ELECTRICAL & ELECTRONIC ENGINEERING
STELLENBOSCH UNIVERSITY

DESIGN (E) 314
TECHNICAL REPORT

Digital Multimeter and Signal Generator

Author:
Lee

Student Number:
Johnson

27 May , 2022

Plagiaatverklaring / Plagiarism Declaration

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.
2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.
3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.
4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.
5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

	24058661
Handtekening / *Signature*	Studentenommer / *Student number*
L.S.Johnson	27/05/2022
Voorletters en van / *Initials and surname*	Datum / *Date*

Abstract

The purpose of this report is to put the complete design of a Digital Multimeter and Signal Generator into words. The report investigates the design and implementation of the hardware and software portions of the project by deconstructing the all the sub systems that make up the entire device and studying the methods used to implement them. We start by giving a high-level description of the system and the user requirements that need to be implemented. The hardware section investigates each hardware element of the system. It describes the hardware elements, design description, design choices and calculations that were used to implement the elements. Circuit diagrams and schematics are also included as visual aids. The software section uses flow, state and other diagrams to convey the logic used to implement software systems that run the device. We also discuss the pin configurations for the STM32 microcontroller that needed to be done for the software to run properly. We then discuss methods for measurement and testing to see which systems work correctly and how well they were implemented before discussing shortcomings of the project and possible improvements.

Contents

List of Figures

List of Tables

List of Abbreviations

PDD Project Description Definition

LCD Liquid Crystal Display

LED Light Emitting Diode

GPIO General Purpose Input Output

ADC Analog-to-Digital Conversion

DAC Digital-to-Analog Conversion

PCB Printed Circuit Board

TIC Test Interface Connection

UART Universal Asynchronous Receiver-Transmitter

List of Symbols

Ω Ohms, S.I unit for resistance

V Voltage across a component

A current through a component

1 Introduction

The system I have designed and implemented has two main functionalities. Firstly, it is a digital multimeter that can measure both DC and AC Voltage as well as current within at least 5% accuracy. Secondly, the system also acts as a signal generator that can produce a DC, sinusoidal or pulse waveform whose parameters can be varied by the user either through serial communication or by using a menu system on the display that is also provided and using the buttons to traverse through it. This report goes through entire design process for the many sub systems that make up the complete project and gives the reader an in depth understanding of the inner workings of the system. Someone with sufficient knowledge should be able to read this report and be able to recreate the system themselves.

2 System description

The system will work with a 9V power supply, but in order to power its own hardware components, the power is regulated down to two separate 5V and 3.3V power supplies. During the measurement mode, the system will be able to measure a DC or AC voltage between 0.1V and 2V and a DC or AC current between 0mA and 9mA. During signal generation, the system will be able to generate a DC, sinusoidal or pulse signal with configurable parameters such as offset (between 0.1V and 3.2V), peak-to-peak amplitude (between 0.1V and 3.2V), frequency (at 0Hz, or between 10Hz and 5kHz) and the pulse duty cycle (between 0% to 100%). The device will also make use of UART serial communication for both transmitting and receiving messages in the correct format. The system will respond to UART configurations and upon requests, can also send back messages to provide information on the current measurements or system state. An LCD is used as the main display for the system and consists of three core elements, the display of the active measurement, the display of the active signal generated and the menu system which can be traversed by the user by using push buttons. These push buttons will be able to change operating modes, select measurement type, set output signal parameters and turn the output on or off all in the menu system.

3 Hardware design and implementation

3.1 High Level Description

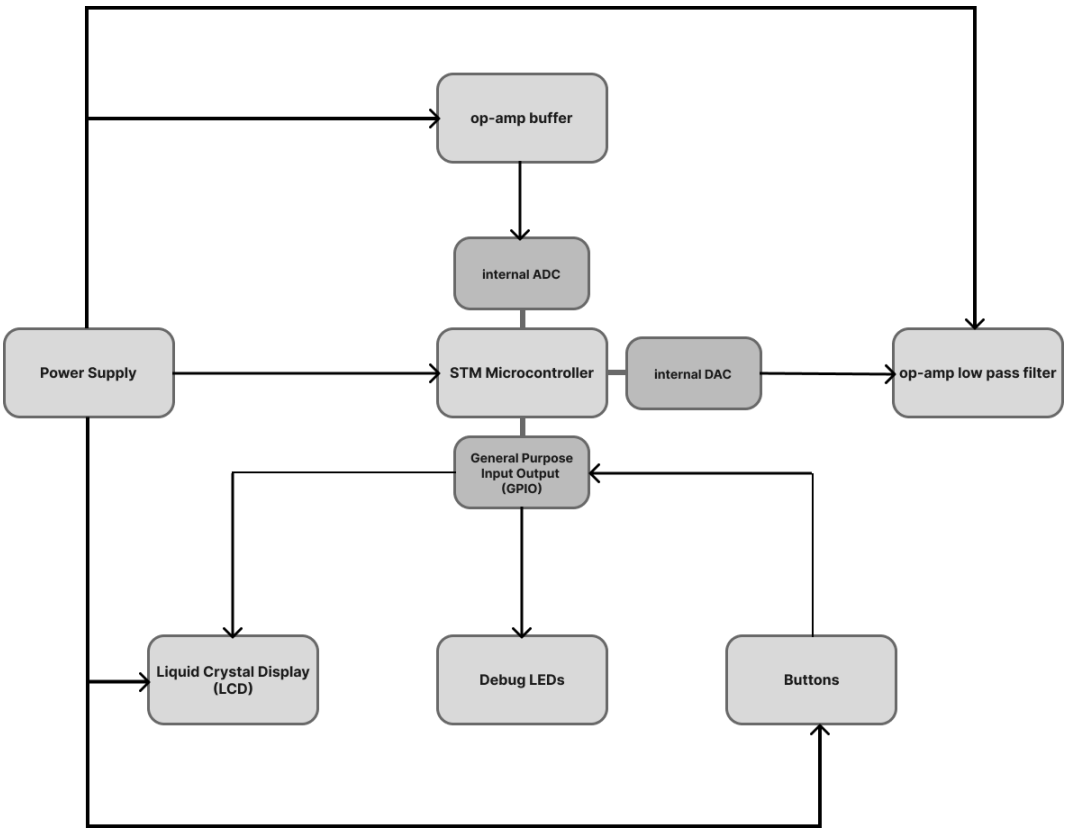

Figure 1: High Level Hardware Implementation Block Diagram

The core to the system's hardware is the STM microcontroller and its peripherals which control or are controlled by the various hardware components. Power is provided through the power supply which consists of two options for voltage regulation which regulates the voltage down to the required voltage for the components. The power supply powers all the hardware components including the STM microcontroller. A single operational amplifier is used as both a buffer and a low pass filter which interacts with the internal ADC and the internal DAC of the microcontroller respectively. The LCD is controlled by the GPIO output ports of the microcontroller in order to display the required information. The debug LEDs are also controlled by the various GPIO output ports in order to provide information about the system state. The system buttons are connected to the microcontroller's GPIO input ports in for system control.

3.2 Power supply

The system is powered by a 9V battery source in general use, however during development and testing the use of a bench power supply at 9V also adequately powers the board. Power can either be provided through a barrel jack and wires or through a 9V battery connector. Two voltage regulators, 5V and 3.3V, needed to be implemented for the various hardware components to be powered correctly and not be overpowered. These two voltage regulators were chosen as most of the implemented hardware runs at either 5V or 3.3V. On the PCB these 5V and 3.3V supplies are ran to various jumpers across the board to ensure that the hardware components have easy access to the supply.

3.2.1 5V Power Supply

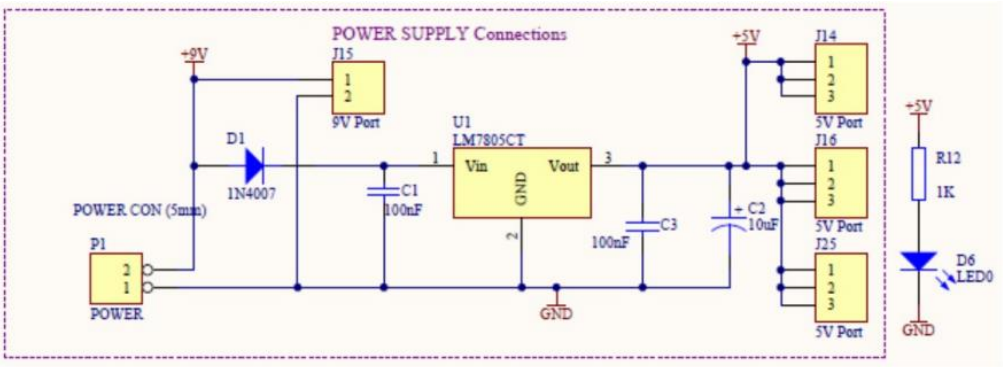

Figure 2: 5V Power Supply Circuit and Indicator LED

The 5V power supply circuit makes use of the LM7805CT regulator in a TO220 package, this regulator regulates the voltage down from the input voltage, 9V, to the desired 5V. The above circuit was provided in the PDD and thus no design or calculations were required, the circuit only needed to be implemented on the PCB. An LED is also implemented in order to display when the 5V power supply is actively supplying power. The design and values were given in the PDD.

3.2.2 3.3V Power Supply

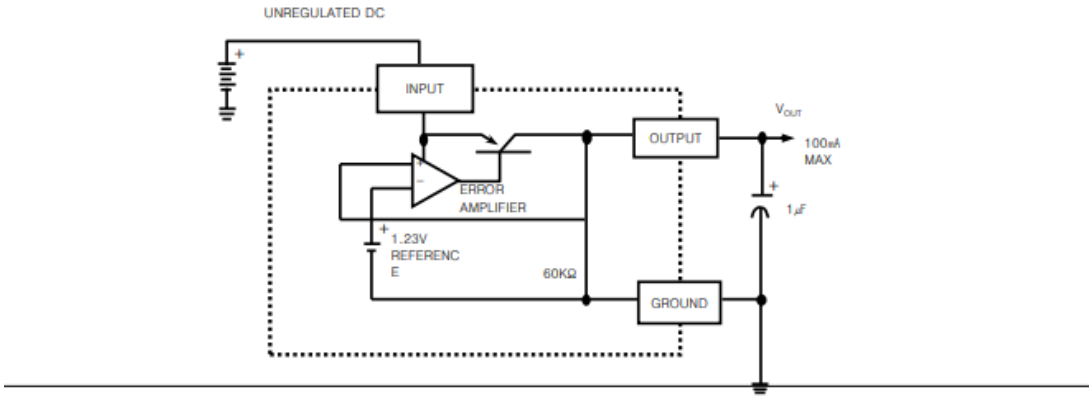

Figure 3: 3.3V Power Supply Circuit

The above circuit was obtained from the LM2950 datasheet and the decision to use this specific circuit was made since no design decisions or calculations would be required for implementation as all values are already provided. The circuit is also supremely simple to both understand and implement. The LM2950 regulator is provided within a TO-92 package. The 5V voltage is taken as the input and the above circuit regulates it down to 3.3V for use by other hardware components.

3.2.3 Voltage requirements

The following table discusses the voltage requirements by each hardware component and as such, these requirements will not be mentioned again in the report.

Table 1: Hardware Component Voltage Requirements

Hardware Component	Operating Voltage
STM 32 Microcontroller	5V
Operational Amplifier	3.3V
LCD	5V
Debug LEDs	3.3V (From Microcontroller GPIO Outputs)
Buttons	3.3V

3.3 Buttons

In this project, five active low push buttons were implemented. The buttons are required to send a low input signal to the GPIO input pins on the nucleo board and a high input when they are not pressed.

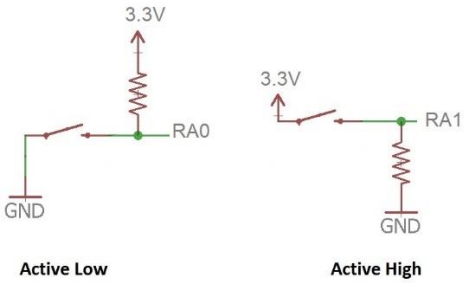

Figure 4: Active Low and Active High Button Circuit Implementation

The resistor between the input voltage and ground and the GPIO input prevents us from shorting the power supply directly to ground. This resistor may either be implemented in hardware or in software by making use of the nucleo board's internal pull-up resistor function at the GPIO input pins.

