2245
DSP & Microprocessor Project
COSC6143C, EEET7062C, EEET7043C

S3834951

Mitchell Reynolds

Project Report

Contents

3 – Project outline and requirements

5 – Timelines

6 – Schematics

8 – Bill of Materials

9 – Building the board

13 – Build errors and hardware troubleshooting

14 – Testing input filter and limiting circuit

18 – Testing DAC output

21 – Testing input/output program

26 – Final program and testing

33 – Square wave harmonics demonstration

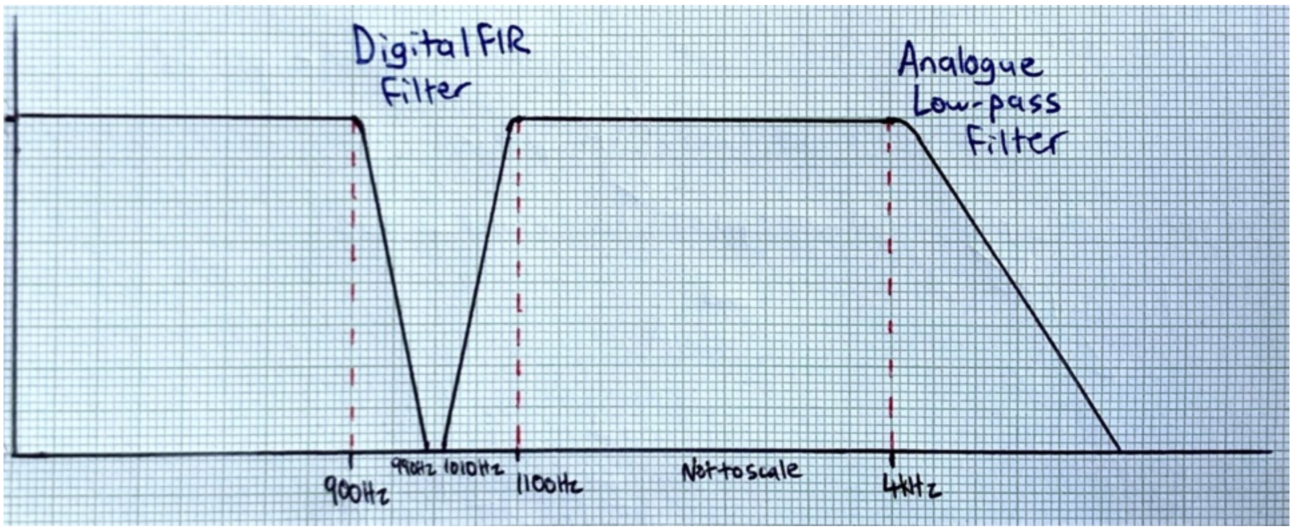
35 – Reflections

Project outline and requirements

The aim of this project is to "design, develop, implement, test and document a digital filter using an embedded controller". The filter will remove a 1kHz frequency from an input signal of 0-4kHz with a max amplitude of 3.3V.
The maximum frequency (4 kHz) will be limited by the hardware we will build, with 4-stage low-pass filters at the input and output, as well as a limiting circuit cutting off the max input amplitude at 3.3V to protect the embedded controller from damage.

The digital filter's specifications are as follows:

Filter Implementation	FIR
Pass band ripple	0.5dB
Stop band attenuation	25dB
Pass band	900 Hz – 1100Hz
Stop band edge frequencies	990Hz – 1010Hz
Sampling frequency	9.6 KHz



The embedded controller that we are using is a **Motorola DSP56F803**. The datasheet can be found here.

Important points for our project regarding the Motorola DSP56F803 are:
- It is a 16-bit controller
- It uses Serial Communication Interface (SCI) and Serial Peripheral Interface (SPI)
- It has two 4-channel 12-bit ADCs. We will only require one.
- DO and REP loops are carried out on the hardware.
- As many as 40 Million Instructions Per Second (MIPS) at 80MHz core frequency, so our processing of a max 4kHz signal is very light work for this processor.
- It uses parallel instructions for certain commands, so we can process data at the same time as moving other data (as an example). It utilises Harvard-style architecture

consisting of three execution units operating in parallel, allowing as many as six operations per instruction cycle.
- The DSP core has a pipelined nature, and some instructions cannot be done immediately after one another. Any time you write to an AGU register immediately followed by an instruction using that same register in an address arithmetic calculation, you will encounter this pipeline issue. This can be solved simply by adding a "nop" line between these commands (or finding an alternate method). An example of commands which would not work is below:

```
MOVE   #$7,N              ; Write to the N register
MOVE   X:(R2)+N,X0        ; N register used in address
                          ; arithmetic calculation
```

