VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Bioimpedance Hydration Measurement Device

Gareth Clay

Supervisor: Dr Ciaran Moore, Dr Sapi Mukerji

October 16, 2016

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours in Computer
Systems and Electronics.

Abstract

The hydration level of a patient is extremely important to the treatment options that a medical doctor prescribes, especially upon admission in an emergency room setting. Low levels of hydration in patients who are already weak due to sickness or injury can lead to complications that compromise patient health. Several clinical techniques exist to measure the hydration levels of a patient, however they tend to be costly, invasive and in general require several hours of lead time. Bioimpedance analysis has proven useful in clinical studies to measure body composition in health-care assessment systems, however existing devices are costly and complicated. This project sets out to design an inexpensive bioimpedance measurement device specific to measuring patient hydration. The developed device was found to be precise to within 1%, with less than 1% and 5% error in passive impedance and phase measurements. Bioimpedance measurements performed with the device were found to be repeatable, and impedance shifts with hydration level were in line with theory. The device measured the expected change in extracellular water upon intravenous administering of fluids. Intracellular water measurements were found to be less reliable, with significant variation between measurements and deviation from the expected result.

Contents

Figures

Chapter 1

Introduction

1.1 Motivation

The hydration levels of a patient are extremely important to the treatment options that a medical doctor prescribes. Low levels of hydration in patients who are already weak due to sickness or injury can lead to complications, including the loss of thermoregulation [1]. Treatment for dehydration needs to be prescribed quickly on admission, these treatments are administered by doctors generally within the emergency department (ED) of a hospital. Of course, without prior testing of the patients hydration level, incorrect treatment options can also be prescribed causing overhydration, also known as *water intoxication*.

Blood pressure is one metric used to determine a patients hydration level, however elevations indicating overhydration may not be noticed even when the degree of water intoxication is severe enough to be threatening to the patients health. ED doctors also make use of blood testing and urinalysis, the measure of specific gravity of urine [2], in order to determine dehydration and overhydration. Both procedures require lab testing, making them slow and impractical for use in the diagnosis of an emergency room admission. Other techniques exist to measure hydration, such as using isotopes injected into the patient, known as radio-isotopic dilution. Typical processes to measure hydration use bromide [3] or a radioactive potassium isotope K^{40}, included in body potassium [4]. The dilution of these isotopes when passed through the body gives an indication of the subjects current level of hydration. However, these procedures are both invasive as they require blood samples, expensive due to the requirement for dosage via mass spectrometry and the time needed for the isotope to pass through the patients system means they cannot be performed at short intervals.

With the clear disadvantages to the current procedures employed in a typical emergency room, a non-invasive test is desirable, one which could be performed on a patient, prior to and during treatment. This would allow doctors to monitor the effects of the treatment and make changes to the prescribed treatment option should this test prove it necessary. Bioimpedance analysis is such a technique, which has proven useful in clinical studies to measure body composition in health care assessment systems [5]. There exist techniques for using this analysis to measure the hydration of a patient, which would make procedure both non-invasive and timely. The only necessary equipment is a simple set of skin-electrodes. The use of bioimpedance analysis to measure hydration would therefore eliminate the need for lab processing time present with other hydration measurement techniques. Such a device would be a beneficial addition to hospital emergency departments.

1.2 Theoretical Background

1.2.1 Bioimpedance and Body Composition

The electrical response of a biomaterial (e.g., the total body, skin, muscle, fat, or blood), either dead or living, to an applied current is referred to as its bioimpedance [6]. More specifically, like the impedance of a material, bioimpedance is the bodys ability to oppose current flow. As multiple systems within the body contribute to the overall bioimpedance, changes in this bioimpedance can indicate changes in the state of the biomaterial of the body.

A bioimpedance measurement is complex, as it consists of both a real and imaginary component. The real component is the resistivity of the body to the current flow, which is typically of an alternating current (AC) signal. The imaginary component is due to a phase shift of voltage relative to current, which is predominately caused by capacitances in the body at the cell membranes. The conduction path through the body has a significant influence on both these factors of the bioimpedance measurement. Broadly, the conducting sections of the body can be separated into the extracellular water (ECW) and intracellular water (ICW). As the names suggest, ECW is the conductive water external to the tissue cells, whereas ICW is the internal cell water.

The measurement of ICW, ECW, and their sum, total body water (TBW) are useful in many pathologies. TBW is strongly related to the fat-free-mass, which is considered the most conducting portion of the body, containing 73.2% of a healthy individuals body water [7]. In a similar manner, body cell mass (BCM) is closely related to ICW. The challenge is in measuring the correct factors and determining the different compartments of the body through models. There are a number of models, with more research still being performed, which describe the composition of the body. Such models are useful, as they can generally be represented electrically, allowing simulation of the body to be performed.

1.2.2 Electrical Properties of Tissue and Modelling

The ionic composition of ECW can be measured directly, as it consists of plasma interstitial fluid. This composition testing has found ECW to have a resistivity similar to that of Saline, around 40Ωcm [8]. The ionic composition of ICW however, cannot be directly measured as it is dependent on the type of cell, meaning resistivity cannot be as easily measured. Instead, a measure of TBW is required in order to determine ICW.

Given that ICW and ECW contain ions, they are conducting and therefore are one of the bodys contributing factors to bioimpedance. In order to measure their volume, a measurement of complex impedance is required over resistance due to the capacitive nature of cell membrane at low and intermediate frequencies. Tissue is composed of cells, and these cells have a cell membrane. The membrane, due to its polarity, acts as a dielectric between two conductors and prevents conduction through the intracellular fluid. Of course at higher frequencies, this capacitance has negligible effect as the cell membrane barrier begins to conduct. This conduction is represented in Figure 1.1, outlining the difference in current path at low and high frequencies. From this behaviour, a simple electrical model can be determined as shown in Figure 1.2. The cell membrane is modelled by capacitor C_m, the ECW and ICW resistivities are then modelled by resistors R_e and R_i respectively.

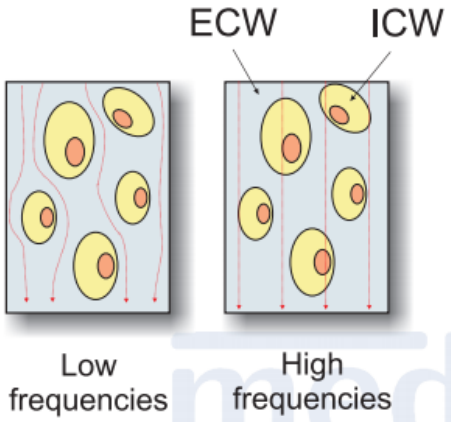


Figure 1.1: Current flow within tissue and the electrical equivalent circuit [9].

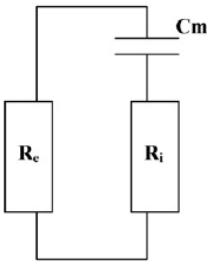


Figure 1.2: Simple equivalent model of tissues with parallel resistances [9].

The consequence of the capacitive behaviour is that ECW resistance must be measured at low frequency (<1kHz), while a measure of both ICW and ECW will be achieved at high frequency (>5MHz). These frequencies cannot be readily achieved in practice. Instead, the data must be extrapolated to zero and infinite frequencies in order to calculate R_0 (ECW resistance) and R_∞ (TBW resistance) [8]. This extrapolation is enabled by the observation that the impedance data on the resistance-reactance plane forms a semi-circle, with intercepts located on the horizontal axis. This observation is according to the Cole-Cole impedance model, which is an elegant and popular method for characterising the electrochemical properties of biological tissues [10]. A typical Cole-Cole model for the measurement of ECW and TBW resistance can be seen in Figure 1.3. The intercept points of the model, which can be extrapolated from the impedance data forming the model, are R_0 (ECW resistance) and R_∞ (TBW resistance). Determination of these two resistances is the first step in typical models for the calculation of ICW and ECW. The resistance at 50kHz, identified as R_{50} in the figure, is also of particular interest as this is considered the point where both the extracellular and intracellular spaces conduct [8]. Above this point, the majority of the signal flows through the intracellular space, while below this point the majority of the conduction is through the extracellular space. For this reason, 50kHz is often used in single frequency bioimpedance measurements.

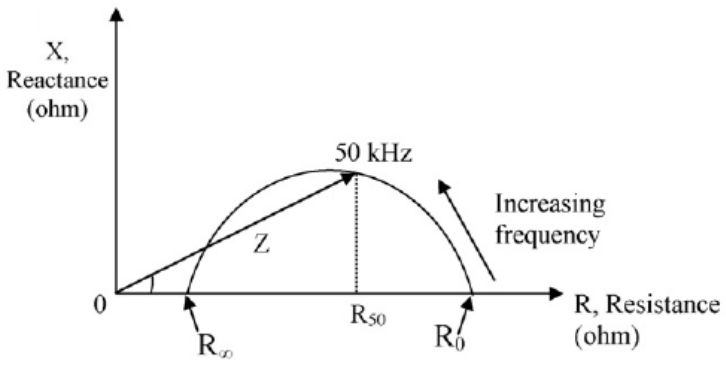


Figure 1.3: The Cole-Cole impedance model. Resistances extrapolated at zero (R_0) and infinite (R_∞) frequencies [8].

1.2.3 Intracellular and Extracellular Water Calculations

Analytically, it is necessary to calculate R_0 and R_∞ in order to calculate ECW and ICW. Experimentally, these values cannot be measured due to limitations on the maximum frequency and the lack of conduction through skin at DC potential. Instead, these points can found by the means of least-square regression, which aims to determine a set of parameters in order to find the line of best fit of a function. The masters thesis [11] explores several methods for this parameter fitting, and concludes a Cole-Cole fit of the resistance against the natural frequency provides the nearest approximation of the actual parameters. This function is detailed in Equation 1.1. The function has four free parameters R_∞, R_0, τ and α, with natural frequency ω being the independent variable.

$$R(\omega) = R_\infty + \frac{(R_0 - R_\infty)\left(1 + (\omega\tau)^\alpha \cos(\alpha\frac{\pi}{2})\right)}{1 + \left(2(\omega\tau)^\alpha \cos(\alpha\frac{\pi}{2}) + (\omega\tau)^{2\alpha}\right)} \tag{1.1}$$

To visibly verify the fitting accuracy, the determined parameters can be used to plot the semi-circular Cole-Cole plot. The fitting accuracy of this plot to the experimental data gives an indication of the accuracy and precision of the Cole-Cole parameters. The real centre ($\Re\{C\}$), imaginary centre ($\Im\{C\}$) and radius (r) of this plot can be determined using Equations 1.2, 1.3 and 1.4 respectively.

$$\Re\{C\} = \frac{R_0 + R_\infty}{2} \tag{1.2}$$

$$\Im\{C\} = \frac{R_0 - R_\infty}{2} \frac{\cos\left(\frac{\alpha\pi}{2}\right)}{\sin\left(\frac{\alpha\pi}{2}\right)} \tag{1.3}$$

$$r = \frac{\frac{R_0 - R_\infty}{2}}{2\sin\left(\frac{\alpha\pi}{2}\right)} \tag{1.4}$$

With R_0 and R_∞ attained from Equation 1.1, a variation of the Hanai equations proposed by [12] can be used to calculate ECW and ICW:

$$\text{ECW} = k_{ECW}\left(\frac{H^2\sqrt{W}}{R_0}\right)^{2/3} \tag{1.5}$$

$$\text{ICW} = k_{ICW}\left(\frac{H^2\sqrt{W}}{R_I}\right)^{2/3} \tag{1.6}$$

where R_I is the intracellular resistance from Equation 1.9 [8] and k_{ECW} and k_{ICW}:

$$k_{ECW} = \frac{a}{BMI} + b \tag{1.7}$$

$$k_{ICW} = \frac{c}{BMI} + d \tag{1.8}$$

$$R_I^{-1} = R_\infty^{-1} - R_0^{-1} \tag{1.9}$$

These functions have been found empirically by regressing BMI against the true values of k_{ECW} and k_{ICW}. The parameters of which are found to be $a = 0.188$, $b = 0.2883$, $c = 5.8758$ and $d = 0.4194$ via cross validation also described in [8].

1.2.4 Commercial Bioimpedance Analysers

There are a number of commercially available bioimpedance analyser devices. They are typically separated into several select categories: bioimpedance spectroscopy, multiple-frequency, segmental multiple frequency, single-frequency and segmental single-frequency bioimpedance analysis. Of this list, only bioimpedance spectroscopy devices are suitable for hydration measurements due to the other devices having a limited number of measurement frequencies (less than 5). This is not suitable for performing the data extrapolation necessary for the Cole-Cole model, as was described in section 1.2.3. A study into BIS devices found the price in excess of $10,000 for the devices with available pricing [13]. A device specific to body hydration measurement that costs significantly less to manufacture would increase the availability of such devices to physicians desiring to measure patient hydration trend data.

The same study also found that devices often were a "black box", in that the equations used to calculate the hydration parameters were unknown. In some cases, the raw bioimpedance data could also not be retrieved, preventing recalculations from being performed. By providing a means to alter these equations in software and a data log of raw data, clinicians will be able to perform their own data analysis should an appropriate method for their application be identified.

1.3 Goals & Design Specifications

The goal of this project is therefore to create an electrical device that uses bioimpedance analysis to measure the hydration level of a patient. To this end, the main goals of the project can be separated into stages as follows:

- Design and build a device capable of measuring bioimpedance.
- Create an interface for reporting the data visually and storing trend data over time.
- Devise and perform test procedures on passive resistor-capacitor networks to verify accuracy and precision.
- Create a suitable enclosure and supporting circuitry to allow external testing of the device.
- Perform testing in a controlled setting, where fluids in measured dosages are administered to the subject being tested.
- From testing, measure a trend in resistance and reactance, and therefore ICW and ECW, appropriate to the level of fluids administered to the patient.

Specifically, a device and supporting software will be developed to the following specifications:

- A minimum 100kHz bandwidth inclusive of the 50kHz centre frequency.
- Impedance and phase measurement accuracy to within 1% and 5% respectively.
- Measure with 1% precision the bioimpedance parameters R_∞ and R_0 across a minimum of 20 tests.
- Provide a suitable interface to communicate with a PC to perform data analysis and logging.
- Hydration analysis software to provide ICW & ECW readings at the time of testing.
- Portability, with an enclosure size and weight that allows easy transportation in a bag or by hand, with battery and data storage for operation without a PC.

Chapter 2

Bioimpedance Measurement Device: Hardware Design

Three revisions of the bioimpedance measurement device have been developed. The final revision of the device fits into a small portable box, which interfaces with a PC via USB. The bioimpedance analysis is performed using a tetra-polar electrode arrangement, requiring four leads and electrodes. This chapter will discuss each of the three revisions of the device detailing the electronic design, the method for calibration and impedance calculation and the resulting accuracy of the device on passive impedances.

2.1 General System

All revisions of the hydration monitor device feature an ADuC702X series microcontroller. Revision 1 utilises the ADuC7020 whereas revisions 2 and 3 utilise the ADuC7024 for the additional general-purpose input/output (GPIO). This microcontroller allows communication via a universal asynchronous receiver/transmitter (UART). The microcontroller operates at V_{DD} = 3.3V, and as such, a switching regulator has been included to step down the input supply from a battery or USB. In order to control the device with readily available feedback, an LCD port has been added and the supporting software written to support HD44780 driven devices. A break-out of the GPIO pins of the microcontroller has also been included. Currently the devices use two such pins configured to provide an input via push buttons. Using these buttons, the user can navigate the menu to configure the device.

2.2 Hydration Monitor Revision 1

2.2.1 Electronic Design

The bioimpedance measurement circuitry is based around the AD593X series of chips by Analog Devices, the functional block diagram can be seen in Figure 2.1. The two chips, the AD5934 and the AD5933, are 12-bit impedance spectral analyser devices which use a 1024-bit Discrete Fourier Transform (DFT) engine to attain the real and imaginary spectral component of an unknown impedance. An excitation signal of known frequency and amplitude is passed through the unknown impedance, the current is then measured via the

receiver stage current-to-voltage converter. A DFT of the returning signal allows the signal phase, and hence impedance, to be determined. The data sheet specifies an accruacy of 0.5% for measurements between 1kΩ and 10MΩ.

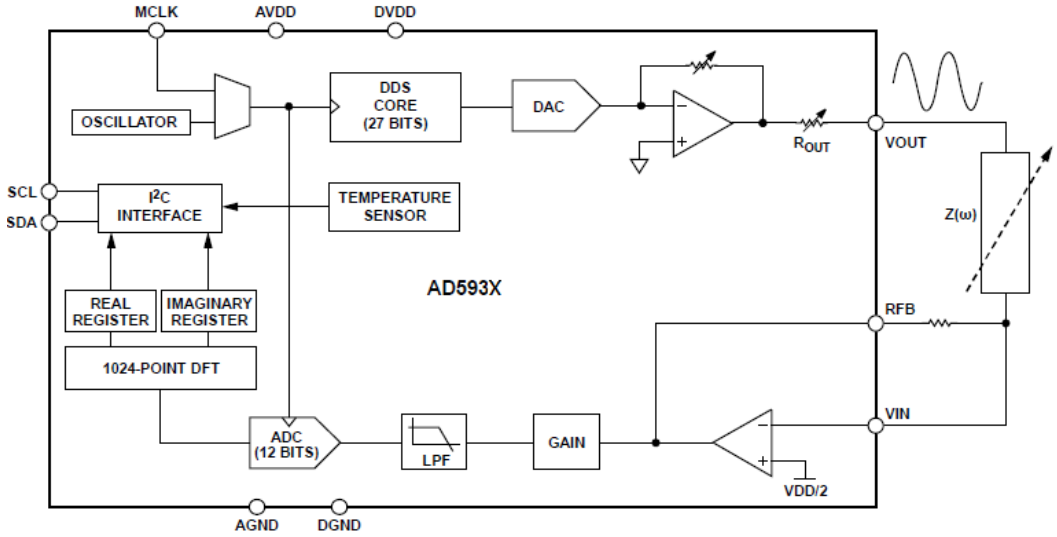


Figure 2.1: AD593X series functional block diagram[14].

The chips are programmable via I_2C, allowing the selection of output gain, amplitude and frequency. The devices can also be programmed to perform a frequency sweep where a start frequency, the number of increments and the frequency increment are specified, and then the chip performs impedance measurements at each point along the sweep. For the first revision of the bioimpedance measurement device, the AD5934 was chosen due to the lower cost, with the trade-off of a 250kSPS sample rate compared with a 1MSPS sample rate of the AD5933. This lower sample rate is likely to cause some small errors in the returned impedance measurement, these errors will increase with the output excitation frequency.

As previously mentioned, the AD593X series chip has a system accuracy of 0.5% at impedances exceeding 1kΩ. This minimum impedance is likely to be the upper limit for a bioimpedance measurement which typically fall within a 300Ω to 800Ω range. This renders the IC unsuitable for the task of measuring bioimpedance without additional circuitry. The main issue with the stand-alone IC is that there exists a significant output resistance influenced by the output excitation amplitude. Typically, this value can exceed 2.4kΩ. Due to the low ohm range of a typical bioimpedance measurement, this output resistance is significant and can affect the unknown impedance measurement. A simple voltage follower, with high input impedance and low output impedance will prevent the influence of this AD5934 output impedance. The AD8606 operational amplifier (op-amp) is a suitable choice, as it has a low output impedance of approximately 1Ω and a unity gain bandwidth of 10MHz, far exceeding the 100kHz bandwidth of the AD5934. The output excitation signal also has a DC offset, due to the difference in biasing of the transmit and receive stages. This DC offset can cause polarisation across the unknown impedance, leading to errors in the measurement. A simple first-order high pass filter in the low frequency range can be used to remove the DC offset. It is then necessary to re-bias the signal, as the receive stage expects a signal offset by $V_{DD}/2$.

Minor inaccuracies are also caused by the current-to-voltage conversion of the receiver

stage. This is due to the internal amplifiers offset bias sensitivity, offset voltage and low common-mode rejection ratio (CMRR). By instead performing this conversion externally, a higher performance device can be used. A second AD8606 op-amp has been selected as it provides excellent CMRR, low offset voltage and has a high tolerance to bias current. The designed analog front end can be seen in Figure 2.2, where signals *MCLK*, *I2C_SCL0* and *I2C_SDA0* are provided by the microcontroller.

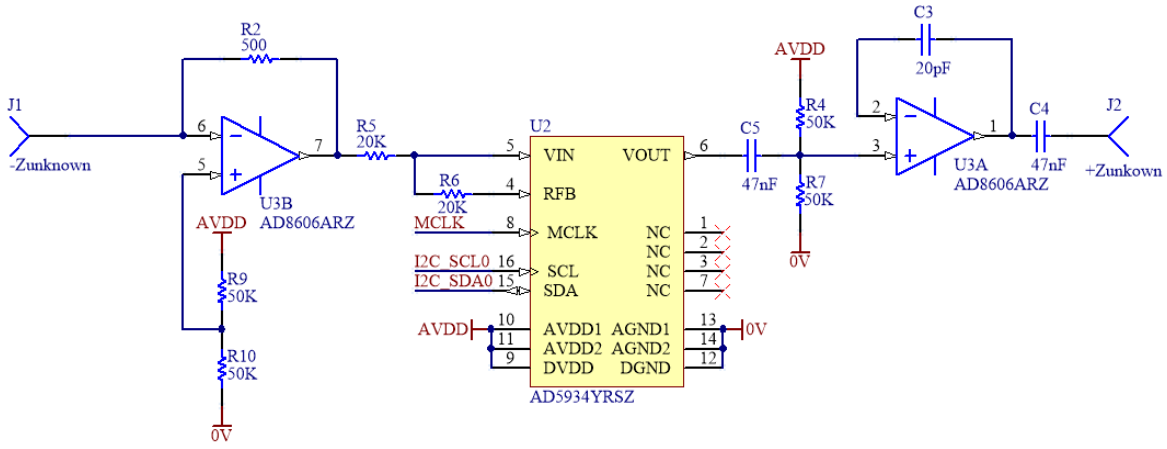


Figure 2.2: The optimised analog front end for the AD5934.

2.2.2 Impedance Calculation and Calibration Procedure

To ensure the correct operation of the proposed analog front end, care must be taken in the calibration of the circuit. The AD5934 makes use of what is known as a *gain factor*, which compensates for gain irregularities throughout the system. The gain factor is typically calculated using a known calibration impedance through Equation 2.1, where *magnitude* is the measured value through the calibration impedance and Z is the actual calibration impedance. The magnitude of the impedance is calculated from the real (R) and imaginary (I) registers of the AD5934, using Equation 2.2.

$$gainfactor = \left(\frac{\frac{1}{Z}}{magnitude} \right) \tag{2.1}$$

$$magnitude = \sqrt{R^2 + I^2} \tag{2.2}$$

To perform calibration without needing to manually swap the system load, a AD849 analog multiplexer has been used. This device can switch the load between the unknown impedance and the calibration impedance using a control signal from the system microcontroller. As the system gain is frequency dependant, by calculating this factor at each frequency on the frequency sweep, the frequency dependent gain can be compensated for if a purely resistive calibration impedance is used. The unknown impedance value can then be calculated using Equation 2.3. The phase shift through the unknown impedance can be found using the data retrieved from the real and imaginary registers of the AD5934, applied to Equation 2.4.

$$Z_{\text{unknown}} = \frac{1}{gain\ factor\ \times\ magnitude} \tag{2.3}$$

$$\theta_{unknown} = tan^{-1}(I/R) \tag{2.4}$$

Of course, the circuitry of the system also introduces a phase shift separate from the phase shift through the unknown impedance. First, the system phase can be measured using the calibration circuit, given that there is zero phase shift across a purely resistive load. The phase can then be measured again through the unknown impedance, subtracting the system phase to find the phase shift solely through the unknown impedance.

The final remaining step in the processing of the bioimpedance data is a calculation of resistance and reactance. These two measures, as mentioned in Section 1.2.2, are required to calculate the ICW and ECW of the patient under test. The two equations for resistance and reactance are detailed in Equation 2.5 and Equation 2.6 where θ is the measured phase, and Z the measured impedance magnitude.

$$|R| = |Z| \times cos(\theta) \tag{2.5}$$

$$|X| = |Z| \times sin(\theta) \tag{2.6}$$

2.2.3 Test Impedance Measurements

The testing has been performed on a typical body bioimpedance range, approximately 300Ω to 800Ω. The measured values have been confined to a 5-100kHz bandwidth, the specified bandwidth of the AD5934. Measurements below this bandwidth tend to be inaccurate, due to the clock divisions necessary to achieve low frequencies from a 16MHz master clock. The device may well perform above the maximum bandwith of 100kHz, due to the gain factor being calculated at each frequency interval, which could account for the signal attenuation present outside of the bandwidth. In the interest of testing the core functionality of the device, the testing for this revision has been performed within the manufacturer specified bandwidth.

The results of this impedance testing on two circuits can be seen in Figure 2.3. It can be observed that the measured data largely conforms to the theoretical values, within ±5.5Ω and ±1.7° (±2.4% and ±2.8% respectively) when measuring circuit 2 impedance and phase. However it is observable in the circuit 1 measurements that a ±27.6Ω error in impedance occurs for frequencies above 20kHz. There also exists a 10.94° phase error peak at 95kHz. This indicates a decrease in accuracy between a more capacitive load such as circuit 2, and a more resistive load as is the case with circuit 1. Lead impedances and other parasitics not present within the calibration circuit used to calculate the device gain factor will also contribute to the measurement error. A parasitic component at resonance can explain the phase peak observed in the circuit 1 data.

As has been mentioned in Section 1.2.2, a simplistic model of the impedance of a human known as a phantom can be represented using a parallel capacitor and resistor in series with a resistor. From measuring the impedance across this load, and then calculating the

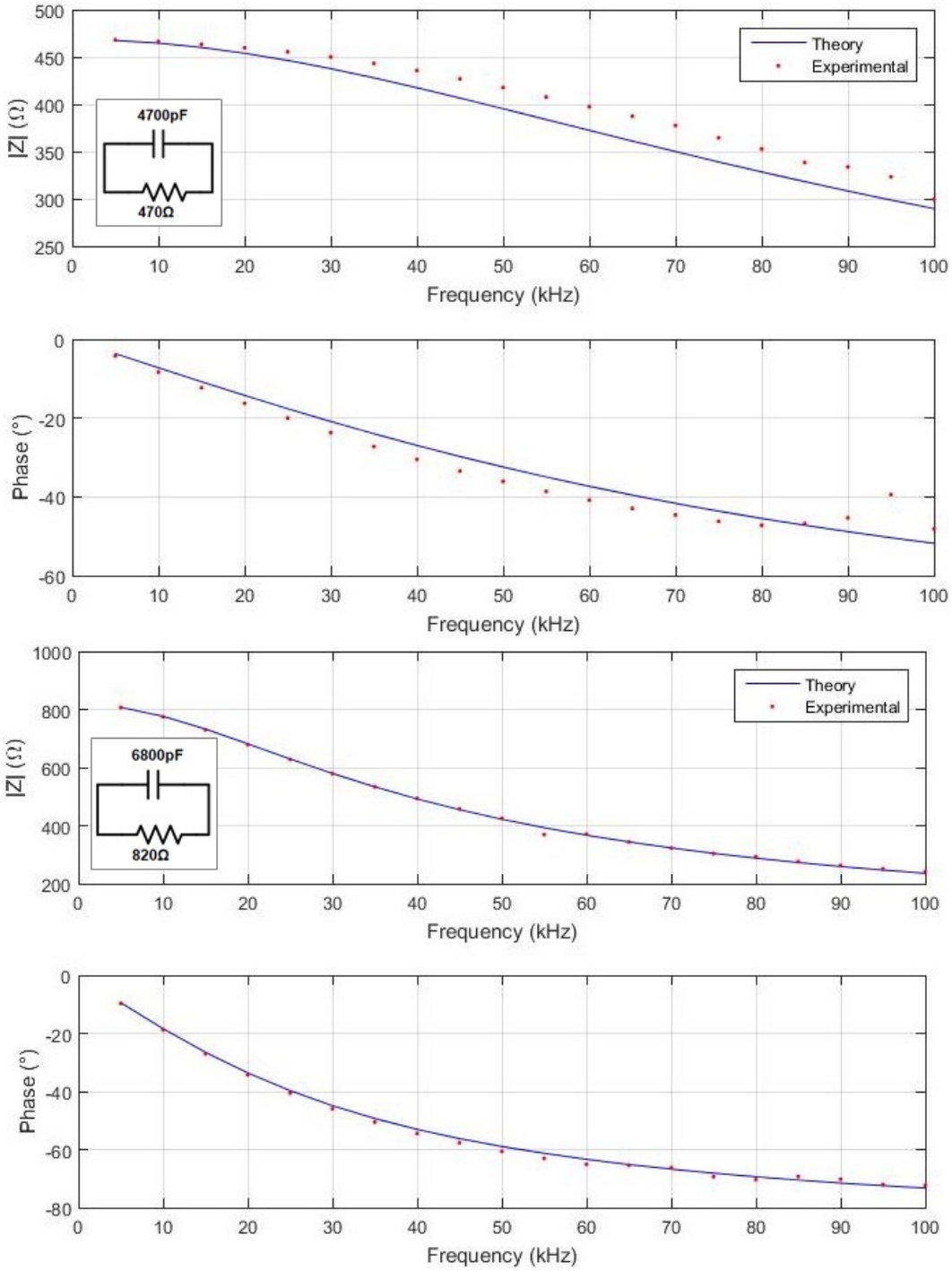


Figure 2.3: Calculated and measured impedance (top) and phase spectrum (bottom) vs frequency for two test circuits (inset) using the revision 1 device.

