7074CEM Coursework Submission

Jerome Samuels-Clarke

Student Number: 10473050

Module Leader - Dr Sam Amiri

October 2021

Table of Contents

1. Introduction

A signal generator is an electronic instrument tool, that can be used for creating a variety of signals to stimulate electronic circuits. In addition, being able to generate a signal, it can also customise the waveform properties such as the amplitude, offset, and wave shape (Tektronix, 2008, p. 6).

The design and cost of signal generators can vary, as depending on the application either cheaper or more expensive models might be necessary. Using discrete components, a low-cost signal generator can be created using a microcontroller, Digital-to-Analog Converter (DAC) and depending on the design the supporting components can allow for higher precision and finer control of the output.

The following report is about designing and implementing a low cost, portable, PIC based signal generator from a given specification. The development of the implementation is discussed in the subsequent chapters, starting with the Design Requirements. This chapter highlights the specification, gives an insight into existing products, as well as outlines the hardware and software methodologies. The Hardware Development chapter covers all aspects the physical design, from component selection down to the final PCB. The Software Development chapter shows flowcharts that were used to develop the firmware, and the implementation of a simulation is also shown. Finally, the Verification and Optimisation chapter discusses the results, testing and debugging for the design. Also mentioned are future improvements from the original specification.

2. Design Requirements

Before covering the specification, it is important to understand what other devices are on the market. This way it covers a common ground for analysing the final design, and how it compares with that currently existing.

Signal generators can come in a range of sizes, costs and functionalities. Traditionally, signal generators are not portable, and will be in a lab or placed on a desk. High performance devices can cost thousands of pounds, such as ones developed by Tektronix. The AFG31000 generator costs around £9,300 but is a high-performance generator capable of creating a wide variety of waveforms, has high resolution and a touch screen panel as seen in Figure 1. As it's a function generator, the capabilities are more than a traditional signal generator, be able to fully customise the waveform. On the other end of the spectrum a signal generator can be less than £100, and even cheaper if depending on the required performance. Commonly, hobbyist create their own or buy kits which just need to be assembled by soldering (V. S. Nair & A. S. Nair, 2018, p. 1).

Figure 1: Tektronix AFG31000 function generator (https://uk.tek.com/signal-generator/afg31000-function-generator).

2.1 Specification

With a rough understanding of what the current market has, the given specification to system being designed is given below:

Output signal specification:

- **Output waveforms:**
  - Basic: Triangle, Ramp Up Sawtooth, Ramp Down Sawtooth
  - Advanced: Square, Sine, Pulse
- **Output frequencies:**
  - Basic: 2Hz, 5Hz, 10Hz
  - Advanced: 20Hz, 50Hz, 100Hz
- **Output amplitude:**
  - Basic: Non-adjustable
  - Advanced: Adjustable
- **Output offset:**
  - Basic: Non-adjustable
  - Advanced: Adjustable

Interface specification:

- **Power:**

    - Basic: Powered through a USB port. Device auto power off after one minute of no user interaction.
    - Advanced: Further optimisations for low power consumption.
- **User interface:**
    - Basic: Physical user interface
    - Advanced: USB control interface

Production specification:

- **Physical size:**
    - Basic: PCB area smaller than 50cm^2 (final PCB layout)
    - Advanced: PCB area smaller than 25cm^2 (final PCB layout)
- **Cost per unit:**
    - Basic: < 20 GBP (Bill of Materials and PCB only, for a production run of 1000 units)
    - Advanced: < 10 GBP (Bill of Materials and PCB only, for a production run of 1000 units)

Taking each section of the specification and breaking it down, makes it a lot easy to manage. With the output signal specification, generating the waveforms can be done several different ways, from incrementing a variable, using look up tables (LUT), timers or a combination. However, when thinking about how to generate the different output frequencies, the most common approach is to use the peripherals the PIC microcontroller has. When dealing with accurate timings, using timers is a much better approach, than a rudimentary method such as delays. In addition, adjusting the amplitude and offset can either by done in software or hardware. There are pros and cons to both methods depending on where the trade-offs in the overall system will be. If higher performance is needed, then there will be higher costs. If lower costs are needed, the performance will drop.

With the production specification this is a relatively simple, in getting the PCB size under 25cm^2 and costing under £10. Using surface mount components greatly reduce the overall size of the PCB, and in some cases cheaper than it's through hole counterpart. Reducing the cost comes from sourcing cheaper components, as well as using alternative components. For example, using a microcontroller with many peripherals and ports, would cost more than ones with less. The selection of the choice of microcontroller used is mentioned in the next chapter.

3. Hardware Development

3.1 Component Selection

3.1.1 Microcontroller

Selecting the components for any design is a crucial part, where every possibility must be analysed. The microcontroller being the brains of the system should be one of the first components to be selected. Ideally without knowing what microcontroller to pick from (thousands of different ones exists), developing code on one might not be compatible with the microcontroller that gets used in the final product. Keeping in theme with the specification, the line of microcontrollers is the PIC18F, using C18. The initial code was developed on a PIC18F4520 out of familiarity. The datasheet for this microcontroller contains the device features, but most importantly it shows other similar microcontrollers as seem in Figure 2. This is useful, as the code developed on the PIC18F4520 can be used on these compatible microcontrollers, with minor changes. From Figure 2 the PIC18F4520 can be seen as the most powerful out of the four. For example, this microcontroller may have more ports, program memory, ADC channels a combination of these and or other features.

Features	PIC18F2420	PIC18F2520	PIC18F4420	PIC18F4520
Operating Frequency	DC – 40 MHz	DC – 40 MHz	DC – 40 MHz	DC – 40 MHz
Program Memory (Bytes)	16384	32768	16384	32768
Program Memory (Instructions)	8192	16384	8192	16384
Data Memory (Bytes)	768	1536	768	1536
Data EEPROM Memory (Bytes)	256	256	256	256
Interrupt Sources	19	19	20	20
I/O Ports	Ports A, B, C, (E)	Ports A, B, C, (E)	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers	4	4	4	4
Capture/Compare/PWM Modules	2	2	1	1
Enhanced Capture/Compare/PWM Modules	0	0	1	1
Serial Communications	MSSP, Enhanced USART	MSSP, Enhanced USART	MSSP, Enhanced USART	MSSP, Enhanced USART
Parallel Communications (PSP)	No	No	Yes	Yes
10-Bit Analog-to-Digital Module	10 Input Channels	10 Input Channels	13 Input Channels	13 Input Channels
Resets (and Delays)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), MCLR (optional), WDT	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), MCLR (optional), WDT	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), MCLR (optional), WDT	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), MCLR (optional), WDT
Programmable High/Low-Voltage Detect	Yes	Yes	Yes	Yes
Programmable Brown-out Reset	Yes	Yes	Yes	Yes
Instruction Set	75 Instructions; 83 with Extended Instruction Set Enabled	75 Instructions; 83 with Extended Instruction Set Enabled	75 Instructions; 83 with Extended Instruction Set Enabled	75 Instructions; 83 with Extended Instruction Set Enabled
Packages	28-Pin SPDIP 28-Pin SOIC 28-Pin QFN	28-Pin SPDIP 28-Pin SOIC 28-Pin QFN	40-Pin PDIP 44-Pin QFN 44-Pin TQFP	40-Pin PDIP 44-Pin QFN 44-Pin TQFP

Figure 2: A summary of the device features, which shows the similar microcontrollers in the PIC18F line (https://ww1.microchip.com/downloads/en/DeviceDoc/39631E.pdf).

From this the PIC18F2420 was chosen as the dedicated microcontroller. The reasoning for this comes down to cost, without sacrificing performance. Out of the four microcontrollers, the PIC18F2420 is the cheapest (per 1000 units). The reason for why it's cheaper is less ports resulting in a smaller footprint,

less program memory, data memory, interrupt sources, ADC channels and no parallel communications.

Referencing back to the specification, features that are most important for the system can be identified. Generating the waveforms will be from a Digital-to-Analog Converter (DAC), which uses a SPI interface. This is referred to as MSSP (Master Synchronous Serial Port) in the datasheet, which is discussed in the next section. Anything to do with timing in a microcontroller, is best done with timers. This gives an accurate method of monitoring the elapsed time until an event is needed, luckily the PIC18F2420 has four of these hardware timers. Lastly, there are two methods of interfacing with the system, being physically or using a communication method, in this case UART. The PIC18F2420 has an Enhanced USART that is used to do the communication, omitting the need for physical buttons. Also, less physical buttons can make the final PCB a smaller footprint, which can also reduce the overall cost.