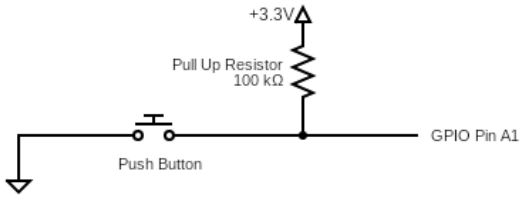

Figure 5: Final design active low circuit for first push button

The first push button's implementation is shown above. We need to select a resistor value for the circuit. We want a resistor that is high enough to stop too much current from flowing into the GPIO input, but low enough so that it does not lead to an unnecessarily high time constant (Every component has parasitic capacitances, thus a higher resistance will lead to a higher time constant $\tau = RC$, which will lead to the button having a slower rising curve). We would want our button to respond within at most a millisecond. The lecturer recommended a resistor with a value of between 20-100 $k\Omega$, so, this is where I started.

I started my design with a 100 $k\Omega$ resistor. By using the oscilloscope, I was able to measure a response time of 24.4 microseconds, which was quick enough and meant the time constant was still quite low. Thus, I will be using the 100 $k\Omega$ resistor in my push button circuit.

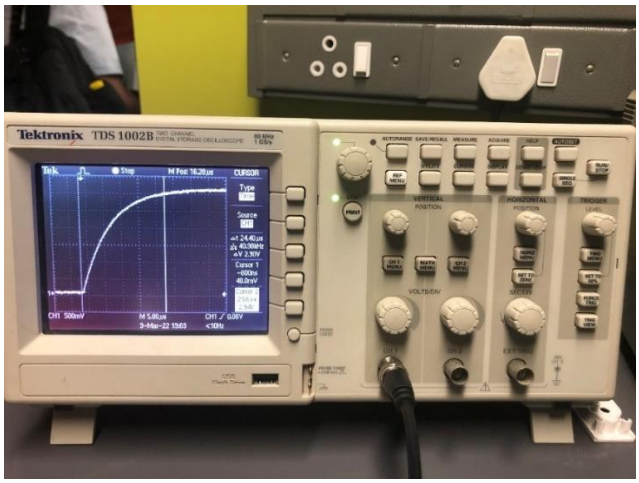

Figure 6: Using the Oscilloscope to measure the button response time

For the other 4 push buttons I implemented, I decided to make use of the internal pull-up resistors instead. This decision was made as it allowed me to implement a simpler design that did not require any calculation as all calibration was automatically done in software by the Nucleo board.

Pin selection on the Nucleo board was done in a way to not only interfere with any pins that were already being used but also to make software implementation easier. I knew that the buttons would be implemented with interrupts in software and thus each button should be connected to a different pin number and their pin group (A, B, C, D) would be irrelevant (see the button software section for more information). Thus, the following pins were chosen to be button inputs: PA1, PB6, PA7, PB8, PB9.

3.4 Debug LEDs

The debug LEDs turn on or off depending on whether we have a high or low output from our GPIO pin on the nucleo board. In the following paragraphs I will describe the design of one LED and the same principles will apply to the rest. We need to ensure that we do not draw more current than what the GPIO pin provides, which is 8 mA. The LED indicates a state change; thus, it needs to be bright enough for us to be able to see if the LED is on or off but does not have to be so bright that it draws unnecessary amounts of current.

The design problem for the LED circuit is the choice of a resistor that will be able to go in series between the GPIO output pin and the LED and control the current flowing through the circuit, and thus the LED brightness as well. An effective way for us to select this resistor is by using a debug circuit where we can select a resistor based on trial and error. We build this debug circuit onto a breadboard and connect it to a DC power supply where we will be able to change the input voltage.

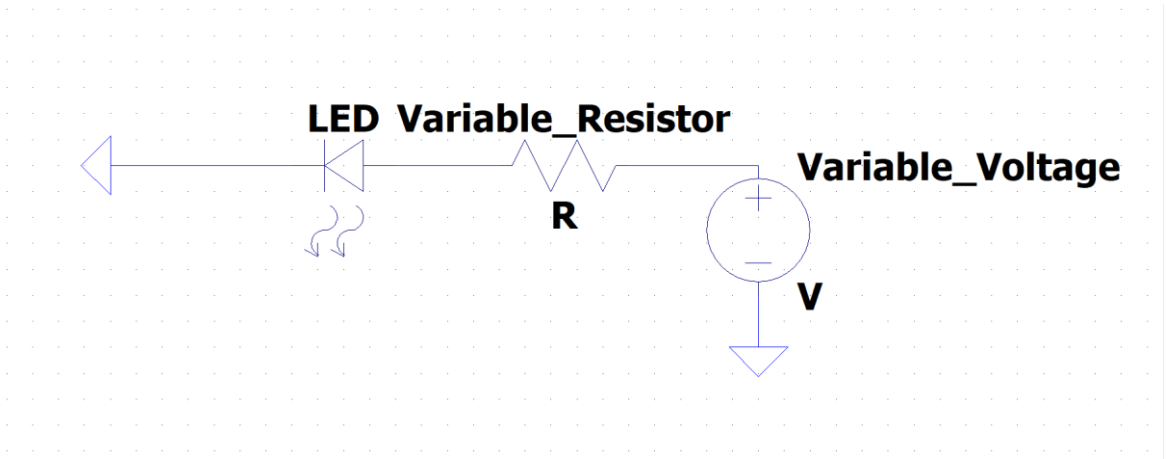

Figure 7: LED design debug circuit

The GPIO pin outputs 8mA but we would like to use much less than that for safety reasons and to ensure that we do not short the LED. By using the debug circuit, the first arbitrary resistor value I connect is 680Ω. I slowly increase the input voltage from the DC power supply until I believe the LED is bright enough for anyone to see a noticeable change between on and off. Using my multimeter, I can measure the voltage across the resistor as well as accurately measure the correct resistance and using Ohm's law I am able to get the current flowing through the resistor, and hence the LED (As the LED and resistor are connected in series). With a measured voltage drop across the resistor of 1.41V and a measured resistance of 680Ω, I obtain:

$$I_{LED} = \frac{V_{Resistor}}{R} = \frac{1.41}{680} = 2.08\ mA$$

This amount of current is well below the 8mA that the GPIO pin outputs, therefore the selected resistor in series with the LED will work perfectly.

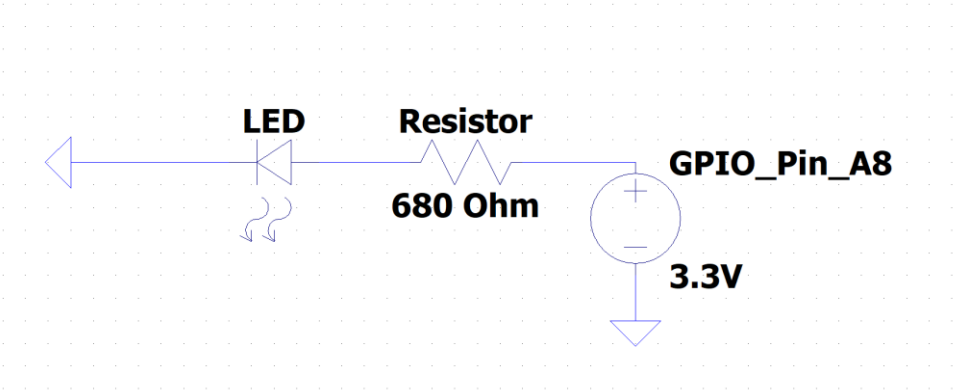

Figure 8: Final design for the Debug LED circuit

This LED circuit was implemented for each of the four debug LEDs. As the pins for the LEDs had no other requirements besides being GPIO output pins, they were selected to be as generic as possible and not block other hardware components from accessing more limited functionality. Thus, the following pins were chosen to be outputs for the debug LEDs: PA8, PB10, PB4, PB5.

3.5 ADC (Input Stage)

Our ADC consists of two stages, an input signal conditioning stage and the STM microcontroller's internal ADC circuit stage. The internal ADC circuitry will be configured via software and is not discussed here. We need to design the input signal conditioning stage which will consist of an op-amp buffer, through which the input signal will travel through in order to protect the microprocessor. The operational amplifier that is used to implement this buffer is the MCP602

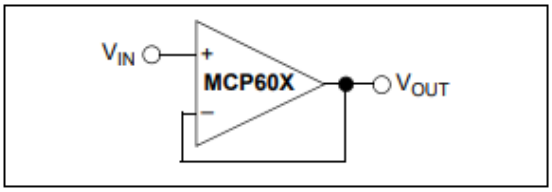

Figure 9: ADC Input Buffer Circuit

The buffer circuit above was found in the datasheet for the MCP602 and was chosen due to its supreme simplicity and few components required. As discussed in the voltage requirements section, V_{DD} = 3.3V and V_{SS} = ground. The input signal is provided through the TIC. An 8-pin socket was soldered onto the PCB to allow us to insert and remove the MCP602 from the PCB with ease instead of soldering it directly onto the board itself. This way if the IC is damaged it can easily be removed. With the above circuit few design choices and calculations are needed due to its simplicity.

Pin PA0 was used as the input pin for the ADC. This decision was made as the first input line of the internal ADC circuit can be found at pin PA0.

3.6 DAC (Output Stage)

Our DAC consists of two stages, the STM microcontroller's internal DAC circuit stage and the output signal conditioning stage. The internal DAC circuitry is configured via software and is not discussed here. We need to design the output signal conditioning stage which will consist of an op-amp low pass filter through which the DAC output will travel, and experience gain and smoothing so that a smooth output signal is outputted. This signal conditioning stage will also be implemented on the MCP602 operational amplifier. The MCP602 has 8 pins, meaning that while the ADC signal conditioning stage can be implemented on the one side of the IC, this stage can be implemented on the other.

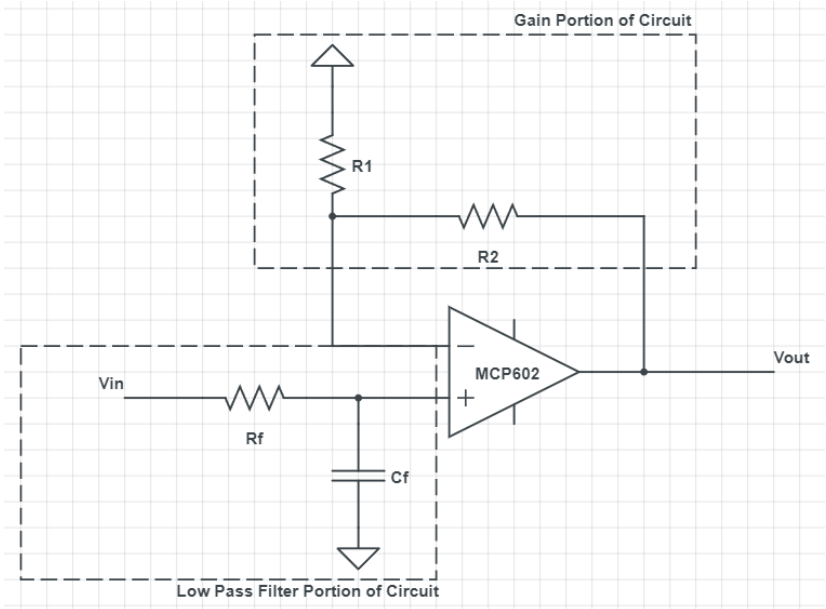

Figure 10: First order low pass Butterworth filter used for DAC output signal conditioning

For the lowpass filter I decided to use a non-inverting first order low pass Butterworth filter due to its simplicity, common use and my previous experience working with the circuit. As discussed in the voltage requirements section, V_{DD} = 3.3V and V_{SS} = ground. The input signal comes from the STM microcontroller's internal DAC circuit and the output goes to the TIC. The MCP602 is connected to an 8-pin socket.