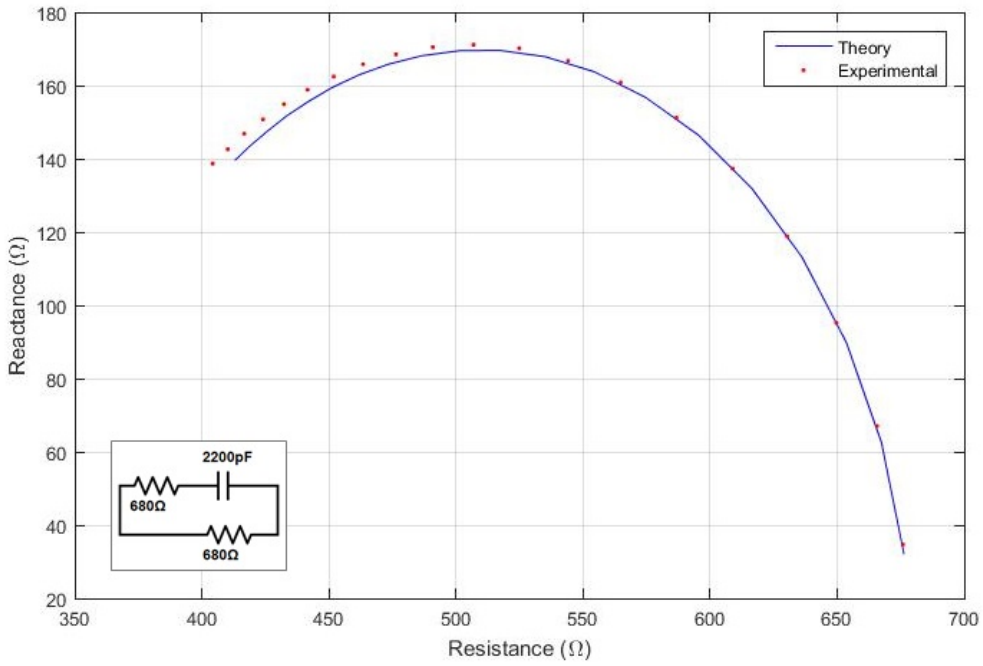


Figure 2.4: Calculated and measured Cole-Cole plot for a bioimpedance phantom circuit (inset) using the revision 1 device.

reactance and resistance, a Cole-Cole plot can be produced with behaviour similar to a human body bioimpedance measurement. Figure 2.4 details a Cole-Cole plot, where it can be seen that the theoretical and measured values conform to within $\pm 2.1\%$ and $\pm 0.9\%$ error for resistance and reactance respectively. There exists a small offset of resistance across the bandwidth, the actual value being larger than the measured value. This is possibly caused by small parasitics present in the calibration circuitry, which are not present when measuring the external impedance.

The accuracy of the measured data on an impedance 'phantom' indicates the device has a suitable level of accuracy to attempt bioimpedance measurements. However, when a full-body test was performed with an electrode attached to the left foot and right hand, the measured data did not take a form one would expect. Instead of a plot similar in shape to Figure 2.4, an extremely high resistance and reactance was measured across the frequency sweep. The likely cause for the inaccuracies of the full body measurement is the electrode-skin interface, which introduces a high impedance into the measurement due its capacitive nature, along with the high impedance of skin. The revision 1 device takes a bipolar measurement using two electrodes, injecting a known excitation voltage at a known frequency and measuring the current flowing through the unknown impedance. With only two electrodes and a measurement dictated by the returning current, there is no compensation for this high impedance interface which significantly reduces the current flowing through the unknown impedance body. Instead, this electrode impedance 'masks' the unknown impedance due to it being significantly larger than the bioimpedance of the body. This issue renders the design unsuitable for bioimpedance measurements. However, this design has proven that accurate impedance measurements in the impedance range of a human body can be measured using the AD593X series chips. A new approach must be taken to measuring bioimpedance, one which can compensate for the electrode-skin interface inherent to human bioimpedance measurements.

2.3 Hydration Monitor Revision 2

2.3.1 Electronic Design

The first revision of the bioimpedance measurement device is limited by its bipolar measurements, relying solely on the current flowing through the load to measure impedance via a known frequency and amplitude excitation signal. The circuitry did remove several issues present with the stand-alone AD5934 chip, namely the present DC bias and the influence of the output resistance on the unknown impedance. However, the bipolar measurements of the system severely limit the performance of the device when presented with electrode impedances, as has been outlined in Section 2.2. The second revision to the analog front end will therefore attempt to resolve this limitation by making use of a tetra-polar measurement system.

In a tetra-polar approach to measuring bioimpedance, two pairs of electrodes known as polarising and measuring electrodes are used. One electrode injects the measurement excitation signal, while a second electrode measures the return current through the unknown impedance. The second pair of electrodes then measures the potential difference across this impedance, the voltage across the unknown impedance due to the excitation signal. The system now has a measure of the current flowing through the unknown impedance, and a measure of the potential difference created by this current. From Ohm's law, the unknown impedance can be easily calculated. Of course, as very little current flows through the electrode interface, the differential voltage drop is almost entirely due to the body impedance. As a result, the measured impedance is now uninfluenced by the electrode interfaces.

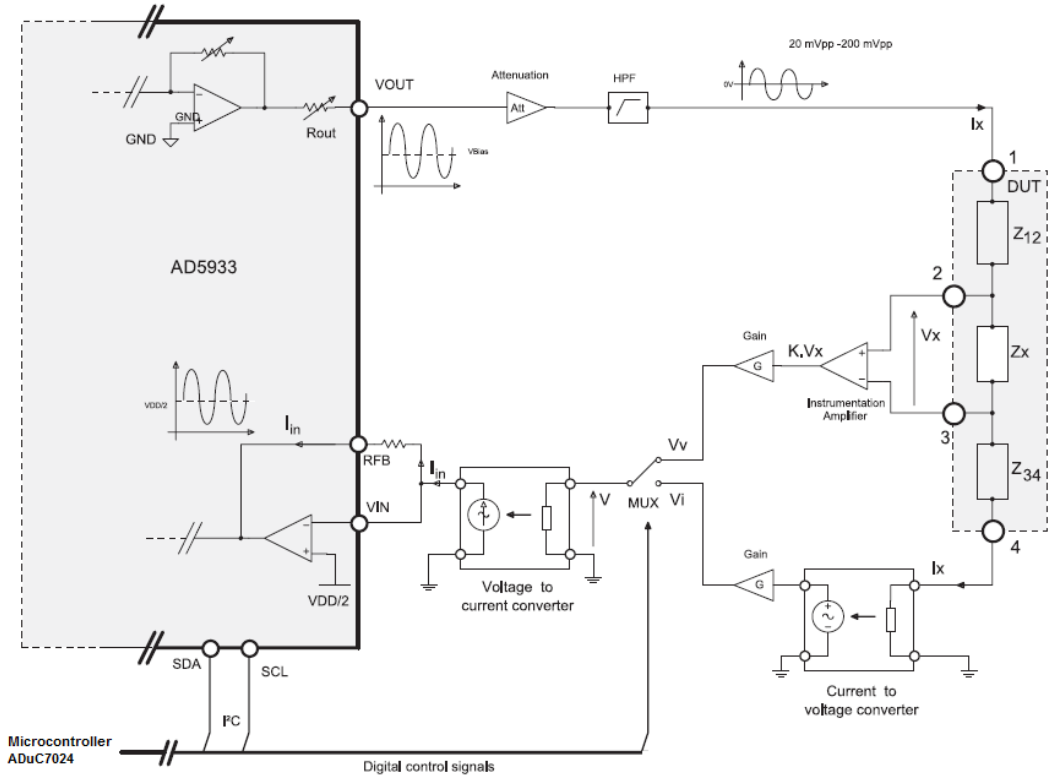


Figure 2.5: Version 2 of the optimised analog front end for the AD5933 [15].

The AD5933 impedance spectral analyser chip has been chosen over the AD5934 for the second revision device. This device has been chosen due to the higher 1MSPS sampling rate over the 250kSPS rate of the AD5934, which will help to improve the accuracy of the system at high excitation frequencies. The two chips remain identical in functionality, the AD5933 is still communicated with over I_2C using the same set of control functions. A circuit for the conversion of the AD5933 to a four electrode measurement system has been proposed in [15]. Although this paper focuses on the measurement of microscopic bio-matter, the circuit utilises tetra-polar measurements and should prove to be readily adaptable to body bioimpedance measurements when integrated with the current aspects of the bioimpedance measurement device. A proposed block diagram for the system can be seen in Figure 2.5. The specific circuitry for each section will be discussed briefly hereafter, along with the additional features to be added to the second revision of the device.

Attenuation and Biasing

The initial stage of the analog front end retains the same characteristics as the first version of the design, with an additional attenuation stage. The excitation signal is first attenuated by a factor of 10 using an inverting amplifier configuration. This attenuation lowers the excitation signal voltage, in turn lowering the current flowing through the unknown impedance. The signal is then high-pass filtered to prevent a DC potential across the unknown impedance, and finally buffered using a voltage follower to prevent the influence of the AD5933 output resistance on the measured unknown impedance.

Voltage Measurement

To achieve a measurement of the potential difference between measuring electrodes, a differential measurement approach with high common-mode rejection rate (CMRR) is desired. The AD8250 instrumentational amplifier by Analog Devices serves this purpose, providing 70dB and above CMRR for frequencies under 150KHz, while the gain of the device is flat across this frequency range at the maximum programmable gain. The device has high-impedance inputs, which prevent significant current from flowing through the measurement electrodes, altering the voltage being measured. The final desirable design feature of this device is the ability to digitally control discrete output gain levels of 1, 2, 4 and 10. This allows the gain to be controlled directly by the microcontroller, making gain adjustment during prototyping much more accessible.

Current to Voltage Conversion

It is necessary to measure the current signal flowing through the unknown impedance. This measurement can be achieved using a transimpedance amplifier, a variant of a non-inverting amplifier which can be seen in Figure 2.6. The high gain of the op-amp causes the input current to flow through the feedback resistor R_x, converting the current to a voltage proportional to R_x. A parallel feedback capacitor C_x is also included to provide stability in the feedback loop. Due to the initial attenuation of the excitation signal and the high-impedance electrode interfaces, the current signal through the unknown impedance is small. It is therefore necessary for the gain of the transimpedance amplifier to be large, a value of 10kΩ for R_x has been chosen.

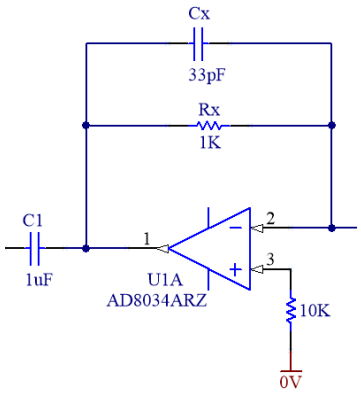


Figure 2.6: Current measurement transimpedance amplifier for the revision 2 device [15].

Amplification and Channel Selection

At this stage, the amplitudes of the voltage and current signals present when performing a bioimpedance measurement are unknown. As the returning signal amplitudes rely on the measured impedance, the placement of the electrodes and the electrode type have a significant influence on the signal amplitude. To ensure there is sufficient amplification of the measurement signals, it is desirable that there be some level of adjustable gain, to ensure the signal is large enough to prevent the introduction of noise, while avoiding saturation of the input transistors of the AD5933.

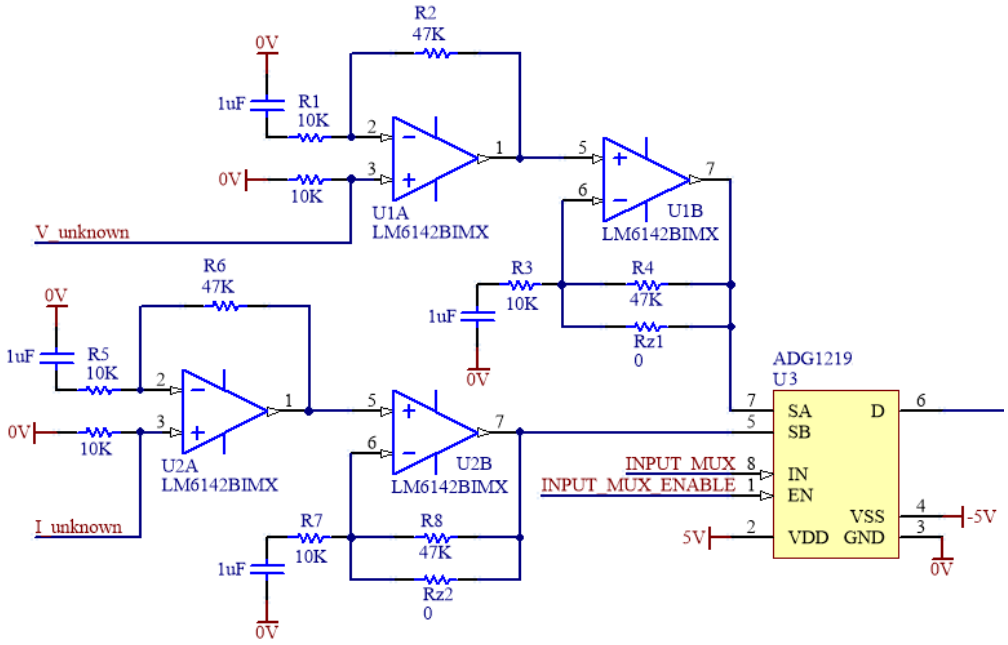


Figure 2.7: Voltage and current signal amplification circuitry, followed by signal selection multiplexer.

The signals are first high-pass filtered to remove any remaining DC offset from the previous gain sections. Both the measured voltage and current signals are then fed through two non-inverting cascaded amplifier arrangements, detailed in Figure 2.7. The first set of amplifiers provide an approximate 5.7 times gain factor, which can be altered by adjusting the feedback

resistance. This resistance has to be altered manually, by changing the resistor. The second set of non-inverting amplifiers can also provide 5.7 times gain for a total of 32 times gain, or fitted with zero ohm resistors for unity gain. A limiting factor of this arrangement is the signal cannot be attenuated, should it prove necessary.

To feed both the voltage and current signal back into the AD5933, the two signals are passed through an ADG1219 analog multiplexer. This multiplexer can be controlled digitally by the system microcontroller using control signals *INPUT_MUX* and *INPUT_MUX_ENABLE*. This particular multiplexer has been chosen for its low off capacitance of 2.5pF and its dual-supply operation which is necessary as all DC offset has been removed from both signals. The bandwidth of the switch exceeds the frequencies of interest by an order of magnitude.

Voltage to Current Conversion

The AD5933 chip is designed to apply a known frequency and voltage signal to an unknown impedance, and measure the returning current by converting the signal back to a voltage using the internal transimpedance amplifier arrangement. In our case, the input signals are a voltage representation of the current through the impedance, and the potential difference across the impedance. Both signals need to be converted back to a current signal, which will then be converted to a voltage internally by the chip. The AD5933 also expects that the signal will be biased by $V_{DD}/2$ due to its single supply operation. This biasing needs to be applied prior to the signal being input to the chip. This circuit is depicted in Figure 2.8.

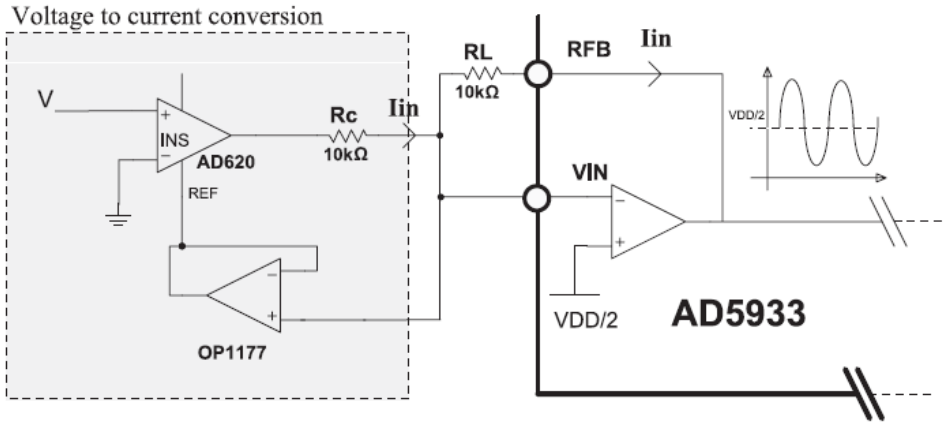


Figure 2.8: Voltage to current measurement stage for the revision 2 device [15].

The AD620 instrumentation amplifier is fed the voltage to convert with reference to ground, the differential output voltage is applied across R_c. This produces a current I_{in}, proportional to the applied voltage. This signal is then converted back to a voltage with the AD5933s internal transimpedance amplifier. In order to provide the necessary DC offset of $V_{DD}/2$, the AD620 reference pin is used. The AD620 simply adds the DC level present on this pin to the output signal. As the internal amplifier non-inverting pin is connected to $V_{DD}/2$, the V_{in} pin of the AD5933 is also $V_{DD}/2$. This is passed through a simple voltage follower to prevent any current flow, and then connects to the AD620 reference pin to provide the necessary input bias.

Low Frequency Measurements

A limitation of the revision 1 device was the lower cut-off frequency of 5kHz, due to the clocking errors caused when dividing the 16MHz master clock below this point. Ideally, it would be desirable to measure at frequencies below 5kHz to provide more data points to extrapolate. The AD5933 datasheet specifies a potential solution, whereby a frequency bandwidth of 300Hz to 5kHz can be achieved using a 2MHz master clock.

To achieve a combination of the lower 300Hz to 5kHz bandwidth using a 2MHz master clock, and the higher frequency bandwidth of the 16MHz master clock, both clocks need to be supplied to the AD5933. An ADG849 analog multiplexer has been chosen to toggle the enable pins of two such 2MHz and 16MHz oscillators. This will allow the microcontroller to switch between the two master clocks supplied to the impedance analyser. By then calibrating the AD5933 for the respective master clock, a wider measuring frequency bandwidth can be achieved by the device.

Supporting Circuitry

The output signal of the AD5933 is high-pass filtered to remove DC biasing from the measurement signal. This filtering helps to remove bias current present within the op-amp circuitry, and to prevent polarising the electrodes, as has been previously mentioned. However, a limitation of this filtering, is that the devices used in the analog front end are now required to operate in a dual-supply configuration, to prevent the excitation signal from being attenuated when the amplitude falls below $V_{DD}/2$. To support this operation, in addition to the 3.3V switching regulator present from revision 1, a second 5V switching regulator has been added. The 3.3V supply has been retained due to the embedded microcontroller logic operating at this level, while an additional 5V supply allows the selection of certain integrated-circuits that require this voltage level. A TL7660 buck-boost converter then inverts the 5V signal, providing the -5V supply necessary for dual-supply operation.

2.3.2 Impedance Calculation and Calibration Procedure

In operation, the proposed second revision device takes voltage and current measurements of the unknown impedance, then using Ohm's law, the impedance can be calculated. However, in a similar fashion to the first revision, it is first necessary to calculate the system gain factor using a calibration load. The complex calibration vector or gain factor, $K(f)$, can be deduced by Equation 2.7.

$$K(f) = Z_c(f).\frac{X_i(f)}{X_v(f)} \tag{2.7}$$

where $Z_c(f)$ is a known calibration impedance and $X_i(f)$ and $X_v(f)$ are the measured current and voltage magnitudes, calculated in the same fashion as the first revision using Equation 2.2. This calibration vector can then be used to calculate the unknown impedance, $Z_x(f)$, as seen in Equation 2.8.

$$Z_x(f) = \frac{X_v(f)}{X_i(f)}.K(f) \tag{2.8}$$

The phase through the unknown impedance must be calculated for both the current and voltage signal. The phase can again be calculated in a similar fashion to the first revision, using Equation 2.4. The final phase shift through the unknown impedance can then be calculated using Equation 2.9.

$$\theta_{unknown} = \theta_{current} - \theta_{voltage} \tag{2.9}$$

In order to calibrate the system, a test load is connected as shown in Figure 2.9. To model the bioimpedance measurements as closely as possible, the impedances between the measuring and polarising electrodes are included, along with the unknown impedance Z_x. As there exists a significant impedance between the electrode-skin interface, these impedances, Z_i, are also represented in the calibration circuit.

It is desirable, as with revision 1, to calculate the system gain factor at each excitation frequency in order to compensate for frequency dependent gain within the system. To achieve this, a ADG1634 4-channel 2:1 analog multiplexer has been used. This device allows the microcontroller to switch measurements between the external electrodes and the on-board calibration circuit using a single control line. The ADG1634 offers low off capacitance of 19pF, and a low on resistance of 4.5Ω.

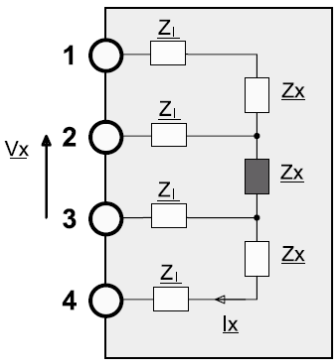
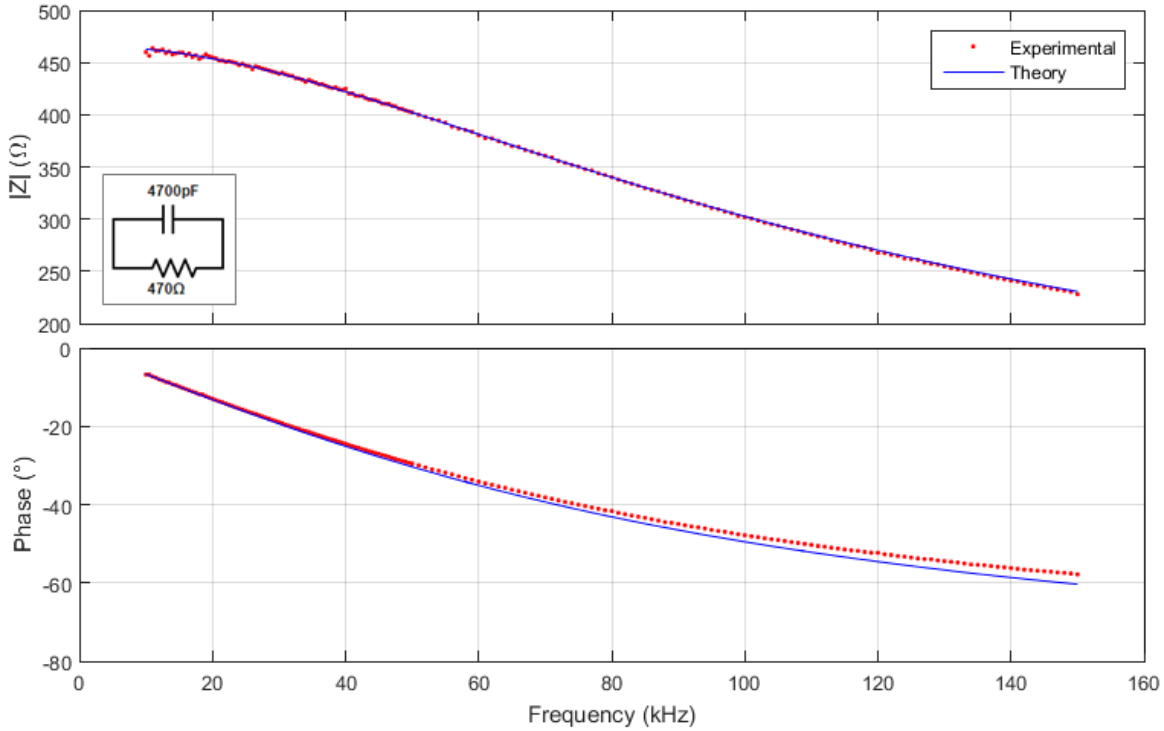
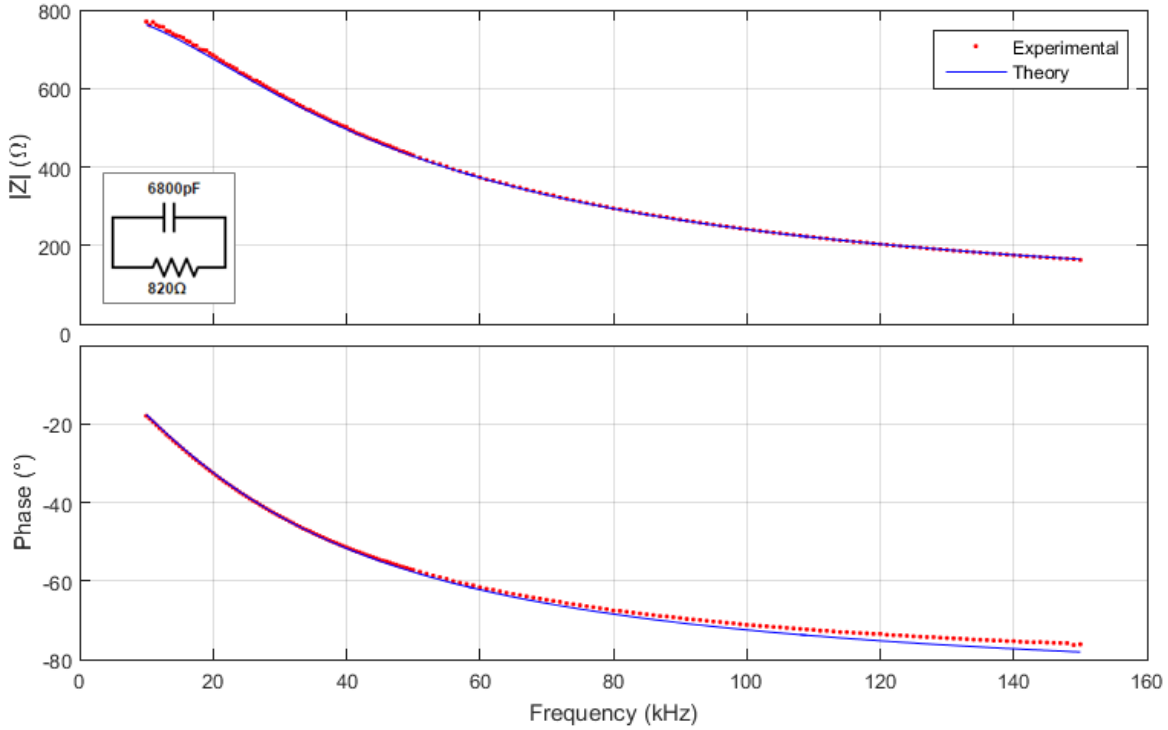


Figure 2.9: Calibration circuit for the revision 2 device [15].

2.3.3 Test Impedance Measurements

The results of passive impedance circuit testing with the revision 2 device can be seen in Figure 2.10, the first two circuits tested were also used in revision 1 testing. The frequency bandwidth for the second revision of the board has been extended to cover a 10kHz to 150kHz bandwidth. The lower band of 10kHz was chosen due to poor performance below this frequency, that tended to deviate so significantly as to not be meaningful in many cases. The use of a 2MHz oscillator did not appear to rectify this issue, instead causing a significant deviation between measurements when the clock was switched to the 16MHz oscillator. It appears to be that the 2MHz clock has significant error above 5kHz due to attenuation within the signal, while the 16MHz oscillator has significant error below 10kHz due to division errors. This issue is more consistently rectified by starting measurements at 10kHz rather than skipping the 5-10kHz band. In a future revision, it would instead be better to use a flip-flop based clock divider, which could increment the clock more gradually than the 14MHz increase in clock present in the current circuitry.


Experimental
Theory
|Z| (Ω)
4700pF
470Ω
Phase (°)
Frequency (kHz)


Experimental
Theory
|Z| (Ω)
6800pF
820Ω
Phase (°)
Frequency (kHz)

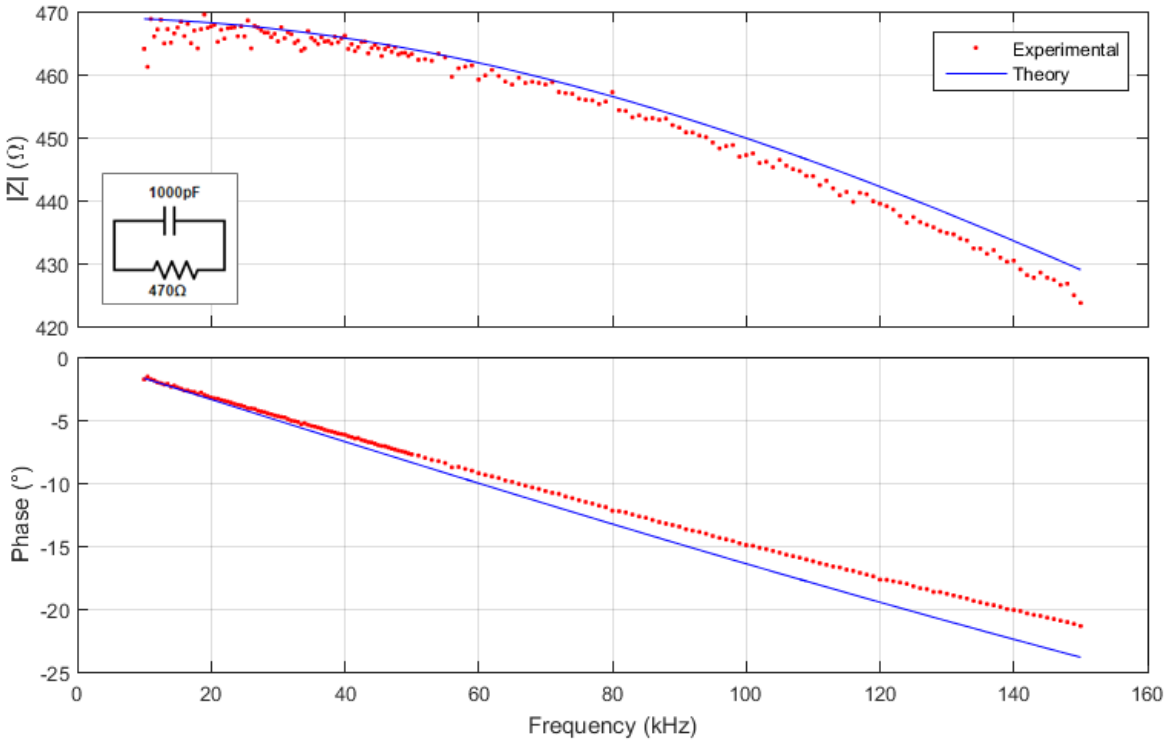


Figure 2.10: Calculated and measured impedance (top) and phase spectrum (bottom) vs frequency for three test circuits (inset) using the revision 2 device.

It can be seen from comparing the resulting impedance and phase measurements of the first two circuits to that of the revision 1 results, that the revision 2 circuitry has significantly improved the device accuracy. Circuit 1 impedance and phase maximum errors are now $\pm 1.3\%$ and $\pm 4.3\%$ respectively, compared with the $\pm 8.3\%$ and $\pm 21.7\%$ maximum errors from revision 1. Revision 2 also covers an additional 50kHz above the 100kHz limit of revision 1, and takes significantly more data points (179 compared to 18). Given that the maximum error is much lower, even with significantly more data points being taken, in can be concluded the revision 2 device has a much higher level of precision than revision 1. Measurements on circuit 2 using the revision 2 device have a maximum error of $\pm 1.8\%$ and $\pm 2.7\%$ for impedance and phase measurements respectively, also improving on the $\pm 2.4\%$ and $\pm 2.8\%$ results of revision 1. However, it can be observed, that there still remains some slight offset of phase past 50kHz that increases with frequency.