3.1.2 Digital-to-Analog Converter

For simplicity the DAC has already been selected, being the MCP4921 from microchip. The MCP4921 is a 12-bit DAC, which uses SPI to interface with. From the datasheet important characteristics can be extracted, for example input voltage and current. The voltage ranges from 2.7V to 5.5V, and the input current is typically 175uA. This low voltage, and relatively low current, means the signal generator could be powered from a single cell battery. Other information that is useful when it comes to the software implementation is the mode of SPI. The MCP4921 supports mode 0,0 and 1,1. This gives flexibility, as any other components that uses SPI to communicate with can be selected based on if they are using the same mode. Finally, one key information that is mentioned in the next section is the external VREF pin of the MCP4921. The datasheet mentions in section 6.4.1.1. that the output voltage is dependent on the voltage on the reference pin. Adjusting this value will adjust the amplitude of the signal.

3.1.3 Other Components

The two main components have been talked about in the previous sections. However, there are two other components that are specific to the system. As the user interface with the system is done via UART, the information first needs to be sent through UART converter. There are many different converters on the market, and the FT230X was chosen. The reasoning behind picking this, is the simplicity with interfacing with the module, as well as the reduced package size.

The specification gives the option to control the amplitude. This can be done in code, however there are some limitations to doing this, that is discussed in the next chapter. As mentioned in the previous section, the amplitude of the MCP4921 can be controlled by VREF pin. A simple voltage divider can be used on the pin, being controlled by a potentiometer. However, as the system uses no physical interfacing (other than the reset button), another way this can be controlled is through a digital potentiometer. The MCP4151 was chosen for this, which is a 10K digital potentiometer, that is controlled via SPI. The SPI interface allows for multiple peripherals to be connected, controlled by pulling the selected chip select signal low. The SPI mode is the same used for the DAC, as well as controlling the module is as simple as sending a byte of data to the address of the module.

3.2 Simulating the System

Proteus is a simulation software for microcontrollers was used throughout the system design. This allows for the system to be tested without the need of a physical development board. The modular approach when writing the code, allowed each section to be developed and tested individually. The UART terminal provided the feedback when developing the user interface. The oscilloscope was used to see the output waveforms and measure the selected frequencies. Figure 3 shows the schematic from Proteus. A benefit to using Proteus is that the full code (attached in the Appendix), can be done in the same software as well as debugged. The simulation results are discussed in the Verification and Optimisation chapter.

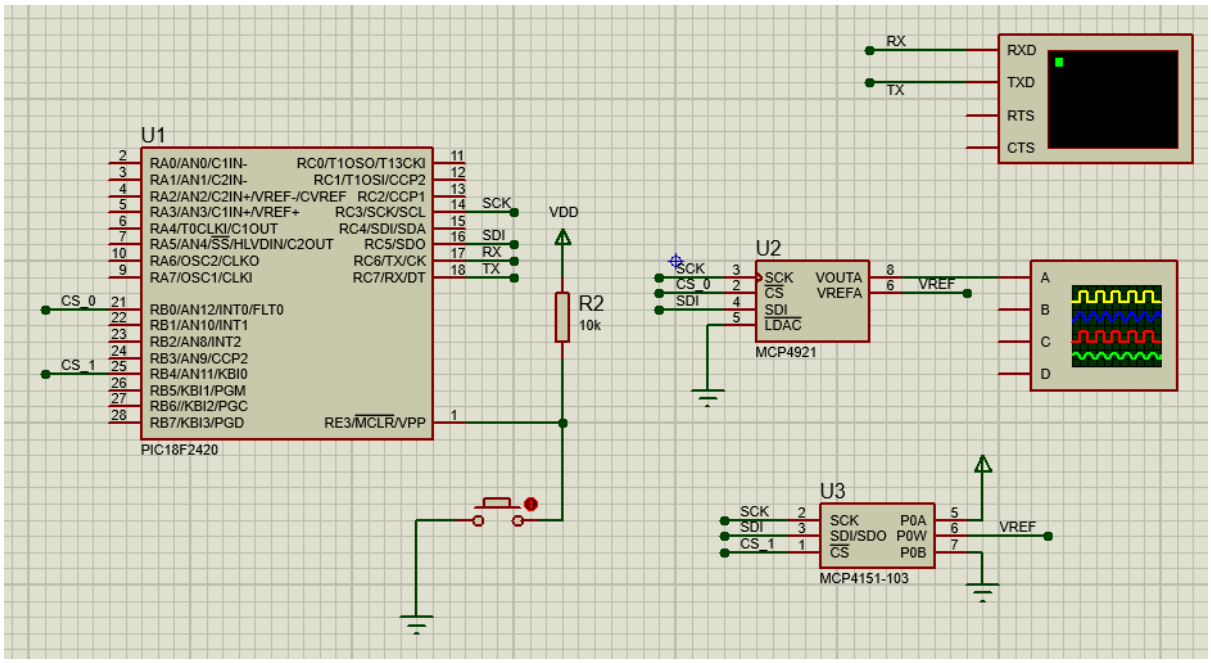

Figure 3: Schematic of the circuit done in Proteus, showing the PIC microcontroller, MCP4921, MCP4151, UART and oscilloscope.

Although Proteus is a good software to use, there are some limitations. One being the schematic doesn't show the full system components needed to make the system work efficiently. Another limitation is the component availability. If a component does not exist in the Proteus database, then a work around of this would have to be done.

3.3 PCB

Although Proteus comes with the ability to create the schematic into a PCB, it is not as powerful or reliable as dedicated PCB software available. Altium was used to create the final PCB. Figure 4 shows the final schematic created in Altium, which is different from the one done in Proteus (Figure 3). The main differences are the UART converter module, In-Serial-Circuit-Programmer (ICSP) to reprogram the PIC microcontroller and supporting components such as decoupling capacitors.

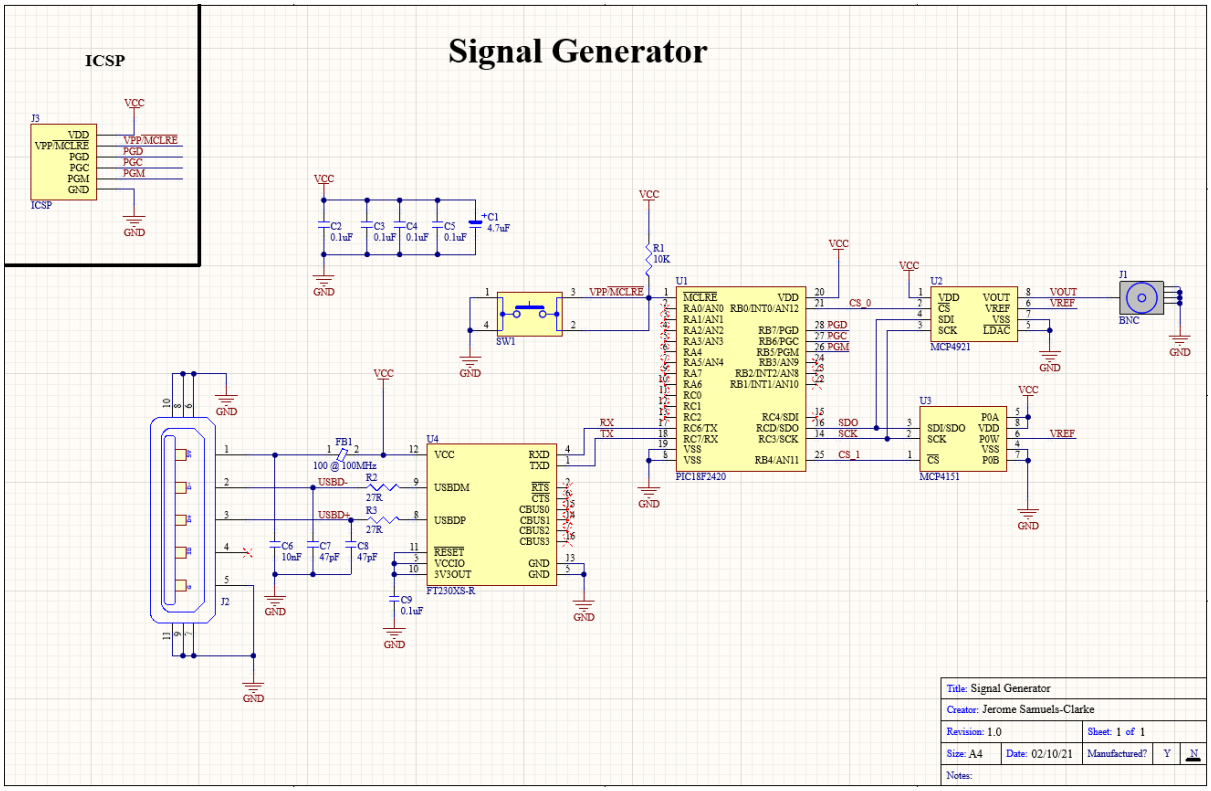

Figure 4: Final schematic of the signal generator done on Altium.