As seen from figure 10, the signal conditioning op-amp circuit can be split up into two parts, the gain portion and the low pass portion. The op-amp gain is provided through the equation

$$G = 1 + \frac{R1}{R2}$$

I will choose to have a gain of 2. The reason I make this decision is because a gain of two makes choosing the resistors very simple as I would just choose the two resistors to be equal and it is also easily dealt with in software. Thus, I choose $R1 = R2 = 10k\Omega$, which yields

$$G = 1 + \frac{10000}{10000} = 2$$

For the filter portion of the circuit, for design we must decide what we want our low pass cut-off frequency to be and then select appropriate values for R_f and C_f to achieve that cut-off.

I chose to design for a low pass cut-off frequency of approximately 10.5 kHz. I made this decision as I believe that it will allow most of the signal that we want to output through but will be able to sufficiently cut-off enough frequencies that will cause the signal to be glitchy and unsmooth. We design the lowpass filter to smooth the output. We know the relationship between frequency and the time constant is described by

$$f = \frac{1}{2\pi\tau}$$

With $\tau = R_f C_f$

To obtain a cut-off frequency of 10.5kHz,

$$\tau = \frac{1}{2\pi f} = \frac{1}{2\pi(10500)} = 15\mu s$$

Thus, we choose $R_f = 10k\Omega$ and $C_f = 1.5nF$ which yields our desired time constant. We have thus completed the design for our DAC output signal conditioning stage.

Pin PA4 was used as the output pin for the DAC. This decision was made as the first output line of the internal DAC circuit can be found at pin PA4.

3.7 LCD display

The liquid crystal display (LCD) that is used in this project is a 1602A, which is a 16x2 character display, with $Vcc = 5V$ and $V_{SS} = 0V$. The LCD will be used to display various states and formats that is discussed more in depth in the software section. We can consider the hardware design of the LCD to take shape in 3 separate stages: Designing for the LCD backlight, designing for the LCD display contrast and the connection of the LCD's data lines.

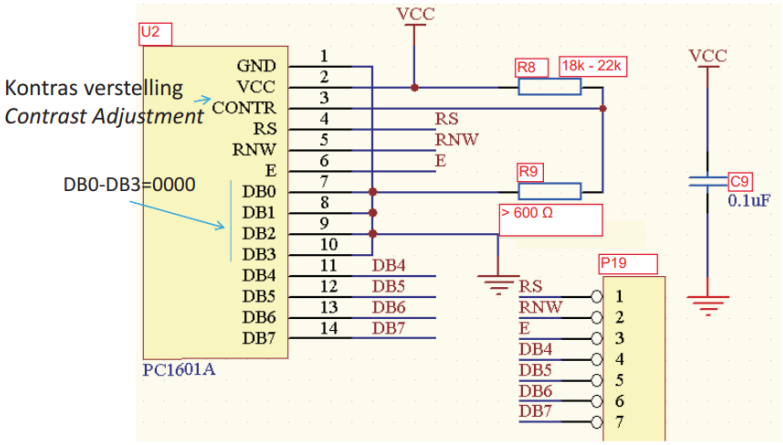

Figure 11: LCD Circuit for the 1601A

Although the above circuit applies to the 1601A, the hardware implementation is the same for the 1602A.

3.7.1 Backlight

This specific LCD model has a display that is difficult to read when the backlight is not turned on, and hence we design for the backlight to be always on. We design the backlight first because once the backlight is on it becomes easier for us to design and test the contrast and data lines as well. The backlight itself is just another LED and thus its design is like the design we performed for the debug LEDs.

We want the backlight to clearly turn on; however, we want to limit the current into the LED to be below 10mA. We are given that the typical voltage drop across the backlight is approximately 4.1V. Using this information combined with the fact that we will use a 5V input voltage, we can select a resistance that will yield a current less than 10mA. Using the equation

$$I_{backlight} = \frac{V_{backlight}}{R} = \frac{5 - 4.1}{R}$$

We choose an arbitrary resistance value of 150Ω and calculate the backlight current to be

$$I_{backlight} = \frac{0.9}{150} = 6mA$$

This is lower than our maximum current rating of 10mA and thus our resistance selection is good, and we will make use of a 150Ω resistance in series between the input voltage and LED backlight.

3.7.2 Contrast

As we can see from figure 11, on line 3 we have a contrast adjustment pin. Here we can adjust the two resistors R8 and R9 in order to obtain a suitable contrast on the display. It needs to be understood that every LCD is unique, and the contrast adjustment values on one LCD may not work on another. As soon as the LCD backlight is on, we can see the display blocks and can use those as a measure for our contrast.

According to the data sheet, R8 should be between $18k\Omega$ and 22kΩ, while R9 needs to be larger than 600Ω.The lecturer had informed us that when our contrast was successfully adjusted, the first line would be barely visible. By building a simple debug circuit on a breadboard and connecting it to the contrast adjustment pin of the LCD, I was able to test various resistor combinations until the contrast was adjusted to the correct level (I discuss this in more depth in the measurements and testing section).

At the correct contrast level I found,

$R8 = 22k\Omega \; and \; R9 = 4.7k\Omega$.

3.7.3 Pin Connections

From figure 11, we can see that the first 4 data lines (DB0-DB3) of the LCD are hard wired to ground, which means the LCD will operate in nibble (4-bit) mode. Hence, there are 4 data lines (DB4-DB7) and 3 control lines (Reset, Read Not Write, Enable) that need to be connected to the microcontroller appropriately. We will connect all of these to GPIO output ports of the microcontroller, thus giving the microcontroller full control over the LCD. Similarly, to the debug LEDs, as the pins for the LCD inputs have no other requirements besides being GPIO output pins, they are selected to be as generic as possible and not block other hardware components from accessing more limited functionality. Thus, the following pins were chosen to be outputs for the LCD: Reset = PB13, Read Not Write = PB14, Enable = PB12, DB4 = PA11, DB5 = PA12, DB6 = PC6, DB7 = PC8.

4 Software design and implementation

4.1 High Level Description

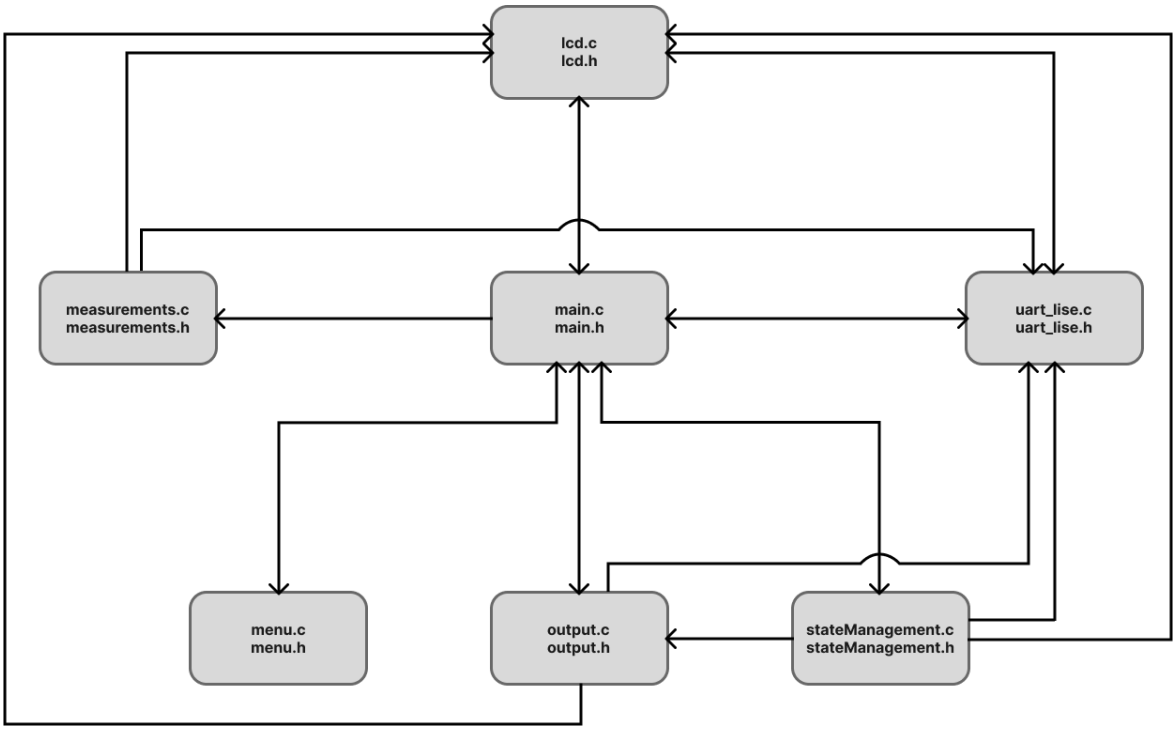

Figure 12: High Level Software Implementation Block Diagram

In the above block diagram, describing the software implementation of the project at a high level, each block represents a source file and its paired header file and displays how each of these files interact with one another in order to form the base functionality of the system. There are more files in the system such as the STM32 base libraries but those were not included as I have not made any modifications to them, even the interrupt file was left untouched.

The software core is found in the main file, where all the functionality from the other files is brought together in order to run the system. All initialisations, pin configurations and interrupt callback functions (for button presses and timer period elapsed) take place in the main file. The UART receiver is also primed in the main file. Various flags are set up in the main function that handle LCD, state change and UART functionality in the infinite while loop. DAC functionality from the output file is also made use of in the infinite while loop.

Although the core of the software system is the main file, in terms of importance, a close second goes to the state management system. The state management file keeps control over the system's input, output and LCD states and handles the state dynamics (such as when the state changes from one state to another).

The LCD file consists of the LCD driver that we define and takes in information from our output (DAC), measurements (ADC), state management and UART files in order to display the correct information in the correct state.

The UART file handles the entire UART protocol and works in tandem with the state management system, the measurements file (ADC), the output file (DAC) and the LCD driver in order to correctly process configurations/commands and send the correct data.

The menu files deal with the operation of the menu on the LCD using the push buttons to traverse through it, thus, it works together with the LCD driver, the state management system as well as the output file to change the DAC's output parameters.

The output and measurement files implement the functionality for the DAC and the ADC respectively. Within the output file we deal with the dynamics of the output system as the different desired output parameters are given and the separate output types according to state. The measurement file deals with retrieving valuable information from the input signal. Both files deal with calibration for the input and output signals.

4.2 Button Bounce Handling

Button debouncing can either be handled in hardware or in software. I've decided to handle it in software in this system as I believe it is simpler and more time efficient to do so. It would also be more consistent if I were to make a number of these systems. The button debouncing that is applied to one push button is applied in the exact same way for the rest of the buttons, and thus I will discuss my design here for one of the push buttons and that will also describe the button bounce handling for all 5.

When the button is pressed, our system changes state or causes movement in the menu, I will consider both to be state changes. However, if we do not correctly implement button debouncing, it is possible that sometimes our state change would be unsmooth or incorrect all together. We need to implement button debounce in code to ensure that the system state only changes every time a button is pressed and is not affected by the mechanical button bouncing which occurs after the press.

To filter out the button bounce, we want to capture the state of the button and the time when the button has a falling edge and then completely ignore any following button inputs within a predetermined amount of time. This amount of time will be predicted through testing the button bounce length with the use of an oscilloscope.

The logic for button bounce handling is shown below using a flow diagram, this entire system is implemented in the main file using two call back functions, one for the external interrupt of the button press, and one for the timer's period ending. The button press logic is then handled in the while loop.