Design requirements as quoted in the brief include using:
1. Built-in timers to generate an interrupt for time reference.
2. SPI sub-system to transmit data serially
3. Built-in A/D converter of the micro-controller
4. Serial D/A converter for this project
5. Assembly and C as coding languages with the linker linking both programs

All quality requirements for the build design were met. We received a schematic and the circuit design was standard (with very minor variations), but the board layout was entirely up to the student.
The requirements were:
- Using veroboard
- Neat soldering with no wires soldered on the underside
- Short soldered wire which does not hang in the air
- BNC connectors for input/output AC signals
- Test points on the board, and additional pins to connect ground to oscilloscope
- The ability to isolate each circuit from each other with jumpers
- Indenting the software properly for readability and adding comments
- Keeping the hardware cost under $100 AUD

Estimated Timeline (previously submitted)

Phase 1:
Week 5 – created components list from schematic and information in lecture
Week 6 – ordered in bulk all components from Mouser, board from Core Electronics
Week 7 – components and board arrived, begun planning board layout on a solderless breadboard
Week 8 – finish plan, begin soldering on the board
Mid semester break – finish soldering and testing

Week 10 – bring the board into class. Test that we can transmit the data to the board and have its input filter remove the undesired high frequency signals as per the specs. Compare the input and output signals on the oscilloscope.

Phase 2:
Week 11 – Begin modifying the DSP system from phase 1 to implement a FIR filter which will remove 1kHz interference as per the specs.
Week 12 – Continue working on DSP modification for rest of semester.
Week 13 -
Week 14 – Write report, and submit.
Week 15 -

Actual Timeline (upon reflection)

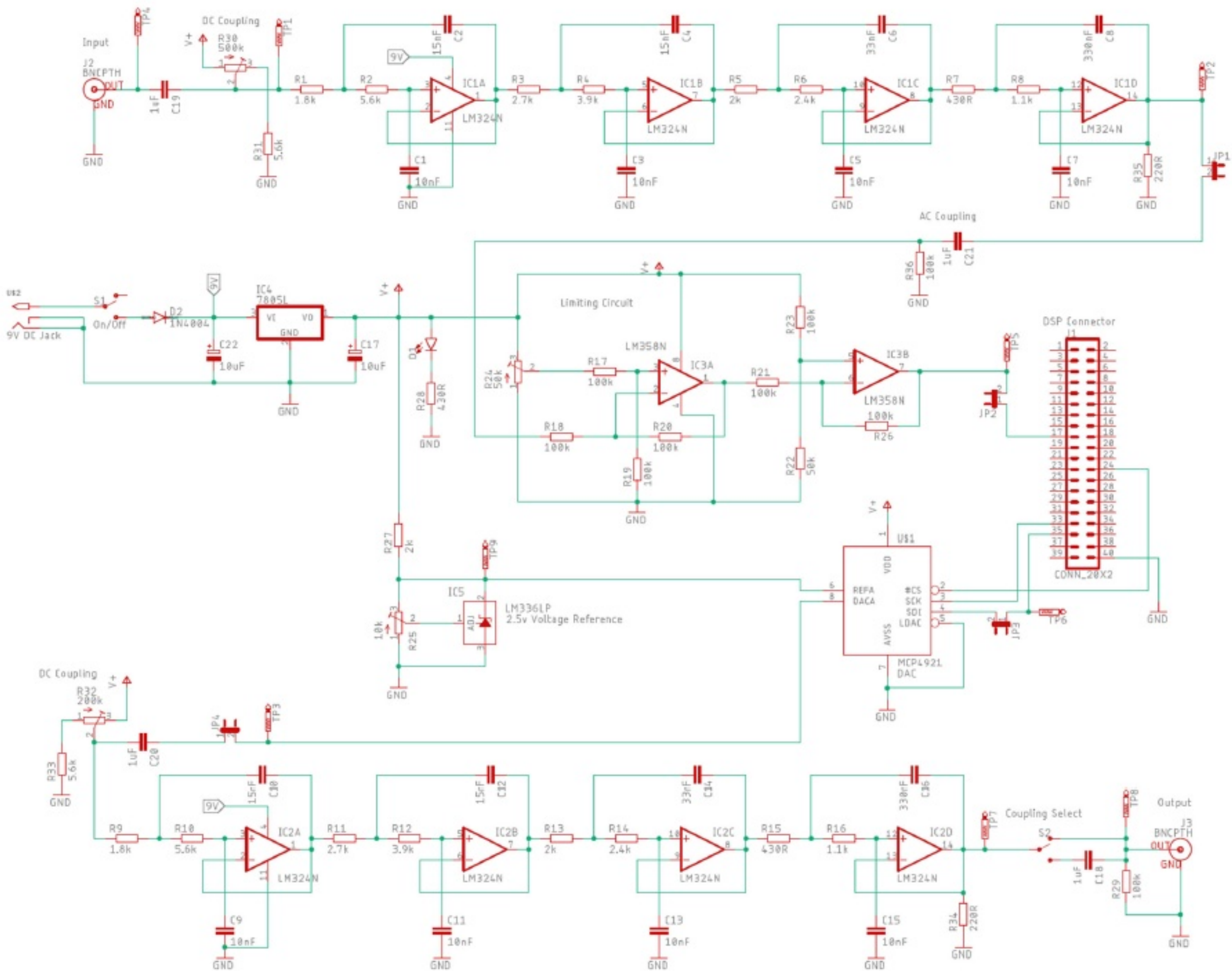
Phase 1:
Week 5 – created components list from schematic and information in lecture
Week 6 – ordered in bulk all components from Mouser, board from Core Electronics
Week 7 – components and board arrived, begun planning board layout on a solderless breadboard
Week 8 – finish plan, begin soldering on the board
Mid semester break – finish soldering and testing