The decoupling capacitors are taken from the datasheets of the selected components, mainly being 0.1uF in values. The UART converter connections is taken from the datasheet and uses a micro-USB connector to send and receive the UART commands. This also brings power to the system at 5V or 3.3V, depending on where the source is coming from. The only physical interface is a reset button to reset the program if it were to crash. Lastly a BNC connector is used as the output. This is a common connection that is used in signal generators and oscilloscopes.

As per the specification the PCB must be under 25cm^2 and cost under £10 for a production of 1000 units. The cost to manufacture the PCB varies with suppliers. Digi-Key was the supplier used and the total cost of components came up to £9.09 as seen in the BOM (Figure 5). Figure 6 shows the final PCB, with the dimensions being 15.4cm^2. The packaging of the components is mainly surface mount, which reduces cost and size. Four mounting holes have added, which a case can be made around, giving access to the reset button, output connector and micro-USB port.

Description	Quantity	Case/Package	Category	Supplier 1	Currency	Supplier Order Qty 1	Supplier Part Number 1	Cost Per 1000 Boards	Cost Per Board
	1								
Cap Aluminum L	1	Radial	Aluminum Electrolytic Capacitors	Digi-Key	GBP	1000	PCE4667CT-ND	£68.56	£0.07
Cap Ceramic 0.1	5	0603	Ceramic Capacitors	Digi-Key	GBP	5000	311-1776-1-ND	£70.98	£0.07
Cap Ceramic 47p	2	0603	Ceramic Capacitors	Digi-Key	GBP	2000	311-4037-1-ND	£21.53	£0.02
Surface Mount C	1	0603	Ceramic Capacitors	Digi-Key	GBP	1000	311-3995-1-ND	£13.45	£0.01
Res General Pur	1	0603	Chip SMD Resistors	Digi-Key	GBP	1000	311-10.0KHRCT-ND	£3.22	£0.00
YAGEO - RC0603	2	0603	Chip SMD Resistors	Digi-Key	GBP	2000	311-27GRCT-ND	£5.32	£0.01
BNC Connector F	1		Connectors	Digi-Key	GBP	1000	2057-RF1-01A-D-00-75-M-ND	£733.65	£0.73
USB - micro B US	1		Connectors	Digi-Key	GBP	1000	609-4616-1-ND	£209.92	£0.21
MICROCHIP - MC	1	SOIC N	Digital Potentiometers	Digi-Key	GBP	1000	MCP4151T-103E/SNCT-ND	£620.15	£0.62
MCP4921 Series	1	MSOP	Digital to Analog Converters (DACs)	Digi-Key	USD	1000	MCP4921T-E/MSTR-ND	£1,343.60	£1.79
FERRITE BEAD 10	1	0603	Ferrite Beads and Chips	Digi-Key	GBP	1000	399-9608-1-ND	£45.56	£0.05
USB-to-UART 1-C	1	SSOP	Interface ICs	Digi-Key	GBP	1000	768-1135-6-ND	£1,190.46	£1.19
MICROCHIP - PIC	1	SOIC	Microcontrollers	Mouser	GBP	1000	579-PIC18F2420-I/SO	£4,119.54	£4.12
IP40 Black Butto	1		Tactile Switches	Digi-Key	GBP	1000	486-3471-ND	£200.22	£0.20
Total	21							£8,646.16	£9.09

Figure 5: Bill of Materials for the system, outlined in red coming to £9.09 for 1 unit.

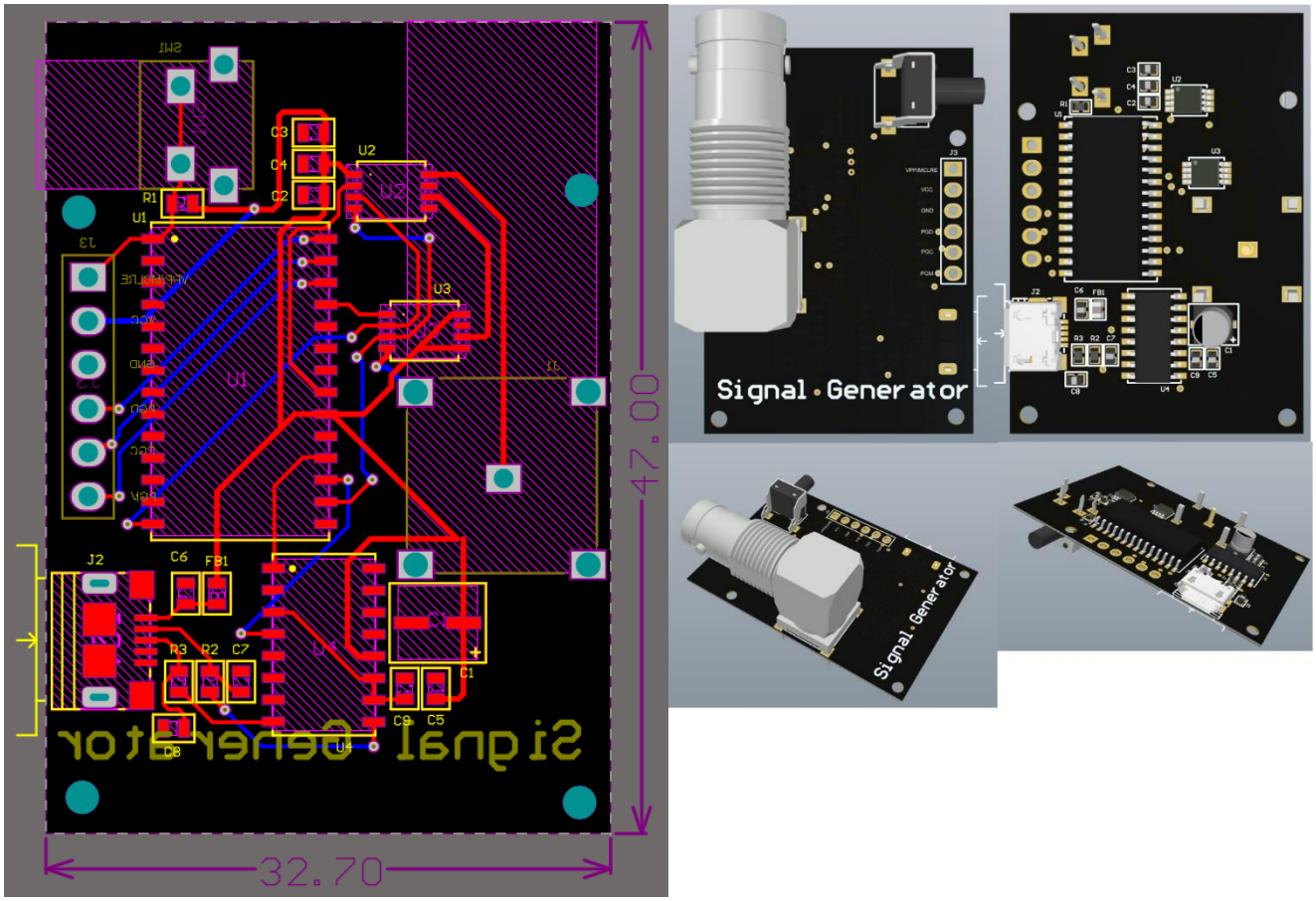

Figure 6: Final PCB of the system. The left image shows the PCB routing (hidden polygon pour connecting the ground pins), as well as the dimensions of the PCB (mm). The right images are a 3D view of the PCB, showing the connectors, reset button and circuitry.

4. Software Development

The approach for the software development of the system is modular. Each sub-system is coded and tested separately, then once the outcome has been met, it is then merged with the main code. The sub-systems are based on the specification, these are identified as:

- **User Interface** – The user interface is done via a UART communication. This will control the whole system, such as selecting which waveforms to run, the frequencies, and adjusting the amplitude, offset, and optionally putting the microcontroller to sleep.
- **Sending Data to the DAC** – To generate the waveforms the values need to be sent to the DAC, every time the timer overflows.
- **Auto Power-off** – The device should power itself off if there is no user interacting (data received from the UART), after one minute.