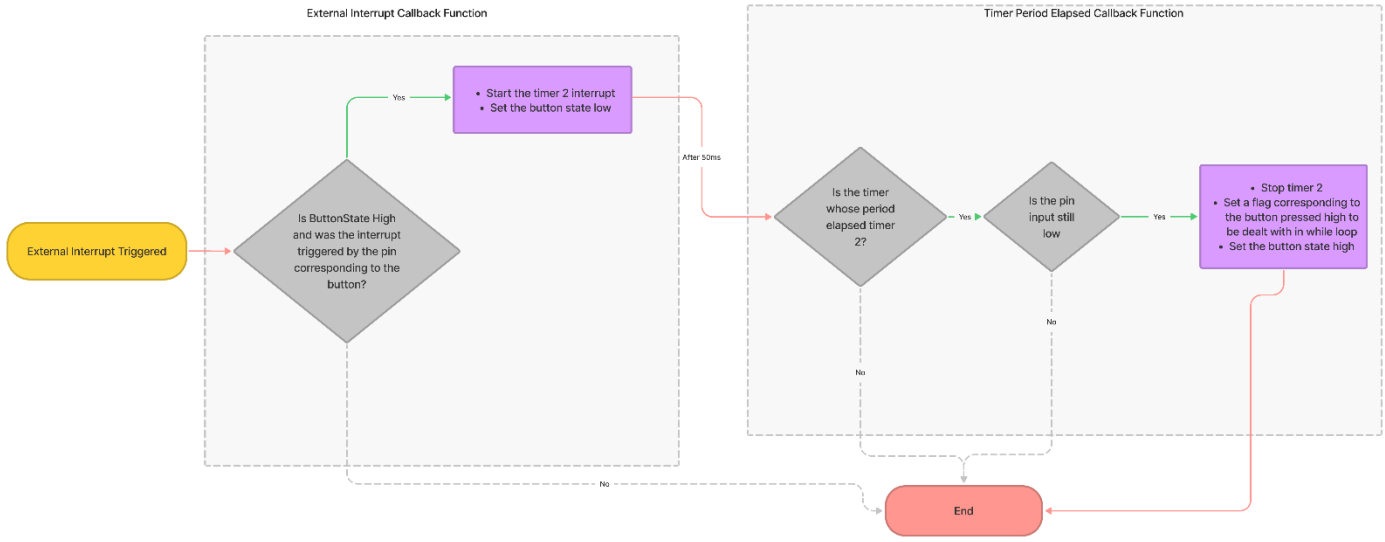

Figure 13: Logic Flow Diagram for Button Bounce Handling

For peripherals, our input pins connected to the push buttons were set up as external interrupts so that when the buttons were pushed, the callback function was called. A timer, timer 2, was also setup to handle the waiting period after a button press for the button bounce and was set up in interrupt mode. Through testing and measurement, discussed in the measurement section of this report, we found that the maximum bounce time was around 1.38ms thus a period of 50ms was chosen as it provided adequate safety for button bounce.

The timer operates at 72 MHz, we set a pre-scaler of 7200 which yields

$$Clock\ Ticks\ per\ Second = \frac{72000000}{7200} = 10000$$

We set our counter period (Auto Reload register) to 500 ticks which yields a period of

$$Timer\ Period = \frac{500}{10000} = 50ms$$

4.3 UART communication (protocol and timing)

All UART functionality is defined in the UART file, and this functionality is continuously being used in the while loop of the main function when it is needed. The UART system will be used to send and receive messages from the TIC. Our UART system will follow a certain protocol as defined in Table 4 and Table 5 of the PDD. This protocol will ensure that sent and received messages have a consistent format and will be well understood by both the TIC and the microcontroller. The UART system makes use of information from the state management system, the LCD driver, the measurements (ADC) system and the output (DAC) system.

The first time the UART system is used is when the system starts up and we send our student number to the TIC. I decided to perform this right before the while loop starts in the main function. Immediately

after this initialization message, I also primed my receiver to work on interrupt mode for the first time. The receiver takes in one byte at a time, this is because our received messages will be of variable length so we cannot assume at the start of the programming what the length of the received message will be.

When it comes to the timing of the UART, it is set up on interrupt mode, which means that as soon as a message is received, dealing with the UART becomes a priority. However, it would be bad practice to deal with the entire processing of the UART in the interrupt handler, thus we rather set a flag in the handler when the entire message has been received and then in the while loop in the main function, we check if that flag is high, at which point we process the received message. We use flags to determine when we need to send messages as well and then these messages will be sent in basic blocking mode. This method allows the rest of the system to continue its processes without having to completely freeze up to deal with the UART. It also allows other interrupts to trigger when dealing with the UART, which is crucial in a system that is as interrupt heavy as this one. Please see the testing and measurement section of the report to see values on the response time of the various UART messages.

In the UART Receive Callback function we can see that we fill an input message buffer with the received bytes until the line feed byte (0x0a or 10) is received, at which point we reset the buffer length and set an input complete flag high to be used in the while loop in the main function. This call back function is implemented in the UART file.

The following flow diagrams describes the logic behind the UART system and how it was implemented.