The third test circuit tested is a lower capacitance circuit, to test the precision of nearby frequency measurements and to give some insight into the level of noise present in the measurements. The resulting impedance measurements deviate a maximum of $\pm 5.26\Omega$ ($\pm 1.2\%$) from the theoretical value. Upon visual inspection it can be seen that there remains significant variations between neighbouring frequency measurements. It is likely that noise is introduced more readily due to the decrease in signal-to-noise ratio (SNR) caused by attenuating the output signal of the AD5933. It would be beneficial to attempt to reduce this attenuation in a following version of the device. A phase offset increasing with frequency can be observed, leading to a maximum error of $\pm 2.5^\circ$ ($\pm 10.67\%$). This phase error is likely caused by a parastic within the external measurement circuitry, which reduces the effective bandwidth of the device and causes attenuation at high frequencies. This needs to be addressed through the board layout and PCB design.

In a similar fashion to the revision 1 testing, a simplistic human bioimpedance phantom has been used to evaluate the performance of the device on a circuit with behaviour similar to a human. The passive component values themselves have been chosen to cover the resistance and reactance ranges of a typical human. Two plots of resistance vs frequency and the subsequent Cole-Cole plot will be used in the hydration analysis detailed in Chapter 4 and as such, these two plots are included in Figure 2.11 to better predict the device's performance when performing actual bioimpedance hydration analysis.

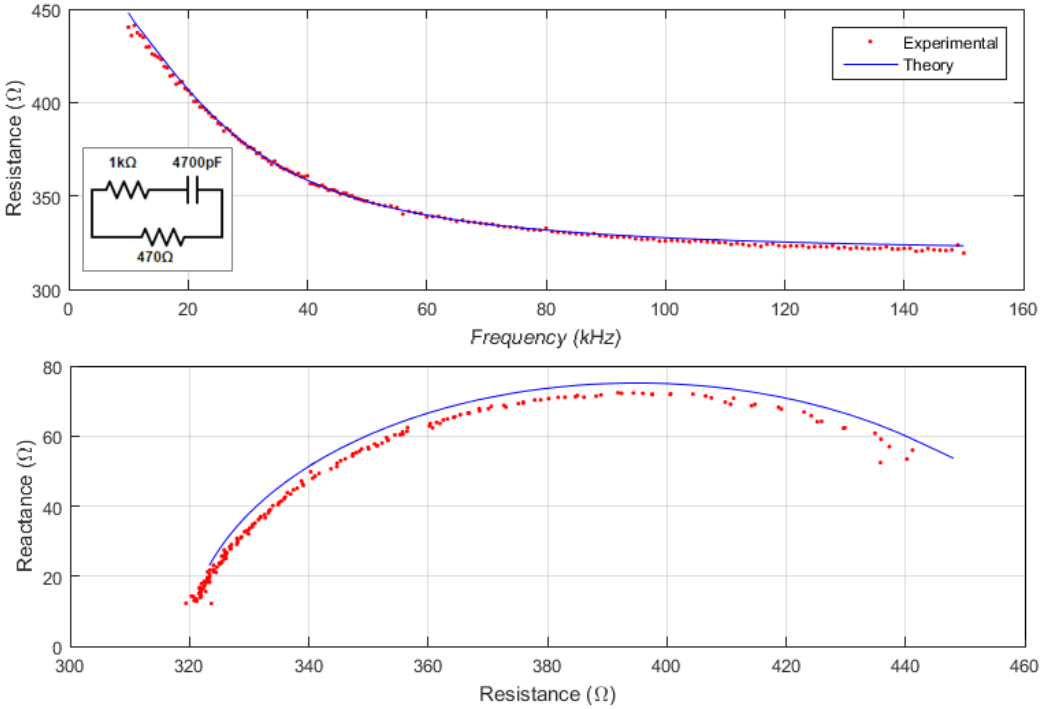


Figure 2.11: Calculated and measured resistance vs frequency (top) and Cole-Cole plot (bottom) for an bioimpedance phantom circuit (inset) using the revision 2 device.

Again, a reasonable agreement can be seen between the theoretical and measured data. There exists a small $\pm 3.91\Omega$ ($\pm 1.2\%$) resistance maximum error, and a much more significant reactance maximum error of $\pm 10.9\Omega$ ($\pm 47.2\%$). This error occurs at high frequency, as can be observed within the figure. As has been mentioned previously, the high frequency errors can be attributed to parasitics present in the external measurement circuit, cables and the connections to the test circuitry. These issues will be addressed in the future revision PCB and mechanical design.

2.4 Hydration Monitor Revision 3

2.4.1 Electronic Design

The third and final revision of the hydration monitor device mostly targets the usability of the device, to enable thorough testing. As has been observed within the results in section 2.3.3, the device provides a reasonable level of accuracy on passive components. While this indicates the current design is suitable for the bioimpedance measurements, the accuracy and precision are still outside the specified 1% and 5% accuracy ranges for impedance and

phase. In addition to targeting accuracy and precision, in order to effectively test the device a number of other necessary changes have been made. These changes attempt to improve device safety, digitise gain control and increase the generally functionality of the device to allow third party testing.

AD5933 Output Circuitry

The second revision of the analog front end for the AD5933 has no control over the current flowing from it's output. While this current is unlikely to exceed safe levels due to the small amplitude of the excitation signal and the significant impedance present at the electrodes and through the body, it is still desirable for this current to be controlled given that excess current however unlikely, could have serious side effects for patients. One such approach is to use a voltage controlled constant current source, which is placed at the output of the AD5933 IC. The circuitry used to measure the returning signal can therefore remain unchanged. A Howland current pump [16] has been chosen, due to its easy implementation and voltage-controlled nature. The proposed circuit is shown in Figure 2.12.

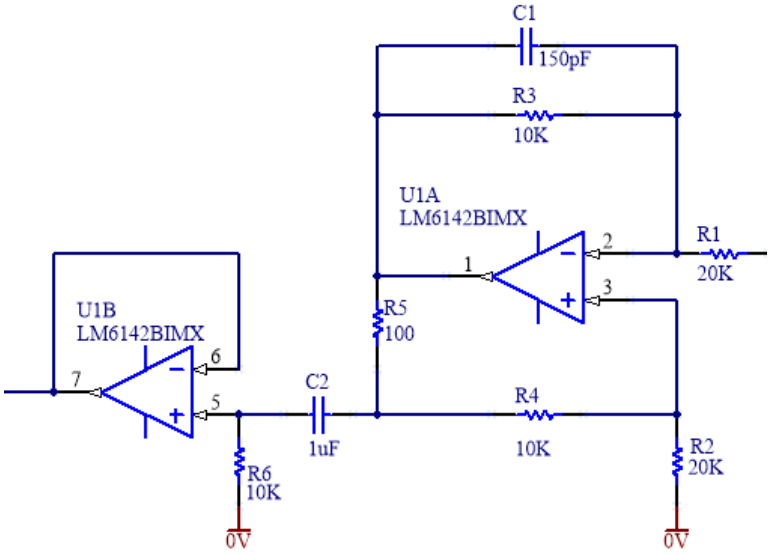


Figure 2.12: Voltage controlled Howland constant current pump.

The Howland current pump utilises both positive and negative feedback, which maintains the voltage potential at both input terminals of the op-amp as equal, with gain controlled by the feedback resistor ratios. Due to this fact, the output current is constant, with dependency only on the input voltage. Capacitor C1 is included to prevent oscillations from occurring by introducing a large pole in the feedback loop. Finally, the output current signal is high-pass filtered to remove any DC components present from the AD5933 output. The equation for the current output from this circuit is detailed in Equation 2.10.

$$I_{output} = \frac{-R_3}{R_1.R_5}.V_{input} \tag{2.10}$$

Issues were quickly encountered with the Howland current pump upon testing. The most significant of which being a limited bandwidth to which the current signal remains constant. It was found that above 30kHz, the signal begins to attenuate significantly, causing a

reduction in signal-to-noise ratio (SNR). As a 10kHz to 150kHz bandwidth is desired, this behaviour is unacceptable for the device. Instead, the initial arrangement from revision 2 of the device was reverted to. In an effort to increase SNR, the inverting-amplifier used to attenuate the signal in revision 2 was replaced by a simple voltage follower. The larger amplitude signal is less susceptible to smaller amplitude noise that might otherwise cause significant error. In addition, the noise component of the signal is no longer amplified along with the desired signal within the gain sections of the device. The larger amplitude signal does however present an issue where the measurement signals exceed the acceptable 0V to 3.3V range of the AD5933 transimpedance amplifier, which needs to be accounted for within the amplification section of the device.

Signal Amplification

The revision 2 device utilises an arrangement of two non-inverting cascaded amplifiers to amplify the signal. With the removal of the initial attenuation of the signal output from the AD5933, the system now requires the return signals to be attenuated rather than amplified. To this end, the non-inverting amplifiers have been replaced with inverting amplifiers, with gain controlled by the feedback resistor. By reducing the feedback resistor value below the input resistance, the signal will now be attenuated by the ratio $-R_{feedback}/R_{input}$. The input resistor values have been chosen to be 47kΩ.

The level of desired attenuation is still dependant on the magnitude of the load impedance, as this influences the measured voltage and current signals. The resistors used to control this attenuation are difficult to alter in the field, which makes altering gain in practice difficult using simple passive elements. The feedback resistance has instead been replaced with the MCP4661 dual-channel 50kΩ digital potentiometer IC. Controllable over the same I^2C bus used to communicate with the AD5933 IC, this potentiometer allows the gain to be altered programmatically, in turn allowing the gain to be altered as needed in the field. Possible applications could be different electrode positions, such as a chest cavity measurement as opposed to a full body measurement, where the impedance changes enough to require an alteration of the system gain.

Functionality Improvements

Until this point, a breakout level-shifter/USB-to-serial converter board has been used for communication between the hydration monitor and a PC. To remove the need for a secondary board, a micro-USB port has been added to the revision 3 device. An additional FTDI FT232 chip has been added to perform the USB-to-serial conversion, making the device completely plug and play, with no addition software drivers or hardware adaptors needed.

The secondary benefit of the inclusion of a micro-USB port on the device is the use of the 5V USB supply as a power source. To still allow the device to be portable, ideally the device could operate off battery or USB power interchangeably, with priority given to the USB power source. With this in mind, a circuit has been designed to allow battery powered operation, which automatically switches to USB power when plugged in. The circuit is shown in Figure 2.13. The p-type MOSFET transistor is ON when no voltage is present on the USB line, and power is provided to the circuit from the battery. When the USB plug is connected, the 5V signal on the USB line switches the MOSFET OFF, and now the power

flow is from the USB source. The diode $D1$ is included to prevent current flow through $R2$ when the USB power is disconnected. The poly-switch $PS1$ has also been added to limit the current draw of the circuit to 200mA in the case of a fault condition.

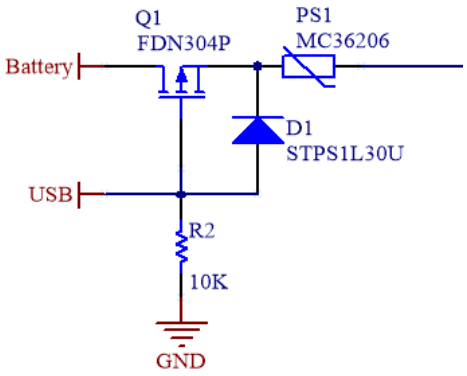


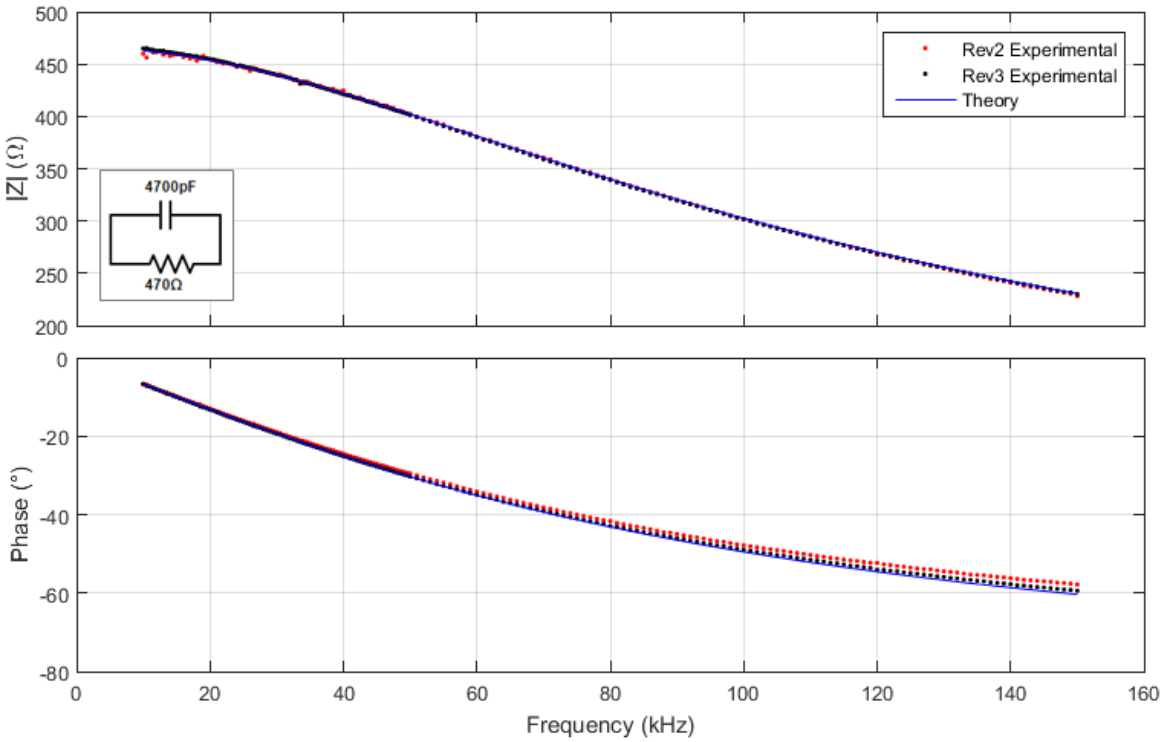
Figure 2.13: Automatic power supply switching circuit.

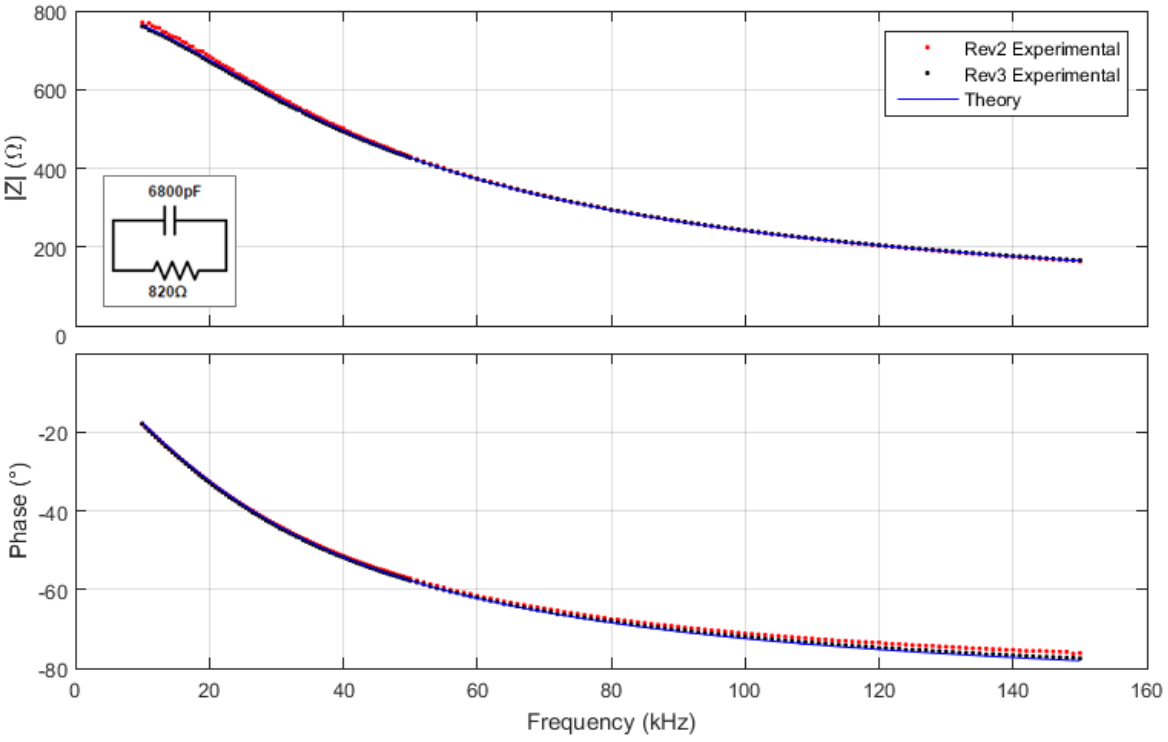
The final major addition to the revision 3 circuitry is USB storage capabilities. While limitations in software, which are detailed in section 3, prevent the calculation of the hydration parameters by the embedded hardware, it is still desirable to store the bioimpedance data for later analysis if a PC is unavailable. The VNC1L USB storage IC has been added to achieve this functionality. The IC connects directly to a type-A USB port also included on the PCB, and when connected to a storage device such as a FAT32 flash drive, carries out the data writing and reading protocols. The data to be written, or the read data, can then be communicated from the embedded microcontroller. In the case of this device, SPI was chosen as the serial communication protocol between the microcontroller and VNC1L chip due to the UART port already being used to communicate with the PC.

2.4.2 Test Impedance Measurements

The results of passive impedance circuit testing using the revision 3 device can be seen in Figure 2.14. The same three circuits from the revision 2 testing have been used, the results are also displayed to highlight the differences in the measurements of the two devices. It can be seen that the first two figures show a very close relationship between the theoretical and actual data. The revision 2 device achieved an accuracy of $\pm 1.3\%$ and $\pm 4.3\%$ in impedance and phase respectively. This error has been reduced further, the revision 3 device achieves a maximum error of only $\pm 0.6\%$ ($\pm 2.82\Omega$) and $\pm 1.8\%$ ($\pm 0.22^\circ$) for impedance and phase. Similar results were achieved on circuit 2, an improvement from $\pm 1.8\%$ and $\pm 2.7\%$ error in impedance and phase to $\pm 1.1\%$ ($\pm 1.84\Omega$) and $\pm 1.8\%$ ($\pm 0.46^\circ$).

The third test circuit is a lower capacitance circuit used to test the precision of nearby frequency measurements. It also gives some insight into the level of noise present in the measurements. It was found that the revision 2 device had a maximum error of $\pm 1.2\%$ ($\pm 5.26\Omega$) and $\pm 10.67\%$ ($\pm 2.5^\circ$) for impedance and phase respectively. It can be seen from the third circuit results in Figure 2.14 that the precision and accuracy have been significantly improved when compared with the previous device. The revision 3 device achieves an accuracy of $\pm 0.39\%$ ($\pm 1.80\Omega$) and $\pm 5.69\%$ ($\pm 0.38^\circ$) for impedance and phase. In addition, the frequency dependant phase offset present in revision 2 has been reduced, a 1.09° offset is present at 150kHz over the 2.51° offset at the same frequency using the revision 2 device.


Rev2 Experimental
Rev3 Experimental
Theory
4700pF
470Ω
|Z| (Ω)
Phase (°)
Frequency (kHz)


Rev2 Experimental
Rev3 Experimental
Theory
6800pF
820Ω
|Z| (Ω)
Phase (°)
Frequency (kHz)

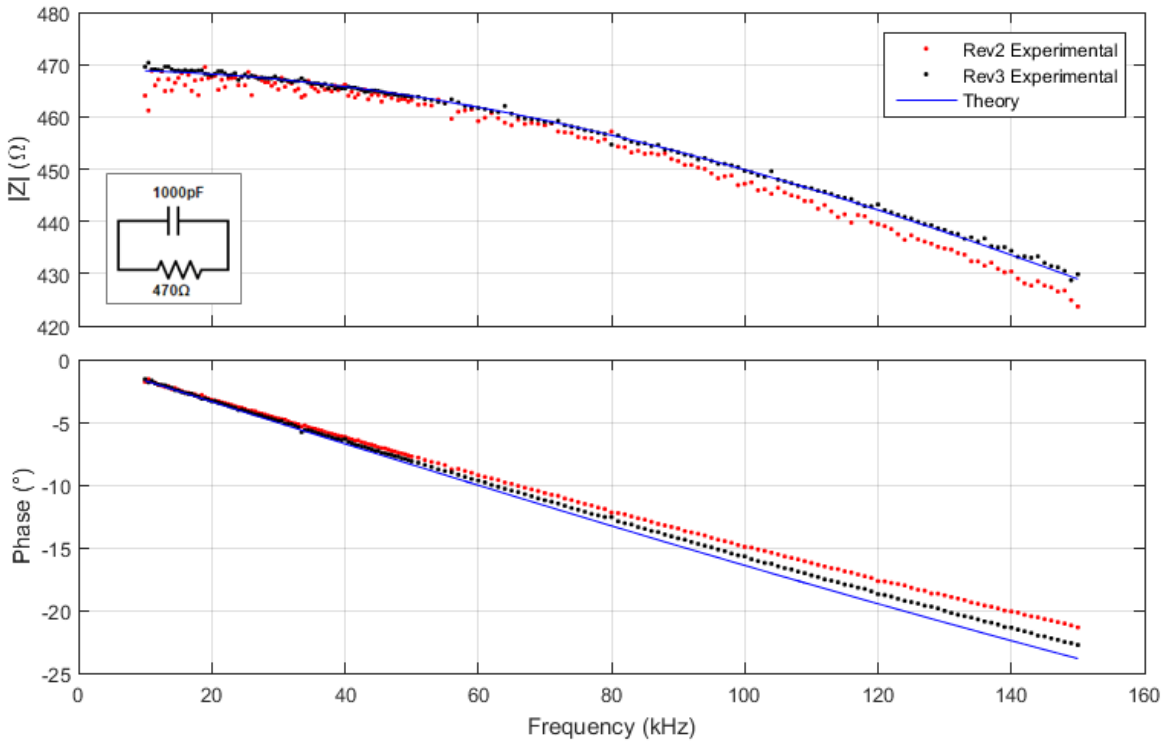


Figure 2.14: Calculated and measured impedance (top) and phase spectrum (bottom) vs frequency for three test circuits (inset) using the revision 3 device.

The increase in precision can be attributed to the removal of the attenuation of the 2Vp-p signal from the AD5933, therefore increasing SNR. The accuracy improvements are again likely due to a reduction in parasitics due to layout improvements.

Depicted in Figure 2.15 are the results of testing the revision 3 device on an impedance phantom, the revision 2 results are also plotted for comparison. Visually, revision 3 device provides are much better fit of the theoretical curve, with only a small reactance offset. When testing the revision 2 device, it was found that there existed a significant $\pm 47.2\%$ ($\pm 10.9\Omega$) error of the reactance at higher frequencies, along with an resistance error of $\pm 1.2\%$ ($\pm 3.91\Omega$). These issues were combated with improved board layout, an increase in SNR and an improved system for calibrating the gain digitally. As a result, when observing the revision 3 experimental results for the same circuit, the reactance error has been significantly reduced to $\pm 11.1\%$ ($\pm 2.56\Omega$). While the error is still a significant percentage of the value, bioimpedance measurements are not expected to have such a reactance value under 30Ω. The error is less than $\pm 5\%$ for reactance values over 30Ω. The resistance error has also been reduced to $\pm 0.34\%$ ($\pm 1.25\Omega$).

It can be concluded that the revision 3 device is a precise impedance measurement device, capable of an accuracy within 1% of the real impedance and 5% of the actual phase, meeting the necessary performance outlined within the specifications. These are the maximum errors, the average impedance error was $\pm 0.3\%$ while the phase error was $\pm 1.7\%$ across the 3 RC test circuits. Given that the initial electrode impedance issue present in the revision 1 device has been removed with the tetra-polar measurement approach used in revisions 2 and 3, the revision 3 device is ready to be tested for bioimpedance performance. This testing and subsequent hydration analysis will be covered in Chapter 4.

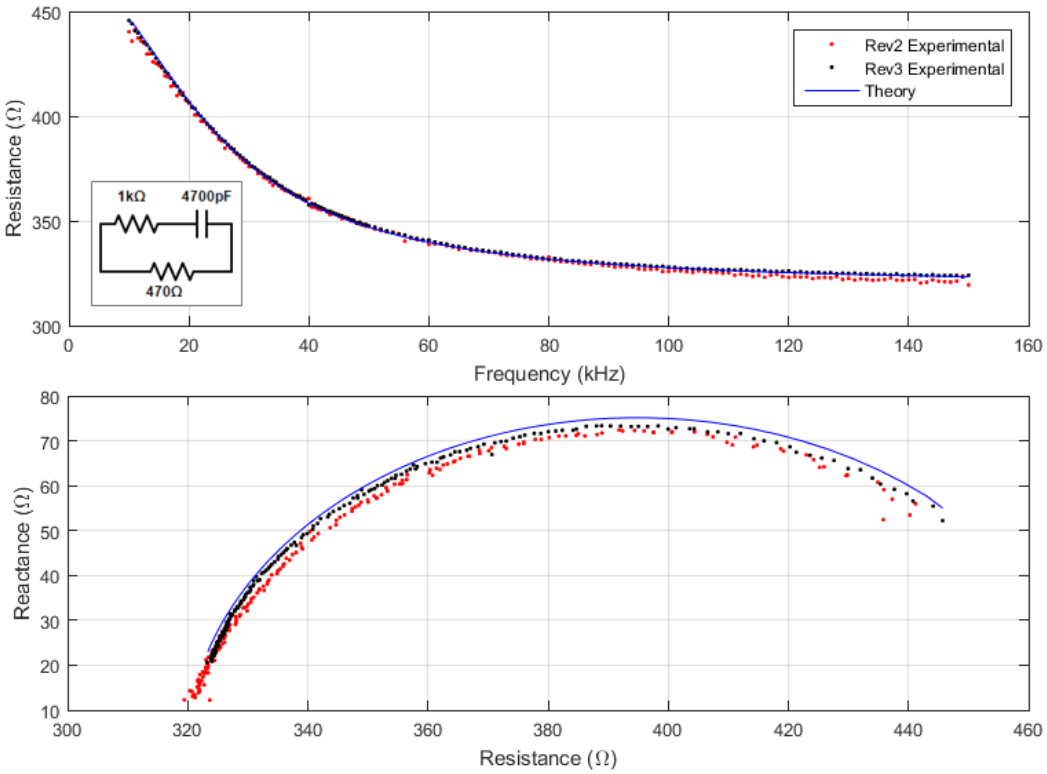


Figure 2.15: Calculated and measured resistance vs frequency (top) and Cole-Cole plot (bottom) for an bioimpedance phantom circuit (inset) using the revision 3 device.

2.5 Hardware Design Summary

The revision 3 device is a tetrapolar based bioimpedance analyser device, performing bioimpedance spectroscopy across a 10kHz to 150kHz bandwidth. The impedance measurement circuitry is based off an AD5933 spectral impedance analyser IC, fitted with additional circuitry to measure the returning current and differential voltage across the unknown impedance. A transimpedance amplifier performs the necessary current to voltage conversion, while an instrumentational amplifier measures the differential voltage across the unknown impedance. Digitally controlled gain allows the varying impedance magnitudes to be measured such as a full body or chest cavity bioimpedance measurement. Two sets of multiplexers allow the AD5933 IC input to be switched between the current and differential voltage across the unknown impedance, and to switch measurement between a calibration circuit and the unknown external impedance.

High SNR & CMRR components, filters and bypass capacitors have been fitted to minimise noise. The embedded microcontroller, the ADuc7024, communicates with the AD5933 IC via I^2C, then across USB to a connected PC. The device can be powered via this USB connection or via battery. The power source selection is automatic, USB power takes precedence over battery to preserve charge, the power source automatically switches if a power source is disconnected. Finally, a type-A USB port and the VNC1L USB storage IC enables the device to store data on a USB FAT32 device.

A case has also been designed in order to allow the bioimpedance measurement device to be portable while still protecting the electronics. A case also allows a battery to be carried, as well as mounting for the buttons, LCD and probe ports. The case has been designed

in SolidWorks, a computer aided design program, and then laser cut from 6mm acrylic. Figure 2.16 depicts the device PCB mounted within the case, holes have been included in the sides for the fitted USB ports.

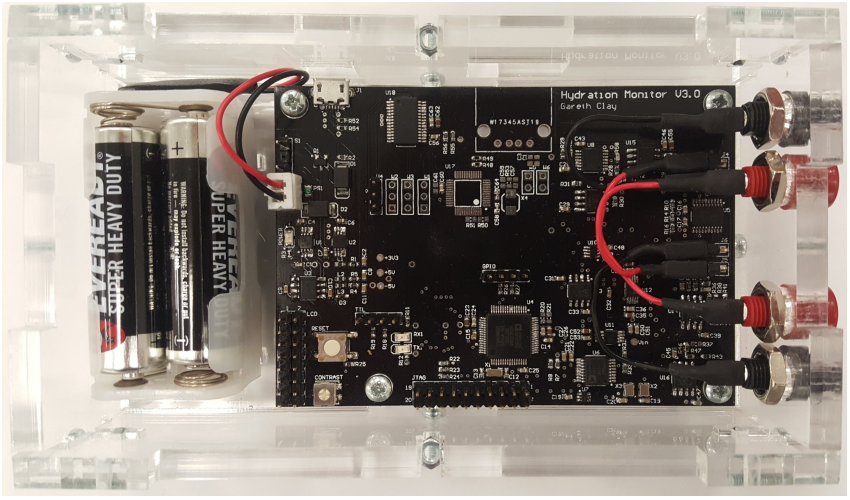


Figure 2.16: Top-down view of the hydration monitoring device PCB and battery pack mounted in case.

The full device with the lid mounted is shown in Figure 2.17. To allow easy access to the internal electronics and battery, the case lid has been designed to simply unscrew from the base by removing two bolts. As can be seen in the figure, the LCD, buttons and power switch are all readily accessible to the user. The case bottom has been fitted with rubber stands prevent the device slipping off surface during use. Finally, the probe ports have been mounted on the left end, colour coded with red for the voltage electrodes and black for the current injection and receiver electrodes.