The language used for the programming is Embedded C, a subset of the C programming language. An alternative to this is writing in Assembly, which generates less instructions than writing in C, and faster at executing instructions. However, writing a large program like this is cumbersome if done in Assembly. There are two common compliers to choose from, being XC8 and C18. There isn't much difference between the two, and code developed on one complier can be used on another, with a few changes. The complier of choice is C18, and the full code can be seen in the Appendix.

Before going into the software design, the libraries used are listed below:

- **p18f2420.h** – This header file is used to access the registers of the microcontroller.
- **timers.h** – This header file contains functions for configuring the timers and gives functions that can write to the timers.
- **usart.h** – To communicate with the USART. It contains functions for easily setting up the USART communication protocol.
- **waveform_lut.h** – This is a custom header file, where the look up tables of each waveform is stored. Adding all the code on one page makes it harder to locate functions. In addition, adding code in separate head files, promotes code reusability.

4.1 User Interface

The UART is used to communicate with the device. This protocol is flexible, in that the settings can be configured for many different needs. This pins for this are located on RC6 (TX) and RC7 (RX), RC6 is set as an output and RC7 set as an input. The USART need to be set as a high priority interrupt, as any changes coming from the UART e.g. adjusting the amplitude, needs to be addressed quickly.

The OpenUSART function is used to configure the USART, and the following parameters were set: TX interrupt off, RX interrupt on, asynchronous mode, 8 bits data mode, 2 stop bits, continuous RX reception, high baud rate, and a baud rate value of 51, to generate a 9600 baud rate. This value of 51 can be worked out using formula in the datasheet, or using a table provided in the datasheet (Figure 7).

BAUD RATE (K)	SYNC = 0, BRGH = 1, BRG16 = 0											
	Fosc = 40.000 MHz			Fosc = 20.000 MHz			Fosc = 10.000 MHz			Fosc = 8.000 MHz		
	Actual Rate (K)	% Error	SPBRG Value (decimal)	Actual Rate (K)	% Error	SPBRG Value (decimal)	Actual Rate (K)	% Error	SPBRG Value (decimal)	Actual Rate (K)	% Error	SPBRG Value (decimal)
0.3	—	—	—	—	—	—	—	—	—	—	—	—
1.2	—	—	—	—	—	—	—	—	—	—	—	—
2.4	—	—	—	—	—	—	2.441	1.73	255	2.403	-0.16	207
9.6	9.766	1.73	255	9.615	0.16	129	9.615	0.16	64	9.615	-0.16	51
19.2	19.231	0.16	129	19.231	0.16	64	19.531	1.73	31	19.230	-0.16	25
57.6	58.140	0.94	42	56.818	-1.36	21	56.818	-1.36	10	55.555	3.55	8
115.2	113.636	-1.36	21	113.636	-1.36	10	125.000	8.51	4	—	—	—

Figure 7: Taken from the PIC18F2420 datasheet, the SPBRG value for the 8MHz clock can be seen (https://ww1.microchip.com/downloads/en/DeviceDoc/39631E.pdf).

The UART was tested using a simple echo program, just to check everything works. After this a flowchart was developed and shows the process of the UART in the code (Figure 9). After the initialisation, the UART is set up that any data on the RX pin will cause an interrupt. The character received is read then analysed to see what the outcome should be. If the value is to do with changing the waveforms, the data will be stored and analysed in the Sending Data to the DAC section. If the data is to do with changing the frequency, a variable that is used to reload the timer will be updated for the selected frequency. If the character is an 'n' or 'm', this will either increase or decrease the amplitude, by incrementing or decrementing a variable and calling the amplitude function. Similarly, the offset can also be adjusted if the 'i' or 'o' characters are received. To adjust the amplitude a value is added onto the data being sent to the data function; for finer control the value can be reduced.

A user interface was created to visually see the different options. Figure 8 the characters '1' and 'y' was received from the UART, which would change the wave to a triangle wave, and set the frequency to 100Hz.

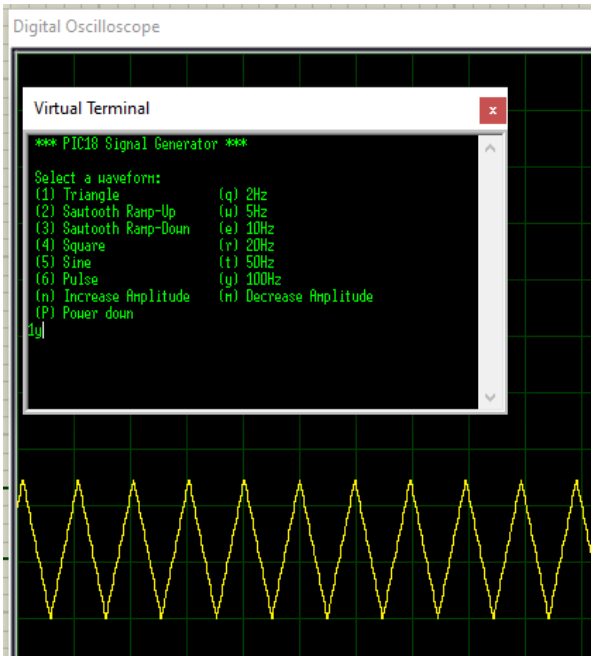

Figure 8: A capture of the user interface selected a triangle wave and output 100Hz.

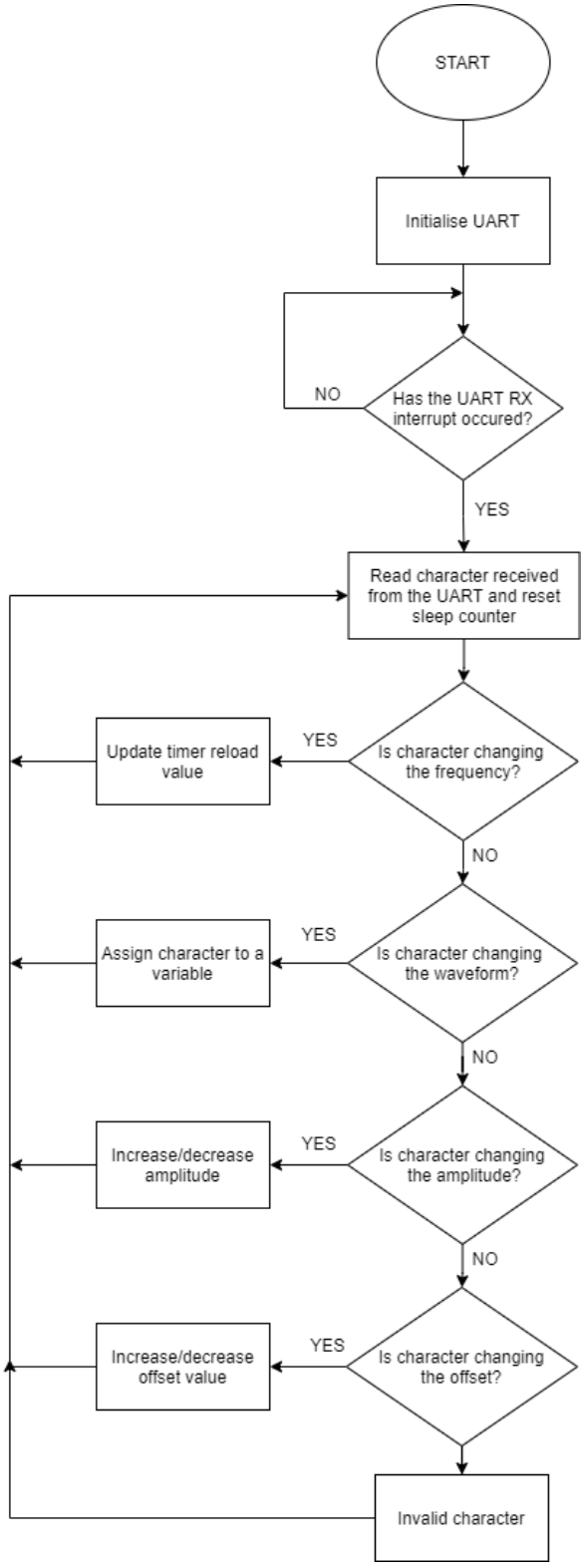

Figure 9: Flowchart of the UART process.