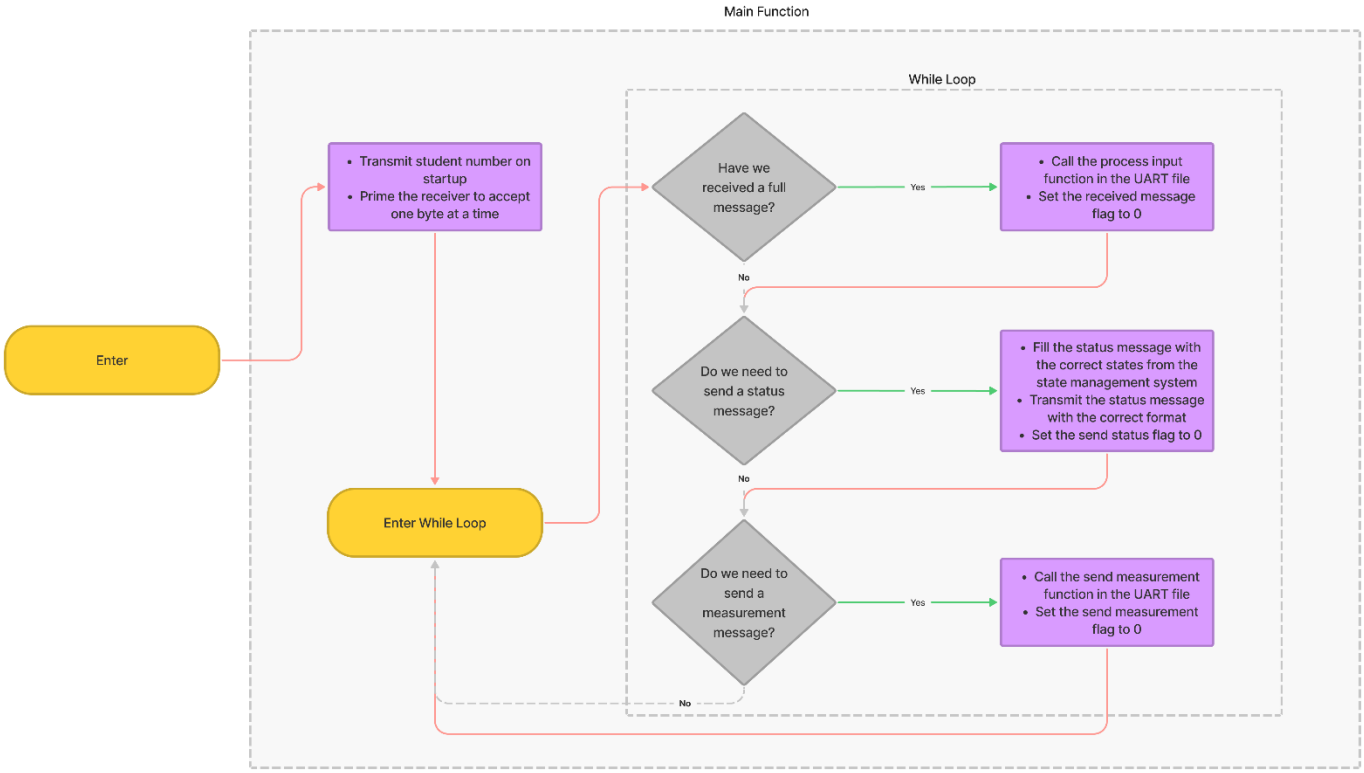

Figure 14: UART flow diagram describing the logic in the main function

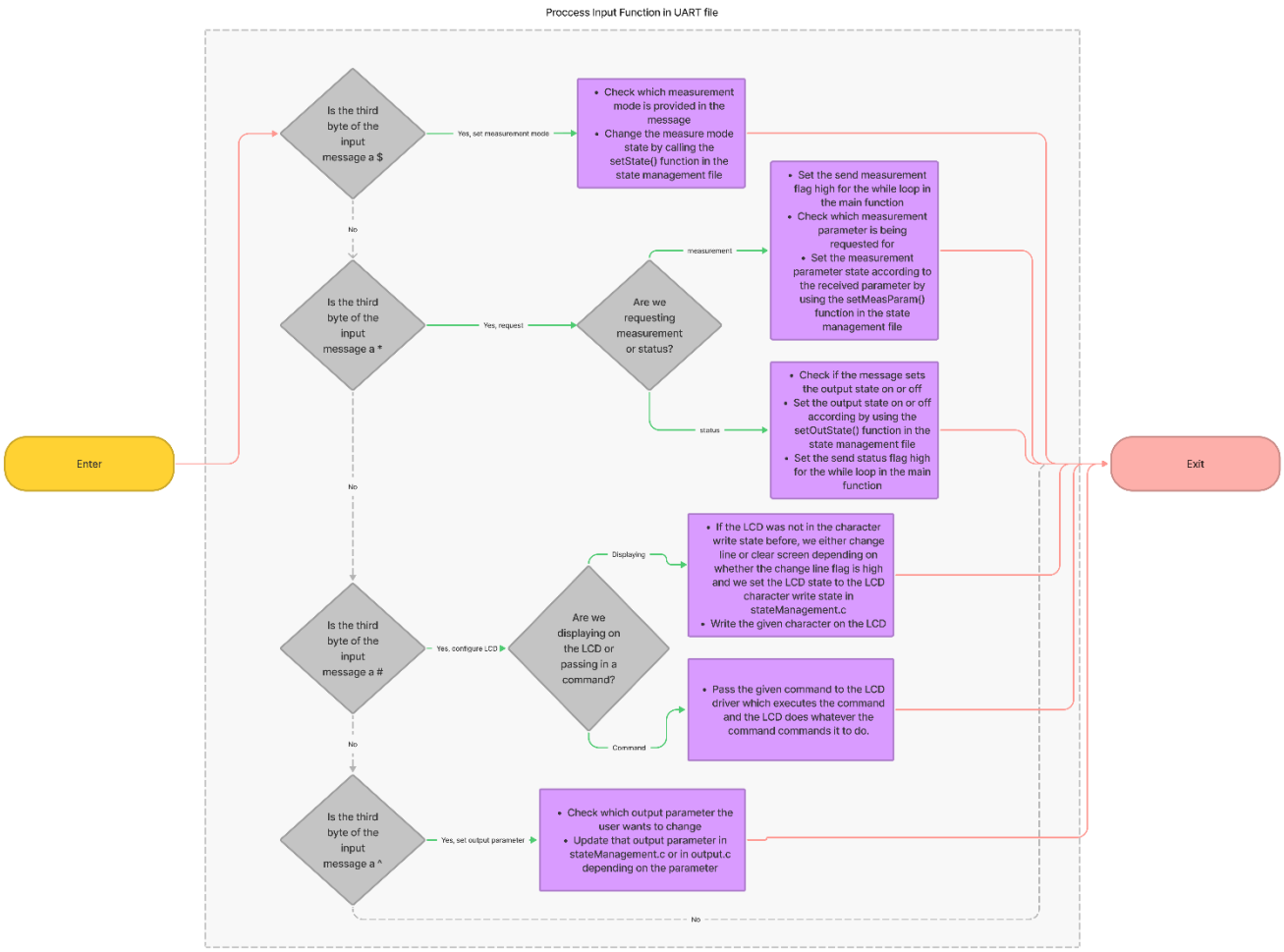

Figure 15: Process input function in the UART file

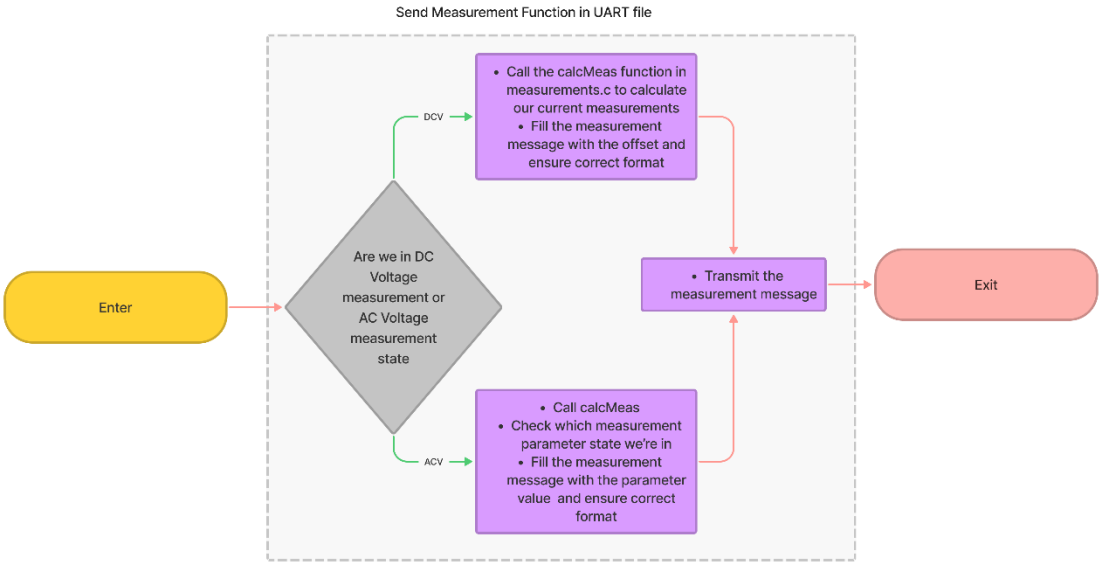

Figure 16: Send Measurement function in the UART file

Pin PA2 and PA3 are set up as the USART TX and RX pins respectively for the STM32F303RE microcontroller that we use and are used with the uart2 handler, huart2. We configure USART2 with a global interrupt so that whenever we receive a byte, we can deal with it immediately no matter what the current state of the system is. We set our Baud Rate to be 115200 Bits/s, with 8 data bits, one stop bit and no parity checking. When transmitting data we make use of blocking mode, while for receiving data we use non-blocking mode.

4.4 ADC, Data Flow and Processing

The ADC's functionality is split up into two different files, the main file, and the measurements file. In the main file we will deal with the initialisation and functionality of the timer, while in the measurements file, we will deal with the data flow and the processing of the input signal to calculate its parameters. The signal is passed into the microcontroller after traveling through a buffer from the TIC. The ADC will be used to measure both DC and AC voltage signals. The measurements file is also often used by the LCD and UART systems for their functionality.

To understand the data flow of the ADC, I have provided a flow diagram which shows how the timer in the main file and sampling function in the measurement file work together to sample the signal. The timer has a period of 50μs and thus we sample the input signal at a frequency of 20kHz.

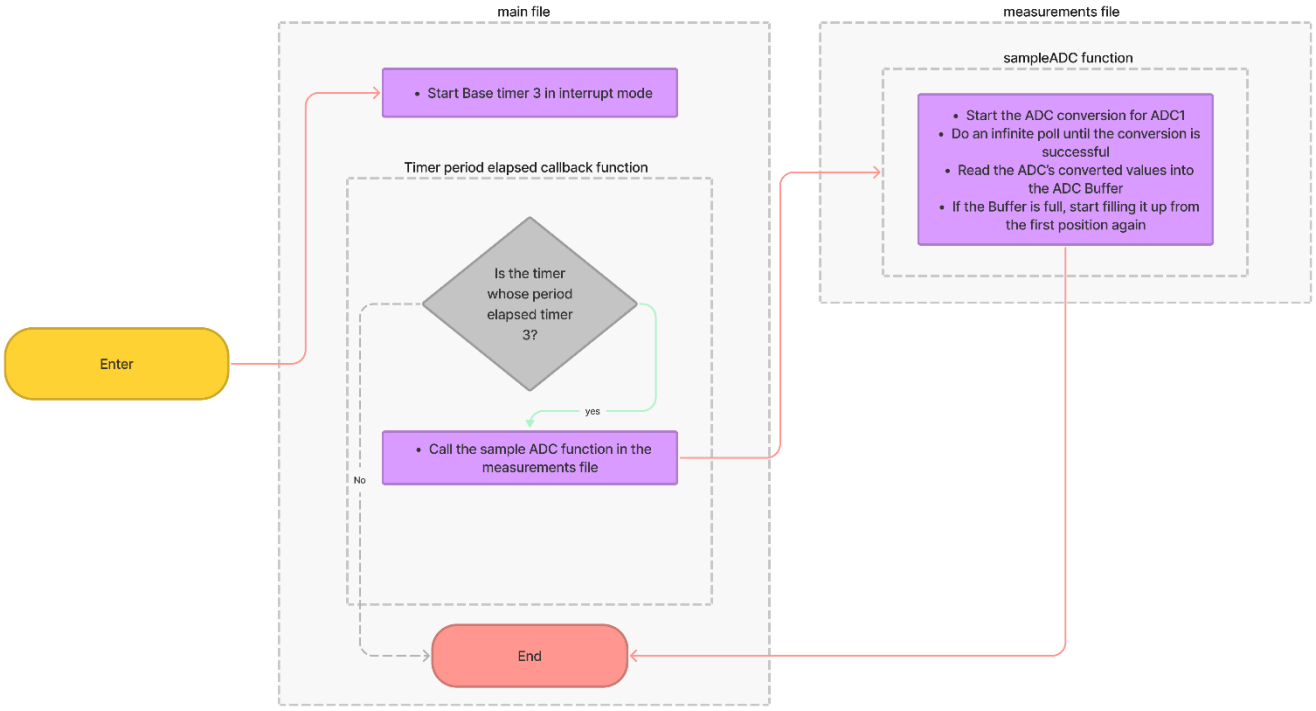

Figure 17: Timer functionality in main and sampling in measurements

The following diagram shows how the processing of the input signal works in the measurement file:

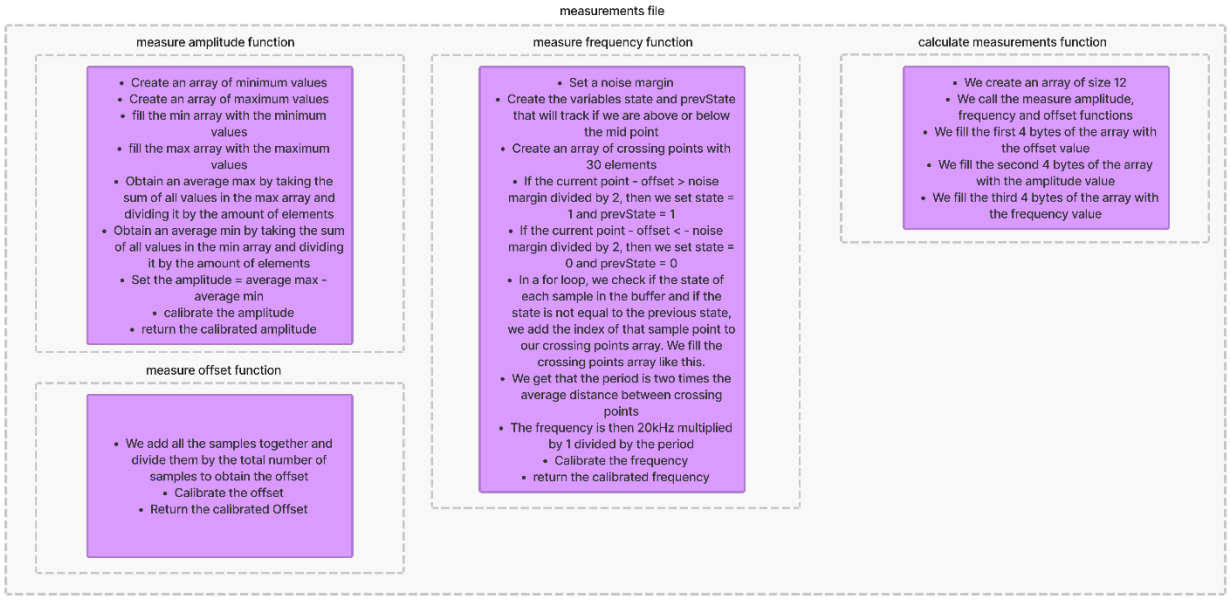

Figure 18: Diagram describing the measurement functions in the measurement file

As you see in the diagram, all the measurement values are calibrated before they are finalized. This is to increase accuracy and is discussed in further depth in the testing and measurement section of the report.

For the peripherals, PA0 was used as the ADC1 input. The ADC was set to 12-bit resolution. Timer 3 was used for the ADC sampling and needed to be set up correctly to obtain the correct sampling rate of 20kHz. The timer was set up in interrupt mode. The timer operates at 72 MHz, we set a pre-scaler of seventy-two which yields

$$Clock\ Ticks\ per\ Second = \frac{72000000}{72} = 1000000$$

We set our counter period (Auto Reload register) to 50 ticks which yields a period of

$$Timer\ Period = \frac{50}{1000000} = 50\mu s$$

This gets a sampling frequency of

$$f = \frac{1}{T} = \frac{1}{50x10^{-6}} = 20000Hz$$

Which is what we wanted.

4.5 DAC, Data Flow and Processing

The system DAC functionality is split up into two files, the main file, and the output file. Within the main file the DAC's trigger, which is set up to correspond to timer four, is initialized. The functions in the output file are also important to the LCD and UART system. Four main functions which are the basis for the DAC's operation are implemented in the output file and each of these functions are continuously called in the while loop in the main function.

```
while (1) {
 // Other functionality in the while loop skipped for display purposes
 dcOut();
 calcDAC();
 acOut();
 pulseOut();
}
```

The DAC should be able to generate a DC, AC or Pulse signal for which the user can modify the signal's offset, amplitude, frequency, and duty cycle depending on the output type. The DAC should also be able to turn on and off. The DAC's output signal is also passed through an op-amp that has been designed and discussed in the hardware section of the report, and thus the software must take the op-amp gain into account when generating a signal.

The following flow diagrams discuss the four main functions in the output file and the logic behind them.