Week 10 – bring the board into class. **Found minor errors on board, fixed some connections (errors documented later in report), and successfully checked input filter on oscilloscope.**

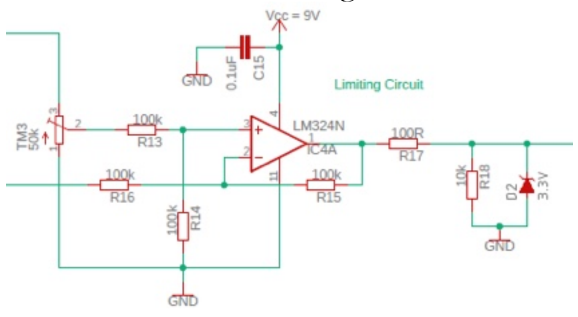
Phase 2:
Week 11 – **Successfully tested the DAC chip, separated from the rest of the circuit.**
Week 12 – **Begin writing code in C and Assembly** to create an FIR filter which will remove 1kHz interference as per the specs.
Week 13 – **Continue writing code and testing SPI transmission to the board.**
Week 14 – **Continue writing code and testing the FIR filter.**
Week 15 - Write report, and submit.

Schematics

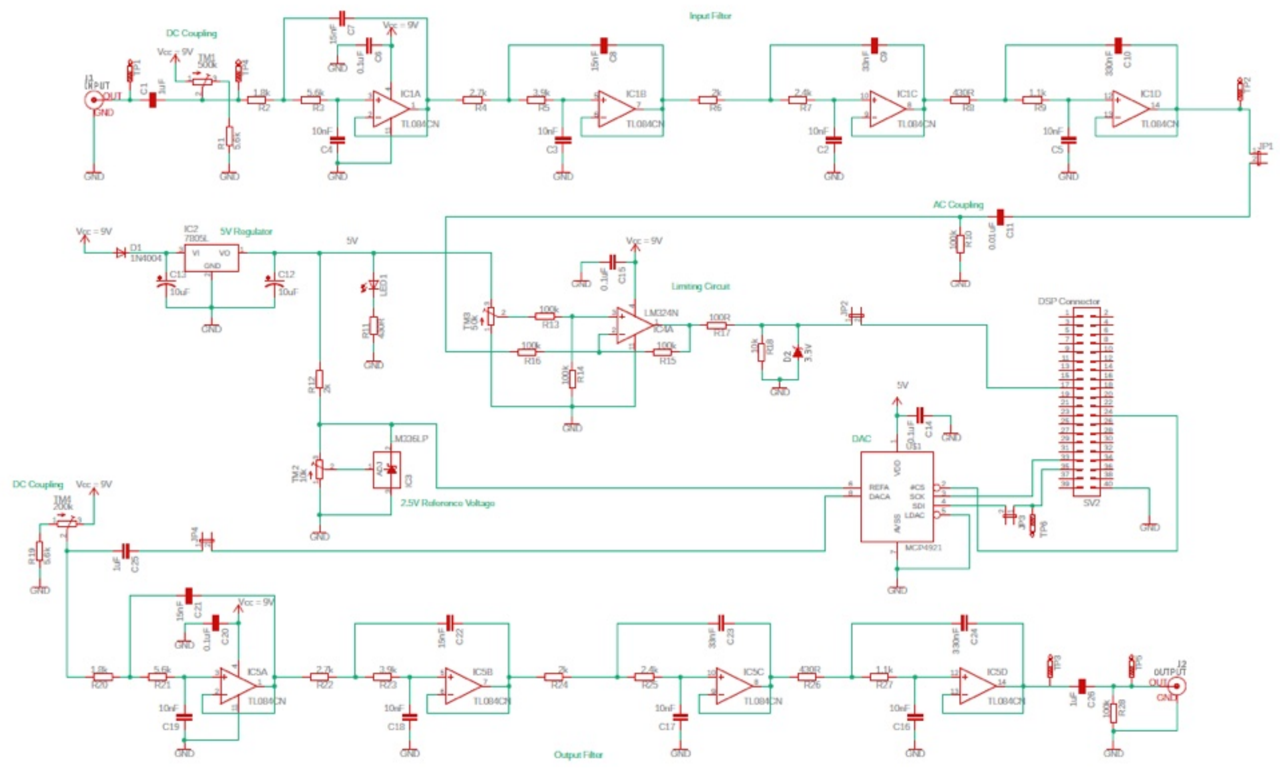
The original schematic provided is not the final version of the board. It includes an op-amp at the output of the limiting circuit (IC3B) that was not used (we used a zener instead), it has no coupling capacitors on the Vcc for the ICs, and it uses an incorrect op-amp for the input & output filters.
Nevertheless, it provided a starting point which we could use to build the BOM.



We removed IC3B, R26, R23 and R22 (the op-amp and resisters at output of limiting circuit and changed R21 from 100k ohm to 100 ohm. The circuit that we built using a 3.3V zener in reverse as this provides an exact limit, anything over 3.3V is transferred to ground through the diode at breakdown voltage. The modified circuit is as follows:



New schematic, created in Eagle after final build and modifications:
Link to full quality pdf here.



Notes on modifications:

- 0.1 µF ceramic capacitors added to Vcc pin of each IC in parallel to ground.

- Female pin headers used for 9V and GND inputs, so input jack has been removed.

- No DC/AC switch was used at the end, as we can simply use the testing point at final output of the filter if we want to check the DC biased output, otherwise the BNC connector will provide the AC coupled output.

- Texas Instruments TL084CN quad op-amps were used for the input and output filters, as this has a very high slew rate. We required it to be more than 0.55V/µs and it has a slew rate of 13 V/µs – refer to specs at Mouser.

- A Central Semiconductors 1N4620 BK zener diode was used in the limiting circuit, as the recommended specs we were provided in class was that it should be low power, only 0.25-0.5W and cut off at 3.3V. Refer to specs at Mouser.

Bill of Materials

Sourced almost entirely from Mouser, with the protoboard being purchased from Core Electronics.