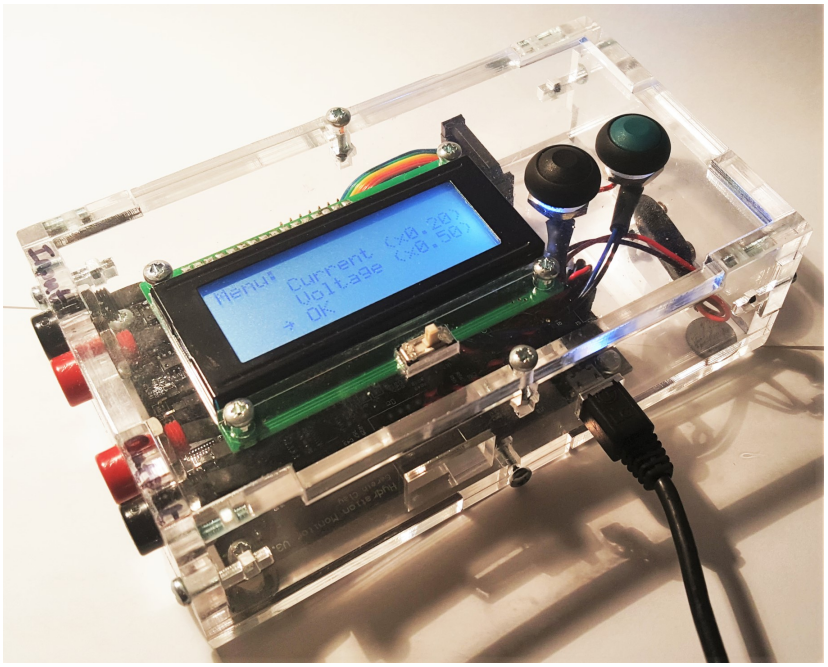


Figure 2.17: Isometric view of the fully enclosed bioimpedance measurement device.

Chapter 3

Bioimpedance Measurement Device: Software Design

3.1 Embedded Software

The ADuC702X series of microcontrollers by Analog Devices are the embedded platfrom used to control the hydration monitor device, described in section 2.1. This series of controllers use ARM7TDMI 32-bit architecture [17]. The C programming language is the language of choice for embedded microcontrollers, the embedded software of which has been developed using the Kiel μVision integrated development environment (IDE). This particular IDE has been chosen as it integrates the ARM7 compiler needed to upload to the microcontroller, with support for a SEGGER J-link, the JTAG debug probe used to perform the hardware upload to the embedded chip.

The following sections detail the more relevant embedded software sections that are used to perform the bioimpedance measurements. In addition to these software sections, supporting libraries were written for the configuration of the various hardware and communication protocols needed for the device to function. These include libraries for the system LCD, the microcontroller clocking and delay configurations, interrupts and the communication protocols required by the system (I^2C, SPI and UART).

3.1.1 Device Configuration

Controllable System Gain

Powering on, the microcontroller first runs through a configuration assembly file to perform the register assignment required by the embedded software. Next, the written user-defined library initialisation methods are run to setup system interrupts, configure the registers and assign GPIO. An initialisation routine then configures the system gain, which as detailed in section 2.4, involves setting a MCP4661 digital potentiometer IC. The I^2C communication protocol is used to configure this device, the same bus used by the AD5933 to communicate with the microcontroller. This protocol allows the use of multiple devices on the same bus due to its unique address identification. Each device is identified using a 7-bit address, packets of data with an address not matching the receiving device are simply ignored. The MCP4661 device data formatting scheme is detailed in Figure 3.1.

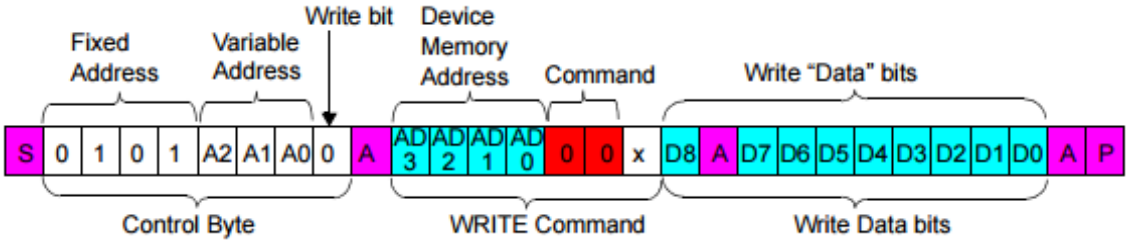


Figure 3.1: The MCP4661 I^2C communication scheme [18].

The first byte of data written is the address control byte, the address of which has been assigned in hardware as 0x28. The final bit of the byte is to determine a read or write operation, this bit has been set to zero for a write. Next, the write command is transmitted to specify the memory address and the command bytes to be written. In order to set the wiper position of the two potentiometers, the command byte is set programmatically as $0x8 \,\|(pot << 4)$, where *pot* is either 0 or 1 for potentiometer 0 or 1 respectively. Finally, a data byte is written to set the wiper position, which ranges from 0 to 255. A separate method converts a resistance in ohms to a wiper position by dividing the resistance in ohms by 181.3 (the resistance change between two adjacent wiper positions), then integer rounding the result. The gain of the returning current and voltage signals can now be controlled programmatically by writing the two resistance values to the potentiometer.

It is desired that the user is able to control the gain in regular operation of the device, without the need for re-programming the microcontroller. The actual gain is found by dividing the potentiometer resistance by the input resistance of the inverting amplifier circuitry detailed in section 2.4.1. The user can therefore simply select a gain, and the conversion to the correct resistance and subsequent potentiometer setting can be done within the embedded software automatically. A simple state machine-based menu has been implemented, which is shown in Figure 3.2. The user pushes one button to move the arrow cursor between options, and a second button to select the corresponding option. Selecting the current or voltage gain options provides a secondary menu, where the same select button can be used to increment the gain value, ranging between 0.05 and 1 times gain in increments of 0.05. When the appropriate gain has been set, the user can select OK to begin the measurements.

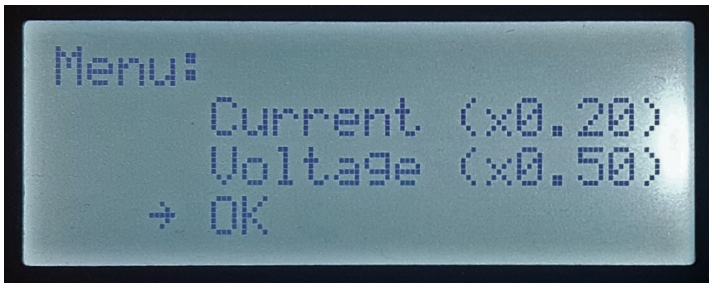


Figure 3.2: LCD displayed user menu to conFigure the hydration measurement device gain.

AD5933 Frequency Sweep Configuration

The AD5933 IC must be configured before a frequency sweep can be performed of the unknown impedance. As has been mentioned, this configuration and the subsequent data

retrieval are performed via I^2C communication. The AD5933 supports three operations: a single byte read/write, a block read/write and a pointer address set for selecting a particular register prior to performing a block operation. Similar to the process in section 3.1.1, the data stream is formatted as the device address, followed by the command byte and the subsequent data bytes appropriate to the operation being performed. To configure the device, only data write operations are necessary. The two write operations are detailed in Figure 3.3.

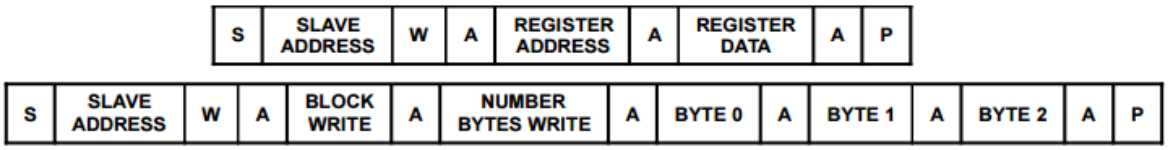


Figure 3.3: The AD5933 I^2C single byte (top) and block write (bottom) communication schemes [14].

It is necessary to perform block write operations, where multiple data bytes are written, due to the 8-bit nature of the control registers. The starting address is determined by setting a pointer to the starting register address, as detailed in Figure 3.4. The pointer address is incremented by one for each subsequent data byte written, allowing a 24-bit parameter to be stored in the memory.

Figure 3.4: The AD5933 I^2C pointer set communication scheme [14].

The configuration parameters to then be written in order to perform a frequency sweep are:

- System clock source & frequency.
- Excitation signal amplitude.
- Internal transimpedance amplifier gain.
- Start frequency.
- Frequency increment.
- Number of frequency increments.
- Settling time.

The system clock source is an external 16MHz oscillator, and as such the lower control register 0x81 is written 16×10^6 via a single byte write. The output excitation signal desired amplitude is 2Vp-p, with a transimpedance gain of 1. The corresponding byte is 0x01, which is written to the high control register 0x82. To set a frequency, a conversion must first be made to the appropriate 24-bit code required by the AD5933. This conversion accounts for the particular system clock in use, and allows the data to be written in 3 bytes using a block write. The frequency code for a particular frequency f_x is:

$$Frequency\ Code = \left(\frac{f_x}{(\frac{16MHz}{4})} \right) \times 2^{27} \tag{3.1}$$

The desired start and increment frequencies are written to registers 0x82 and 0x85 respectively. The number of increments are written to register 0x88. Finally the settling time is written to register 0x8A, this is the number of frequency cycles before the ADC is triggered to perform a conversion. Data byte 0x2 is written to the high control register to begin the frequency sweep.

3.1.2 System State Machine

The general operation of the bioimpedance measurement device works within a state machine, the states of which are shown in Figure 3.5. On start up, after completing the initialisation routines, the device enters the gain select menu detailed in section 3.1.1. Following user input to select the configuration, the device enters a low frequency configuration state. This state communicates the desired parameters for the frequency sweep, the first sweep covers the 10kHz to 50kHz range in 500Hz increments. On completion of this setup, the device communicates a status done byte to the connected PC. This byte indicates the beginning of the data logging, which will be covered within the Python GUI section described in 3.2. The state is then transitioned to the low frequency sweep state.

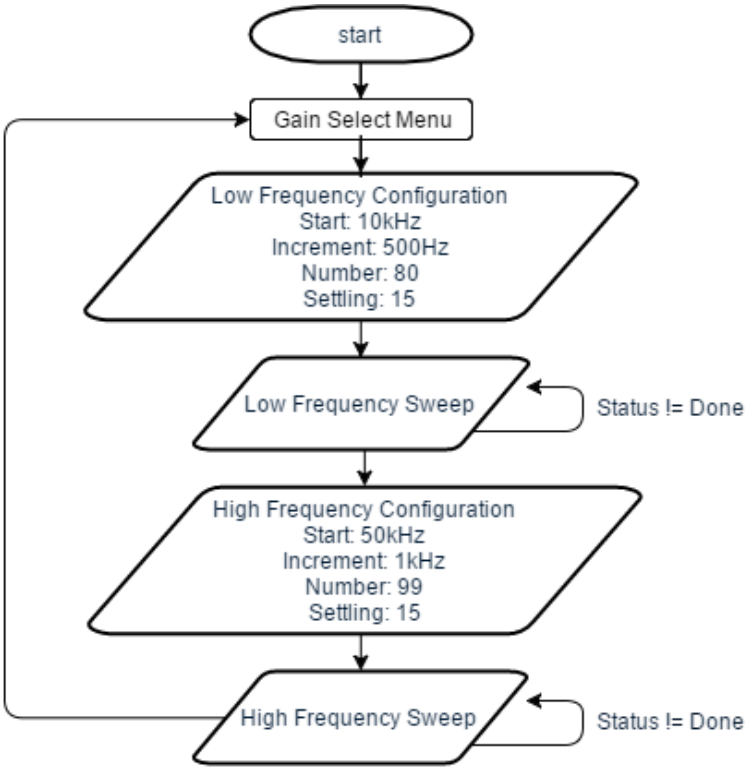


Figure 3.5: The bioimpedance measurement device state machine.

Using the relevant GPIO control signal and multiplexer, the load impedance is switched to the on-board known test resistors. The AD5933 can receive one of two commands during a frequency sweep, a 0x4 byte indicates to take a impedance measurement and not increment

the frequency, while 0x3 indicates a impedance measurement followed by an increment in the frequency. An average of 5 impedance measurements are taken of the known test resistors, using the repeat-frequency command. As both the current and voltage are measured through the load, it is necessary to toggle a second multiplexer to record both measurements. These values can then be used to find the system phase shift and gain factor, the calculations of which have been detailed in section 2.2.2. The multiplexer is then switched to measuring the external impedance, and the process of averaging across 5 measurements is repeated. The final impedance, resistance, reactance and phase values can then be calculated from the gain factor and system phase, using the process detailed in section 2.3.2.

At the end of each respective loop of a frequency, the AD5933 status register (0x8F) is polled to check if the frequency sweep has been completed. If not, the frequency is incremented and the process repeats for the next frequency. Otherwise, the loop exits and the state shifts to the high frequency configuration. This frequency sweep covers the 51kHz to 150kHz frequency range in increments of 1kHz. The configuration and subsequent frequency sweep are identical to the low frequency range, and the device returns to the gain select menu following the end of the sweep.

3.1.3 Data Transmission

The hydration measurement device records the unknown impedance magnitude, phase, resistance, reactance and the frequency at which these results were measured. The data is transmitted to a PC via UART serial, via a micro-USB cable. Each piece of data is transmitted with an identifier byte to distinguish between the pieces of data should the order be changed in a later revision, followed by the data double formatted in ASCII 32. Each piece of data and its identifier byte are tab separated, while each subsequent frequency measurement is delimited by a newline character.

3.2 Python GUI

A python GUI has been developed to work in conjunction with the bioimpedance measurement device, performing the necessary data fitting and data visualisation. The decision to not perform these calculations within embedded software was made due to a limitation of the Kiel μVision licensing, which limited the program size to only 32kB. This was not enough space to develop the desired data fitting algorithm to perform hydration calculations. Instead, it was deemed more efficient to develop a PC-based program, Python was chosen due to its ease of use and readily accessible GUI and scientific libraries.

The GUI has been designed to provide a means to compare the measured impedance against theoretical parallel RC circuits and theoretical R||RC circuits, and also to acquire and calculate hydration parameters. Each of these functions can be selected via buttons in the main page of the GUI, which can be returned to via a return button located on each function page.

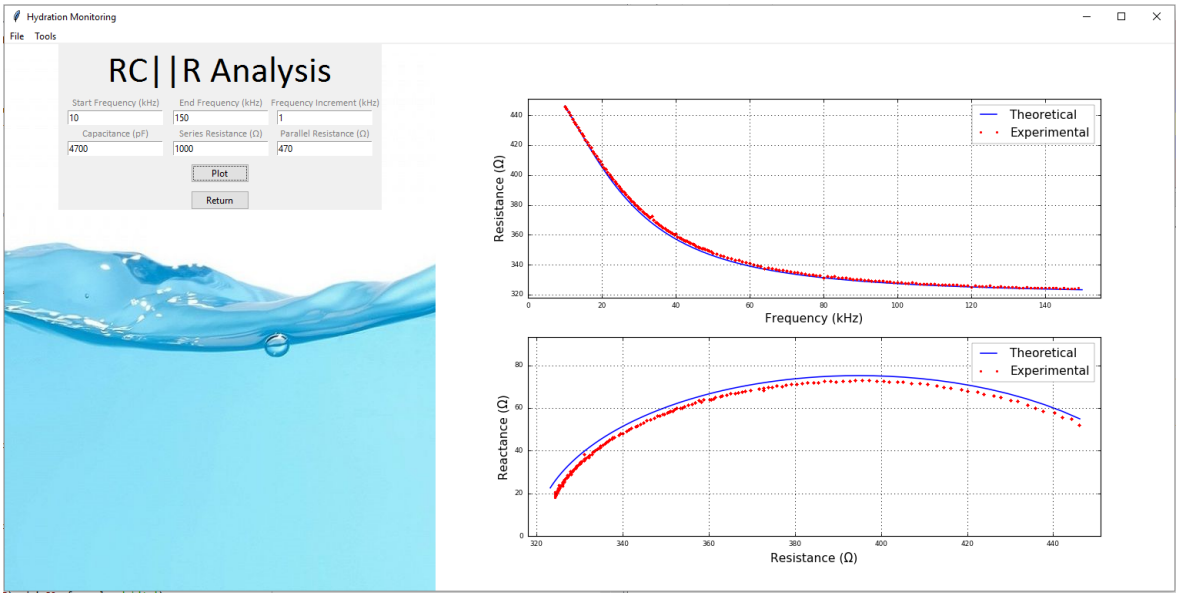


Figure 3.6: R||RC circuit theoretical values comparison GUI page.

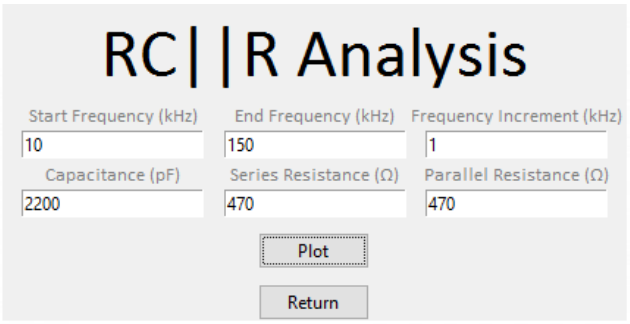


Figure 3.7: R||RC circuit theoretical parameters to plot.

The two theoretical circuit comparison functions are relatively similar, so only the R||RC circuit page will be detailed. Observable within Figure 3.6 is this function page, with example theoretical curves plotted in blue. The top plot is impedance magnitude against frequency, while the bottom is a Cole-Cole plot of reactance against resistance. The actual points of data from the hydration monitor are plotted in red, easily comparable to the theory curve. All of the circuit parameters and the frequency sweep range can be adjusted, a close up of the parameter menu located on the left is shown in Figure 3.7. The ability to compare the measured data to the theoretical curves allows for easy testing of the device's accuracy during development, as well as ensuring accuracy over time by allowing testing to be performed where the readings are easily comparable to the actual theoretical values.

The hydration analysis page can be seen in Figure 3.8. The right hand side of the page is dedicated to data plotting, where the top plot is resistance against frequency and the bottom is a Cole-Cole plot of reactance against resistance. Both plots interactively update during the frequency sweep, adding the data points as they are received from the hydration monitor device. When all the data has been received, the top plot of resistance against frequency is fitted using least-squares regression as detailed in section 1.2.3. This fit is plotted in blue,

and the resultant Cole-Cole plot is also added to the bottom graph in blue as can be seen in the figure.

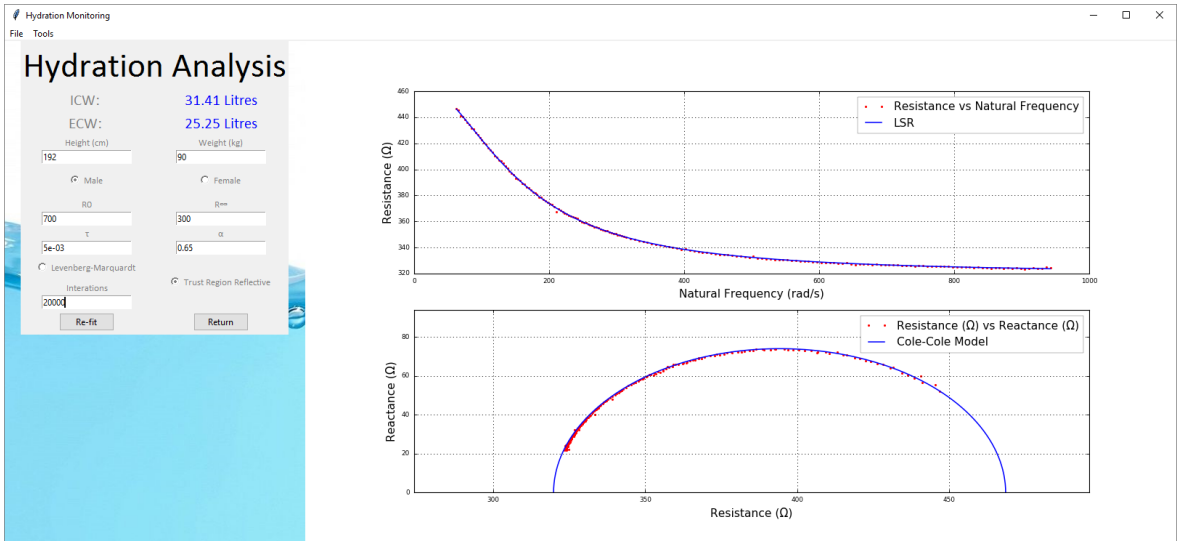


Figure 3.8: Hydration analysis GUI page, with fitting parameters (left) and data plots (right).

On the left of this GUI is the parameter menu, a close-up of which is shown in Figure 3.9. In the figure, it can be seen that the ICW and ECW labels display as 'Select serial port' in blue. The serial port menu, which can be found of the tools menu, is shown in Figure 3.10. This serial port menu offers all the available serial ports on a particular computer, allowing the device to be easily connected to different PCs without needing to change any background code. Upon selection of an appropriate serial port, the labels will change to 'Press start on device'. When this button is pressed and measurements begin, they will again change to 'Analysing...', and finally to the calculated values for ICW and ECW from the data as seen in Figure 3.8. This calculation takes input from the height and weight inputs located on the menu.

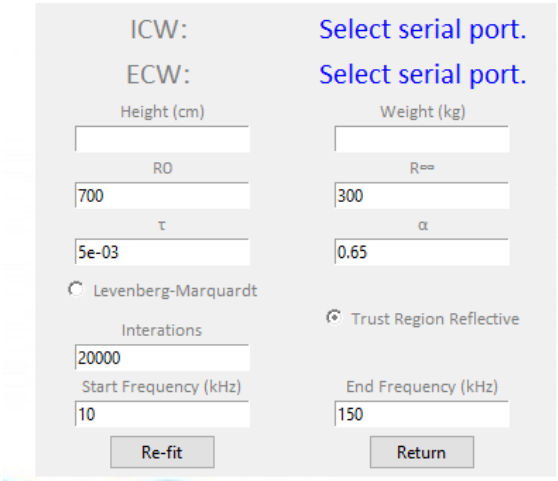


Figure 3.9: Hydration analysis parameters for hydration ICW & ECW calculation and data fitting.

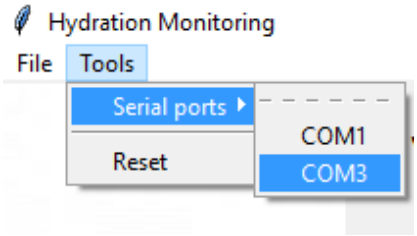


Figure 3.10: Serial port selection menu for the python GUI.

The hydration parameters menu also has a number of options for the data fitting performed when the bioimpedance data has been acquired. Two variants of the least squares algorithm can be selected, either Levenberg-Marquardt (LM) or Trust-Region Reflective (TRR). LM is iteration controllable, better suited to unbounded problems. As such, the number of iterations can be selected for this algorithm, however it does have issues where the lack of bounds fail to find a fit. In these situations, TRR can be used where each of the parameters is bounded to a reasonable region. The starting points for each of the parameters R_0, R_∞, τ and α can also be altered as shown in the figure, the default values are currently shown. The range of frequencies in which to plot and curve-fit can be altered using the 'Start Frequency' and 'End Frequency' boxes. Having control of the algorithms and parameters used to perform the hydration calculations ensure that a fit can always be found through tuning, in the cases that the default values fail to do so.

Following the bioimpedance measurements and hydration calculations, it is desirable that the data be stored. This has been achieved by saving the data to an Excel spreadsheet, the formatting from one such test can be seen in Figure 3.11. As can be seen in this figure, each of the measurements is titled and aligned with the corresponding frequency. The time stamp and date are also included at the top of the document.

16:00:16						
28/09/2016						
Frequency (kHz)	**Impedance (Ω)**	**Phase (degrees)**	**Resistance (Ω)**	**Reactance (Ω)**	**ECW (Litres)**	**ICW (Litres)**
10	564.74	-4.79	562.77	47.15	21.03986676	25.74797662

Figure 3.11: Bioimpedance data Excel formatting.

To enable the user to save the data to a particular directory with a particular naming scheme, a file selection dialog is provided. This dialog can be accessed via a save function found on the file menu, which offers options to save the hydration analysis, RC circuit data or R||RC circuit data. In addition to the Excel spreadsheet, a copy of the bioimpedance plots are also saved to the directory with the same naming scheme.

Chapter 4

Hydration Data Analysis

4.1 Initial Bioimpedance Testing

Before controlled hydration testing of the bioimpedance measurement device can take place, regular bioimpedance tests are needed to calibrate the gain ranges to ensure that the device functions as intended. All bioimpedance testing has been performed on subjects whom have been in the supine position[1] for 10 minutes prior to testing, minimising the variation in the body's state between tests by assuming the restful and at ease position [5]. The electrode probes were initially connected on the right hand and left foot as is standard tetra-polar electrode placement. Ambu Blue Sensor Ag/AgCl gel filled electrodes have been used in testing, notably these electrodes offer high conductivity and adhesion to minimise the gap impedance between the electrode and skin. The electrodes were later re-positioned to the ankle and wrist in an attempt to improve the contact between the electrodes and skin.

An initial bioimpedance test performed on a 90.1kg, 192cm tall male can be seen in Figure 4.1. As hoped, the resultant bioimpedance data when plotted in Cole-Cole form (bottom plot) matches the semi-circular shape of the Cole-Cole estimate. The parameters of this plot have been derived from least-squares regression performed on the resistance against frequency data shown in the top plot of the figure. Twenty subsequent tests were performed, one per-minute, in order to determine the variation between measurements in the impedance data. Table 4.1 holds the data for these twenty tests along with the average, standard deviation and uncertainty for each.

There are a number of studies containing bioimpedance data, one such paper [8] makes note of the resulting R_0, R_∞ and R_{50} results from the testing of 27 male subjects. This study found that the average R_0 was 523.43$\pm$68.07, R_∞ was 407.74$\pm$46.26 and R_{50} was 501.67$\pm$55.97. Comparing with the results from the hydration measurement device, R_0 and R_∞ fall within the uncertainties of the study, while R_{50} is on the lower uncertainty border. This is likely explained by the fact that the average weight of this study was 75.42$\pm$14.53kg, and the average height 1.78$\pm$0.054m. The subject tested in this experiment is on the upper boarder of both uncertainty ranges, which is likely to result in bioimpedance parameters on the edges of the study's uncertainty. The average total body water measured by the device, a sum of ECW and ICW, is 50.46$\pm$0.98L. Again, this fits with the literature which states a typical adult male's weight is made up of 58$\pm$8% water [12], which is 52.26$\pm$7.21L for the tested subject.

[1] Subject lying flat on their back facing upwards.

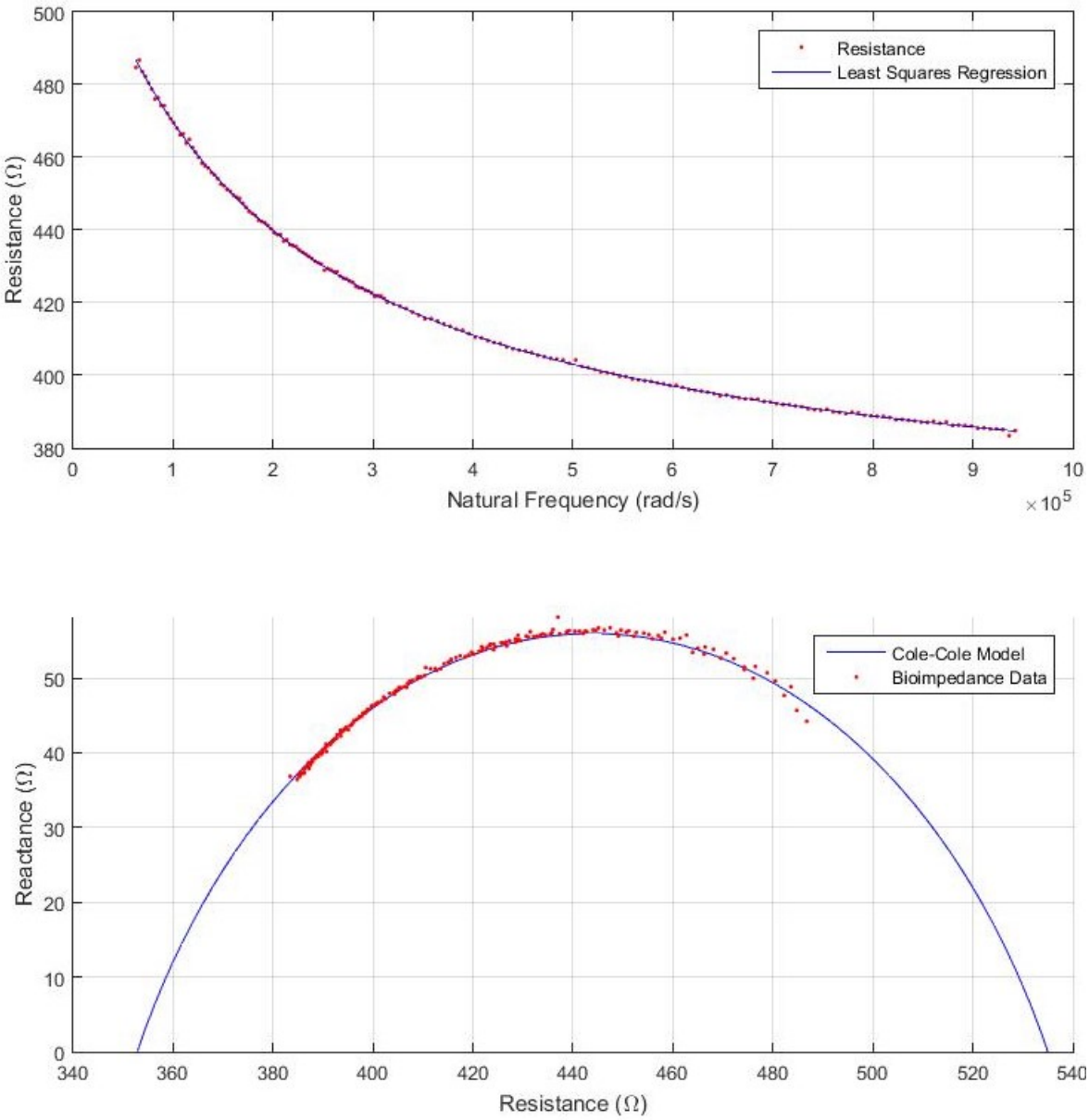


Figure 4.1: A single full body bioimpedance measurement with least-squares regression fit (top) and resultant Cole-Cole estimation (bottom).