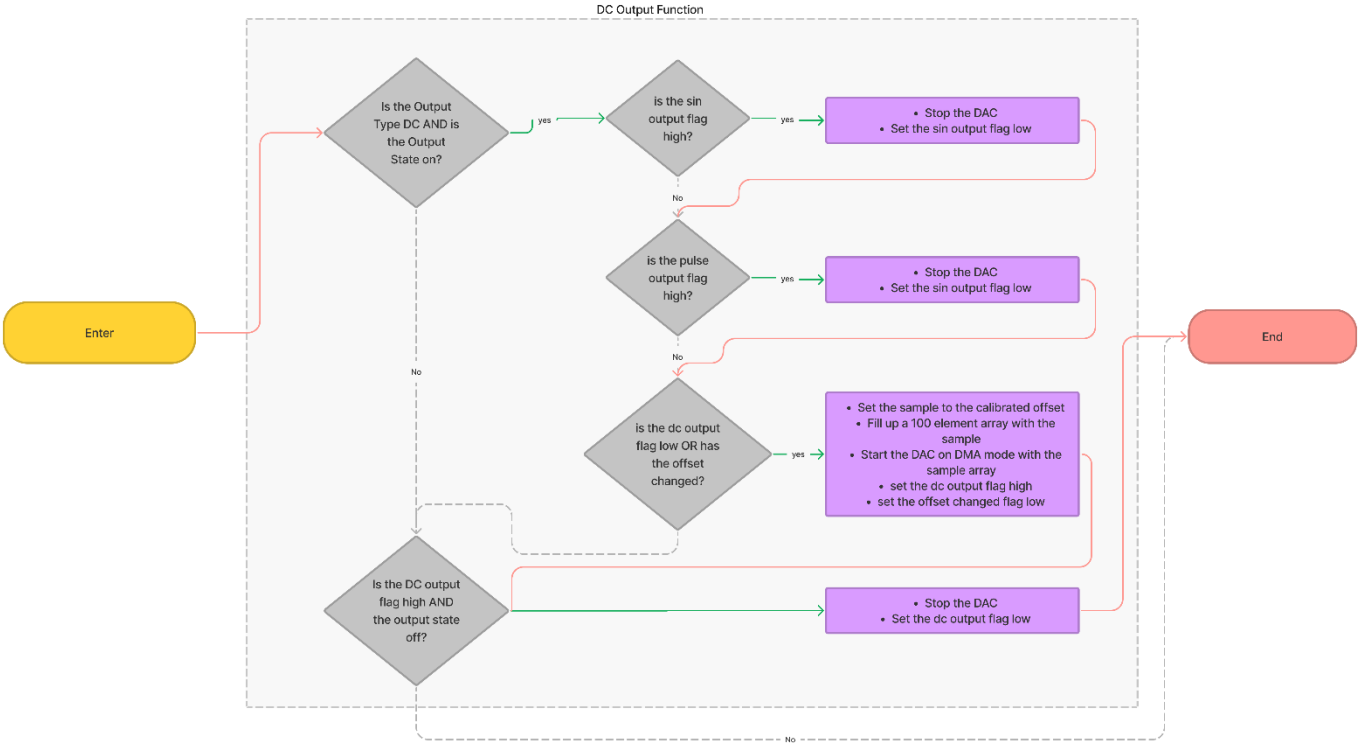

Figure 19: DC Output function in the output file

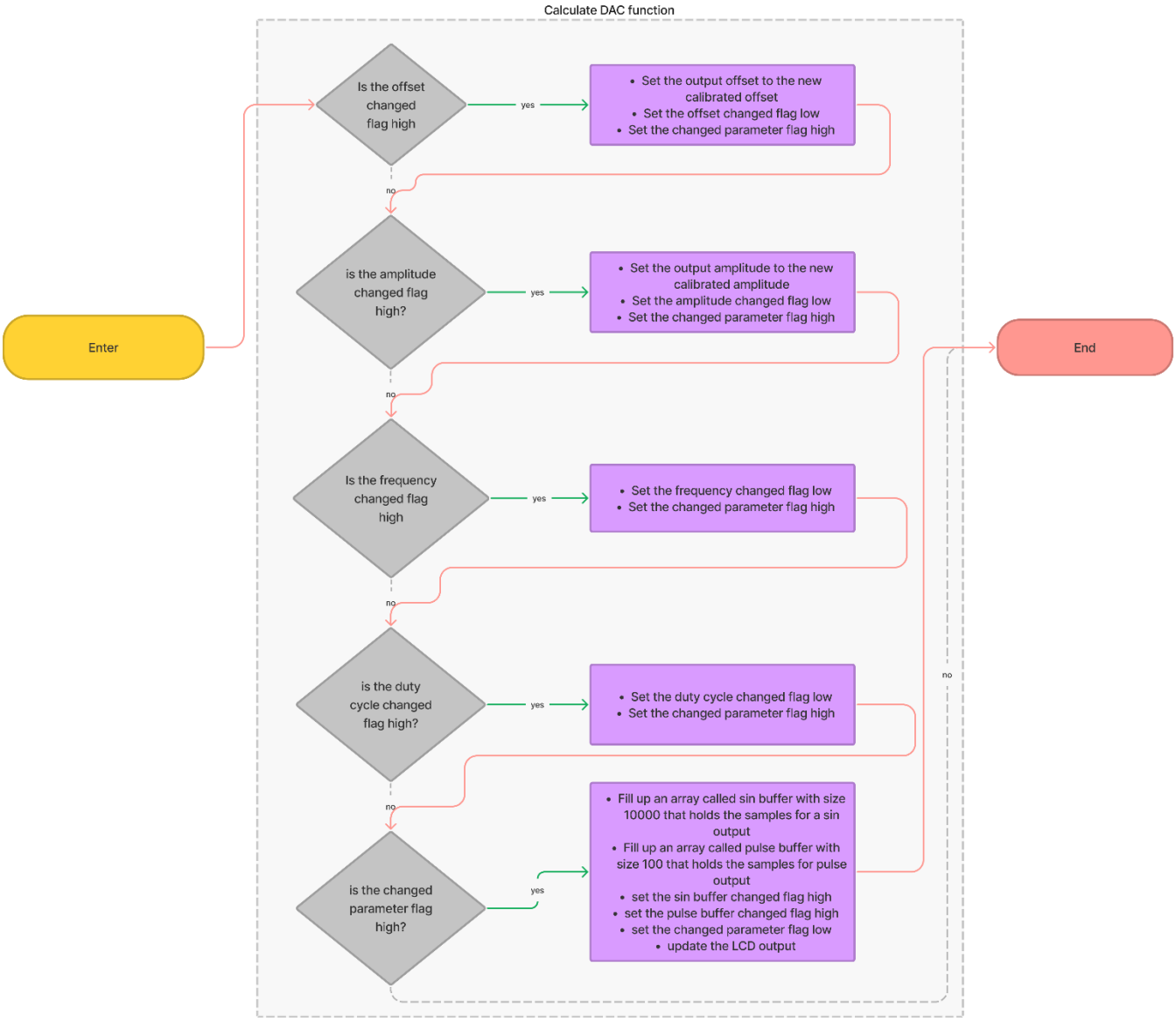

Figure 20: Calculate DAC buffer function in the output file

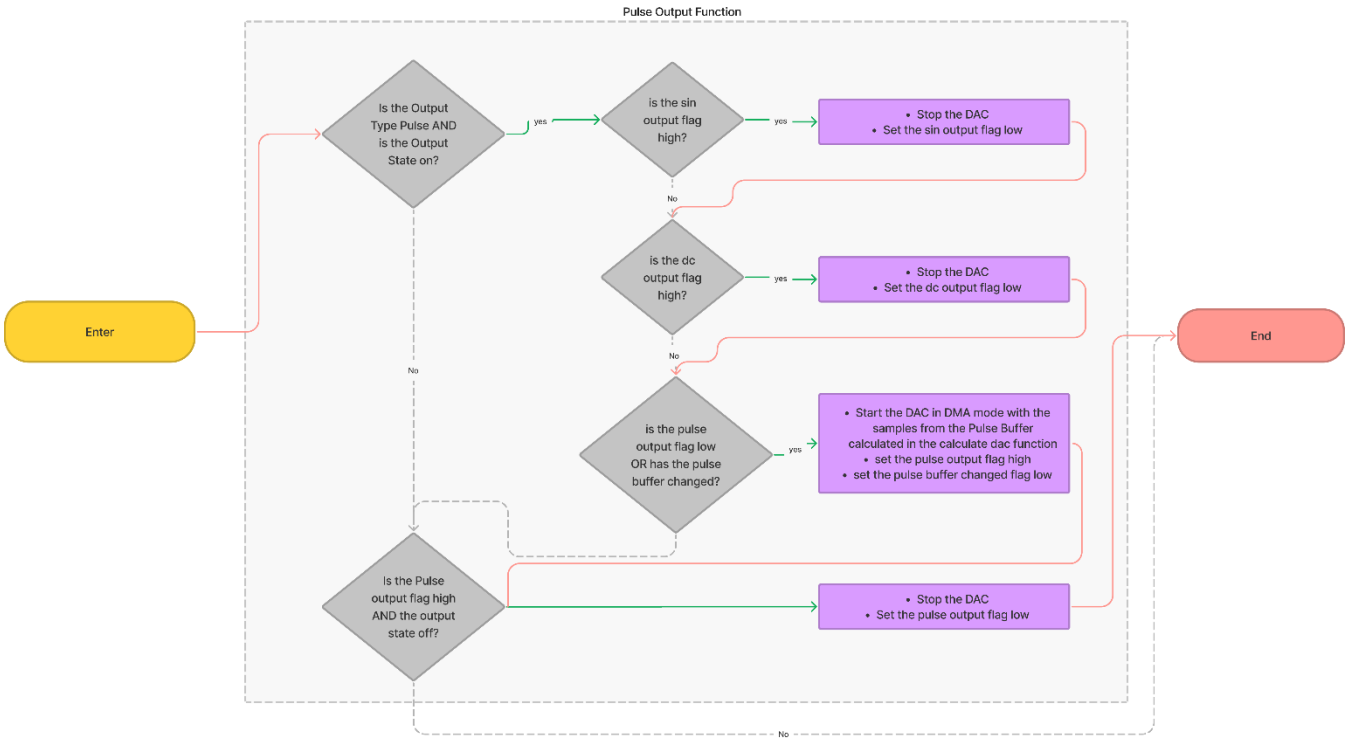

Figure 21: Pulse Output function in the output file

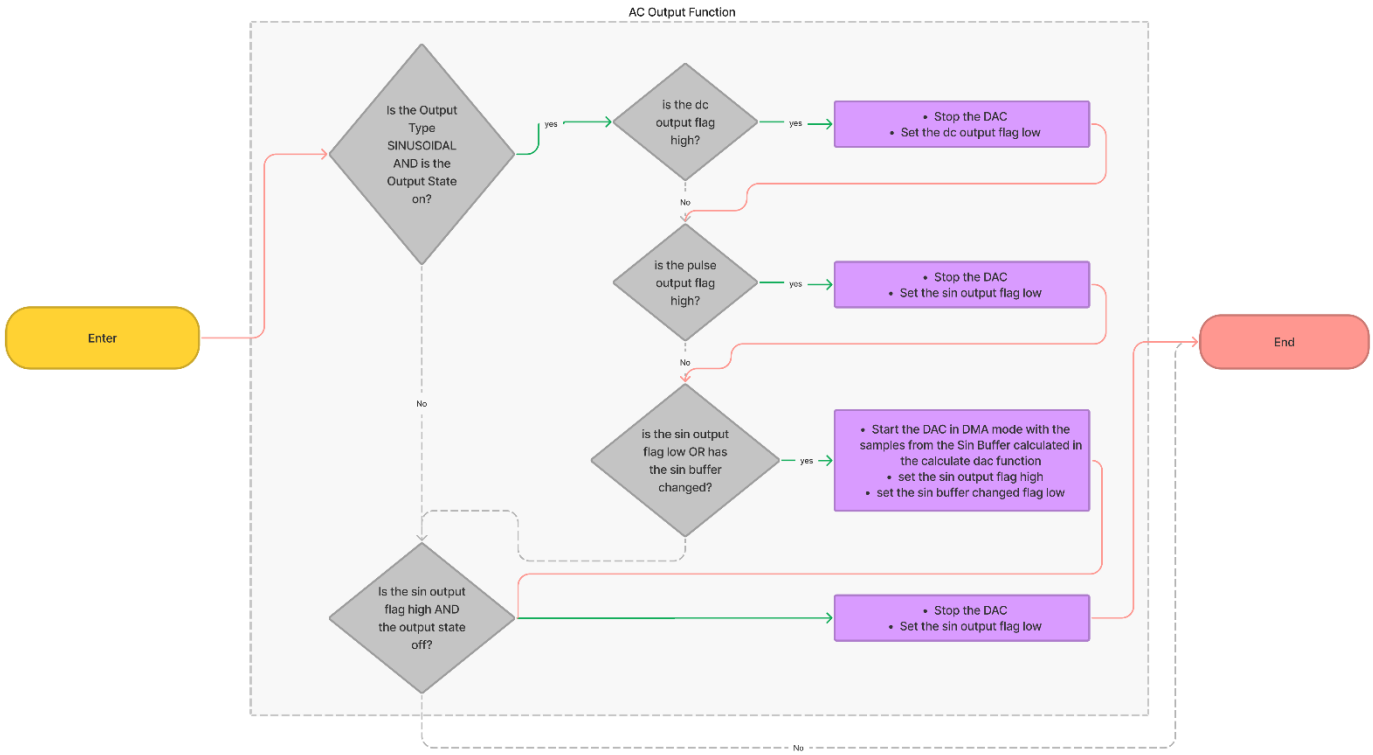

Figure 22: AC output function in the output file

For the calculate DAC function, the sin and pulse buffers are filled up according to the following two equations:

$$y_{SineDigital}(x) = \left(\sin\left(x \cdot \frac{2\pi}{n_s}\right) + 1\right)\left(\frac{(0xFFF + 1)}{2}\right)$$

Figure 23: Equation for the sinusoidal output buffer

$$f_{Sinewave} = \frac{f_{TimerTRGO}}{n_s}$$

Figure 24: Equation for the Sinusoidal output buffer Frequency

$$Y_{pulse}(x) = offset + amplitude\ , x < duty$$

$$Y_{pulse}(x) = offset\ , x \geq\ duty$$

As you can see in the diagrams above, the parameters are calibrated before being outputted to consider the op-amp gain. This calibration was calculated and is discussed in more depth in the testing and measurement section of the report.