4.2 Sending Data to the DAC

To update the waveform requires the DAC to be constantly sending data. This data is either the values for each waveform, or a value to change the frequency. Timer0 is used as the means of generating interrupts depending on the frequency of the waveform. Timer0 is set as a low priority interrupt, as the sleep and UART interrupts are of higher priority. With this being said, there is not much difference and the timer could also be set as high priority. As the timer is a low priority, the INTCON2 register for timer0 needs to be cleared to set it as low priority. The OpenTimer0 function is used to configure the timer with the following configuration: interrupt on, 16-bit mode, use internal clock source, and a prescaler of 1:1. Lastly, Figure 10 shows the flowchart for the timer0 interrupt function. It is straightforward, where the timer0 gets initialised and loaded with an initial value. If the timer has overflown an interrupt will be generated. Inside this interrupt function, the timer is reloaded with the output frequency and the DAC function is called.

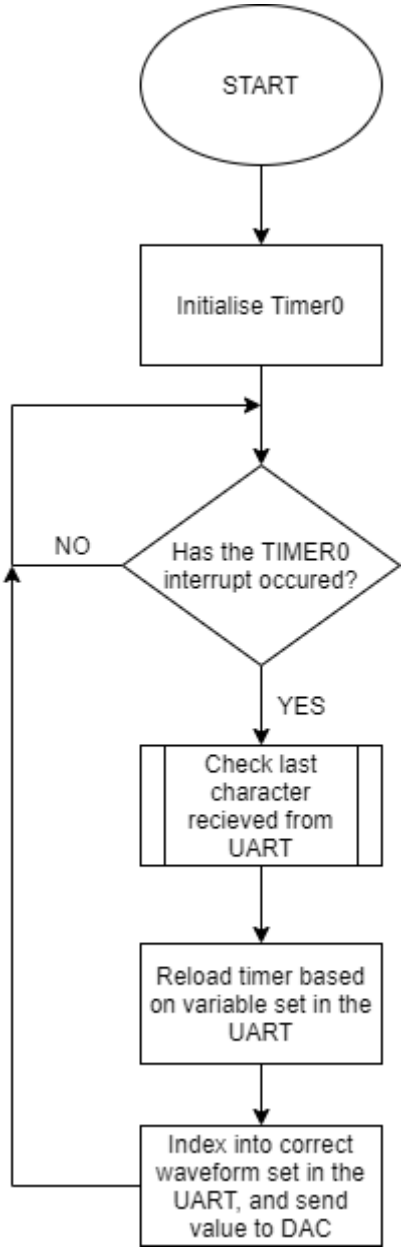

Figure 10: Flowchart for the timer0 interrupt function. Depending on the data received from the UART interrupt function will change either the frequency or output waveform.

4.2.1 Generating the Frequencies

There are many approaches that can be done to generate the frequencies. The approach done for this system is the find the commonalities between each frequency. This being the sample size. If the sample size is the same, then the equations should be the same regardless of the selected waveform, for these equations the sample size is 50. The equations for generating the frequency comes from what value to reload the timer with, in this case timer0. When the UART receives a character to do with selecting a different frequency a global variable is updated, and when timer0 interrupt is triggered this variable will update the frequency at which the timer interrupt is triggered, thus changing the frequency of when the DAC will send data. Equations 1-7 are the calculations for generating a frequency of 2Hz, 5Hz, 10Hz, 20Hz, 50Hz and 100Hz.

$$\frac{8MHz}{4} = 2MHz, \qquad PRESCALER = 1, \qquad \frac{1}{2MHz} = 0.5uS \ (1)$$

$$2Hz = \frac{\frac{1}{2}}{0.5 * 10^{-6}} = 1 * 10^{6} \rightarrow \frac{1 * 10^{6}}{50} = 20{,}000, \rightarrow \ 65535 - 20000 = 45535 \ (2)$$

$$5Hz = \frac{\frac{1}{5}}{0.5 * 10^{-6}} = 0.4 * 10^{6} \rightarrow \frac{0.4 * 10^{6}}{50} = 8000, \rightarrow \ 65535 - 8000 = 57535 \ (3)$$

$$10Hz = \frac{\frac{1}{10}}{0.5 * 10^{-6}} = 0.2 * 10^{6} \rightarrow \frac{0.2 * 10^{6}}{50} = 4000, \rightarrow \ 65535 - 4000 = 61535 \ (4)$$

$$20Hz = \frac{\frac{1}{20}}{0.5 * 10^{-6}} = 0.1 * 10^{6} \rightarrow \frac{0.1 * 10^{6}}{50} = 2000, \rightarrow \ 65535 - 2000 = 63535 \ (5)$$

$$50Hz = \frac{\frac{1}{50}}{0.5 * 10^{-6}} = 0.04 * 10^{6} \rightarrow \frac{0.04 * 10^{6}}{50} = 800, \rightarrow \ 65535 - 800 = 64735 \ (6)$$

$$100Hz = \frac{\frac{1}{100}}{0.5 * 10^{-6}} = 0.02 * 10^{6} \rightarrow \frac{0.02 * 10^{6}}{50} = 400, \rightarrow \ 65535 - 400 = 65135 \ (7)$$

Equation 1-7: Calculations showing the values for the different frequencies. The values calculated are written to the timer reload register.

4.2.2 Generating the waveform

Generating the waveforms can be done in several ways. For example, to generate a triangle wave, a value can be incremented, then decremented. Likewise, with the sawtooth wave, a value can be incremented, then reset to zero once the maximum value has been reached. However, as discussed above keeping the sample size the same makes implementing a lot easier. A Look-Up Table (LUT) was used for each waveform with a sample size of 50. The sine, triangle and sawtooth waves can be done using online LUT generators or using Microsoft Excel. The square wave and pulse wave are just a series of zeroes and 255, representing the upper and lower limits. When timer0 overflows, a variable is indexed in the selected waveform LUT and the value is sent to the DAC function. The function separates the two bytes into lower and upper variables. Data is appended onto the upper variable, which is needed to configure the DAC, then the two variables are sent one after the other.

Despite the calculations being correct the output frequencies do not give the correct frequencies. This can be down to several reasons, but mainly because each instruction takes a set amount of time to

execute. By the time to output is finally sent, this will have increased the output frequency. To fix this, the values sent to the timer has been slightly adjusted to give a more accurate value. Figure 11 shows the final output of the 100Hz signal at 10.16ms or 98.4Hz. The value can be increased more, however depending on where the cursors are positioned on the oscilloscope will make the result vary.

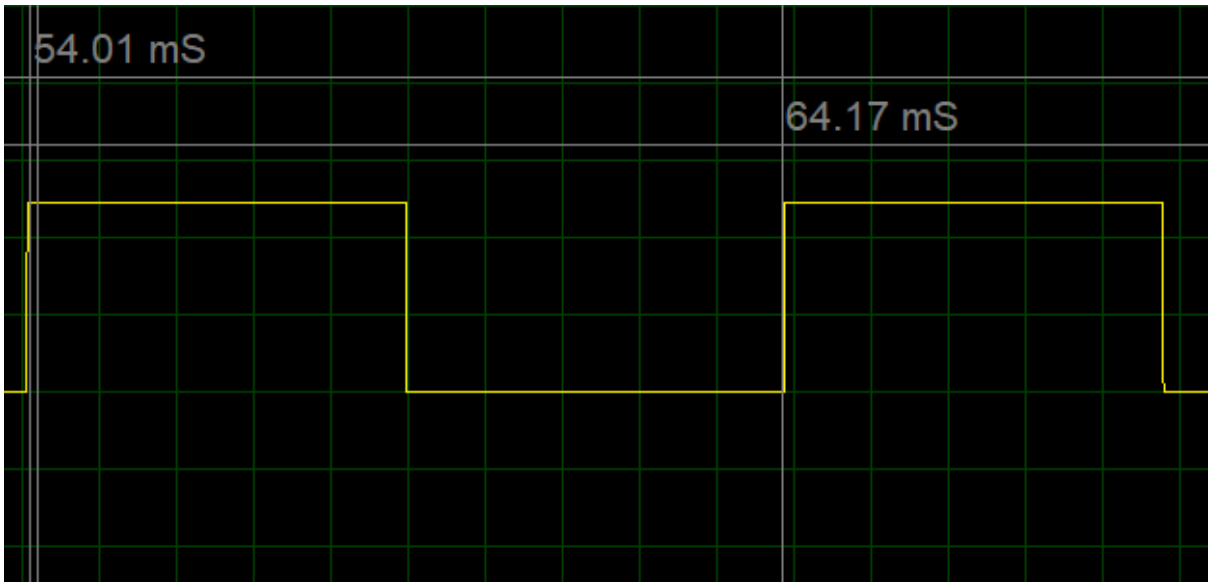

Figure 11: Output of the square wave generating a frequency of 100Hz.

