6.1 Instance TIMER

Description: 8, 16, 24 or 32-bit Timer
Instance type: Timer [v2.80]
Datasheet: online component datasheet for Timer

Table 19. Component Parameters for TIMER

Parameter Name	Value	Description
CaptureAlternatingFall	false	Enables data capture on either edge but not until a valid falling edge is detected first.
CaptureAlternatingRise	false	Enables data capture on either edge but not until a valid rising edge is detected first.

Parameter Name	Value	Description
CaptureCount	2	The CaptureCount parameter works as a divider on the hardware input "capture". A CaptureCount value of 2 would result in an actual capture taking place every other time the input "capture" is changed.
CaptureCounterEnabled	false	Enables the capture counter to count capture events (up to 127) before a capture is triggered.
CaptureMode	None	This parameter defines the capture input signal requirements to trigger a valid capture event
EnableMode	Software Only	This parameter specifies the methods in enabling the component. Hardware mode makes the enable input pin visible. Software mode may reduce the resource usage if not enabled.
FixedFunction	false	Configures the component to use fixed function HW block instead of the UDB implementation.
InterruptOnCapture	false	Parameter to check whether interrupt on a capture event is enabled or disabled.
InterruptOnFIFOFull	false	Parameter to check whether interrupt on a FIFO Full event is enabled disabled.
InterruptOnTC	true	Parameter to check whether interrupt on a TC is enabled or disabled.
NumberOfCaptures	1	Number of captures allowed until the counter is cleared or disabled.
Period	16777215	Defines the timer period (This is also the reload value when terminal count is reached)
Resolution	24	Defines the resolution of the hardware. This parameter affects how many bits are used in the Period counter and defines the maximum resolution of the internal component signals.
RunMode	Continuous	Defines the hardware to run continuously, run until a terminal count is reached or run until an interrupt event is triggered.
TriggerMode	None	Defines the required trigger input signal to cause a valid trigger enable of the timer
User Comments		Instance-specific comments.

8.7 Component type: USBFS [v3.20]

8.7.1 Instance USB

Description: USB 2.0 Full Speed Device Framework
Instance type: USBFS [v3.20]
Datasheet: online component datasheet for USBFS

Table 20. Component Parameters for USB

Parameter Name	Value	Description
EnableBatteryChargDetect	false	This parameter allows to detect a charging supported USB host port using the API function USBFS_DetectPortType().
EnableCDCApi	true	Enables additional high level API's that allow the CDC device to be used similar to a UART device.
EnableMidiApi	true	Enables additional high level MIDI API's.
endpointMA	MA_Static	Endpoint memory allocation
endpointMM	EP_Manual	Endpoint memory management
epDMAautoOptimization	false	This parameter enables resource optimization for DMA with Automatic Memory Management mode. Set this parameter value to true only when a single IN endpoint is present in the device. Enabling this parameter in a multi IN endpoint device configuration causes undesired effects.
extern_cls	false	This parameter allows for user or other component to implement his own handler for Class requests. USBFS_-DispatchClassRqst() function should be implemented if this parameter enabled.
extern_vbus	true	This parameter enables external VBUSDET input.
extern_vnd	false	This parameter allows for user or other component to implement his own handler for Vendor specific requests. USBFS_HandleVendorRqst() function should be implemented if this parameter enabled.
extJackCount	0	Max number of External MIDI IN Jack or OUT Jack descriptors
Gen16bitEpAccessApi	false	This parameter defines whether to generate APIs for the 16-bits endpoint access.
HandleMscRequests	true	This parameter is used to enable handling MSC requests and generate MSC APIs.
isrGroupArbiter	High	This parameter defines the interrupt group of the Arbiter Interrupt.