As discussed already, PA4 is set as the DAC output pin. We set the DAC to output one configuration and it is trigger to the timer four trigger out event. We enable the DMA1 channel 3 global interrupt and set the DMA request settings to have a circular mode, an increment address in memory and a data width of a word.

Timer 4 was used for the DAC sampling and needed to be set up correctly to obtain the correct sampling rate of 100kHz. The timer was set up in interrupt mode. The timer operates at 72 MHz, we set a pre-scaler of seventy-two which yields

$$Clock\ Ticks\ per\ Second = \frac{72000000}{72} = 1000000$$

We set our counter period (Auto Reload register) to ten ticks which yields a period of

$$Timer\ Period = \frac{10}{1000000} = 10\mu s$$

This gets a sampling frequency of

$$f = \frac{1}{T} = \frac{1}{10x10^{-6}} = 100000Hz$$

Which is what we wanted.

4.6 LCD Interface and Timing

The implemented LCD Driver consists of nine base functions and two add-on functions that ensure the correct display formatting is visible for the different system states. By looking at the data sheet and really investigating the timing, we can write a very reliable LCD driver.

4.6.1 Data Line Control

The LCD Data function is required to set the correct data lines high to transmit data to the LCD. For the design, we can do this in a simple case of if-else statements and make use of the HAL libraries to write to each pin. We know that we are working in nibble mode thus we only write to four data lines. We perform bitwise & to see if a data line needs to be set high. If the data line needs to be set high, we write a one to the pin, or else we write a zero to it. We do this for each data line (DB4-DB7). We know this requires no delay and thus we can execute each write immediately.

4.6.2 Commands

The LCD command function will be required to write a command to the LCD for it to execute. We use the HAL write function to set the reset pin to zero, indicating to the LCD that we are sending through a command. We then move the command data to the LCD's data lines by using the above LCD Data function. We set the enable line high and then low again, on the falling edge, the command is given to the LCD, and it can now be performed. Timing wise, after we raise the enable line and after we lower it, on both occasions we need to pause the function for a few microseconds. To implement this, I just used a HAL delay function for 1ms.

4.6.3 Initialisation

The initialisation step is especially important and needs to be done without being interrupted. Thus, the initialisation was performed as soon as possible in the main function, when I knew that no interrupts would be called while it was being initialized. The entire LCD initialisation process can be found in the datasheet. Timing is important and was approximated using the HAL delay function. I decided to set the display, cursor, cursor blink, and increment on when initializing the LCD.

4.6.4 Write Character

The Write Character command is required to write a character to the LCD screen, by using bitwise logic, the HAL libraries, and our previously defined functions, we can implement the following

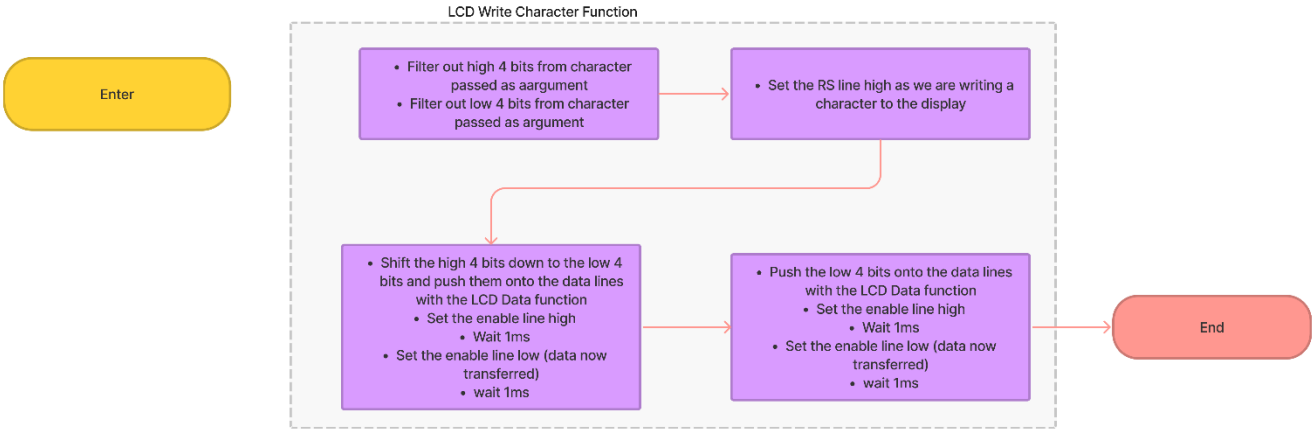

Figure 25: LCD Write Character function logical flow diagram

The timing is incorporated into the function and once again, HAL delay is used after every rise and fall of the enable line.

4.6.5 Write String

The write string function uses a for loop to iterate through a string and write each character to the LCD by using the LCD Character Write function as described above. The timing works out fine as it has already been implemented in the character write function.

4.6.6 Clear

The clear function requires us to completely clear the LCD display. We can easily implement this using the LCD command function. The command byte to clear a screen is 00000001. We first use the LCD

Command with data = 0 and then use it again with data = 1 to clear the screen. Importantly, a 3ms delay is added after the second command to give the LCD time to clear its display.

4.6.7 Set Cursor Position

The cursor position function sets the cursor somewhere on the display depending on the given row and column as arguments. To set the cursor to the first row, we issue an LCD command of 0x80 + column position – 1, and to set the cursor to the second row, we issue a command of 0xC0 + column position -1, using the LCD command function defined above. The timing is already dealt with in the command function.

4.6.8 Shift Right and Shift Left

If we want to shift our LCD display to the right or to the left, we just need to issue the command 0x1C or 0x18 respectively, as specified by the data sheet. This is done with the LCD command function above, which also takes care of the timing of the function.

4.6.9 Menu View

The Menu View puts the LCD into the menu state. It is mostly defined from the menu file, but here we execute the view function and ensure that the correct menu format is displayed. To do that, the LCD Driver that was created is used. The timing is already dealt with in the driver.

4.6.10 Display View

This function manages the LCD's display when the LCD is in the display state where it displays the measurement of the ADC at the top line and the signal output of the DAC at the bottom line. The design and implementation of this is defined with the flow diagram below. All the timing is dealt with by the LCD driver we have defined.

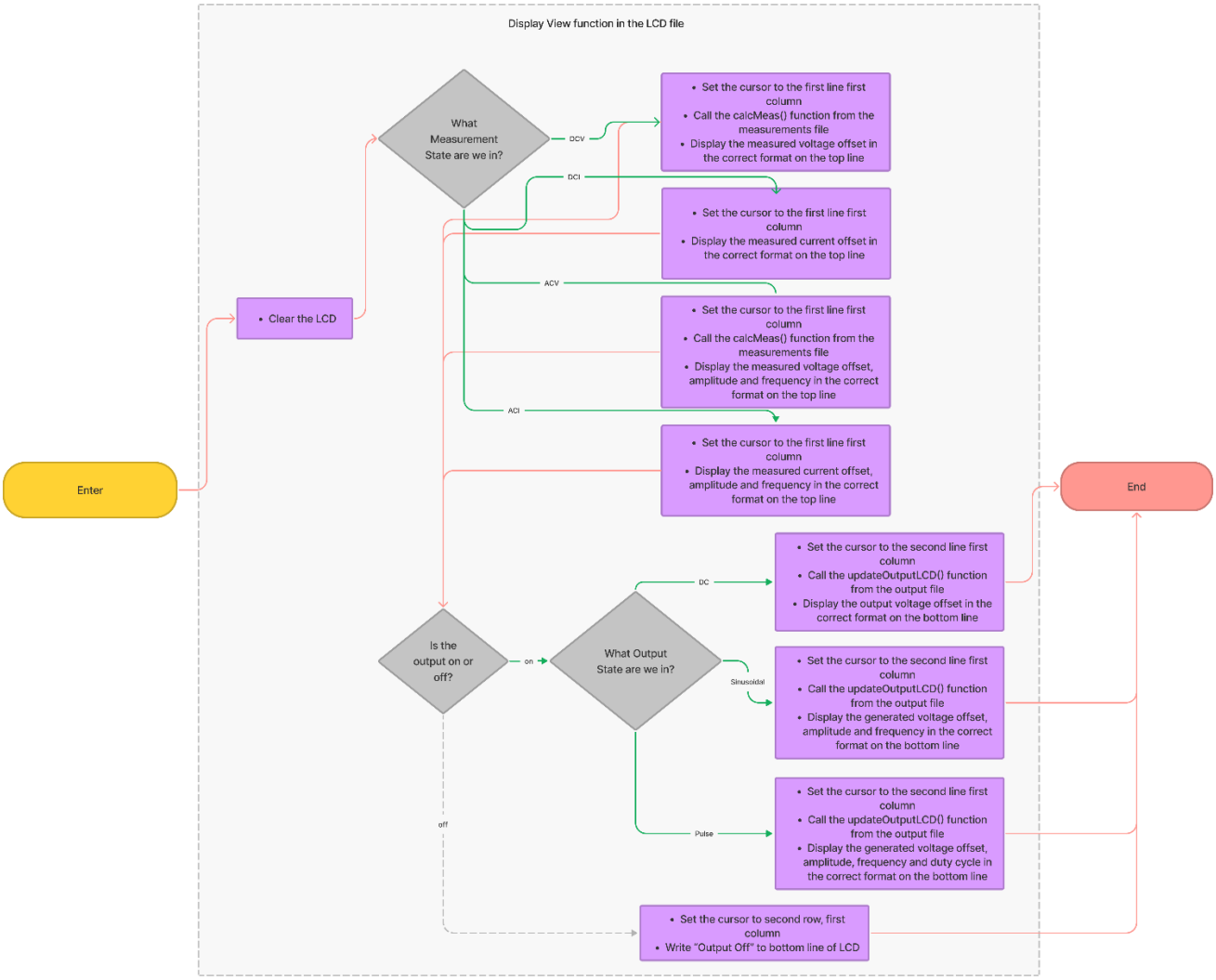

Figure 26: LCD Display View function in the LCD file

4.6.11 Pin Configurations

All the LCD pins have already been discussed in the hardware section and is just set up as standard GPIO outputs ports.

5 Measurements and Results

5.1 Power Supply

In order to test if the power supply was working correctly, I connected my Uni-T multimeter between ground and the power supply, the output of the 5V Voltage Regulator and the 3.3V Voltage Regulator and measured 9V, 5V and 3.3V respectively. Which meant that the board was being correctly powered and correctly regulated the voltage down to 5V and 3.3V.