Observing the uncertainty and standard deviation in the resistance parameters, it can be said that there is a low amount of perceivable deviation, with the most significant uncertainly being in R_∞ by $\pm 1.35\%$. This is promising as it shows that changes between measurements are more likely influenced by changes in the subjects fluid levels, rather than a lack of precision of the device. As for the hydration parameters themselves, ECW has a very small deviation from the mean with a 0.05L standard deviation and ± 0.10L ($\pm 0.45\%$) uncertainty. However, ICW has a much larger variation between measurements with a standard deviation of 0.32L and an uncertainty of ± 0.88L ($\pm 3.13\%$). The ICW calculation using Equation 1.6 involves the product of R_∞ and R_0. Given that R_∞ has the most significant error, and that R_0 also has an associated error, it therefore follows that there would exist a more significant error in ICW. However, the literature does not make mention of large errors in ICW in comparison

Test	ECW (L)	ICW (L)	R_0 (Ω)	R_∞ (Ω)	R_{50} (Ω)
1	22.31	29.02	532.23	368.71	422.03
2	22.37	28.45	530.00	370.98	422.02
3	22.35	28.62	530.94	370.45	422.31
4	22.38	28.28	529.69	371.82	422.31
5	22.36	28.36	530.60	371.82	422.61
6	22.36	28.25	530.52	372.39	422.76
7	22.31	28.33	532.12	372.74	422.99
8	22.37	27.98	529.97	373.76	423.08
9	22.36	27.97	530.60	374.10	423.56
10	22.37	27.81	529.96	374.71	423.72
11	22.33	28.00	531.49	374.36	423.26
12	22.36	27.79	530.64	375.19	423.68
13	22.34	27.75	531.01	375.64	424.29
14	22.27	28.06	533.80	375.15	424.47
15	22.28	27.99	533.27	375.33	424.2
16	22.31	27.78	532.18	376.02	424.69
17	22.24	28.35	534.92	373.99	425.04
18	22.25	27.83	534.36	376.80	425.74
19	22.23	28.26	535.13	374.60	425.36
20	22.22	27.95	535.53	376.69	424.73
Average	22.32	28.14	531.95	373.76	423.64
Standard Deviation	0.05	0.32	1.86	2.12	1.11
Uncertainty	$\pm$0.10 ($\pm$0.45%)	$\pm$0.88 ($\pm$3.13%)	$\pm$3.58 ($\pm$0.67%)	$\pm$5.06 ($\pm$1.35%)	$\pm$2.10 ($\pm$0.50%)

Table 4.1: Results of 20 subsequent (one measurement per-minute) bioimpedance measurements with the bioimpedance measurement device.

to ECW. It could prove to be an issue, as while $\pm$3.13% is a relatively small error, it may well be too large to detect smaller changes in hydration. This will be explored further in the following section.

4.2 Clinical Hydration Testing

Ideally, the bioimpedance measurement device accuracy and hydration analysis performance would be verified by the means of another bioimpedance analysis (BIA) device or via isotope dilution testing. However, as has been described in the introduction, given that one of the key motivations behind this project are the high cost of such machines and procedures, this direct validation could not be performed even with the inside help of the MD supervising the project. Instead, while the equipment for direct validation testing could not be sourced, he could instead perform an intravenous Saline drip test.

The testing involved the subject (a 175cm, 90.6kg male) lying in the supine position while 2L of 0.9% Saline is administered intravenously at a rate of 33mL/minute. Full body measurements were then taken over a period of 80 minutes, the last 20 minutes of testing were performed after the 2L of Saline had been taken. As detailed in section 1.2.2, adding Saline to

the body increases the number of ions and therefore charge carriers, resulting in an increase in conductivity. It would therefore be expected that as the Saline is absorbed, the overall resistance of the body would decrease. All the fluid enters the extracellular space, meaning the intracellular space should not be influenced by the fluid increase over the testing time period. Within the extracellular space, the fluid will shift with time as the extracellular space is comprised of both the interstitial space and the intravascular fluid [19]. However, this project has not differentiated between these two compartments, and instead takes a total ECW volume, which will not be influenced by this shift in fluid location.

There are a number of factors that can influence BIA measurements, they need to be controlled in order to achieve an accurate result. Position changes and limb arrangement of the subject have a large influence on BIA. As such, movement was kept to a minimum with legs spread and arms positioned parallel to the body to prevent contact which could allow conduction. Conductive objects were also removed from contact of the subject. The final and perhaps most influential aspect is the electrode contacts, care must be taken to ensure sufficient conduction through the elctrode-skin interface.

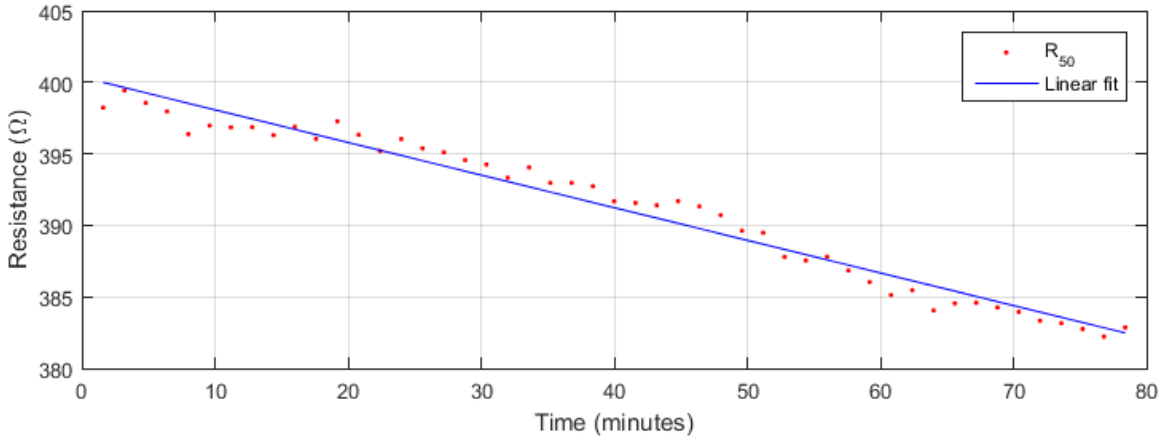


Figure 4.2: R_{50} results from 50 tests of a patient administered 2L of Saline over a period of 80 minutes.

Figure 4.2 shows the R_{50} measurements, the resistance at 50kHz, across the testing period against a linear line of best fit. This particular frequency was chosen due to it being where one would expect conduction through both extracellular and the intracellular spaces [8]. It therefore follows that at this frequency, the measured resistance would of both spaces, giving an indication of the total body resistance. It can be observed that not only does the resistance at this frequency decrease by $17.5 \pm 2.2\Omega$, the relationship is linear as could be expected given that the Saline was administered via a drip at a constant rate of 33mL/minute. This result is promising, as it shows that the device is precise enough to detect the changes in the body's resistance due to an increase in ECW.

Through least-squares regression, the Cole-Cole parameters R_0 and R_∞ can be derived. These measurements are detailed within the top two plots of Figure 4.3. R_0 is often referred to as the extracellular resistance, and is the bioimpedance parameter used to calculate ECW using Equation 1.5. It would therefore be expected that this resistance would decrease with an increase in extracellular water. This is the case, R_0 decreased by 55Ω across the testing period. R_∞ is often referred to as the total-body water resistance, both R_0 and R_∞ are used to calculate the intracellular resistance via Equation 1.9. As intracellular resistance is then used

to calculate ICW using Equation 1.6, and ICW is expected to remain unchanged, R_∞ should decrease by the same 55Ω as R_0. From the observed results, this is not the case, R_∞ decreases by only 7Ω. The results are also lacking precision, the measurements have an uncertainty of $\pm 10\Omega$ about the line of best fit.

The bottom two plots of Figure 4.3 show the resulting ECW and ICW volumes across the testing period. It can be observed that the device measured an initial TBW of $\sim 51L$ (56.6% of body weight), which fits with the literature $58\pm 8\%$ of total body weight. As hoped, an increase of $1.39L$ is observed in ECW. From the residuals of the linear fit to the data, an uncertainty of $\pm 0.42L$ is present within the ECW measurements. The resulting ICW measurements are not so promising. Given that no change should be expected, an observed $-2.09 \pm 2.93L$ change is not desirable. This discrepancy can be explained by R_∞ having high levels of uncertainty, which is likely due to errors caused by the extrapolation of data performed in Cole-Cole analysis. This extrapolation relies on the least-squares regression fit of the higher frequency (>50kHz) measurements and as extrapolation inherently introduces error when lacking data, it is possible that there is not enough high frequency data to get an accurate R_∞. This could be due to the upper frequency limit of 150kHz being too low. The higher frequency measurements are also more heavily distorted by the electrode-skin interface. A second cause of error is therefore a poor electrode connection. As difficulty was encountered in attaining sufficient conduction between the skin-electrode interface with the Ambu Blue electrodes, this is the more cause of the high frequency distortion and error observed in R_∞. Access to larger electrodes with more conductive gel and adhesive tape would help to reduce this issue.

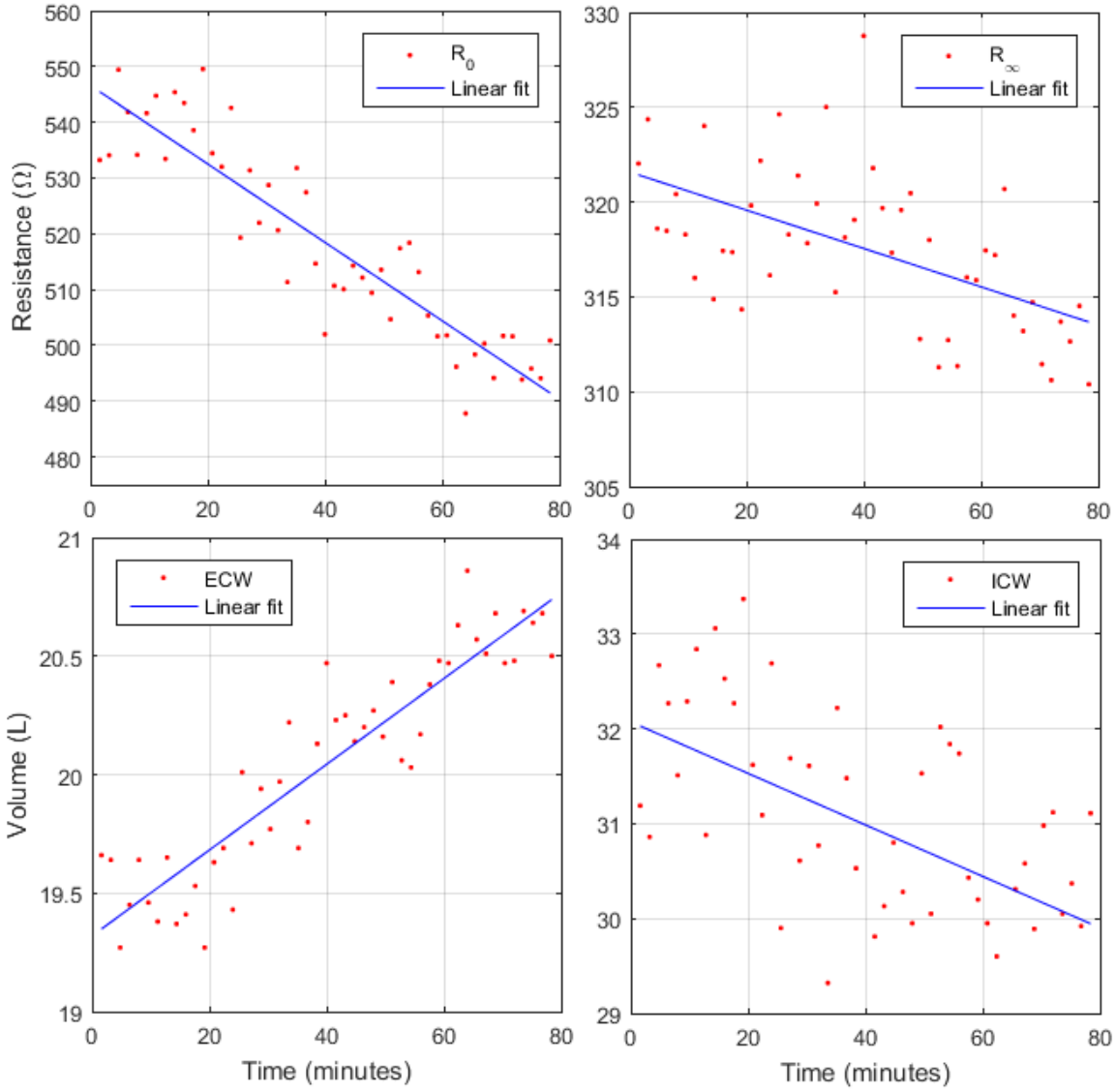


Figure 4.3: R_0 (top left), R_∞ (top right), ECW (bottom left) & ICW (bottom right) results from 50 tests of a patient administered 2L of Saline over a period of 80 minutes.

Chapter 5

Conclusion

A new bioimpedance measurement device has been designed, based off the AD5933 IC and extended with an analog front end to convert the device operation from bipolar to tetrapolar measurements. The device operates across a 10-150kHz bandwidth, and was found to have an accuracy to within 1% and 5% of the true value on passive circuits for impedance and phase measurements respectively. Subsequent bioimpedance measurements determined the Cole-Cole parameters R_0 and R_∞ with $< 1\%$ precision across 20 readings. The resulting Cole-Cole parameters when compared to the literature were found to fall within the ranges of measurements found in previous studies. The device communicates with a PC via a USB cable, the same connection also provides all the necessary power for the device. The impedance data can then be plotted, analysed and saved from a Python-based GUI. This GUI also calculates the ECW and ICW volumes of the patient being tested from the derived Cole-Cole parameters. The device is portable, measuring only 150x90x50mm in dimension, capable of battery operation and data logging via a USB storage device. As such, the specifications determined as necessary to a functional BIA device have been achieved.

Given the high level of accuracy on passive impedances, testing was performed to validate the bioimpedance measurement accuracy used to determine body hydration. Verification of the device performance could not be done directly due to the expense and difficulty acquiring a commercial BIA device. Access to isotope dilution techniques to measure hydration was also not possible due to the cost, even with the inside assistance of the supervising MD. Verification testing was instead performed by administering controlled fluid levels to a test subject. It was found that the R_{50} decreased linearly across the testing as was expected, the low level of uncertainty indicating a high level of precision in the bioimpedance measurements. From extrapolation of the bioimpedance data across the frequency band, R_0 and R_∞ were found and used to calculate ECW and ICW volume. Notably, ECW was found to increase by $1.39 \pm 0.42L$ upon administering 2L of fluids into the the extracellular space. ICW was expected not to change, but was instead found to both decrease and have a high level of uncertainty ($-2.09 \pm 2.93L$). It can be concluded that device provides an accurate measure of bioimpedance at low frequency, and therefore a reasonable ECW estimate. However, the extrapolation of the high frequency data leads to a large error in R_∞ and therefore ICW. Given the device accuracy on passive impedances, this result can largely be attributed to electrode-skin interface issues, which were found to be the main factor in causing poor Cole-Cole fitting in estimating R_∞. Further testing with more suitable electrodes is necessary to fully characterise this error. Despite this issue, the specifications of the device and goals of the project have been met, and as a result the developed bioimpedance measurement device is a successful prototype.

Bibliography

[1] B. Popkin, K. DAnci, and I. Rosenberg, "Water, hydration and health," *Nutrition Reviews*, vol. 68, pp. 439–458, 8 2006.

[2] *Test ID: Specific Gravity, Urine*. [Online]. Available: `http://www.mayomedicallaboratories.com/test-catalog/Clinical+and+Interpretive/9318`.

[3] M. Miller, J. Cosgriff, and G. Forbes, "Bromide space determination using anion-exchange chromatography for measurement of bromide," *Clin Nutr*, vol. 50, pp. 168–171, 1989.

[4] R. Pierson, J. Wan, E. Colt, and P. Neuman, "Body composition measurements in normal man: The potassium, sodium, sulfate and tritium spaces in 58 adults," *J Chronic Dis*, vol. 35, pp. 419–428, 1982.

[5] S. Khalil, M. Mohktar, and F. Ibrahim, "The theory and fundamentals of bioimpedance analysis in clinical status monitoring and diagnosis of diseases," *Sensors*, vol. 14, pp. 10 895–10 928, 2014.

[6] B. J. Nordbotten, *Bioimpedance Measurements Using the Integrated Circuit AD5933*. 2008.

[7] L. Genton, D. Hans, U. Kyle, and C. Pichard, "Dual energy x-ray absorptiometry and body composition: Differences between devices and comparison with reference methods," *Nutrition*, vol. 18, pp. 66–70, 2002.

[8] M. Jaffrin and H. Morel, "Body fluid volumes measurements by impedance: A review of bioimpedance spectroscopy (bis) and bioimpedance analysis (bia) methods," *Medical Engineering & Physics*, vol. 30, pp. 1257–1269, 2008.

[9] L. Rothlingshofer, M. Ulbrich, S. Hahne, and S. Leonhardt, "Monitoring change of body fluid during physical exercise using bioimpedance spectroscopy and finite element simulations," *Journal of Electrical Bioimpedance*, vol. 2, 7985, 2011.

[10] B. Maundy and A. Elwakil, "Extracting the cole-cole impedance model parameters without direct impedance measurements," *Electronics Letters*, vol. 46, 20 2010.

[11] D. Ayllon, "Methods for cole parameter estimation from bioimpedance spectroscopy measurements," *M.E. thesis, University of Boras, Sweden*, 2008.

[12] P. Watson, I. Watson, and R. Batt, "Total body water volumes for adult males and females estimated from simple anthropometric measurements.," *Am J Clin Nutr*, vol. 33, pp. 27–39, 1980.

[13] U. Moissl, P. Wabel, P. Chamney, I. Bosaeus, N. Levin, A. Bosy-Westphal, O. Korth, M. Muller, L. Ellegard, V. Malmros, C. Kaitwatcharachai, M. Kuhlmann, F. Zhu, and N. Fuller, "Body fluid volume determination via body composition spectroscopy in health and disease.," *PHYSIOLOGICAL MEASUREMENT*, vol. 27, 921933, 2006.

[14] AnalogDevices, *AD5933/AD5934 - Converter, Network Analyzer Data Sheet*.

[15] C. Margo, J. Katrib, M. Nadi, and A. Rouane, "A four-electrode low frequency impedance spectroscopy measurement system using the ad5933 measurement chip," *Physiological Measurement*, vol. 34, pp. 391–405, 2013.

[16] D. Sheingold, "Impedance & admittance transformations," *The Lightning Empiricist*, vol. 12, pp. 1–8, 1964.

[17] ARM, *ARM7 Processor Family*. [Online]. Available: http://www.arm.com/products/processors/classic/arm7/index.php?tab=Why+ARM7.

[18] Microchip, *MCP4661 digital potentiometer data sheet*. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/22107B.pdf.

[19] A. Yartsev, "Response to 1l of normal saline," [Online]. Available: http://www.derangedphysiology.com/main/core-topics-intensive-care/manipulation-fluids-and-electrolytes/Chapter%5C%202.3.3/response-1l-normal-saline.

Appendix A

Circuit PCBs & Schematics

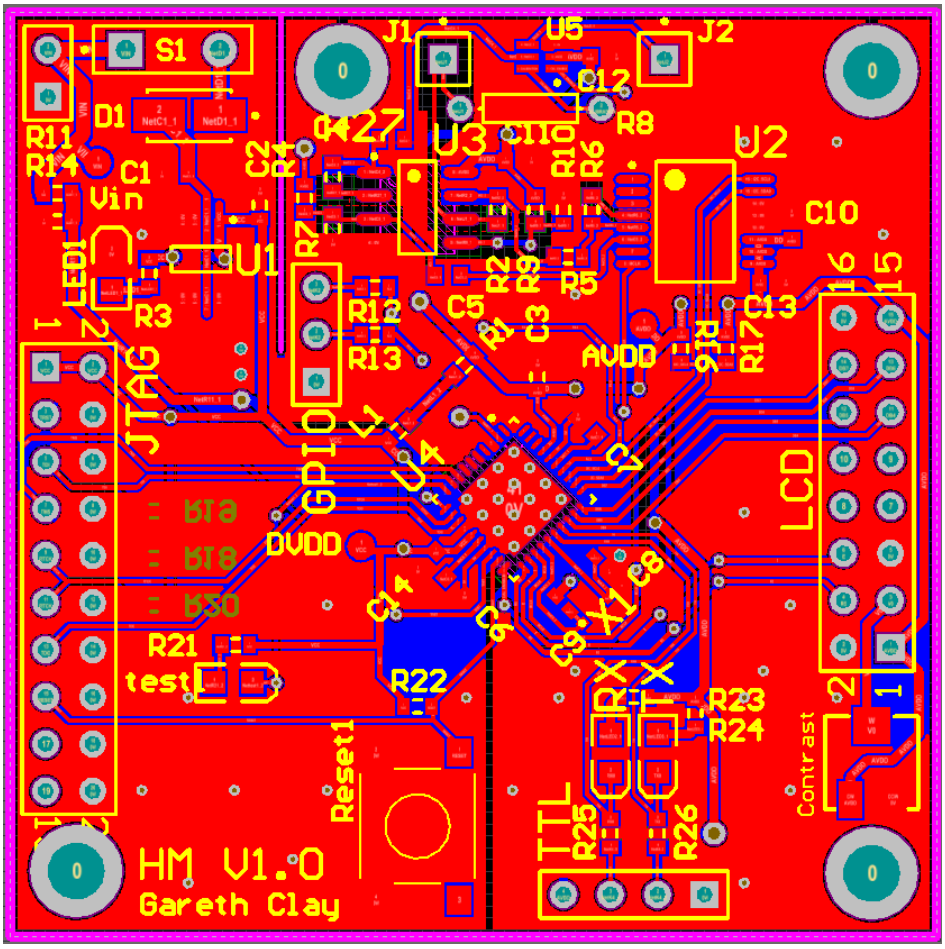


Figure A.1: Revison 1 printed circuit board.

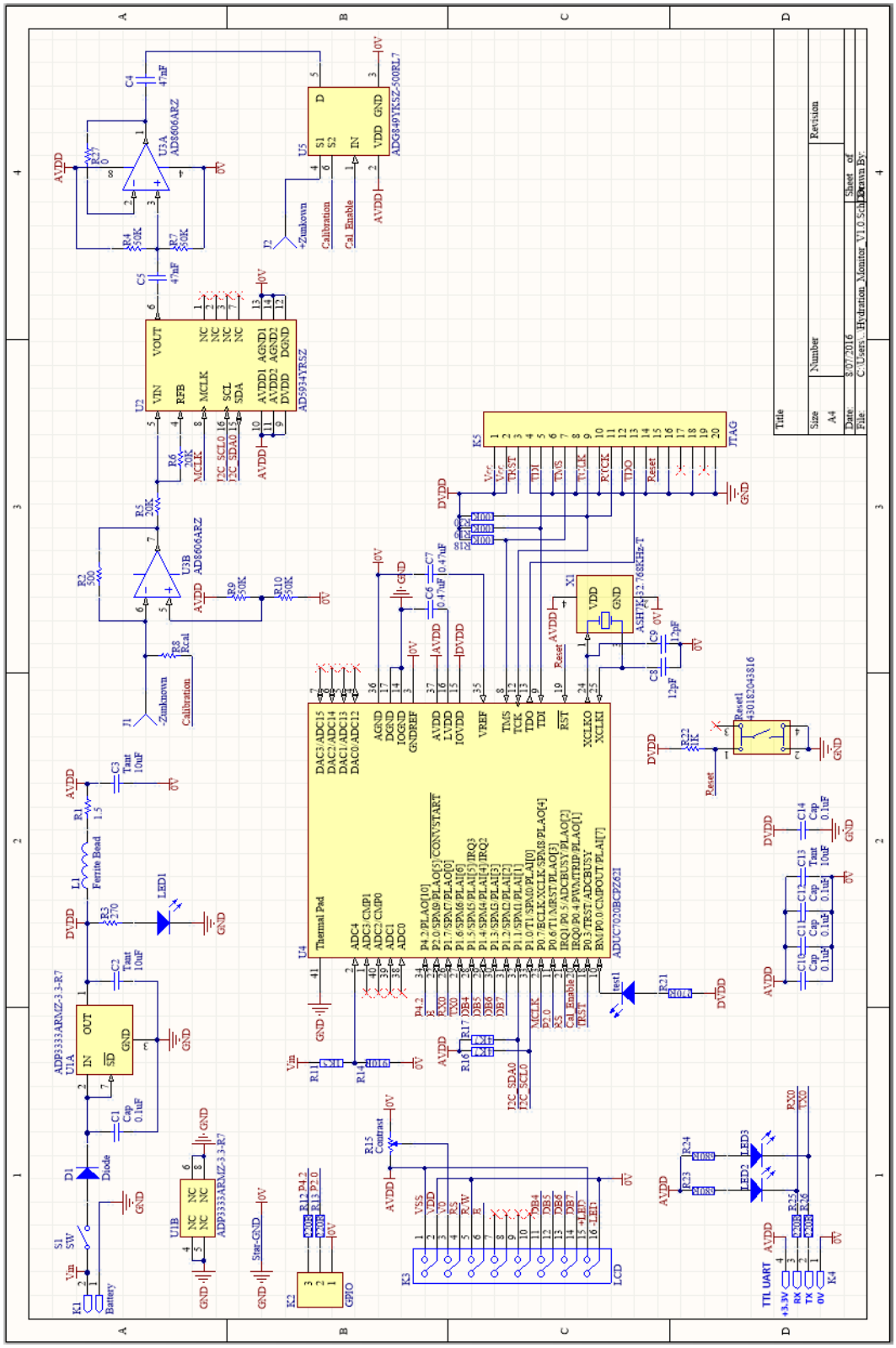


Figure A.2: Revision 1 circuit schematic.

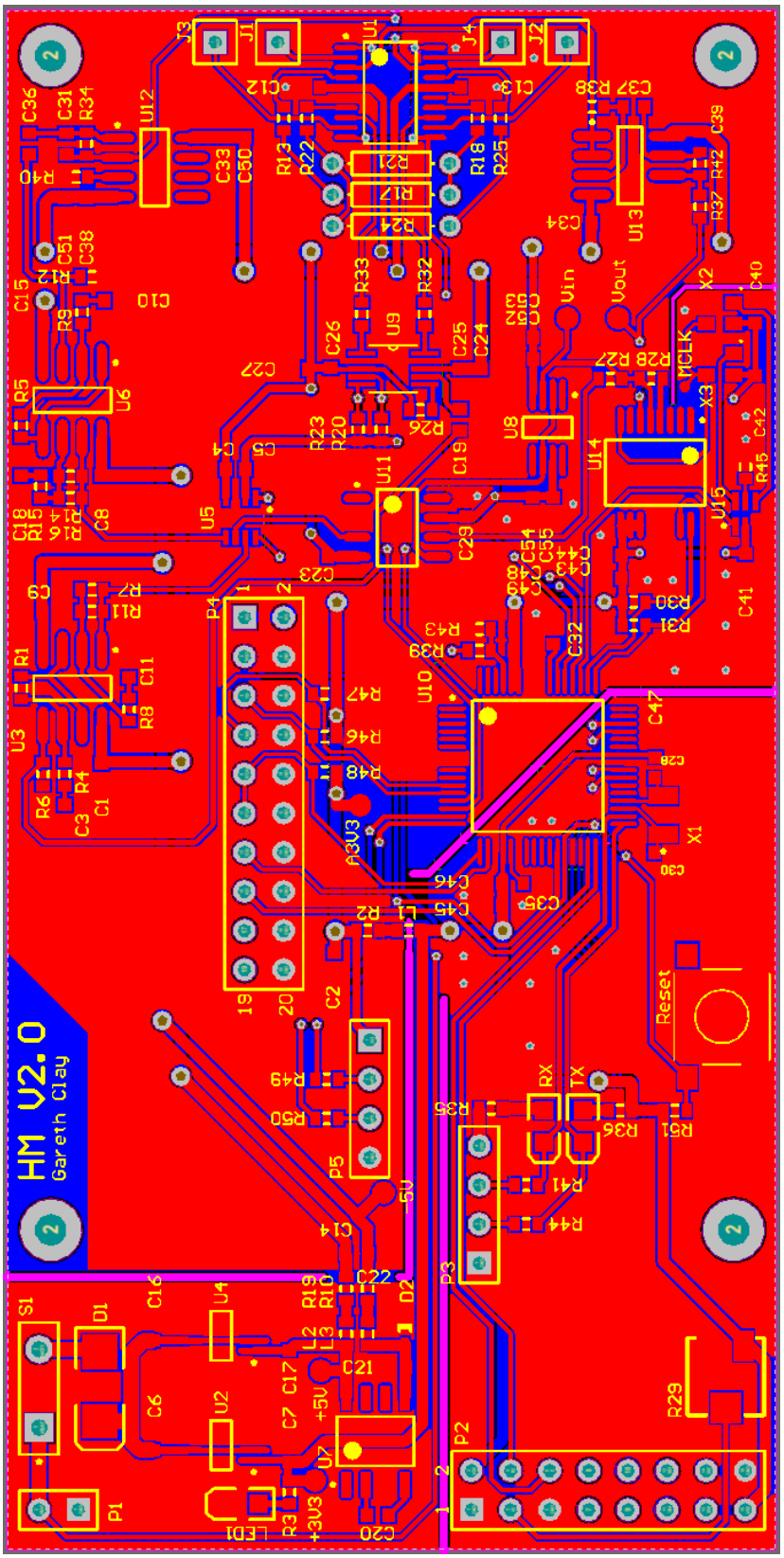


Figure A.3: Revision 2 printed circuit board.