Parameter Name	Value	Description
isrGroupBusReset	Low	This parameter defines the interrupt group of the Bus Reset Interrupt.
isrGroupEp0	Medium	This parameter defines the interrupt group of the Control Endpoint Interrupt (EP0).
isrGroupEp1	Medium	This parameter defines the interrupt group of the Data Endpoint 1 Interrupt.
isrGroupEp2	Medium	This parameter defines the interrupt group of the Data Endpoint 2 Interrupt.
isrGroupEp3	Medium	This parameter defines the interrupt group of the Data Endpoint 3 Interrupt.
isrGroupEp4	Medium	This parameter defines the interrupt group of the Data Endpoint 4 Interrupt.
isrGroupEp5	Medium	This parameter defines the interrupt group of the Data Endpoint 5 Interrupt.
isrGroupEp6	Medium	This parameter defines the interrupt group of the Data Endpoint 6 Interrupt.
isrGroupEp7	Medium	This parameter defines the interrupt group of the Data Endpoint 7 Interrupt.
isrGroupEp8	Medium	This parameter defines the interrupt group of the Data Endpoint 8 Interrupt.
isrGroupLpm	High	This parameter defines the interrupt group of the LPM Interrupt.
isrGroupSof	Low	This parameter defines the interrupt group of the Start of Frame Interrupt.
max_interfaces_num	1	Defines maximum interfaces number
Mode	false	Specifies whether the implementation will create API for interfacing to UART component(s) for a corresponding set of external MIDI connections.
mon_vbus	false	The mon_vbus parameter adds a single VBUS monitor pin to the design. This pin must be connected to VBUS and must be assigned in the pin editor.
MscDescriptors		Mass Storage Class Descriptors
MscLogicalUnitsNum	1	This parameter allows to specify the number of logical units that should be supported by the Mass Storage device.
out_sof	false	The out_sof parameter enables Start-of-Frame output.
Pid	F232	Product ID

Parameter Name	Value	Description
powerpad_vbus	false	This parameter enables VBUS power pad
ProdactName		This string is displayed by the Operating System when it is installing the mass storage device as the Product Name.
ProdactRevision		This string is displayed by the Operating System when it is installing the mass storage device as the Product Revision.
rm_lpm_int	true	Removes LPM ISR
User Comments		Instance-specific comments.
VendorName		This string is displayed by the Operating System when it is installing the mass storage device as the Vendor Name.
Vid	04B4	Vendor ID

8.8 Component type: VDAC8 [v1.90]

8.8.1 Instance VDAC_REF

Description: 8-Bit Voltage DAC
Instance type: VDAC8 [v1.90]
Datasheet: online component datasheet for VDAC8

Table 21. Component Parameters for VDAC_REF

Parameter Name	Value	Description
Data_Source	CPU or DMA (Data Bus)	Selects the method in which the data is written to the vDAC.
Initial_Value	127	Configures the initial vDAC output voltage. The output uses the following relation: Initial output voltage = value*(FullRange/255). This calculated output voltage value is invalid if DAC Bus is used.
Strobe_Mode	Register Write	Selects how the data is strobed into the DAC. For a register write, the data is strobed into the DAC on each CPU or DMA write. If operating in External mode, an external data strobe signal is required.
User Comments		Instance-specific comments.
VDAC_Range	0 - 4.080V (16mV/bit)	Specifies the full voltage scale range of the vDAC
VDAC_Speed	Low Speed	Specifies the vDAC settling speed. Note that the 'Slow Speed' selection consumes less power.
Voltage	2032	This parameter sets the voltage value.

9 Other Resources

The following documents contain important information on Cypress software APIs that might be relevant to this design:

- Standard Types and Defines chapter in the System Reference Guide
  - Software base types
  - Hardware register types
  - Compiler defines
  - Cypress API return codes
  - Interrupt types and macros
- Registers
  - The full PSoC 5LP register map is covered in the PSoC 5LP Registers Technical Reference Manual
  - Register Access chapter in the System Reference Guide
    - CY_GET API routines
    - CY_SET API routines
- System Functions chapter in the System Reference Guide
  - General API routines
  - CyDelay API routines
  - CyVd Voltage Detect API routines
- Power Management
  - Power Supply and Monitoring chapter in the PSoC 5LP Technical Reference Manual
  - Low Power Modes chapter in the PSoC 5LP Technical Reference Manual
  - Power Management chapter in the System Reference Guide
    - CyPm API routines
- Watchdog Timer chapter in the System Reference Guide
  - CyWdt API routines
- Cache Management
  - Cache Controller chapter in the PSoC 5LP Technical Reference Manual
  - Cache chapter in the System Reference Guide
    - CyFlushCache() API routine