4.3 Auto Power-Off and Reducing Power Consumption

As mentioned previously, anything to do with time is best done with the hardware timers. The PIC18F2420 has four timers, each with slight variations. Timer0 was used for generating the output waveforms, and timer1 was used to automatically power the timer off, by executing a sleep instruction. Figure 12 shows the flowchart, showing how the microcontroller is put the sleep if no activity has been detected after a minute.

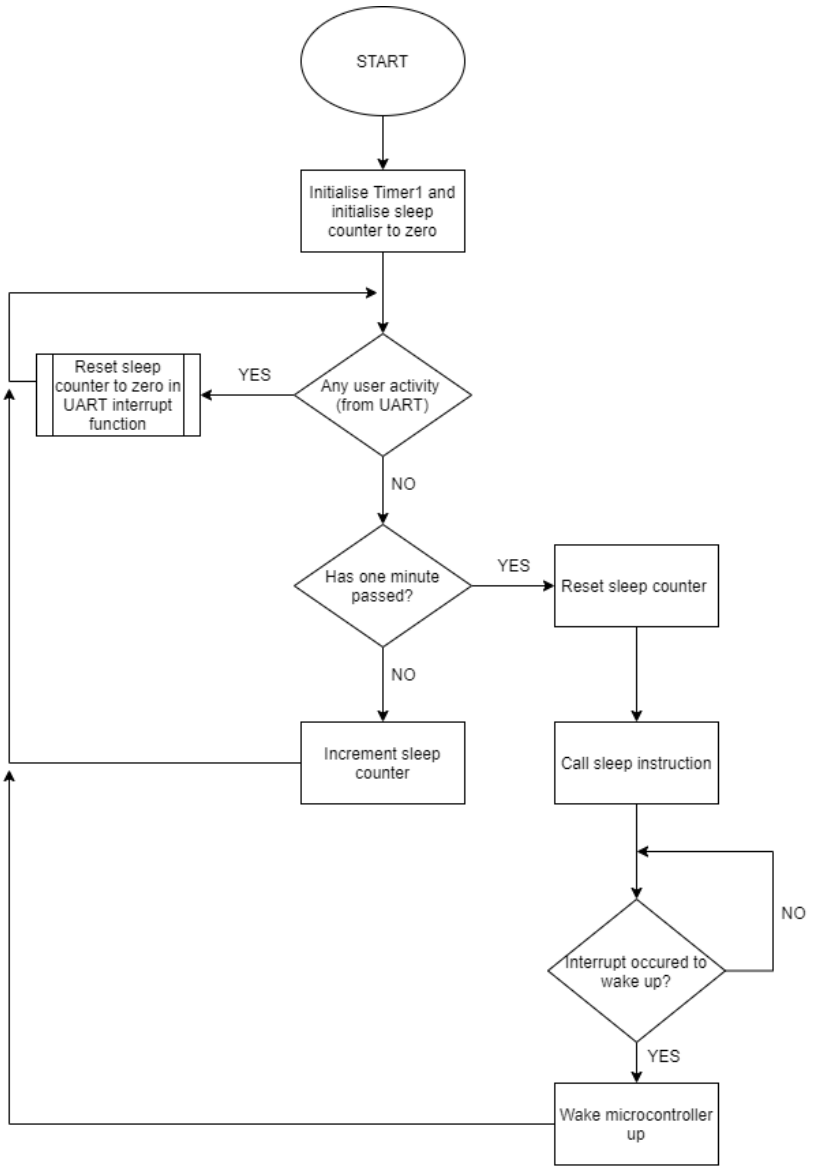

Figure 12: Flowchart for putting the microcontroller in sleep mode, after one minute in being inactive.

The timer is first initialised similar to timer0 in the previous section. As generating a one minute delay is fairly long, a variable is incremented every time the timer overflows by generating an interrupt. The interrupt function also checks if the variable is has reached the target value. If so, the microcontroller is put to sleep. However, if there is any activity (coming from the UART), this variable is reset when entering the UART interrupt function. Equation 8-9 shows the calculations to work out the value to count to is shown below, followed by the simulation result (Figure 13) of microcontroller going to sleep after one minute has passed without any activity.

$$\frac{8MHz}{4} = 2MHz, \qquad PRESCALER = 8, \qquad \frac{2MHz}{8} = 250{,}000 \ (8)$$

$$\frac{250{,}000}{65535} = 3.815 \rightarrow 3.815 * 60 \ seconds = 229 \ (9)$$

Equation 8-9: Calculations for putting the microcontroller to sleep. After the timer has overflown 299 times, one minute will have elapsed of being inactive.

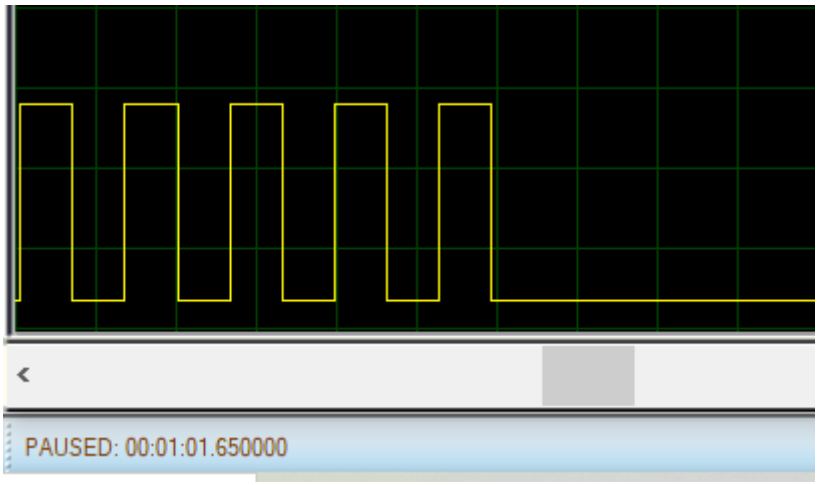

Figure 13: Microcontroller going idle (sleep) after one minute of inactivity.

Although mentioned above is the microcontroller going to sleep, there are different types of sleep modes. The main two being Idle and sleep (can be considered deep sleep). Idle mode is where the peripherals are still clocked, but the main CPU is shutdown. The difference between normal run mode, idle mode and sleep mode is significant in terms of power consumption. From the data sheet run mode's current draw is around 11uA, idle mode is around 2.5uA and sleep mode is around 100nA.

The reasoning behind using idle mode, as opposed to sleep mode is, sleep mode turns off all clocks. Meaning the UART won't be able to trigger an interrupt to wake the microcontroller out of sleep mode. Idle mode keeps the peripheral clocks on, allowing the UART to generate and interrupt and wake up. To switch between run and sleep mode is by setting the IDLEN bit in the OSCCON register, to set the microcontroller to idle mode when a sleep instruction is executed.

Lastly, on the topic of reducing power consumption the main frequency of the microcontroller is running at 8MHz internally. The microcontroller is capable of running at much high frequencies (40MHz), however higher frequencies will draw more current. It is possible to go much lower than 8MHz, however performance will be reduced, along with calculating timer values may turn out to be too small.

5. Verification and Optimisation

Going back to the original specification, the designed system can be analysed to verify if the outcome has been met, as well as where improvements could be made. There were six waveforms to generate, all which have been achieved. The method of using LUTs provided a way to easily generate the required waveform. An alternative method could be to use formulas, especially for generating the sine wave. Although this can give a more accurate representation of the signal, doing complex mathematics in microcontrollers consumes a lot of resources, and may even need to be moved on to a dedicated DSP processor. The downside to using LUTs is it's limited by the size. Bigger LUTs required more memory to store large arrays, which is why in the code the tables were stored in ROM.

A few problems came about when trying to generate an accurate waveform. The output of the DAC was not the correct frequency at higher values. For example, Figure 14 shows the output using the calculations for 100Hz. The output frequency is 53.7Hz, which is too far off 100Hz.

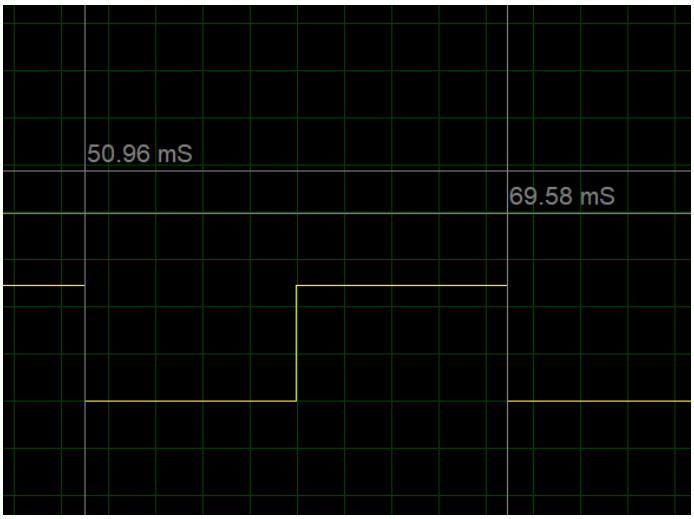

Figure 14: Capture of the oscilloscope showing the incorrect output frequency, despite the correct calculations were used.