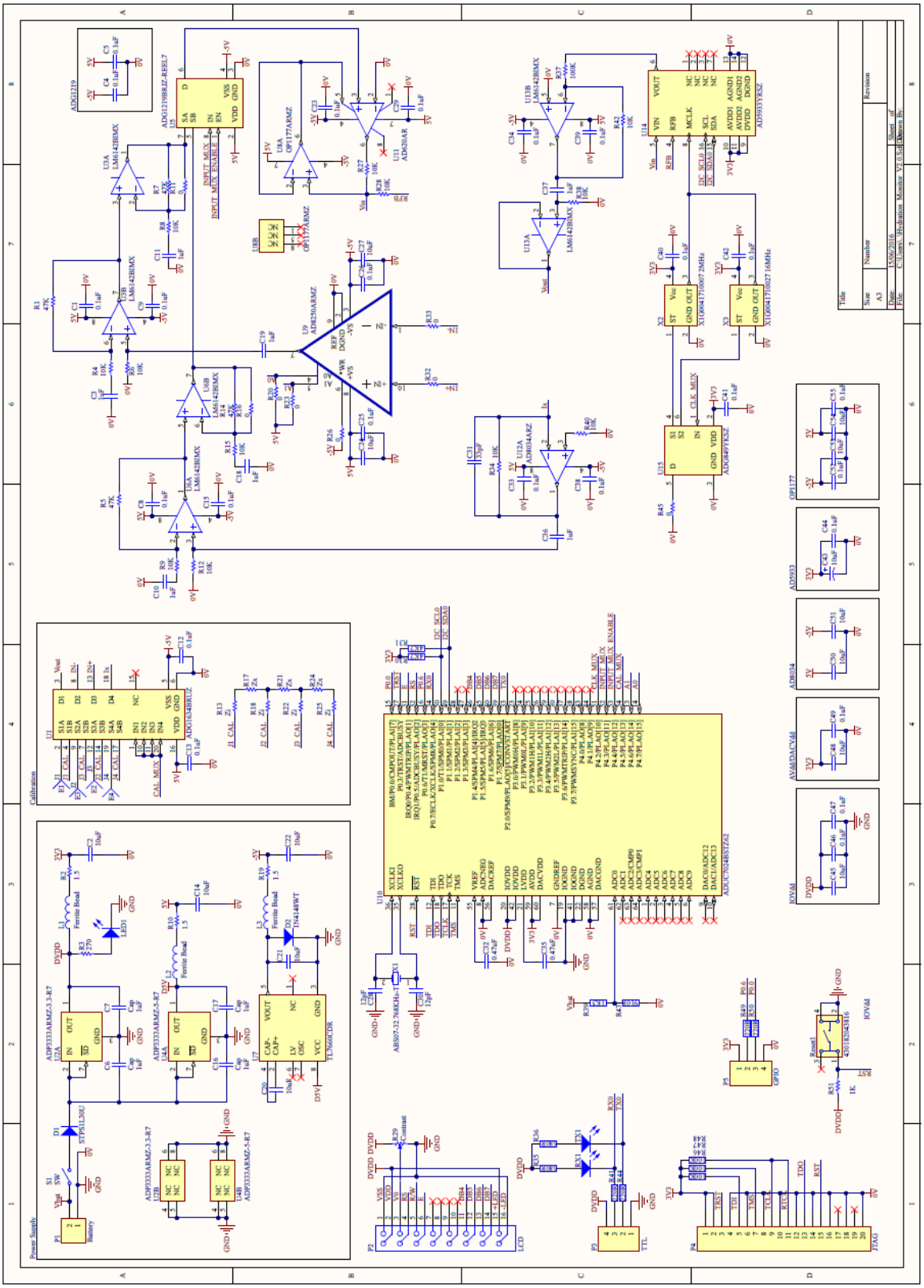


Figure A.4: Revision 2 circuit schematic.

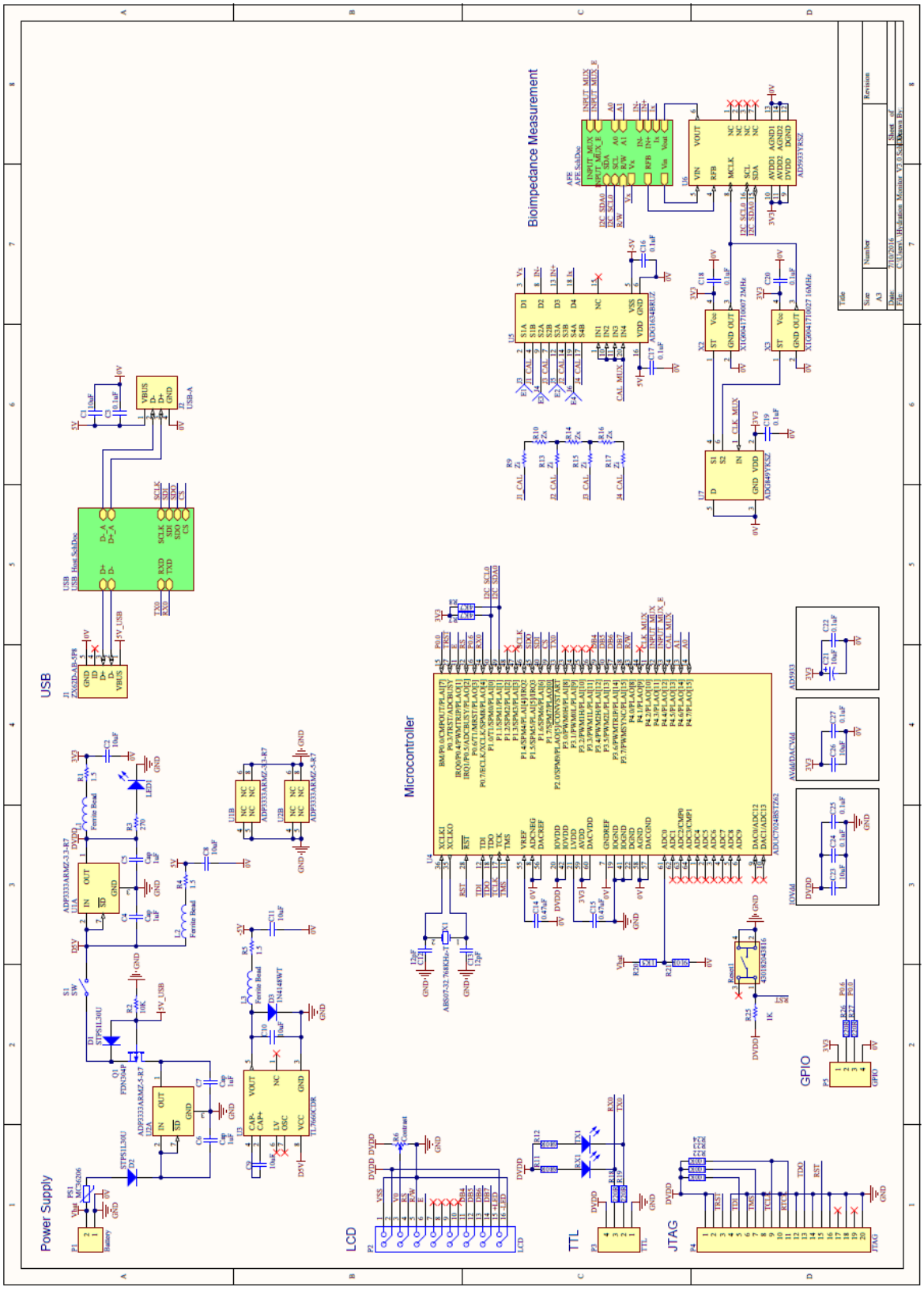


Figure A.5: Revision 3 main sheet circuit schematic.

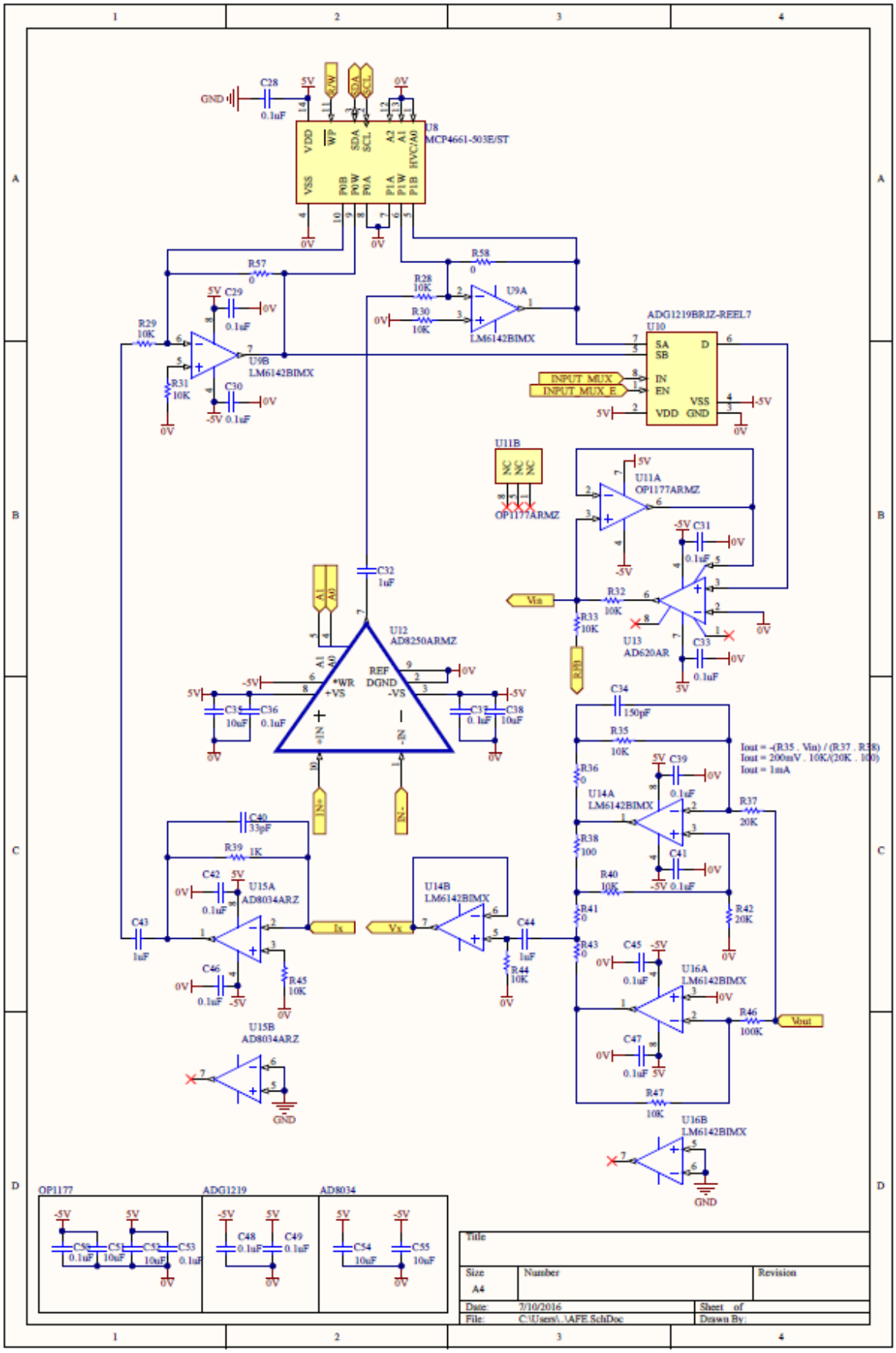


Figure A.6: Revision 3 AFE circuit schematic.

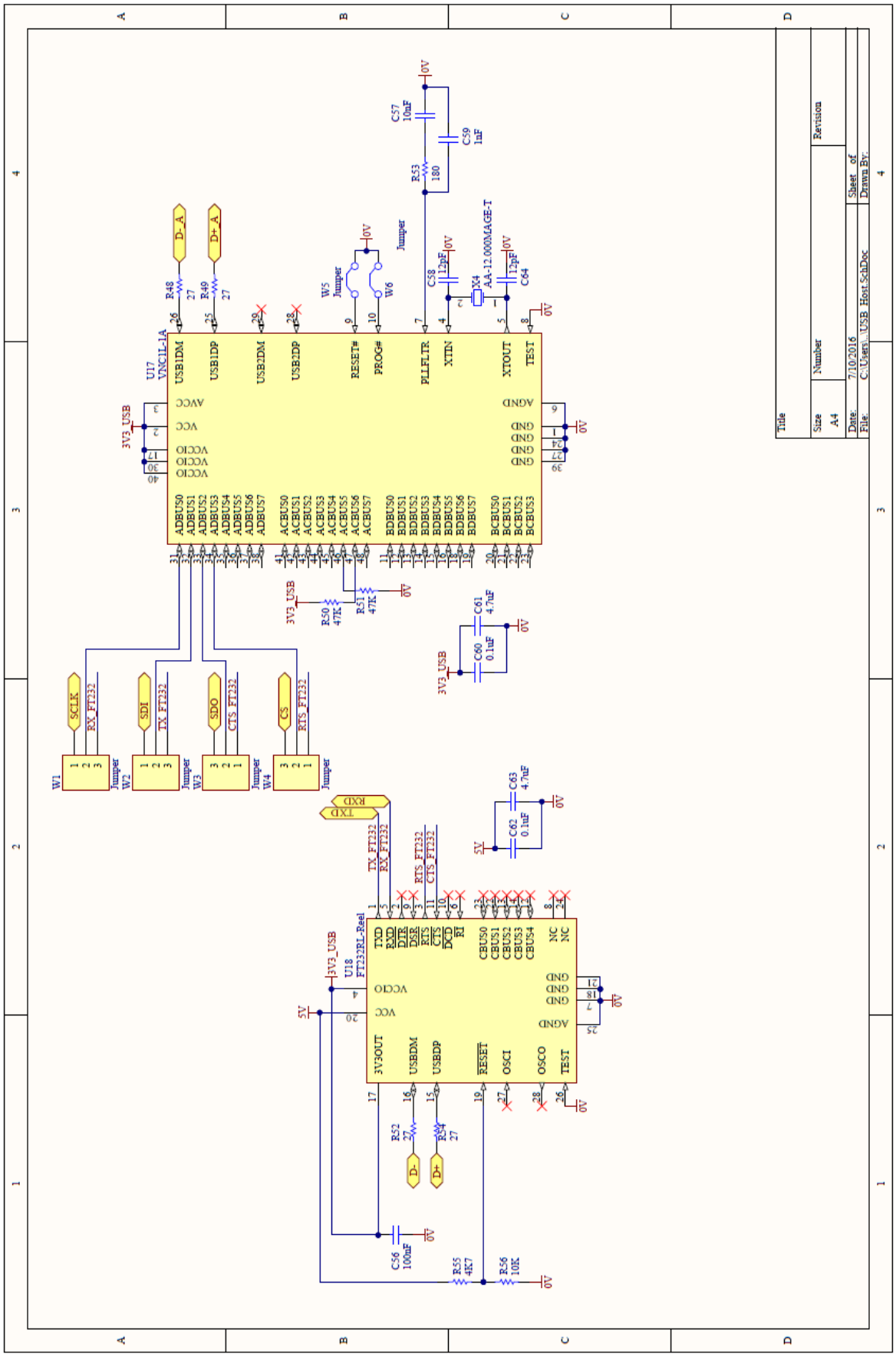


Figure A.7: Revision 3 USB circuit schematic.

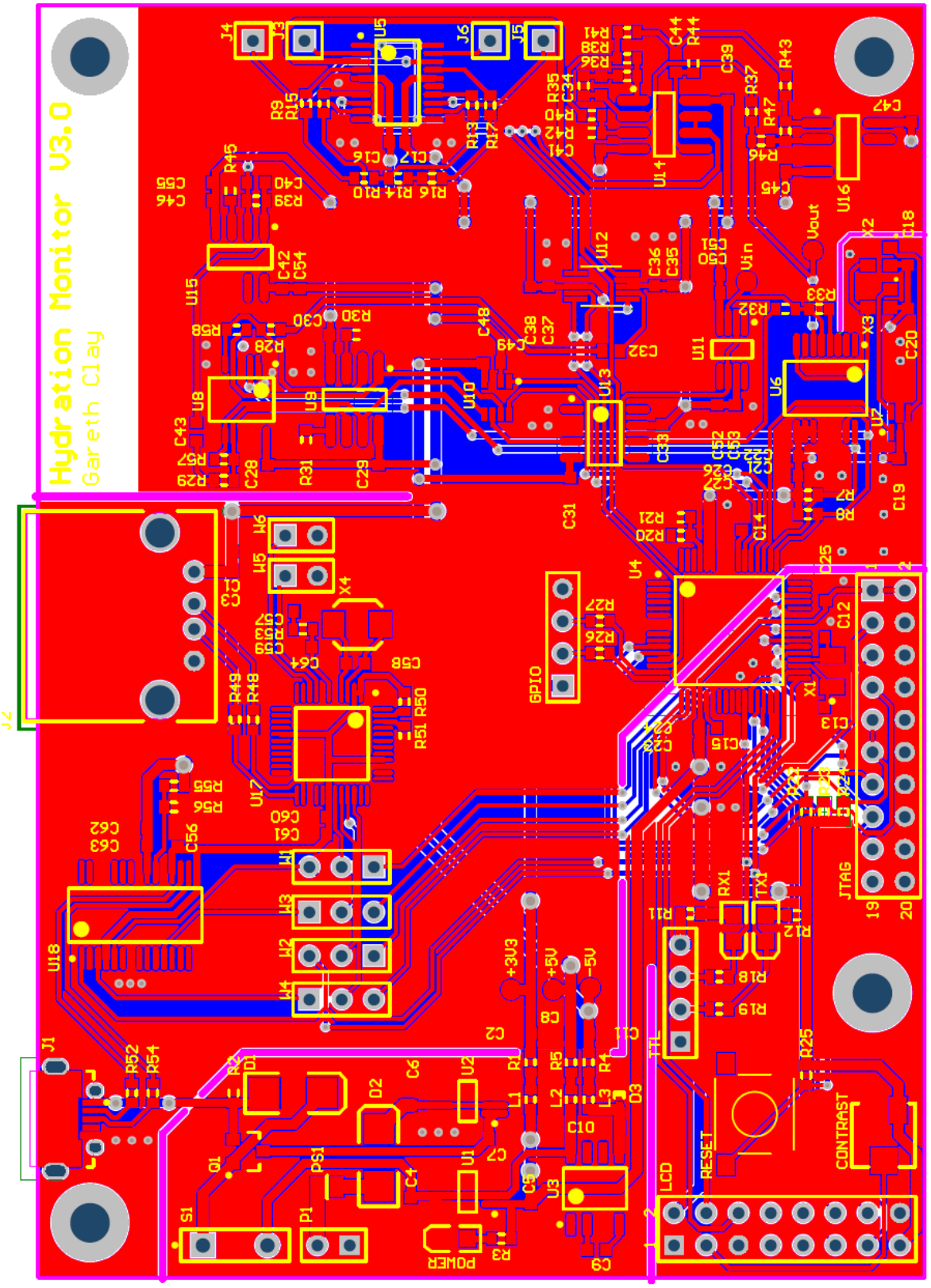


Figure A.8: Revision 3 printed circuit board.

Appendix B

Embedded Software

```
#include "ADuC702x.h"
#include <stdio.h>

#include "util.h"
#include "lcd.h"
#include "i2c.h"
#include "AD5933.h"
#include "uart.h"
#include "MCP4661.h"

typedef enum state
{
  gain_config,
  high_freq_config,
  high_freq_measurement,
  low_freq_config,
  low_freq_measurement
}state;

typedef enum gain_state
{
  current_gain,
  voltage_gain,
  start
}gain_state;

#define PI 3.14159265358979323846

unsigned int low_start_freq = 10000;      // 10000
unsigned int low_increment_freq = 500;  // 1000
unsigned int low_num_increments = 80;  // 141
unsigned int low_settling_time = 15;

unsigned int start_freq = 51000;      // 10000
unsigned int increment_freq = 1000;   // 1000
unsigned int num_increments = 99;  // 141
unsigned int settling_time = 15;

double avg_num = 5;

double test_impedance = 389.1; // Configuration impedance used for calibration

state current_state = gain_config;
gain_state current_gain_state = start;
```

```
45
int voltage_res = 5000; // Passive components: WP1 = Voltage Signal: 10000
47 int current_res = 2000; //                      WP0 = Current Signal: 4000

49 void draw_menu()
{
51   clear_lcd();

53   move_cursor(0,0);
   WriteLCD("Menu:");
55
   move_cursor(5,1);
57   WriteLCD("Current (x");
   WriteDouble(current_res/10000.00);
59   WriteLCD(")");

61   move_cursor(5,2);
   WriteLCD("Voltage (x");
63   WriteDouble(voltage_res/10000.00);
   WriteLCD(")");
65
   move_cursor(5,3);
67   WriteLCD("OK");

69   if(current_gain_state == current_gain)
     move_cursor(3,1);
71   else if(current_gain_state == voltage_gain)
     move_cursor(3,2);
73   else
     move_cursor(3,3);
75
   WriteData(0x7E); // Arrow
77   delay_ms(200);
}
79
void current_menu(unsigned char state)
81 {
   clear_lcd();
83   move_cursor(0,0);
   WriteLCD("Current Gain:");
85
   move_cursor(5,1);
87   WriteLCD("x");
   WriteDouble(current_res/10000.00);
89
   move_cursor(5,2);
91   WriteLCD("OK");

93   if(state)
     move_cursor(3,2);
95   else
     move_cursor(3,1);
97
   WriteData(0x7E); // Arrow
99   delay_ms(200);
}
101
void voltage_menu(unsigned char state)
103 {
   clear_lcd();
105  move_cursor(0,0);
   WriteLCD("Voltage Gain:");
```

```
107
    move_cursor(5,1);
109 WriteLCD("x");
    WriteDouble(voltage_res/10000.00);
111
    move_cursor(5,2);
113 WriteLCD("OK");

115 if(state)
      move_cursor(3,2);
117 else
      move_cursor(3,1);
119
    WriteData(0x7E); // Arrow
121 delay_ms(200);
}
123
void select_current_gain()
125 {
    typedef enum currentGain_state
127 {
      increment,
129   OK
    }currentGain_state;
131
    currentGain_state cur_state;
133 int exit = 0;

135 cur_state = increment;

137 delay_ms(100);
    while(!exit)
139 {
      switch(cur_state)
141   {
        case increment:
143       current_menu(0);

145       if(readButton2())
            cur_state = OK;
147
          if(readButton1())
149       {
            current_res+= 500;
151
            if(current_res>10000)
153           current_res = 500;
          }
155       break;

157     case OK:
          current_menu(1);
159
          if(readButton2())
161         cur_state = increment;

163       if(readButton1())
            exit = 1;
165
          break;
167   }
    }
```

```
169
    delay_ms(100);
171 }

173 void select_voltage_gain()
    {
175   typedef enum voltageGain_state
      {
177     increment,
        OK
179   }voltageGain_state;

181   voltageGain_state vol_state;
      int exit = 0;
183
      vol_state = increment;
185
      delay_ms(100);
187   while(!exit)
      {
189     switch(vol_state)
        {
191       case increment:
            voltage_menu(0);
193
            if(readButton2())
195           vol_state = OK;

197         if(readButton1())
            {
199           voltage_res+= 500;

201           if(voltage_res>10000)
                voltage_res = 500;
203         }
            break;
205
          case OK:
207         voltage_menu(1);

209         if(readButton2())
              vol_state = increment;
211
            if(readButton1())
213           exit = 1;

215         break;
        }
217   }

219   delay_ms(100);
    }
221
    int main (void)  {
223   double Z, theta, gain_factor, R, X, system_phase;
      unsigned char status = 0;
225   int i = 0;
      unsigned char x = 0;
227   unsigned char exit = 0;

229   IRQ = my_IRQ_Handler; //  Specify Interrupt Service Rountine
      init();
```

```
231   InitLCD();
      delay_ms(1000);
233   clear_lcd();
      move_cursor(0,0);
235   InitLCD();
      InitI2C0();
237
      set_voltageGain(voltage_Gain_x2); // Differential voltage gain x2
239
      SETBIT(GP4DAT, 16); // Enable non-volatile memory writing
241                       // for MCP4661 digital pot
243   ConfigUART9600();
245   while(1)
      {
247     switch(current_state)
        {
249       case gain_config:
            while(1)
            {
251           switch(current_gain_state)
              {
253             case current_gain:
                  draw_menu();
255
                  if(readButton2())
257                 current_gain_state = voltage_gain;
259
                  if(readButton1())
261                 select_current_gain();
263
                  break;
265
                case voltage_gain:
                  draw_menu();
267
                  if(readButton2())
269                 current_gain_state = start;
271
                  if(readButton1())
                    select_voltage_gain();
273
                  break;
275
                case start:
                  draw_menu();
277
                  if(readButton2())
279                 current_gain_state = current_gain;
                  if(readButton1())
281                 exit = 1;
283
                  break;
285           }
              if(exit)
287             break;
            }
289         current_state = low_freq_config;
            exit = 0;
291         break;
```

```
293       /*************Configuration*************/
          case low_freq_config:
295           single_write_WP(1, voltage_res);
              single_write_WP(0, current_res);
297
              AD5933_Reset();
299           AD5933_SetSystemClk(AD5933_CONTROL_EXT_SYSCLK, AD5933_16MHZ_SYS_CLK);
              AD5933_ConfigSweep(low_start_freq, low_increment_freq,
      low_num_increments, low_settling_time); // Start freq, freq incr, num incr,
      settling time
301           AD5933_SetRangeAndGain(AD5933_RANGE_2000mVpp, AD5933_GAIN_X1);

303           AD5933_StartSweep();
              current_state = low_freq_measurement;
305
              uart_writeString("Config done\n");
307           break;

309       case high_freq_config:
              //single_write_WP(1, voltage_res);
311           //single_write_WP(0, current_res);

313           AD5933_Reset();
              AD5933_SetSystemClk(AD5933_CONTROL_EXT_SYSCLK, AD5933_16MHZ_SYS_CLK);
315           AD5933_ConfigSweep(start_freq, increment_freq, num_increments,
      settling_time); // Start freq, freq incr, num incr, settling time
              AD5933_SetRangeAndGain(AD5933_RANGE_2000mVpp, AD5933_GAIN_X1);
317
              AD5933_StartSweep();
319           current_state = high_freq_measurement;

321           //uart_writeString("Config done\n");
              break;
323
          /************* Frequency Measurements*************/
325       case low_freq_measurement:
              i++;
327           clear_lcd();
              WriteLCD("Performing sweep...");
329
              /************ System Calibration ************/
331           set_test_load(1); // Set impedance measuring to test load
              delay_ms(1);
333
              gain_factor = system_phase = 0;
335           for(x = 0; x < avg_num; x++) // 5 measurements and average
              {
337             gain_factor += AD5933_CalculateGainFactor(test_impedance,
      AD5933_FUNCTION_REPEAT_FREQ); // Calculate gain factor at each frequency in
      the step
                system_phase += AD5933_CalculatePhase(AD5933_FUNCTION_REPEAT_FREQ);
339           }
              gain_factor = gain_factor/avg_num;
341           system_phase = system_phase/avg_num;

343           set_test_load(0); // Set impedance measuring to electrodes
              delay_ms(1);
345
              /************ Impedance & Phase Measurements ************/
347           Z = theta = 0;
              for(x = 0; x < avg_num; x++) // 5 measurements and average
349           {
```

```
          theta += AD5933_CalculatePhase(AD5933_FUNCTION_REPEAT_FREQ);
          Z += AD5933_CalculateImpedance(gain_factor,
AD5933_FUNCTION_REPEAT_FREQ);
        }
        Z = Z/avg_num;
        theta = theta/avg_num - system_phase;

        R = calculate_resistance(Z, (theta)*(PI/180.0));
        X = calculate_reactance(Z, (theta)*(PI/180.0));
        /************ Reporting Data ************/

        uart_writeDouble((((i-1)*low_increment_freq)+low_start_freq)/1000.0);
        uart_writeString("  ");
        uart_writeDouble(Z);
        uart_writeString("  ");
        uart_writeDouble(theta);
        uart_writeString("  ");
        uart_writeDouble(calculate_resistance(Z, (theta)*(PI/180.0))); // Need
to convert phase from degrees to radians
        uart_writeString("  ");
        uart_writeDouble(calculate_reactance(Z, (theta)*(PI/180.0)));
        uart_writeChar('\n');

        status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1); // Check if
sweep is done
        if((status & AD5933_STAT_SWEEP_DONE))
        {
          i=0;
          current_state = high_freq_config;
          break;
        }

        // Increment to the next frequency
        AD5933_CalculateImpedance(gain_factor, AD5933_FUNCTION_INC_FREQ);
        break;

      case high_freq_measurement:
        i++;
        clear_lcd();
        WriteLCD("Performing sweep...");

        /************ System Calibration ************/
        set_test_load(1); // Set impedance measuring to test load
        delay_ms(1);

        gain_factor = system_phase = 0;
        for(x = 0; x < avg_num; x++) // 5 measurements and average
        {
          gain_factor += AD5933_CalculateGainFactor(test_impedance,
AD5933_FUNCTION_REPEAT_FREQ); // Calculate gain factor at each frequency in
the step
          system_phase += AD5933_CalculatePhase(AD5933_FUNCTION_REPEAT_FREQ);
        }
        gain_factor = gain_factor/avg_num;
        system_phase = system_phase/avg_num;

        set_test_load(0); // Set impedance measuring to electrodes
        delay_ms(1);

        /************ Impedance & Phase Measurements ************/
        Z = theta = 0;
        for(x = 0; x < avg_num; x++) // 5 measurements and average
```

```
407             {
                  theta += AD5933_CalculatePhase(AD5933_FUNCTION_REPEAT_FREQ);
409               Z += AD5933_CalculateImpedance(gain_factor,
    AD5933_FUNCTION_REPEAT_FREQ);
                }
411             Z = Z/avg_num;
                theta = theta/avg_num - system_phase;
413
                R = calculate_resistance(Z, (theta)*(PI/180.0));
415             X = calculate_reactance(Z, (theta)*(PI/180.0));
                /************ Reporting Data ************/
417
                uart_writeDouble((((i-1)*increment_freq)+start_freq)/1000.0);
419             uart_writeString("  ");
                uart_writeDouble(Z);
421             uart_writeString("  ");
                uart_writeDouble(theta);
423             uart_writeString("  ");
                uart_writeDouble(calculate_resistance(Z, (theta)*(PI/180.0))); // Need
    to convert phase from degrees to radians
425             uart_writeString("  ");
                uart_writeDouble(calculate_reactance(Z, (theta)*(PI/180.0)));
427             uart_writeChar('\n');

429             status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1); // Check if
    sweep is done
                if((status & AD5933_STAT_SWEEP_DONE))
431             {
                  i=0;
433               current_state = gain_config;
                  break;
435             }

437             // Increment to the next frequency
                AD5933_CalculateImpedance(gain_factor, AD5933_FUNCTION_INC_FREQ);
439             break;
              }
441   }
    return 0;
443 }
```