5.2 UART Communications

When it came to UART I was able to test the hardware by making use of an oscilloscope and the software by using the Termite serial terminal program on windows. In termite, if I correctly configured the settings to the correct com port, 115200 baud rate, eight data bits, one stop bit, no parity bits and append CR-LF, I was able to receive my student number every time I reset the board which at least gave me some level of confidence in my board's UART communication. After this I tested every command / configuration message that the TIC could send to the board and ensured that the board responded correctly each time. I also made sure that the send measurement and send status commands were working perfectly. On the hardware side, I connected the UART transmit on the TIC (J13 – 5,6) to an oscilloscope. When pressing the reset button on my board and clicking run stop on the oscilloscope, I was able to obtain the following

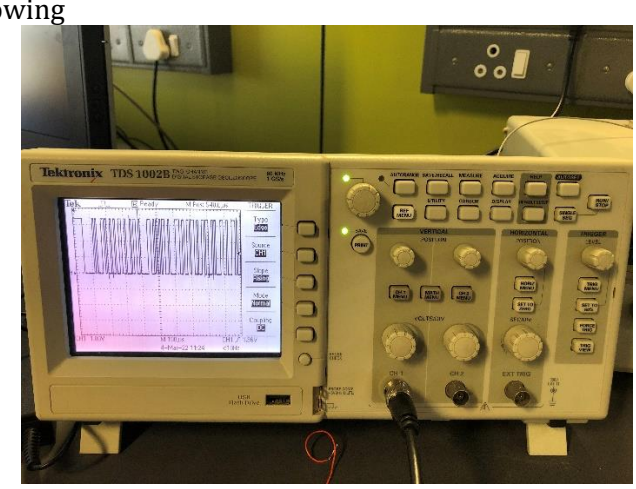

Figure 27: UART Transmit of student number

This meant that my UART transmit was working in hardware.

For timing I made use of the HAL get tick function and two, time variables to measure the amount of time it took to perform each UART command / configuration and the amount of time it took for requests to be sent. The longest UART command was for changing the pulse frequency and it took about 350ms. Every other command and request took much less time to complete, which was well within specifications.

5.3 Buttons

To accurately test my button's hardware implementation, I decided to create the final circuit on a breadboard for testing. I connected the input to a DC power supply at an input voltage of 3.3 volts and a current of 30mA, and I connected the output to an oscilloscope (instead of a GPIO input pin). When the button was not pressed, the Oscilloscope read a 3.1V signal and when it was pressed the Oscilloscope read an input of 0V or sometimes a small negative voltage reading, which means I have accurately created the active low push button circuit. I tested the response time as discussed above in the design choices section and measured a response of 24.4 microseconds. These values meant that I have achieved the specifications and requirements for the hardware implementation of the button. For testing the button bounce time, I used the exact same setup as discussed above on the breadboard. I used the trigger menu on the oscilloscope to capture the rising and falling edges of the button and then made use of the cursors to accurately measure the amount of time the button bounced for. I did about 35 tests where I applied varying amounts of pressure and hold-in-time to the buttons on each press. The measured bounce times ranged from a minimum of 31 microseconds to a maximum of 1.38 milliseconds.

The buttons were proven to work once more once when they were causing the correct state change in the system.

5.4 Debug LEDs

I tested the LED circuit twice. Once on the debug circuit, and once on the PCB. I built the debug circuit as discussed on a breadboard. I was able to test whether the circuit worked correctly by connecting it to a DC power supply in the labs and configured the input voltage to be 3.3V and the current to be 8mA. The LED clearly turned on and drew a current of 2.09 mA which is less than the maximum of 8mA. I then rebuilt this circuit on the PCB and instead of connecting my input to the GPIO output pin, I connected it to the DC power supply once more. Once again, my LED was clearly turning on at 3.3V and 8mA and was pulling 2.09 mA from the power supply. The Debug LEDs turned on for the correct state in software and was clearly visible, which meant it was working correctly.

5.5 ADC

To test the ADC, I had to test the buffer for the hardware implementation and the measurements for the software implementation. To test the buffer, I simply connected a signal generator to the input and an oscilloscope to the output. The buffer was working correctly as the measured signal on the oscilloscope was the same as the signal being generated by the signal generator.

For the software side, I needed to connect a signal generator to the TIC ADC input pin (J13 - 7,8) with a probe set to 1x and then connect another probe from an oscilloscope to this pin with the probe set to 10x attenuation. This was done so that I could achieve the most accurate readings as possible and thus calibrate my system accurately. In order to do this calibration for the offset, amplitude and frequency, I fed in seven test values for each and measured my systems measurements. I then got a transfer function between the input and the output and calibrated the output to match the input. After testing again, all my values were within 5% accuracy which also meant that my ADC software was working very well.

5.6 DAC

To test the DAC, I needed to test both the hardware and software side of the system. For the hardware I needed to test the gain of my op-amp and ensure that it was working as intended. I connected the op-amp on a breadboard and connected an oscilloscope to the output using a 10x probe and a load resistance of $5k\Omega$. I connected a DC power supply to the input and then measured the gain and linear relationship every 200mV. I also connected a signal generator to the input and ensured I was obtaining a non-inverting signal with a gain of about 2 out. From these tests I was able to determine that the op-amp was working correctly and achieving a gain of 2.15 and I would use this value to calibrate in software.

For software testing I would connect the DAC output pin on the TIC (J13 – 3,4) to an oscilloscope with a 10x probe and simply use the UART to turn the output on and see if the output was at least giving me the correct shape for each output type. It was. I then changed different output signal parameters and made sure that my output signals were changing appropriately. Considering the calibration of 2.15 gain, the DAC was outputting the correct signals even for variable parameters well within the 10% accuracy range, which is very good. Both the hardware and software sides were working for the DAC.

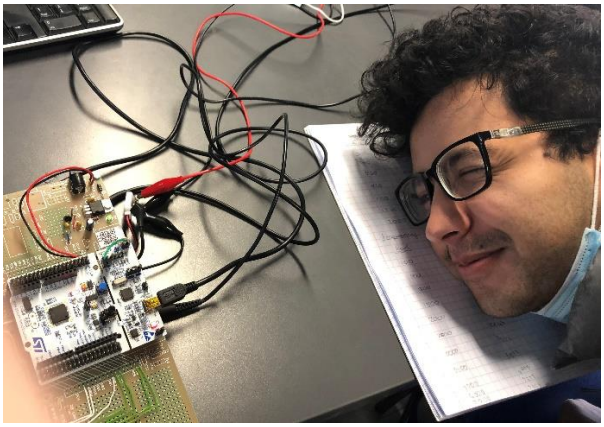

Figure 28: Pro Tip: Have a work buddy check your connections when testing!

For the timing of the DAC, I made use of the HAL get tick function and two, time variables to measure the amount of time it took to turn the DAC on or off and how long it took to change the output parameters. The calculate DAC function took the longest, 800ms to update the DAC buffers. This was too long so I lowered the buffer size from 10000 down to 2000, this will lead to some discrepancies at very high frequencies, but it also minimized the longest DAC response time to below 200ms, which is much more efficient.

5.7 LCD

To test the LCD, I had to create debug circuits on a breadboard to test for the backlight and the contrast before connecting it to the PCB and testing the software. I used the debug circuit to test various resistor values for the backlight circuit but eventually decided on the 150Ω as it provided the brightest backlight while still minimizing the current over the LED to below 10mA. The backlight for the LCD was proven to work as soon as the backlight turned on. I also used the debug circuit to test various combinations for R8 and R9 of the LCD circuit before finding $R8 = 22k\Omega \; and \; R9 = 4.7k\Omega$. I knew I got the contrast working when the display blocks on the top line were barely visible. So, the backlight was proven to work and the contrast, now I just had to test the software.

After setting up my driver and including the initialisation in the beginning of the main function, I just had to flash my code to the board, and the LCD (which was now correctly connected to the PCB and all the right pins) should display a blinking cursor. This is exactly what it did when I ran the code, and I then knew that my LCD display was working perfectly fine.

For the rest of the display formats, I simply had to use the system and test out all the different states and use the buttons to change states and UART etc. to ensure that my LCD was displaying the correct display formats at the correct times.

5.8 Complete System

To test the complete system, I simply had to use the project and naturally go into every state and experience every bit of functionality in order to know if it works. I was able to do this and was happy with the system.

6 Conclusions

Unfortunately, the two main requirements that my design does not satisfy is the menu system and the current sensor. While the base frameworks and state management system for both the menu and the current sensor have been set up correctly, I was unable to implement the current sensor in hardware even though the framework for it in software has already been implemented and the menu system was not implemented in software. This meant that the user requirement for measuring DC or AC current from 0mA to 9mA was not satisfied and the user will not be able to change operating modes, select measurement type, set output signal parameters and turn the output on or off in the menu system, but will still be able to perform these functions with the UART commands.

When it comes to device short comings, one of the main problems my device had was the time it took to fill my DAC output buffer for the sinusoidal output. This unfortunately meant that I had to lower the buffer size, and this ultimately led to lower accuracy for the signal frequency. If I were to redo it, I would look at a new way to implement the function which fills the buffer. I believe it would be a better option to change the DAC timer's auto reload register to alter the frequency, instead of changing the sample size in the buffer. This would mean that I can use a lower buffer size and thus fill the buffer much quicker. For the LCD, using a dedicated timer to deal with timing would be better than using HAL delay which has a minimum delay of 1ms, where most of the LCD time delays were in the microsecond range. This

would make our LCD driver much quicker than what it currently is. Another problem I had for the LCD is it became glitchy and slow when I displayed AC or Pulse measurements or outputs due to the consistent display updating and the implemented scroll. To fix this, I would write a separate driver function which would only update the changed characters, instead of rewriting the entire string every 0.5s.

STM32 Microcontroller Pins and Configuration

Table 2: Microcontroller Pins and Configurations

Pin Used	Hardware Element	Configuration
PA0	Op-amp buffer	ADC1 IN1
PA1	Middle push button	GPIO EXTI1
PA2	TIC	USART TX
PA3	TIC	USART RX
PA4	Op-amp low pass filter	DAC1 OUT1
PA7	Bottom push button	GPIO EXTI7
PB10	LED D3	GPIO Output
PB12	LCD Enable	GPIO Output
PB13	LCD Reset	GPIO Output
PB14	LCD Read not Write	GPIO Output
PC6	LCD DB6	GPIO Output
PC8	LCD DB7	GPIO Output
PA11	LCD DB4	GPIO Output
PA12	LCD DB5	GPIO Output
PA8	LED D2	GPIO Output
PB4	LED D4	GPIO Output
PB5	LED D5	GPIO Output
PB6	Left push button	GPIO EXTI6
PB8	Top push button	GPIO EXTI8
PB9	Right push button	GPIO EXTI9

References

[1] A. Barnard, L. Grootboom, U. Louw and D. Klink, *Design (E) 314 Project Definition*, 7th ed. Stellenbosch University, 2022.

[2] *ST MICRO (L7805CV) 5V 2% REGULATOR*, 13th ed. Jameco Electronics, 2006.

[3] *100mA LOW DROPOUT VOLTAGE REGULATOR LM2950/1*, 1st ed. Texas Instruments, 2015.

[4] C. Coulston, "EENG 383 - Lecture Notes", *Inside.mines.edu*, 2022. [Online]. Available: https://inside.mines.edu/~coulston/courses/EENG383/lecture/lecture01.html.

[5] *UM1724 User manual STM32 Nucleo-64 boards (MB1136)*, 14th ed. ST Microcontrollers, 2020.

[6] *RM0316 Reference manual*, 8th ed. ST Microcontrollers, 2017.

[7] G. Brown, *Discovering the STM32 Microcontroller*. 2016.

[8] *2.7V to 6.0V Single Supply CMOS Op Amps*. Microchip, 2007.

[9] A. Barnard, *LCD - How-to-Slides*. Stellenbosch University, 2017.

[10] *SPLC780D1 16COM/40SEG Controller/Driver*, 1st ed. Orise Tech, 2008.

[11] L. Johnson, *EDesign314 Preliminary Report*, 1st ed. Stellenbosch University, 2022.

[12] "DAC in STM32 » ControllersTech", *ControllersTech*, 2020. [Online]. Available: https://controllerstech.com/dac-in-stm32/.

[13] K. Magdy, "Interfacing 16×2 LCD With PIC Microcontrollers | MPLAB XC8", *Deepbluembedded.com*, 2019. [Online]. Available: https://deepbluembedded.com/interfacing-16x2-lcd-with-pic-microcontrollers-mplab-xc8/.

[14] Sean Muller, student number 23575786, demo 2 code snippets.

[15] Lise Prinsloo, student number 23726059, demo 2 code snippets.