To debug this problem, the main code was stripped down to just using the DAC and timer0. Without the DAC function, and monitoring the frequency on a GPIO pin, gave the correct frequency. Adding the DAC function caused delay in the software from executing, despite no delays being present in the function. By a means of trial and error, a few different methods increased the performance. First is reloading the timer before calling the DAC function. The next is to reduce the sample size. Initially 100 samples were used, however reducing it by half helped with generating the output frequency. Lastly, the original DAC function initialised local variables used with function. Removing these variables from the function and making them global was the last part of the fault finding for this part of the code. This is not a common issue and may even be down to the performance of the machine the simulation is running on.

The amplitude has able to be adjusted by changing the VREF pin of the DAC. An alternative to this is doing this in code. Altering the data sent to the DAC, e.g. the number of bits shifted to the right can also be used to change the amplitude. However, when doing it by software there are limitations, whereas doing it through hardware provides more flexibility. If the cost of the device was to be reduced further, than this is a way to reduce it. This is very similar to adjusting the offset, the limitation of doing it in software, like what was implemented. A much better approach would have been to use either a single or dual rail op-amp at the end of the output that will apply a DC bias to the signal.

The interface of the system is done via the USART, which provide an alternative approach to using physical buttons. The downside to this is a host machine or input device needed to control the signal generator. A more optimised approach to the design, would be to have both buttons and a UART control. This way the device could be battery operated, or powered by the source of the UART, similar to how the PCB was designed. In addition, using physical buttons would allow for the device to go into deep sleep mode, further reducing the power consumption. The physical buttons would be able to generate the interrupts to wake the device up.

Lastly, the cost of the system was under the budget of £10, as well as the size was under 25cm^2. The device is small enough to be carrier around, needing only an output cable and a device to send the UART commands, making the device low cost and portable.

6.0 References

Tektronix. (2008, February 20). *Signal Generator Fundamentals.* Retrieved from Tektronix: https://engineering.case.edu/lab/circuitslab/sites/engineering.case.edu.lab.circuitslab/files/docs/Signal_Generator_Fundamentals-_Tektronix.pdf

V. S. Nair, & A. S. Nair. (2018). (2018). Portable wireless multipurpose signal viewer, analyzer and generator using ATMEGA328P MCU and android. Paper presented at the - 2018 3rd International Conference for Convergence in Technology (I2CT), 1-4. https://doi.org/10.1109/I2CT.2018.8529318