Code/main.c

```
1  /*****************************************************************************/
   /***************************** Include Files *********************************/
3  /*****************************************************************************/
   #include "AD5933.h"
5  #include "ADuC702x.h"
   #include "util.h"
7  #include "i2C.h"
   #include "lcd.h"
9  #include <math.h>

11 #define PI 3.14159265358979323846

13 /*****************************************************************************/
   /************************** Constants Definitions ****************************/
15 /*****************************************************************************/
   const long POW_2_27 = 134217728ul;      // 2 to the power of 27
17
   /*****************************************************************************/
19 /************************ Variables Definitions ******************************/
```

```
/*****************************************************************************/
unsigned long currentSysClk       = AD5933_16MHZ_SYS_CLK;
unsigned char currentClockSource = AD5933_CONTROL_INT_SYSCLK;
unsigned char currentGain         = AD5933_GAIN_X1;
unsigned char currentRange        = AD5933_RANGE_2000mVpp;


int convert(long data)  // Converts 2's compliment to decimal
{
  int nativeInt = 0;
  int negative = (data & (1 << 15)) != 0;

  if (negative)
    nativeInt = data | ~((1 << 16) - 1);
  else
    nativeInt = data;

  return nativeInt;
}

/*****************************************************************************//**
 * @brief Writes data into a register.
 *
 * @param registerAddress - Address of the register.
 * @param registerValue - Data value to write.
 * @param bytesNumber - Number of bytes.
 *
 * @return None.
*****************************************************************************/
void AD5933_SetRegisterValue(unsigned char registerAddress,
                             unsigned long registerValue,
                             unsigned char bytesNumber)
{
    unsigned char byte         = 0;
    unsigned char writeData[2] = {0, 0};

    for(byte = 0;byte < bytesNumber;byte++)
    {
        writeData[0] = registerAddress + bytesNumber - byte - 1;
        writeData[1] = (unsigned char)((registerValue >> (byte * 8)) & 0xFF);
        I2C_write_bytes(AD5933_ADDRESS, writeData, 2);
    }
}

/*****************************************************************************//**
 * @brief Reads the value of a register.
 *
 * @param registerAddress - Address of the register.
 * @param bytesNumber - Number of bytes.
 *
 * @return registerValue - Value of the register.
*****************************************************************************/

unsigned long AD5933_GetRegisterValue(unsigned char registerAddress,
                                      unsigned char bytesNumber)
{
    unsigned long registerValue = 0;
    unsigned char byte          = 0;
    unsigned char writeData[2]  = {0, 0};
    unsigned char readData[2]   = {0, 0};

    for(byte = 0;byte < bytesNumber;byte ++)
```

```
    {
        /* Set the register pointer. */
        writeData[0] = AD5933_ADDR_POINTER;
        writeData[1] = registerAddress + byte;
        I2C_write_bytes(AD5933_ADDRESS, writeData, 2);

        /* Read Register Data. */
        readData[0] = 0xFF;
        I2C_read_bytes(AD5933_ADDRESS, readData, 1);
        registerValue = registerValue << 8;
        registerValue += readData[0];
    }
    return registerValue;
}


/*****************************************************************************//**
 * @brief Initializes the communication peripheral.
 *
 * @return status - The result of the initialization procedure.
 *                  Example: 0x0 - I2C peripheral was not initialized.
 *                           0x1 - I2C peripheral was initialized.
*****************************************************************************/
void AD5933_Init(void)
{
    InitI2C0();
}

/*****************************************************************************//**
 * @brief Resets the device.
 *
 * @return None.
*****************************************************************************/
void AD5933_Reset(void)
{
    AD5933_SetRegisterValue(AD5933_REG_CONTROL_LB,
                            AD5933_CONTROL_RESET | currentClockSource,
                            1);
}


/*****************************************************************************//**
 * @brief Configures the sweep parameters: Start frequency, Frequency increment
 *        and Number of increments.
 *
 * @param startFreq - Start frequency;
 * @param incFreq - Frequency increment;
 * @param incNum - Number of increments.
 *
 * @return None.
*****************************************************************************/
void AD5933_ConfigSweep(unsigned long  startFreq,
                        unsigned long  incFreq,
                        unsigned short incNum,
                        unsigned short settlingTime)
{
    unsigned long  startFreqReg = 0;
    unsigned long  incFreqReg   = 0;
    unsigned short incNumReg    = 0;

    /* Ensure that incNum is a valid data. */
    if(incNum > AD5933_MAX_INC_NUM)
```

```
    {
145         incNumReg = AD5933_MAX_INC_NUM;
    }
147 else
    {
149         incNumReg = incNum;
    }
151
    /* Convert users start frequency to binary code. */
153 startFreqReg = (unsigned long)(((double)startFreq) / (currentSysClk/4) *
                                   POW_2_27);
155
    /* Convert users increment frequency to binary code. */
157 incFreqReg = (unsigned long)(((double)incFreq) / (currentSysClk/4) *
                                   POW_2_27);
159


161
    /* Configure the device with the sweep parameters. */
163 AD5933_SetRegisterValue(AD5933_REG_FREQ_START,
                            startFreqReg,
165                         3);
    AD5933_SetRegisterValue(AD5933_REG_FREQ_INC,
167                         incFreqReg,
                            3);
169
    AD5933_SetRegisterValue(AD5933_REG_SETTLING_CYCLES, settlingTime, 2); //
    Settling cycle to 15
171
    AD5933_SetRegisterValue(AD5933_REG_INC_NUM,
173                         incNumReg,
                            2);
175 }

177 /*****************************************************************************//**
 * @brief Starts the sweep operation.
179 *
 * @return None.
181 *****************************************************************************/
void AD5933_StartSweep(void)
183 {
    unsigned char status = 0;
185
    AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
187                         AD5933_CONTROL_FUNCTION(AD5933_FUNCTION_POWER_DOWN) |
                            AD5933_CONTROL_RANGE(currentRange) |
189                         AD5933_CONTROL_PGA_GAIN(currentGain),
                            1);
191
    AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
193                           AD5933_CONTROL_FUNCTION(AD5933_FUNCTION_STANDBY) |
                              AD5933_CONTROL_RANGE(currentRange) |
195                           AD5933_CONTROL_PGA_GAIN(currentGain),
                              1);
197
    AD5933_Reset();
199 AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
                        AD5933_CONTROL_FUNCTION(AD5933_FUNCTION_INIT_START_FREQ)|
201                     AD5933_CONTROL_RANGE(currentRange) |
                        AD5933_CONTROL_PGA_GAIN(currentGain),
203                     1);
    delay_ms(5);
```

```
205
    AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
207                         AD5933_CONTROL_FUNCTION(AD5933_FUNCTION_START_SWEEP) |
                            AD5933_CONTROL_RANGE(currentRange) |
209                         AD5933_CONTROL_PGA_GAIN(currentGain),
                            1);
211

213     status = 0;
        while((status & AD5933_STAT_DATA_VALID) == 0)
215     {
          status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1);
217     };

219 }

221
/*****************************************************************************//**
223  * @brief Reads the real and the imaginary data and calculates the Gain Factor.
     *
225  * @param calibrationImpedance - The calibration impedance value.
     * @param freqFunction        - Frequency function.
227  *                              Example: AD5933_FUNCTION_INC_FREQ - Increment
                                             freq.;
229  *                                       AD5933_FUNCTION_REPEAT_FREQ - Repeat
                                             freq..
231  *
     * @return gainFactor         - Calculated gain factor.
233 *****************************************************************************/
double AD5933_CalculateGainFactor(double calibrationImpedance,
235                               unsigned char freqFunction)
{
237     double        gainFactor = 0;
        double        voltage_magnitude  = 0;
239     double        current_magnitude  = 0;

241     /* Calculate Voltage Magnitude */
        voltage_magnitude = AD5933_CalculateVoltage(freqFunction);
243
        /* Calculate Current Magnitude */
245     current_magnitude = AD5933_CalculateCurrent(AD5933_FUNCTION_REPEAT_FREQ);

247     /* Calculate gain factor */
        gainFactor = calibrationImpedance*(current_magnitude/voltage_magnitude);
249
        return gainFactor;
251 }

253
/*****************************************************************************//**
255  * @brief Reads the real and the imaginary data and calculates the voltage
        magnitude.
     *
257  * @param freqFunction        - Frequency function.
     *                              Example: AD5933_FUNCTION_INC_FREQ - Increment
259                                          freq.;
     *                                       AD5933_FUNCTION_REPEAT_FREQ - Repeat
261                                          freq..
     *
263  * @return voltage_magnitude          - Calculated voltage magnitude.
*****************************************************************************/
265 double AD5933_CalculateVoltage(unsigned char freqFunction)
```

```
{
  double         voltage_magnitude  = 0;
  signed short   realData    = 0;
  signed short   imagData    = 0;
  unsigned char  status      = 0;

  set_input_mux(1);

  AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
                          AD5933_CONTROL_FUNCTION(freqFunction) |
                          AD5933_CONTROL_RANGE(currentRange) |
                          AD5933_CONTROL_PGA_GAIN(currentGain),
                          1);
    status = 0;
    while((status & AD5933_STAT_DATA_VALID) == 0)
    {
        status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1);
    }

    realData = AD5933_GetRegisterValue(AD5933_REG_REAL_DATA,2);
    imagData = AD5933_GetRegisterValue(AD5933_REG_IMAG_DATA,2);

    realData = convert(realData);
    imagData = convert(imagData);

    voltage_magnitude = sqrt((realData * realData) + (imagData * imagData));

    return voltage_magnitude;
}

/*****************************************************************************//**
 * @brief Reads the real and the imaginary data and calculates the current
    magnitude.
 *
 * @param freqFunction          - Frequency function.
 *                                Example: AD5933_FUNCTION_INC_FREQ - Increment
                                           freq.;
 *                                         AD5933_FUNCTION_REPEAT_FREQ - Repeat
                                           freq..
 *
 * @return current_magnitude          - Calculated current magnitude.
*****************************************************************************/
double AD5933_CalculateCurrent(unsigned char freqFunction)
{
  double         current_magnitude  = 0;
  signed short   realData    = 0;
  signed short   imagData    = 0;
  unsigned char  status      = 0;

  set_input_mux(0);

  AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
                          AD5933_CONTROL_FUNCTION(freqFunction) |
                          AD5933_CONTROL_RANGE(currentRange) |
                          AD5933_CONTROL_PGA_GAIN(currentGain),
                          1);
    status = 0;
    while((status & AD5933_STAT_DATA_VALID) == 0)
    {
        status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1);
    }
```

```
327     realData = AD5933_GetRegisterValue(AD5933_REG_REAL_DATA,2);
        imagData = AD5933_GetRegisterValue(AD5933_REG_IMAG_DATA,2);
329
        realData = convert(realData);
331     imagData = convert(imagData);

333     current_magnitude = sqrt((realData * realData) + (imagData * imagData));

335     return current_magnitude;
    }
337
    /*************************************************************************//**
339  * @brief Reads the real and the imaginary data and calculates the Impedance.
     *
341  * @param gainFactor   - The gain factor.
     * @param freqFunction - Frequency function.
343  *                       Example: AD5933_FUNCTION_INC_FREQ - Increment freq.;
     *                                AD5933_FUNCTION_REPEAT_FREQ - Repeat freq..
345  *
     * @return impedance   - Calculated impedance.
347 *************************************************************************/
    double AD5933_CalculateImpedance(double gainFactor,
349                                  unsigned char freqFunction)
    {
351     signed short  realData  = 0;
        signed short  imagData  = 0;
353     double        voltage_magnitude  = 0;
        double        current_magnitude  = 0;
355     double        impedance = 0;
        unsigned char status    = 0;
357
        /* Calculate Voltage Magnitude */
359     voltage_magnitude = AD5933_CalculateVoltage(freqFunction);

361     /* Calculate Current Magnitude */
        current_magnitude = AD5933_CalculateCurrent(AD5933_FUNCTION_REPEAT_FREQ);
363
        /* Calculate Impedance */
365     impedance =  gainFactor*(voltage_magnitude/current_magnitude);

367     return impedance;
    }
369 /*************************************************************************//**
     * @brief Reads the real and the imaginary data and calculates the voltage phase.
371  *
     * @param systemPhase          - The system phase through calibration circuit.
373  * @param freqFunction         - Frequency function.
     *                               Example: AD5933_FUNCTION_INC_FREQ - Increment
375                                             freq.;
     *                                        AD5933_FUNCTION_REPEAT_FREQ - Repeat
377                                             freq..
     *
379  * @return phase               - Calculated voltage phase.
    *************************************************************************/
381 double AD5933_CalculateVoltagePhase(unsigned char freqFunction)
    {
383   double        phase  = 0;
      signed short  realData   = 0;
385   signed short  imagData   = 0;
      unsigned char status     = 0;
387
      set_input_mux(1);
```

```
389
391   AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
                              AD5933_CONTROL_FUNCTION(freqFunction) |
393                           AD5933_CONTROL_RANGE(currentRange) |
                              AD5933_CONTROL_PGA_GAIN(currentGain),
395                           1);
      status = 0;
397   while((status & AD5933_STAT_DATA_VALID) == 0)
      {
399       status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1);
      }
401
      realData = AD5933_GetRegisterValue(AD5933_REG_REAL_DATA,2);
403   imagData = AD5933_GetRegisterValue(AD5933_REG_IMAG_DATA,2);

405   realData = convert(realData);
      imagData = convert(imagData);
407
      phase = atan2(imagData, realData)*(180.0/PI);
409
      return (phase);
411 }

413 /*************************************************************************//**
    * @brief Reads the real and the imaginary data and calculates the current phase.
415 *
    * @param systemPhase          - The system phase through calibration circuit.
417 * @param freqFunction         - Frequency function.
    *                               Example: AD5933_FUNCTION_INC_FREQ - Increment
419                                           freq.;
    *                                        AD5933_FUNCTION_REPEAT_FREQ - Repeat
421                                           freq..
    *
423 * @return current_phase          - Calculated current phase.
    *************************************************************************//**/
425 double AD5933_CalculateCurrentPhase(unsigned char freqFunction)
    {
427   double         phase  = 0;
      signed short   realData   = 0;
429   signed short   imagData   = 0;
      unsigned char status     = 0;
431
      set_input_mux(0);
433
      AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
435                           AD5933_CONTROL_FUNCTION(freqFunction) |
                              AD5933_CONTROL_RANGE(currentRange) |
437                           AD5933_CONTROL_PGA_GAIN(currentGain),
                              1);
439   status = 0;
      while((status & AD5933_STAT_DATA_VALID) == 0)
441   {
          status = AD5933_GetRegisterValue(AD5933_REG_STATUS,1);
443   }

445   realData = AD5933_GetRegisterValue(AD5933_REG_REAL_DATA,2);
      imagData = AD5933_GetRegisterValue(AD5933_REG_IMAG_DATA,2);
447
      realData = convert(realData);
449   imagData = convert(imagData);
```

```
451     phase = atan2(imagData, realData)*(180.0/PI);
453     return (phase);
}
455
/*****************************************************************************//**
 * @brief Reads the real and the imaginary data and calculates the Phase.
 *
 * @param systemPhase  - The system phase at the current frequency.
 * @param freqFunction - Frequency function.
 *                       Example: AD5933_FUNCTION_INC_FREQ - Increment freq.;
 *                                AD5933_FUNCTION_REPEAT_FREQ - Repeat freq..
 *
 * @return phase   - Calculated load phase.
*****************************************************************************/
467 double AD5933_CalculatePhase(unsigned char freqFunction)
{
469
    double voltage_phase = AD5933_CalculateVoltagePhase(freqFunction);
471     double current_phase = AD5933_CalculateCurrentPhase(freqFunction);
    double phase = voltage_phase - current_phase;
473
475     return (phase);
}
/*****************************************************************************//**
 * @brief Selects the range and gain of the device.
 *
 * @param range - Range option.
 *                Example: AD5933_RANGE_2000mVpp
 *                         AD5933_RANGE_200mVpp
 *                         AD5933_RANGE_400mVpp
 *                         AD5933_RANGE_1000mVpp
 * @param gain  - Gain option.
 *                Example: AD5933_GAIN_X5
 *                         AD5933_GAIN_X1
 *
 * @return None.
*****************************************************************************/
491 void AD5933_SetRangeAndGain(char range, char gain)
{
493     AD5933_SetRegisterValue(AD5933_REG_CONTROL_HB,
                            AD5933_CONTROL_FUNCTION(AD5933_FUNCTION_NOP) |
495                         AD5933_CONTROL_RANGE(range) |
                            AD5933_CONTROL_PGA_GAIN(gain),
497                         1);
    /* Store the last settings made to range and gain. */
499     currentRange = range;
    currentGain = gain;
501 }

503 /*****************************************************************************//**
 * @brief Selects the source of the system clock.
 *
 * @param clkSource  - Selects the source of the system clock.
507  *                     Example: AD5933_CONTROL_INT_SYSCLK
 *                              AD5933_CONTROL_EXT_SYSCLK
509  * @param extClkFreq - Frequency value of the external clock, if used.
 *
511  * @return None.
*****************************************************************************/
```

```
513 void AD5933_SetSystemClk(char clkSource, unsigned long extClkFreq)
{
515     currentClockSource = clkSource;
    if(clkSource == AD5933_CONTROL_EXT_SYSCLK)
517     {
        if(extClkFreq == AD5933_2MHZ_SYS_CLK)
519         {
          // Setting AD5933 clock to 2MHz (P4.2)
521       SETBIT(GP4DAT, 18);
        }
523         else
        {
525       // Setting AD5933 clock to 16MHz (P4.2)
          CLRBIT(GP4DAT, 18);
527       }
        currentSysClk = extClkFreq;                 // External clock frequency
529     }
    else
531     {
        currentSysClk = AD5933_16MHZ_SYS_CLK;     // 16 MHz
533     }
    AD5933_SetRegisterValue(AD5933_REG_CONTROL_LB, currentClockSource, 1);
535 }
```

Code/AD5933.c

```
#include "i2c.h"
2 #include "MCP4661.h"

4 int convert_ohm_to_wiper(float val)
{
6   int position = (int)(val/(181.3));

8   if(position > 255) position = 255;
    if(position < 0) position = 0;
10
    return position;
12 }

14 /* Increments pot x wiper position by one from its' current value */
void increment_WP (char pot)
16 {
    unsigned char cmd = 0x4 | (pot << 4);
18     unsigned char address =   0x28;         /* Address + Write byte */

20     I2C_write_bytes(address, &cmd, 1);
}
22
/* Decrements pot x wiper position by one from its' current value */
24 void decrement_WP (char pot)
{
26     unsigned char cmd = 0x8 | (pot << 4);
    unsigned char address =   0x28;         /* Address + Write byte */
28
    I2C_write_bytes(address, &cmd, 1);
30 }

32 /* Sets pot x wiper position to val current value */
void single_write_WP(char pot, float val)
34 {
/*  Value of write command depends on which pot selected */
36     unsigned char writeData[2]  = {0, 0};
```

```
    unsigned char cmd = 0x0 | (pot << 4);
    unsigned char address =   0x28;          /* Address + Write byte */

    int position = convert_ohm_to_wiper(val);

    writeData[0] = cmd;
    writeData[1] = position;                    /* Data byte - Wiper data */

    I2C_write_bytes(address, writeData, 2);
}
```

Code/MCP4661.c

```
#include "ADuC702x.h"
#include <stdio.h>
#include "math.h"
#include "lcd.h"
#include "i2c.h"
#include "AD5933.h"
#include "util.h"

unsigned int clock_locked = 0;
unsigned int delay_flag = 0;

void init(void)
{
  /*############ Clock Initialisation ############*/
  int t2val_old;

//  GP0CON |= 0x10000000;  // set P0.7 to ECLK - output clock to P0.7

  /* configuring Timer2 for Wake-up */
  t2val_old = T2VAL;
  T2LD = 0x050;   // 5; 5 clocks @32.768kHz = 152ms
  T2CON = 0x480;  // Enable Timer2;
                  // Clock source = internal oscillator, count-down

  while ((T2VAL == t2val_old) || (T2VAL > 3)); // ensures timer is started...

  /* enabling Timer2 interrupt */
  IRQEN = WAKEUP_TIMER_BIT;   // enable TIMER2 Wakeup IRQ

  PLLKEY1 = 0xAA;
  PLLCON  = 0x01;  // External Osc. Select.
  PLLKEY2 = 0x55;

  POWKEY1 = 0x01;
  POWCON  = 0x00;  //
  POWKEY2 = 0xF4;

  while(!clock_locked);

  /*############## Timer Interrupts ##############*/
  IRQEN = GP_TIMER_BIT;          // Timer 0 => Interrupt

  /*############ GPIO Initialisation ############*/
  // Setting AD5933 clock to 2MHz (P4.2)
  SETBIT(GP4DAT, 26);
  SETBIT(GP4DAT, 18);

  // Setting calibration mux to test load (P4.5)
  SETBIT(GP4DAT, 29);
```

```
  CLRBIT(GP4DAT, 21);

  // Enabling A0 and A1 (P4.7 & P4.6)
  SETBIT(GP4DAT, 31);
  SETBIT(GP4DAT, 30);


  // Enabling input mux (P4.4)
  SETBIT(GP4DAT, 28);
  SETBIT(GP4DAT, 20);

  // Setting input mux to voltage (P4.3)
  SETBIT(GP4DAT, 27);
  CLRBIT(GP4DAT, 19);

  // Enabling button pins P0.6 & P1.2 as inputs
  CLRBIT(GP0DAT, 30);
  CLRBIT(GP1DAT, 26);
}

/*############## Differential Voltage Gain ##############*/
void set_voltageGain(unsigned char gain)
{
  if(gain == voltage_Gain_x10)
  {
    SETBIT(GP4DAT, 23);
    SETBIT(GP4DAT, 22);
  }
  else if(gain == voltage_Gain_x5)
  {
    CLRBIT(GP4DAT, 23);
    SETBIT(GP4DAT, 22);
  }
  else if(gain == voltage_Gain_x2)
  {
    SETBIT(GP4DAT, 23);
    CLRBIT(GP4DAT, 22);
  }
  else // Gain of x1 by default
  {
    CLRBIT(GP4DAT, 23);
    CLRBIT(GP4DAT, 22);
  }
}

unsigned int readButton1(void)
{
  if(GP0DAT & 0x40) return 1;
  else return 0;
}
unsigned int readButton2(void)
{
  if(GP1DAT & 0x04) return 1;
  else return 0;
}

void set_test_load(unsigned int test)
{
  if(test)
    CLRBIT(GP4DAT, 21); // Setting calibration mux to test load (P4.5)
  else
    SETBIT(GP4DAT, 21); // Calibration mux set to electrodes
```

```
}

void set_input_mux(unsigned int voltage)
{
  if(voltage) CLRBIT(GP4DAT, 19); // Setting input mux to voltage measurement
  else SETBIT(GP4DAT, 19);        // Setting input mux to current measurement
}

//Function for converting the two hex values to a decimal value
unsigned long int hextodec(unsigned char data_high, unsigned char data_low)
{
  unsigned long int data;
  data=(unsigned long int)data_high*256+data_low;
  return data;
}

double calculate_resistance(double impedance, double phase)
{
  return fabs(impedance*cos(phase)); // Returns absolute value
}

double calculate_reactance(double impedance, double phase)
{
  return fabs(impedance*sin(phase)); // Returns absolute value
}

// Used for delays when interrupts aren't setup yet
void delay (int length)
{
  while (length >=0)
  {
    length--;
  }
}

void delay_ms(unsigned int delay)
{
  /* 41.78MHz/256 => 6.127uS  |  1mS/6.127uS = ~163 */
  T1LD = delay * 163;
  T1CON = 0xC8;       // Timer 0 Enabled, Periodic, /256 prescaler

  while(!delay_flag);

  delay_flag = 0;
}

void delay_us(unsigned int delay)
{
  /* 41.78MHz => 23.934nS |  1uS/23.934nS = ~42 */
  T1LD = delay * 42;
  T1CON = 0xC0;       // Timer 0 Enabled, Periodic, /16 prescaler

  while(!delay_flag);
  delay_flag = 0;
}

void my_IRQ_Handler (void)
{
  static unsigned int statI2C0MSTA = 0;
  static unsigned int statIRQSTA = 0;

  statI2C0MSTA = I2C0MSTA;
```

```
  statIRQSTA = IRQSTA;

  /* Clock initialisation Interrupt */
  if(IRQSIG & WAKEUP_TIMER_BIT)
  {
    /* disable timer, clear interrupt flag */
    T2CON   = 0x00;
    T2CLRI  = 0x01;
    clock_locked = 1;
  }

  /* Delay Interrupt */
  if(IRQSIG & GP_TIMER_BIT)
  {
    T1CLRI = 0x01;                  // Clear Timer 0 Interrupt
    T1CON = 0x00;                   // Disable timer
    delay_flag = 1;
  }

  /* I2C Interrupts */
  if(statIRQSTA & 0x400) // Check for I2C0 Master interrupt
  {
    if (statI2C0MSTA & 0x4) { // Master Transmit IRQ
      if (tx_length > 0) { // Tx FIFO ready for another byte
        I2C0MTX = i2c_tx[--tx_length];
      }
      else {
        TransmitInProgress = 0;
        rx_length = 0;
        rx_count = 0;
      }
    }

    if (statI2C0MSTA & 0x8) { // Master Recieve IRQ
      i2c_rx[rx_length++] = I2C0MRX;

      I2C0CNT = rx_count-1;

      TransmitInProgress = rx_count > 0;
    }
  }
}
```

Code/util.c

Appendix C

Python GUI

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep  8 00:00:41 2016

@author: Gareth
"""

import sys
import glob
import serial
import re

import matplotlib
from tkinter import filedialog

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg,
    NavigationToolbar2TkAgg
from matplotlib.figure import Figure
from matplotlib import style
import tkinter as tk
from tkinter import ttk
import tkinter.messagebox
from tkinter import *

from PIL import Image, ImageTk

import numpy as np
np.seterr(invalid='ignore')

from scipy.optimize import curve_fit
from math import cos, sin, pi, sqrt, atan

import xlsxwriter
import os.path
import time



LARGE_FONT= ("Verdana", 14)
style.use("seaborn-paper")

# Ordering from device: Frequency Impedance Phase Resistance Reactance
w = []
freq = []
```

```
impedance_data = []
phase_data = []
real_data = []
imag_data = []
hydration_parameters = []

# RC Theory Parameters
Z = []
Theta = []
f_theory = []

# RC||R Theory Parameters
resistance = []
reactance = []
f_theory1 = []

# Plot figures
f = Figure(figsize=(15,5), dpi=80, facecolor='white')
LSR = f.add_subplot(2,1,1)
LSR.grid('on')

CC = f.add_subplot(2,1,2)
CC.grid('on')

r = Figure(figsize=(12,5), dpi=80, facecolor='white')
RC = r.add_subplot(2,1,1)
P = r.add_subplot(2,1,2)
RC.grid('on')
P.grid('on')

phantom = Figure(figsize=(12,5), dpi=80, facecolor='white')
R_F = phantom.add_subplot(2,1,1)
R_R = phantom.add_subplot(2,1,2)
R_F.grid('on')
R_R.grid('on')

LSR.set_xlabel('Natural Frequency (rad/s)', fontsize=14)
LSR.set_ylabel('Resistance (  )', fontsize=14)

CC.set_xlabel('Resistance (  )', fontsize=14)
CC.set_ylabel('Reactance (  )', fontsize=14)


RC.set_xlabel('Frequency (kHz)', fontsize=14)
RC.set_ylabel('Impedance (  )', fontsize=14)

P.set_xlabel('Frequency (kHz)', fontsize=14)
P.set_ylabel('Phase (degrees)', fontsize=14)

R_F.set_xlabel('Frequency (kHz)', fontsize=14)
R_F.set_ylabel('Resistance (  )', fontsize=14)

R_R.set_xlabel('Resistance (  )', fontsize=14)
R_R.set_ylabel('Reactance (  )', fontsize=14)

sweep_complete = False
reset = True
quit_ = False

serial_port = None

def serial_ports():
```

```
106     """ Lists serial port names
107
108         :raises EnvironmentError:
109             On unsupported or unknown platforms
110         :returns:
111             A list of the serial ports available on the system
112     """
113     if sys.platform.startswith('win'):
114         ports = ['COM%s' % (i + 1) for i in range(256)]
115     elif sys.platform.startswith('linux') or sys.platform.startswith('cygwin'):
116         # this excludes your current terminal "/dev/tty"
117         ports = glob.glob('/dev/tty[A-Za-z]*')
118     elif sys.platform.startswith('darwin'):
119         ports = glob.glob('/dev/tty.*')
120     else:
121         raise EnvironmentError('Unsupported platform')
122
123     result = []
124     for port in ports:
125         try:
126             s = serial.Serial(port)
127             s.close()
128             result.append(port)
129         except (OSError, serial.SerialException):
130             pass
131     return result
132
133 OPTIONS = list(serial_ports())
134
135 def set_port(port):
136     global serial_port
137
138     serial_port = port
139     return
140
141 def func(x, a, b, c):
142     return a * np.exp(-b * x) + c
143
144 def cole_cole_func(w, a, b, c, d):
145     return b+((a-b)*(1+pow((w*c),d)*cos(d*pi/2)))/(1+2*pow((w*c),d)*cos(d*pi/2)+
    pow((w*c),(2*d)))
146
147 def file_save():
148     file = tkinter.filedialog.asksaveasfilename(filetypes = (("Excel workbook=",
    "*.xlsx"), ("All files", "*.*")), defaultextension=".xlsx")
    ".
149         return
150
151     # Create an new Excel file and add a worksheet.
152     workbook = xlsxwriter.Workbook(file)
153     worksheet = workbook.add_worksheet()
154
155     # Add a bold format to use to highlight cells.
156     bold = workbook.add_format({'bold': True})
157
158     worksheet.write('A1', time.strftime("%H:%M:%S"))
159     worksheet.write('A2', time.strftime("%d/%m/%Y"))
160
161     worksheet.set_column('A:A', 20)
162     worksheet.write('A3', 'Frequency (kHz)', bold)
163     for i in range (3, len(freq)+3):
164
```

```
            worksheet.write(i, 0, freq[i-3])

    worksheet.set_column('B:B', 20)
    worksheet.write('B3', 'Impedance (  )', bold)
    for i in range (3, len(freq)+3):
            worksheet.write(i, 1, impedance_data[i-3])

    worksheet.set_column('C:C', 20)
    worksheet.write('C3', 'Phase (degrees)', bold)
    for i in range (3, len(freq)+3):
            worksheet.write(i, 2, phase_data[i-3])

    worksheet.set_column('D:D', 20)
    worksheet.write('D3', 'Resistance (  )', bold)
    for i in range (3, len(freq)+3):
            worksheet.write(i, 3, real_data[i-3])

    worksheet.set_column('E:E', 20)
    worksheet.write('E3', 'Reactance (  )', bold)
    for i in range (3, len(freq)+3):
            worksheet.write(i, 4, imag_data[i-3])

    figurename, file_extension = os.path.splitext(file)
    figurename = figurename+'.png'

    f.savefig(figurename, bbox_inches='tight', dpi=100)

    if not sweep_complete: # avoid null errors if sweep isn't over
        return

    worksheet.set_column('F:F', 20)
    worksheet.write('F3', 'ECW (Litres)', bold)
    worksheet.write(3, 5, hydration_parameters[0])

    worksheet.set_column('G:G', 20)
    worksheet.write('G3', 'ICW (Litres)', bold)
    worksheet.write(3, 6, hydration_parameters[1])

    worksheet.set_column('H:H', 20)
    worksheet.write('H3', 'R0 (ohms)', bold)
    worksheet.write(3, 7, hydration_parameters[2])

    worksheet.set_column('I:I', 20)
    worksheet.write('I3', 'Rinf (ohms)', bold)
    worksheet.write(3, 8, hydration_parameters[3])

    worksheet.set_column('J:J', 20)
    worksheet.write('J3', 'R50 (ohms)', bold)
    worksheet.write(3, 9, hydration_parameters[4])

    # Insert an image.
    #worksheet.insert_image('B5', 'logo.png')

    workbook.close()

def figure_save():

    file = tkinter.filedialog.asksaveasfilename(filetypes = (("PNG=", "*.png"), (
    "All files", "*.*")), defaultextension=".png")
    if file is None: # asksaveasfile return `None` if dialog closed with "cancel
    ".
        return
```

```
    f.savefig(file, bbox_inches='tight', dpi=100)

def RC_figure_save():

    file = tkinter.filedialog.asksaveasfilename(filetypes = (("PNG=", "*.png"), ("All files", "*.*")), defaultextension=".png")
    if file is None: # asksaveasfile return `None` if dialog closed with "cancel".
        return
    r.savefig(file, bbox_inches='tight', dpi=80)

def RC_R_figure_save():

    file = tkinter.filedialog.asksaveasfilename(filetypes = (("PNG=", "*.png"), ("All files", "*.*")), defaultextension=".png")
    if file is None: # asksaveasfile return `None` if dialog closed with "cancel".
        return
    phantom.savefig(file, bbox_inches='tight', dpi=80)

def plot_cole_cole_data():
    ######################################################
    #############    Cole-Cole Calculations #############
    ######################################################
    global ICW_var
    global ECW_var

    # Convert to numpy arrays from lists
    real_data_np = np.asarray(real_data)
    imag_data_np = np.asarray(imag_data)
    w_np = np.asarray(w)

    Rinf = float(app.frames.get(graphPage).Rinf.get())
    R0 = float(app.frames.get(graphPage).R0.get())
    tau = float(app.frames.get(graphPage).tau.get())
    alpha = float(app.frames.get(graphPage).alpha.get())

    p0 = np.array([R0, Rinf, tau, alpha])

    #p0 = np.array([700, 300, 5e-03, 0.65])

    if app.frames.get(graphPage).fit_type.get():
        print("Using LM")
        iterations = int(app.frames.get(graphPage).iter.get())

        try:
            popt, pcov = curve_fit(cole_cole_func, w_np, real_data_np, p0, maxfev=iterations)
        except RuntimeError:
            print("Error - curve_fit failed")
    else:
        print("Using trf")
        try:
            popt, pcov = curve_fit(cole_cole_func, w_np, real_data_np, p0, bounds=(0, [1000., 1000., 1, 1.]))
        except RuntimeError:
            print("Error - curve_fit failed")

    print(popt)

    cole_cole_fit = cole_cole_func(w_np, popt[0], popt[1], popt[2], popt[3])

```

```
    # define angles (100 evenly spaced)
282 t = np.linspace(0, pi, 100)

284 # define radius
    radius = (popt[0] - popt[1])/(2*sin((popt[3]*pi)/2))
286
    # find real and imaginary centre
288 r_c = (popt[0] + popt[1])/2
    i_c = -((popt[0] - popt[1])/2)*((cos((popt[3]*pi)/2))/(sin((popt[3]*pi)/2)))
290
    real = r_c + radius * np.cos(t)
292 imag = i_c + radius * np.sin(t)

294 pecw = 0
    picw = 0
296
    if app.frames.get(graphPage).gender.get():
298     pecw = 40.5 # male
        picw = 273.9
300 else:
        pecw = 39   # female
302     picw = 264.9

304 Kb = 4.3

306 if app.frames.get(graphPage).height.get():
        H = float(app.frames.get(graphPage).height.get())
308 else:
        H = 0
310
    if app.frames.get(graphPage).weight.get():
312     W = float(app.frames.get(graphPage).weight.get())
    else:
314     W = 0

316 D = 1.05

318 ECW = (1/100)*pow(((pecw*Kb*pow(H,2)*sqrt(W))/(sqrt(D)*popt[0])),(2/3))

320 ptbw = picw - ((picw - pecw)*pow((popt[1]/popt[0]),(2/3)))

322 ICW = ECW*(pow(((ptbw*popt[0])/(pecw*popt[1])),(2/3)) - 1)


324
    hydration_parameters.append(ECW)
326 hydration_parameters.append(ICW)
    hydration_parameters.append(popt[0])
328 hydration_parameters.append(popt[1])
    hydration_parameters.append(real_data[80])
330
    CC.clear()
332 LSR.clear()

334 LSR.plot(w_np, real_data_np, 'r.', label="Resistance vs Natural Frequency")
    LSR.plot(w_np, cole_cole_fit, label="LSR")
336 LSR.legend(prop={'size':14})

338 CC.plot(real_data_np, imag_data_np, 'r.', label="Resistance (  ) vs Reactance
     (  )")
    CC.plot(real, imag, label="Cole-Cole Model")
340 CC.legend(prop={'size':14})
```

```
342     max_data = 0
344     for r_data in imag_data_np:
            if r_data > max_data:
346             max_data = r_data
348
        CC.set_xlim([int(real_data_np[-1])-50, int(real_data_np[0])+50])
350     CC.set_ylim([0, max_data+20])
352     LSR.set_xlabel('Natural Frequency (rad/s)', fontsize=14)
        LSR.set_ylabel('Resistance ( )', fontsize=14)
354
        CC.set_xlabel('Resistance ( )', fontsize=14)
356     CC.set_ylabel('Reactance ( )', fontsize=14)
358     CC.grid('on')
        LSR.grid('on')
360
362     app.frames.get(graphPage).update_hydration_paramaters(ECW, ICW)
364     phantom.canvas.draw_idle()
        r.canvas.draw_idle()
366     f.canvas.draw_idle()
368     ######################################################
        return
370
    def plot_RC_data(f_theory, Z, Theta):
372     RC.clear()
        P.clear()
374
        RC.plot(np.asarray(f_theory), np.asarray(Z), 'b', label="Theoretical")
376     RC.set_xlim([0, int(f_theory[-1])+5])
        RC.set_ylim([int(Z[-1]-5), int(Z[0]+5)])
378
        P.plot(np.asarray(f_theory), np.asarray(Theta), 'b', label="Theoretical")
380     P.set_xlim([0, int(f_theory[-1])+5])
        P.set_ylim([int(Theta[-1]-5), int(Theta[0]+5)])
382
        RC.plot(np.asarray(freq), np.asarray(impedance_data), 'r.', label="
        Experimental")
384     RC.set_xlabel('Frequency (kHz)', fontsize=14)
        RC.set_ylabel('|Z| ( )', fontsize=14)
386
        P.plot(np.asarray(freq), np.asarray(phase_data), 'r.', label="Experimental")
388     P.set_xlabel('Frequency (kHz)', fontsize=14)
        P.set_ylabel('Phase ( )', fontsize=14)
390
        RC.legend(prop={'size':14})
392     P.legend(prop={'size':14})
394     RC.grid('on')
        P.grid('on')
396
        r.canvas.draw_idle()
398     return
400 def plot_RC_R_data(f_theory1, resistance, reactance):
        R_F.clear()
402     R_R.clear()
```

```
404     R_F.plot(np.asarray(f_theory1), np.asarray(resistance), 'b', label="
        Theoretical")
        R_F.set_xlim([0, int(f_theory1[-1])+5])
406     R_F.set_ylim([int(resistance[-1]-5), int(resistance[0]+5)])

408     R_R.plot(np.asarray(resistance), np.asarray(reactance), 'b', label="
        Theoretical")
        R_R.set_xlim([int(resistance[-1]-5), int(resistance[0]+5)])
410
        maximum = 0
412     pos = 0
        y = 0
414
        for z in reactance:
416         if z>maximum:
                maximum = z
418             pos = y

420         y+=1

422     R_R.set_ylim([0, int(reactance[pos]+40)])

424     R_F.plot(np.asarray(freq), np.asarray(real_data), 'r.', label="Experimental")
        R_F.set_xlabel('Frequency (kHz)', fontsize=14)
426     R_F.set_ylabel('Resistance (  )', fontsize=14)

428     R_R.plot(np.asarray(real_data), np.asarray(imag_data), 'r.', label="
        Experimental")
        R_R.set_xlabel('Resistance (  )', fontsize=14)
430     R_R.set_ylabel('Reactance (  )', fontsize=14)

432     R_R.legend(prop={'size':14})
        R_F.legend(prop={'size':14})
434
        R_R.grid('on')
436     R_F.grid('on')

438     phantom.canvas.draw_idle()
        return
440
    class GUI(tk.Tk):
442     def __init__(self, *args, **kwargs):
            tk.Tk.__init__(self, *args, **kwargs)
444         tk.Tk.wm_title(self, "Hydration Monitoring")

446         container = tk.Frame(self)
            container.pack(side="top", fill="both", expand = True)
448         container.grid_rowconfigure(0, weight=1)
            container.grid_columnconfigure(0, weight=1)
450
            self.frames = {}
452
            for F in (startPage, graphPage, rcPage, phantomPage):
454             frame = F(container, self)
                self.frames[F] = frame
456             frame.grid(row=0, column=0, sticky="nsew")

458         self.show_frame(startPage)
            w, h = self.winfo_screenwidth(), self.winfo_screenheight()
460         self.geometry("%dx%d+0+0" % (w-400, h-400))
```

```
462         # Menu Bar
            self.menubar = Menu(self)
464         self.filemenu = Menu(self.menubar, tearoff=0)
            self.filemenu.add_command(label = "Save data", command = file_save)
466         self.filemenu.add_command(label = "Save hydration figure", command =
    figure_save)
            self.filemenu.add_command(label = "Save RC figure", command =
    RC_figure_save)
468         self.filemenu.add_command(label = "Save RC||R figure", command =
    RC_R_figure_save)

470         self.filemenu.add_separator()

472         self.filemenu.add_command(label = "Exit", command = callback)
            self.menubar.add_cascade(label = 'File', menu = self.filemenu)
474
            self.toolsmenu=Menu(self.menubar,tearoff=0)
476
            self.submenu = Menu(self.toolsmenu)
478         self.toolsmenu.add_cascade(label='Serial ports', menu=self.submenu,
    underline=0)

480         for port in OPTIONS:
                self.submenu.add_command(label=port, command=lambda p=port: set_port(
    p))
482


484         self.toolsmenu.add_separator()
            self.toolsmenu.add_command(label="Reset", command=self.reset)
486         self.menubar.add_cascade(label="Tools", menu=self.toolsmenu)

488         self.config(menu = self.menubar)

490     def show_frame(self, cont):
            frame = self.frames[cont]
492         frame.tkraise()

494     def reset(self):
            global reset
496         global sweep_complete

498         LSR.clear()
            CC.clear()
500         P.clear()
            RC.clear()
502         R_F.clear()
            R_R.clear()
504
            real_data[:] = []
506         imag_data[:] = []
            freq[:] = []
508         w[:] = []
            impedance_data[:] = []
510         phase_data[:] = []
            hydration_parameters[:] = []
512
            resistance[:] = []
514         reactance[:] = []
            f_theory1[:] = []
516
            Z[:] = []
518         Theta[:] = []
```

```
        f_theory[:] = []

        CC.grid('on')
        LSR.grid('on')

        phantom.canvas.draw_idle()
        r.canvas.draw_idle()
        f.canvas.draw_idle()

        app.frames.get(graphPage).update_hydration_status("Press start button.")

        reset = True
        sweep_complete = False

def callback():
    global quit_
    if tkinter.messagebox.askokcancel("Quit", "Do you really wish to quit?"):
        quit_ = True


class startPage(tk.Frame):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self,parent)

        background_image= tk.PhotoImage(master = self, file="background.png")
        background_label = tk.Label(self, image=background_image)
        background_label.place(x=0, y=0, relwidth=1, relheight=1)

        background_label.image = background_image

        tk.Label(self, text="Hydration Monitor V3.0", font=("Calibri", 50)).grid(
    row=0, column=0)

        tk.Button(self, text="Hydration Analysis", font=("Calibri", 20),
                  command=lambda: controller.show_frame(graphPage)).grid(row=1,
     column=0, sticky=E, padx = 10)


        tk.Button(self, text="RC Circuit Analysis", font=("Calibri", 20),
                  command=lambda: controller.show_frame(rcPage)).grid(row=2,
    column=0, sticky=E, padx = 10)

        tk.Button(self, text="RC||R Circuit Analysis", font=("Calibri", 20),
            command=lambda: controller.show_frame(phantomPage)).grid(row=3,
    column=0, sticky=E, padx = 10)

class graphPage(tk.Frame):
    def __init__(self, parent, controller):

        tk.Frame.__init__(self, parent)

        background_image= tk.PhotoImage(master = self, file="background.png")
        background_label = tk.Label(self, image=background_image)
        background_label.place(x=0, y=0, relwidth=1, relheight=1)

        background_label.image = background_image

        Grid.rowconfigure(self, 0, weight=1)
        Grid.columnconfigure(self, 0, weight=1)

        self.ICW = tk.StringVar(self)
```

```
        self.ECW = tk.StringVar(self)
        self.gender = tk.IntVar(self)
        self.fit_type = tk.IntVar(self)

        label_frame = tk.Frame(self)
        label_frame.grid(row=0, column=0, sticky=N)

        tk.Label(label_frame, text="Hydration Analysis", font=("Calibri", 36)).
    grid(row=0, column=0, columnspan=2)

        tk.Label(label_frame, textvariable=self.ICW, font=("Calibri", 16), fg = "
    blue").grid(row=1, column=1)
        tk.Label(label_frame, textvariable=self.ECW, font=("Calibri", 16), fg = "
    blue").grid(row=2, column=1)

        tk.Label(label_frame, text="ICW: ", font=("Calibri", 16), fg = "grey").
    grid(row=1, column=0)
        tk.Label(label_frame, text="ECW: ", font=("Calibri", 16), fg = "grey").
    grid(row=2, column=0)


        tk.Label(label_frame, text="Height (cm)", font=("Calibri", 10), fg = "
    grey").grid(row=3,column=0)
        self.height = tk.Entry(label_frame)
        self.height.grid(row=4,column=0)

        tk.Label(label_frame, text="Weight (kg)", font=("Calibri", 10), fg = "
    grey").grid(row=3,column=1)
        self.weight = tk.Entry(label_frame)
        self.weight.grid(row=4,column=1)

        tk.Radiobutton(label_frame, text="Male", variable=self.gender, value=1,
    font=("Calibri", 10), fg = "grey").grid(row=5,column=0, pady=10)
        tk.Radiobutton(label_frame, text="Female", variable=self.gender, value=0,
     font=("Calibri", 10), fg = "grey").grid(row=5,column=1, pady=10)

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

        tk.Label(label_frame, text="R0", font=("Calibri", 10), fg = "grey").grid(
    row=6,column=0)
        self.R0 = tk.Entry(label_frame)
        self.R0.insert(END, '700')
        self.R0.grid(row=7,column=0)

        tk.Label(label_frame, text="R  ", font=("Calibri", 10), fg = "grey").
    grid(row=6,column=1)
        self.Rinf = tk.Entry(label_frame)
        self.Rinf.insert(END, '300')
        self.Rinf.grid(row=7,column=1)

        tk.Label(label_frame, text="  ", font=("Calibri", 10), fg = "grey").grid(
    row=8,column=0)
        self.tau = tk.Entry(label_frame)
        self.tau.insert(END, '5e-03')
        self.tau.grid(row=9,column=0)

        tk.Label(label_frame, text="  ", font=("Calibri", 10), fg = "grey").grid(
    row=8,column=1)
        self.alpha = tk.Entry(label_frame)
        self.alpha.insert(END, '0.65')
        self.alpha.grid(row=9,column=1)
```

```
626         tk.Radiobutton(label_frame, text="Levenberg-Marquardt", variable=self.
    fit_type, value=1, font=("Calibri", 10), fg = "grey").grid(row=10,column=0,
    pady=5)
            tk.Radiobutton(label_frame, text="Trust Region Reflective", variable=self
    .fit_type, value=0, font=("Calibri", 10), fg = "grey").grid(row=10,column=1,
    rowspan=3)
628
            tk.Label(label_frame, text="Interations", font=("Calibri", 10), fg = "
    grey").grid(row=11,column=0)
630         self.iter = tk.Entry(label_frame)
            self.iter.insert(END, '20000')
632         self.iter.grid(row=12,column=0)


634
            ########################
636
            ttk.Button(label_frame, text="Re-fit", command = plot_cole_cole_data).
    grid(row=13, column=0, pady=5)
638
            ttk.Button(label_frame, text="Return",
640             command=lambda: controller.show_frame(startPage)).grid(row=13, column
    =1)

642         canvas = FigureCanvasTkAgg(f, self)
            canvas.show()
644         canvas.get_tk_widget().grid(row = 0, column = 1, sticky=N+E+W+S)


646
    def update_hydration_paramaters(self, ECW_val, ICW_val):
648
        ICW_str = "{:.2f}".format(ICW_val)
650     ECW_str = "{:.2f}".format(ECW_val)

652     self.ICW.set(ICW_str + " Litres")
        self.ECW.set(ECW_str + " Litres")
654
    def update_hydration_status(self, status):
656
        self.ICW.set(status)
658     self.ECW.set(status)

660 class rcPage(tk.Frame):
    def __init__(self, parent, controller):
662
        tk.Frame.__init__(self, parent)
664
        background_image= tk.PhotoImage(master = self, file="background.png")
666     background_label = tk.Label(self, image=background_image)
        background_label.place(x=0, y=0, relwidth=1, relheight=1)
668
        background_label.image = background_image
670
        Grid.rowconfigure(self, 0, weight=1)
672     Grid.columnconfigure(self, 0, weight=1)

674     label_frame = tk.Frame(self)
        label_frame.grid(row=0, column=0, sticky=N)
676
        tk.Label(label_frame, text="RC Analysis", font=("Calibri", 36)).grid(row
    =0,column=0, columnspan=2, sticky=N)
678
```

```
        tk.Label(label_frame, text="Start Frequency (kHz)", font=("Calibri", 10),
 fg = "grey").grid(row=1,column=0)
680     freq_start= tk.Entry(label_frame)
        freq_start.grid(row=2,column=0, padx=5)
682
        tk.Label(label_frame, text="End Frequency (kHz)", font=("Calibri", 10),
fg = "grey").grid(row=1,column=1)
684     freq_end = tk.Entry(label_frame)
        freq_end.grid(row=2,column=1, padx=5)
686
        tk.Label(label_frame, text="Frequency Increment (kHz)", font=("Calibri",
10), fg = "grey").grid(row=3,column=0)
688     freq_incr= tk.Entry(label_frame)
        freq_incr.grid(row=4,column=0)
690
        tk.Label(label_frame, text="Capacitance (pF)", font=("Calibri", 10), fg =
 "grey").grid(row=5,column=0)
692     cap = tk.Entry(label_frame)
        cap.grid(row=6,column=0)
694
        tk.Label(label_frame, text="Resistance (  )", font=("Calibri", 10), fg =
"grey").grid(row=5,column=1)
696     res= tk.Entry(label_frame)
        res.grid(row=6,column=1)
698
        ttk.Button(label_frame, text="Plot",
700                 command=lambda: self.plot_RC(freq_start, freq_end, freq_incr,
cap, res)).grid(row=7,column=0, columnspan=2, pady=10)

702     canvas = FigureCanvasTkAgg(r, self)
        canvas.show()
704     canvas.get_tk_widget().grid(row = 0, column = 1, sticky=N+S+E+W)

706     ttk.Button(label_frame, text="Return",
                   command=lambda: controller.show_frame(startPage)).grid(row=8,
column=0, columnspan=2)
708
    def plot_RC(self, freq_start, freq_end, freq_incr, cap, res):
710     fs = fe = fi = c = r = 0.00

712     if freq_start.get():
            fs = float(freq_start.get())
714
        if freq_end.get():
716         fe = float(freq_end.get())

718     if freq_incr.get():
            fi = float(freq_incr.get())
720
        if cap.get():
722         c = float(cap.get())

724     if res.get():
            r = float(res.get())
726
        incr = int(((fe - fs)/fi) + 1)
728
        for i in range(0, incr):
730         f_theory.append(fs + (fi*i))
            Z.append((1/(sqrt(pow(1/r, 2) + pow(2*pi*f_theory[i]*1000*c*pow(10,
-12), 2)))))
732         Theta.append(atan(-2*pi*f_theory[i]*1000*c*pow(10, -12)*r)*(180/pi))
```

```
734         plot_RC_data(f_theory, Z, Theta)

736         return

738 class phantomPage(tk.Frame):
    def __init__(self, parent, controller):
740
        tk.Frame.__init__(self, parent)
742
        background_image= tk.PhotoImage(master = self, file="background.png")
744         background_label = tk.Label(self, image=background_image)
        background_label.place(x=0, y=0, relwidth=1, relheight=1)
746
        background_label.image = background_image
748
        Grid.rowconfigure(self, 0, weight=1)
750         Grid.columnconfigure(self, 0, weight=1)

752         label_frame = tk.Frame(self)
        label_frame.grid(row=0, column=0, sticky=N)
754
        tk.Label(label_frame, text="RC||R Analysis", font=("Calibri", 36)).grid(
    row=0,column=0, columnspan=3)
756
        tk.Label(label_frame, text="Start Frequency (kHz)", font=("Calibri", 10),
     fg = "grey").grid(row=1,column=0, padx=15)
758         freq_start= tk.Entry(label_frame)
        freq_start.grid(row=2,column=0)
760
        tk.Label(label_frame, text="End Frequency (kHz)", font=("Calibri", 10),
    fg = "grey").grid(row=1,column=1)
762         freq_end = tk.Entry(label_frame)
        freq_end.grid(row=2,column=1)
764
        tk.Label(label_frame, text="Frequency Increment (kHz)", font=("Calibri",
    10), fg = "grey").grid(row=1,column=2)
766         freq_incr= tk.Entry(label_frame)
        freq_incr.grid(row=2,column=2)
768
        tk.Label(label_frame, text="Capacitance (pF)", font=("Calibri", 10), fg =
     "grey").grid(row=3,column=0)
770         cap = tk.Entry(label_frame)
        cap.grid(row=4,column=0)
772


774         tk.Label(label_frame, text="Series Resistance (  )", font=("Calibri", 10)
    , fg = "grey").grid(row=3,column=1)
        res1 = tk.Entry(label_frame)
776         res1.grid(row=4,column=1)

778         tk.Label(label_frame, text="Parallel Resistance (  )", font=("Calibri",
    10), fg = "grey").grid(row=3,column=2)
        res2 = tk.Entry(label_frame)
780         res2.grid(row=4,column=2)

782         ttk.Button(label_frame, text="Plot",
                    command=lambda: self.plot_RC_R(freq_start, freq_end, freq_incr
    , cap, res1, res2)).grid(row=5,column=0, pady=10, columnspan=3)
784
        canvas = FigureCanvasTkAgg(phantom, self)
786         canvas.show()
```

```
        canvas.get_tk_widget().grid(row = 0, column = 1, sticky=N+S+E+W)

        ttk.Button(label_frame, text="Return",
                   command=lambda: controller.show_frame(startPage)).grid(row=6, column=0, columnspan=3)

    def plot_RC_R(self, freq_start, freq_end, freq_incr, cap, res1 ,res2):
        fs = fe = fi = c = r1 = r2 = 0.00

        if freq_start.get():
            fs = float(freq_start.get())

        if freq_end.get():
            fe = float(freq_end.get())

        if freq_incr.get():
            fi = float(freq_incr.get())

        if cap.get():
            c = float(cap.get())

        if res1.get():
            r1 = float(res1.get())

        if res2.get():
            r2 = float(res2.get())

        incr = int(((fe - fs)/fi) + 1)

        for i in range(0, incr):
            f_theory1.append(fs + (fi*i))

            2*pi*f_theory1[i]*1000
            c*pow(10, -12)

            resistance.append(((r1*r2)*(r1+r2) + r2*pow((1/(2*pi*f_theory1[i]*1000*c*pow(10, -12))), 2))/(pow((r1+r2), 2)+ pow((1/(2*pi*f_theory1[i]*1000*c*pow(10, -12))), 2)))
            reactance.append(-((-(r2/(2*pi*f_theory1[i]*1000*c*pow(10, -12)))*(r1+r2) + (r1*r2)*(1/(2*pi*f_theory1[i]*1000*c*pow(10, -12))))/(pow((r1+r2), 2)+ pow((1/(2*pi*f_theory1[i]*1000*c*pow(10, -12))), 2))))

        plot_RC_R_data(f_theory1, resistance, reactance)

        return

def update_plots(i):
        global sweep_complete
        global ready

        if sweep_complete is not True:
            line = ser.readline().strip()       # Read next data line
                                                # Strip removes the \r\n carriage return
            line = line.decode("utf-8") # converts back to a string from a byte

            line_data = re.split(r'\t+', line)     # removing tab delimiters between doubles, returns an array of strings for each data piece in the line

            y = len(line_data)  # Number of pieces of data

            for x in range(0, y):
```

```
842                 if line_data[x] is '':
                        return False
844
                    if x==0:
846                     w.append(float(line_data[x])*2*pi)
                        freq.append(float(line_data[x]))
848
                    elif x==1:
850                     impedance_data.append(float(line_data[x]))
                    elif x==2:
852                     phase_data.append(float(line_data[x]))
                    elif x==3:
854                     real_data.append(float(line_data[x]))
                    elif x==4:
856                     imag_data.append(float(line_data[x]))

858             if float(line_data[0]) == 150:   # Frequency sweep has finished
                    plot_cole_cole_data()
860                 sweep_complete = True
                    return True
862
                LSR.plot(w, real_data, 'r.', label="Resistance vs Natural Frequency")
864             LSR.set_xlabel('Natural Frequency (rad/s)', fontsize=14)
                LSR.set_ylabel('Resistance (  )', fontsize=14)
866
                CC.plot(real_data, imag_data, 'b.', label="Resistance (  ) vs
    Reactance (  )")
868             CC.set_xlabel('Resistance (  )', fontsize=14)
                CC.set_ylabel('Reactance (  )', fontsize=14)
870             CC.set_ylim([0, 100])


872
                RC.plot(np.asarray(freq), np.asarray(impedance_data), 'r.', label="
    Experimental")
874             RC.set_xlabel('Frequency (kHz)', fontsize=14)
                RC.set_ylabel('|Z| (  )', fontsize=14)
876
                P.plot(np.asarray(freq), np.asarray(phase_data), 'r.', label="
    Experimental")
878             P.set_xlabel('Frequency (kHz)', fontsize=14)
                P.set_ylabel('Phase (  )', fontsize=14)
880
                R_F.plot(np.asarray(freq), np.asarray(real_data), 'r.', label="
    Experimental")
882             R_F.set_xlabel('Frequency (kHz)', fontsize=14)
                R_F.set_ylabel('Resistance (  )', fontsize=14)
884
                R_R.plot(np.asarray(real_data), np.asarray(imag_data), 'r.', label="
    Experimental")
886             R_R.set_xlabel('Resistance (  )', fontsize=14)
                R_R.set_ylabel('Reactance (  )', fontsize=14)
888


890             f.canvas.draw_idle()
                #phantom.canvas.draw_idle()
892             #r.canvas.draw_idle()

894         return False

896 ########################################################################
    ######################### Begin Main Program ###########################
898 ########################################################################
```

```
if __name__ == "__main__":
    window_state = True
    serial_port_state = False

    i = 0

    app = GUI()
    app.protocol("WM_DELETE_WINDOW", callback)

    while True:
        if not window_state: # Ensuring we haven't exited the program in the meantime
            break

        while True:
            if reset:
                if quit_:
                    window_state = False
                    app.destroy()
                    break

                if serial_port is not None:

                    if not serial_port_state: # Don't want to initilise more than once
                        ser = serial.Serial(serial_port, 9600, bytesize=8, parity='N', stopbits=1, timeout=0.5)
                        serial_port_state = True
                        app.frames.get(graphPage).update_hydration_status("Press start button.")

                    start = ser.readline().strip()
                    if start.decode("utf-8") == "Config done":
                       reset = False
                       app.frames.get(graphPage).update_hydration_status("Analysing...")
                       break

                    app.update_idletasks()
                    app.update()

                else:
                    app.frames.get(graphPage).update_hydration_status("Select serial port.")

                    app.update_idletasks()
                    app.update()

            else:
                if not quit_:
                    app.update_idletasks()
                    app.update()
                else:
                    window_state = False
                    app.destroy()
                    break

        if not window_state: # Ensuring we haven't exited the program in the meantime
            break

        while True:
```

```
954             if reset:
                    break
956
                if not quit_:
958                 app.update_idletasks()
                    app.update()
960             else:
                    window_state = False
962                 app.destroy()
                    break
964
                status = update_plots(i)
966             i += 1

968             if status:
                    break
970
    ser.close()
```

Code/Hydration_Monitor_GUI.py