7.0 Appendix

7.1 main.c

```
/*      File name:      Signal Generator                        */
/*      Version:  1.0                                           */
/*      Author:         Jerome Samuels-Clarke                   */
/*      Company:        Coventry University                     */
/*      Date:           02/10/21

/*      Program function: This program creates a signal generator, using the UART  to select different waveforms, frequencies and amplitudes. */

/* The following configure operational parameters of the PIC                                    */
#pragma config OSC = INTIO67 //set osc mode to INTIO67
#pragma config WDT=OFF //Watch Dog Timer disabled
#pragma config LVP=OFF //Low Voltage Programming option OFF
#pragma config MCLRE = ON //Master Clear Pin ON
#pragma config DEBUG = OFF // Debug off

// Include the following headers
#include <p18f2420.h> // Device used is the PICF4520
#include <timers.h> // Include the Timer Library
#include <usart.h>  // Include the Usart Library
#include "waveform_lut.h" // Include waveform look up tables

#define CS_0    LATBbits.LATB0 // LATB0 is used as the Chip Select pin for DAC
#define CS_1    LATBbits.LATB4 // LATB4 is used as the Chip Select pin for MCP4151

// Variables used in DAC Note: Initialising these variables inside DAC function, slows down processor
unsigned int c;
unsigned int lower_bits;
unsigned int upper_bits;
unsigned char offset = 0;

unsigned int timer0_value = 45200;      // Variable for storing timer0 value
unsigned char uart_data;                // Variable for storing UART data
unsigned char wave_sel;                 // Variable for which wave to select from UART
```

```
unsigned char amplitude_adj = 128;                  // Variable for adjusting the ampltiude
unsigned char i = 0;                                // Variable for indexing into the look up tables

unsigned int sleep_timer = 0; // Variable for storing the timer1 overflows

// This function is used to communicate with the MCP4151 (digital potentiometer)
void MCP4151(unsigned char data) {
   CS_1 = 0; // Chip Select low
   SSPBUF = 0x00; // Send configuration and address
   while (!SSPSTATbits.BF); // wait until the first byte is sent
   SSPBUF = data; // Send data
   while (!SSPSTATbits.BF); // wait until the second byte is sent
   CS_1 = 1; // Chip Select high
}

// This function is used to communicate with the DAC (MCP4921)
void DAC(unsigned int data) {
  c = (((data+offset) + 1)*8) - 1; // Shift data to make it 12 bits
//c=data+offset;
   upper_bits = c / 256;   // Store upper 8 bits
   upper_bits = (48) | upper_bits; // Append configurations on upper 8 bits
   lower_bits = 255 & c; // Store lower 8 bits
   CS_0 = 0; // Chip Select low
   SSPBUF = upper_bits; // Send upper 8 bits
   while (!SSPSTATbits.BF); // wait until the first byte is sent
   SSPBUF = lower_bits; // Send lower 8 bits
   while (!SSPSTATbits.BF); // wait until the second byte is sent
   CS_0 = 1; // Chip Select high
}

/* This is the interrupt service routine for Timer1. When the sleep_timer reaches
the value, the micrcontroller will go to sleep */
void TMR1_ISR(void)
{
   PIE1bits.TMR1IE = 0;          // Disable timer1 interrupts
```

```
    WriteTimer1(0); // Clear Timer

    sleep_timer++;
    if (sleep_timer >= 229) {
        sleep_timer = 0;
        Sleep();
    }

    PIE1bits.TMR1IE = 1; // Enable timer1 interrupts
    PIR1bits.TMR1IF = 0; // Clear timer1 flag
}

/* ----- Interrupt service routine UART_ISR bellow: ---- */
void UART_ISR(void) {

    PIE1bits.RCIE = 0; // Disable timer1 interrupts

    sleep_timer = 0; // Reset sleep timer if there is any activity

    // wait for user input and make sure data is ready
    while (!DataRdyUSART());

// Get character from buffer store in variable. Depending on data, perform a certain action
    uart_data = getcUSART();
    switch (uart_data) {
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
            wave_sel = uart_data; // If data is between 1:6, store for selecting waveform
            break;
        case 'q':
```

```
        timer0_value = 45200;              // Set timer for 2Hz frequency
        break;
    case 'w':
        timer0_value = 57500;                      // Set timer for 5Hz frequency
        break;
    case 'e':
        timer0_value = 61550;                      // Set timer for 10Hz frequency
        break;
    case 'r':
        timer0_value = 63620;              // Set timer for 20Hz frequency
        break;
    case 't':
        timer0_value = 64830;              // Set timer for 50Hz frequency
        break;
    case 'y':
        timer0_value = 65227;              // Set timer for 100Hz frequency
        break;
    case 'n':
        MCP4151(amplitude_adj++);                  // Increase amplitude
        break;
    case 'm':
        MCP4151(amplitude_adj--);                  // Decrease amplitude
        break;
    case 'i':
        offset += 25;              // Increase offset
        break;
    case 'o':
        offset -= 25;              // Decrease offset
        break;
      //default:
       //    putrsUSART("\r\nInvalid option selected.\r\n\n");                    // Uncomment to see when data in invalid
}

  // putcUSART(uart_data); // display character entered                // Uncomment to see what characters are being typed
```

```
    PIE1bits.RCIE = 1;                          // Enable timer1 interrupts
    PIR1bits.RCIF = 0;              // Clear UART flag
}



// ----- Interrupt routine TMR0_ISR follows bellow: ----
void TMR0_ISR(void)
{
    INTCONbits.TMR0IE = 0; // Disable timer0 interrupts


    WriteTimer0(timer0_value); // Reset timer0 to selected frequency


            // If the size of the look up table as has been reached, rest variable to zero
    if (i >= SIZE - 1)
        i = 0;


            // Increment value in look up table, depending on what waveform has been selected
    switch (wave_sel) {
        case '1':
            DAC(triangle_lut[i++]);
            break;
        case '2':
            DAC(STRU_lut[i++]);
            break;
        case '3':
            DAC(STRD_lut[i++]);
            break;
        case '4':
            DAC(square_lut[i++]);
            break;
        case '5':
            DAC(sine_lut[i++]);
            break;
        case '6':
            DAC(pulse_lut[i++]);
```

```
            break;
        default:
            DAC(square_lut[i++]);

    }

    INTCONbits.TMR0IE = 1; // Enable timer0 interrupts
    INTCONbits.TMR0IF = 0; // Clear timer0 flag
}

// ----- Bellow we test for the source of the low interrupt ----
#pragma interrupt low_isr
void low_isr(void) {
    if (INTCONbits.TMR0IF == 1) // Was interrupt caused by Timer 0?
    {
        TMR0_ISR(); // Yes , execute TMR0 ISR program
    }
}

#pragma code low_interrupt = 0x18 // High Interrupt vector @ 0x08
void low_interrupt(void) // At location 0x18 instruction GOTO
{
    _asm // This is the assembly code at vector location
    GOTO low_isr
    _endasm
}
#pragma code // used to allow linker to locate remaining code


// ----- Bellow we test for the source of the high interrupt ----
#pragma interrupt high_isr
void high_isr(void) {
    if (PIR1bits.TMR1IF == 1) // Was interrupt caused by Timer1?
        TMR1_ISR(); // Yes , execute TMR1 ISR program
    if (PIR1bits.RCIF == 1) // Was interrupt caused by UART ?
```

```
      UART_ISR(); // If yes , execute UART program
}

#pragma code high_interrupt = 0x08 // High Interrupt vector @ 0x08
void high_interrupt(void) // At location 0x08 instruction GOTO
{
   _asm // This is the assembly code at vector location
   GOTO high_isr
   _endasm
}
#pragma code // used to allow linker to locate remaining code

void main(void)
{
   OSCCONbits.IDLEN = 1; // Device enters an Idle mode on SLEEP instruction

// Internal Oscillator Frequency Select bits set to 8MHz and use internal oscillator block
   OSCCONbits.IRCF0 = 1;
   OSCCONbits.IRCF1 = 1;
   OSCCONbits.IRCF2 = 1;
   OSCCONbits.SCS0 = 1;
   OSCCONbits.SCS1 = 1;

   ADCON1 = 0x0F; // Set Ports as Digital I/O rather than analogue

            // Set RB0 and RB4 as outputs and clear the bits
   TRISBbits.RB0 = 0;
   TRISBbits.RB4 = 0;
   LATBbits.LATB0 = 1;
   LATBbits.LATB4 = 1;

            // Set RC3, RC5 and RC6 as outputs, RC7 as an input and clear the register
   TRISCbits.RC3 = 0;          // SPI SCK
   TRISCbits.RC5 = 0;          // SPI SDO
   TRISCbits.RC6 = 0;          // UART TX
```

```
    TRISCbits.RC7 = 1;              // UART RX
    PORTC = 0;

            // SPI Initialisation
    SSPSTAT = 0xC0; //Status Register SSPSTAT=11000000
    SSPCON1 = 0x20; //Enables serial port pins & set the SPI clock as clock = FOSC/4

    // Enable Interrupts Registers and Bits

    IPR1bits.RCIP = 1; // EUSART Receive Interrupt Priority bit - High Priority
    INTCONbits.GIEL = 1; // Enables all low ?priority peripheral interrupts
    RCONbits.IPEN = 1; // Enables interrupt priority

    PIE1bits.RCIE = 1; // EUSART Receive Interrupt Enable bit - Enable
    PIR1bits.RCIF = 0; // Clear the EUSART Interrupt Flag

    INTCON2bits.TMR0IP = 0;   // Set timer0 as low priority
    INTCONbits.TMR0IE = 1;              // Enable Timer 0 Interrupt
    INTCONbits.TMR0IF = 0;              // Clear the Timer 0 Interrupt Flag

    IPR1bits.TMR1IP = 1;                        // Set timer1 as high priority
    PIE1bits.TMR1IE = 1;                        // Enable Timer1 Interrupt
    PIR1bits.TMR1IF = 0;                        // Clear the Timer1 Interrupt Flag

    INTCONbits.GIE = 1; // Enable global interrupts

/* Timer0 Initialisation
Close The Timer 0 if it was previously Open
Configure the Timer 0 with Interrupt, ON, 16 bits, internal clock source and Prescaler of 1:1
Set timer at a default value(100Hz) */
    CloseTimer0();
    OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_1);
    WriteTimer0(timer0_value);
```

```
/* Timer1 Initialisation

Close The Timer1 if it was previously Open

Configure the Timer1 with Interrupt, ON, 16 bits, internal clock source and Prescaler of 1:8

Set timer at a default value(100Hz) */

   CloseTimer1(); // Close The Timer 0 if it was previously Open

   OpenTimer1(TIMER_INT_ON & T1_16BIT_RW & T1_SOURCE_INT & T1_PS_1_8);

   WriteTimer1(0); // Clear Timer


   // Set USART Parameters Baud rate 9600 Baud, 8 data bits 2 stop bits , RX interrupt and Asynchonous mode

   OpenUSART(USART_TX_INT_OFF & USART_RX_INT_ON & USART_ASYNCH_MODE & USART_EIGHT_BIT &
USART_CONT_RX & USART_BRGH_HIGH, 51);


            //Sent the string(s) to terminal

   putrsUSART(" *** PIC18 Signal Generator *** \r\r\n\n ");

   putrsUSART("Select a waveform:\r\n (1) Triangle          (q) 2Hz\r\n (2) Sawtooth Ramp-Up     (w) 5Hz\r\n (3) Sawtooth
Ramp-Down    (e) 10Hz\r\n (4) Square              \

            (r) 20Hz\r\n (5) Sine              (t) 50Hz\r\n (6) Pulse             (y) 100Hz\r\n (n) Increase Amplitude    (m) Decrease
Amplitude\r\n (i) Increase Offset      \

            (o) Decrease Offset\r\n");

   while (1) {

   }

}
```

7.2 waveform_lut.h

```
#ifndef WAVEFORM_LUT_H
#define    WAVEFORM_LUT_H

#define SIZE              50   // Size of lut

// Look up table for generating a triangle wave
unsigned char rom triangle_lut[SIZE] = {
    10,  20,  31,  41,  51,  61,  71,  82,  92, 102,
   112, 122, 133, 143, 153, 163, 173, 184, 194, 204,
   214, 224, 235, 245, 255, 245, 235, 224, 214, 204,
   194, 184, 173, 163, 153, 143, 133, 122, 112, 102,
    92,  82,  71,  61,  51,  41,  31,  20,  10,   0
};

// Look up table for generating a ramp-up sawtooth wave
unsigned char rom STRU_lut[SIZE] = {
     5,  10,  15,  20,  26,  31,  36,  41,  46,  51,
    56,  61,  66,  71,  77,  82,  87,  92,  97, 102,
   107, 112, 117, 122, 128, 133, 138, 143, 148, 153,
   158, 163, 168, 173, 179, 184, 189, 194, 199, 204,
   209, 214, 219, 224, 230, 235, 240, 245, 250, 255
};

// Look up table for generating a ramp-down sawtooth wave
unsigned char rom STRD_lut[SIZE] = {
   255, 250, 245, 240, 235, 230, 224, 219, 214, 209,
   204, 199, 194, 189, 184, 179, 173, 168, 163, 158,
   153, 148, 143, 138, 133, 128, 122, 117, 112, 107,
   102,  97,  92,  87,  82,  77,  71,  66,  61,  56,
    51,  46,  41,  36,  31,  26,  20,  15,  10,   5,
};

// Look up table for generating a square wave
```

```
unsigned char rom square_lut[SIZE] = {
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255
};

// Look up table for generating a sine wave
unsigned int rom sine_lut[SIZE] = {
  128, 144, 160, 175, 190, 204, 216, 227, 237, 244,
  250, 253, 255, 254, 252, 247, 241, 232, 222, 210,
  197, 183, 168, 152, 136, 119, 103,  87,  72,  58,
   45,  33,  23,  14,   8,   3,   1,   0,   2,   5,
   11,  18,  28,  39,  51,  65,  80,  95, 111, 128
};

// Look up table for generating a pulse wave at 20% duty cycle
unsigned char rom pulse_lut[SIZE] = {
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
};

#endif     /* WAVEFORM_LUT_H */
```