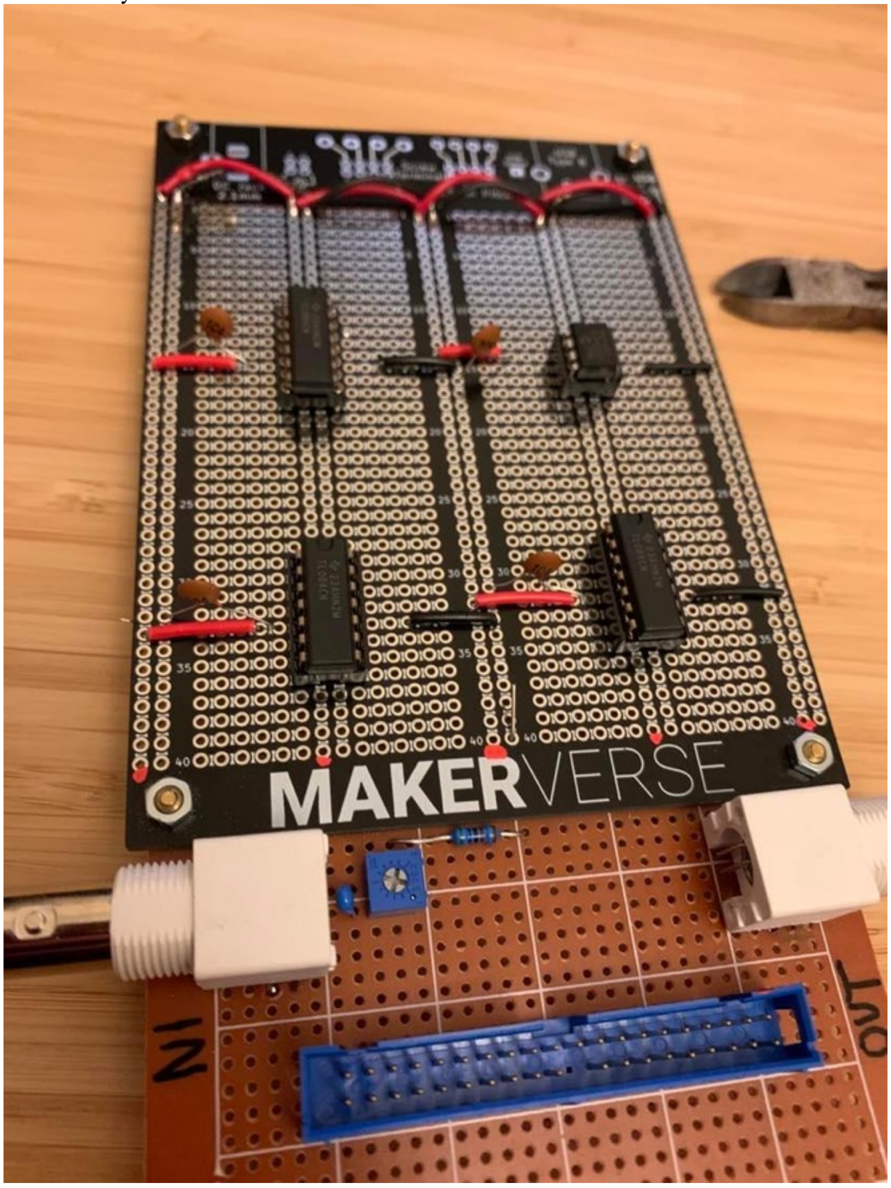
Component	Quantity	Price
MCP4921-E/P DAC IC	1	$4.16
1N4620 Zener Diodes 3.3V	1	$1.74
TL084CNE4 quad op-amp dual supply	3	$2.73
Jumper shunts M7583-46	10	$0.95
Right angle BNC connectors	2	$9.06
1uF 10% Multilayer Ceramic Capacitors	4	$1.94
0.01uF Multilayer Ceramic Capacitors	7	$2.46
0.015uF 10% Multilayer Ceramic Capacitors	3	$1.10
0.033uF 10% Multilayer Ceramic Capacitors	2	$0.70
0.33uF 10% Multilayer Ceramic Capacitors	2	$1.06
10uF Aluminium Electrolytic Capacitors	2	$1.14
500K OHMS trimpot Bourns	1	$1.50
200K OHMS trimpot Bourns	1	$1.50
50K OHMS trimpot Bourns	1	$1.64
430 ohm 1/4W 1% resistor	3	$0.53
1.1k ohm 1/4W 1% resistor	2	$0.32
220 ohm 1/4W 1% resistor	3	$0.48
100k ohm 1/4W 1% resistor	9	$1.32
50k ohm 1/4W 1% resistor	1	$0.16
5.6k ohm 1/4W 1% resistor	4	$0.64
1.8k ohm 1/4W 1% resistor	2	$0.29
2.7k ohm 1/4W 1% resistor	2	$0.29
3.9k ohm 1/4W 1% resistor	2	$0.29
2k ohm 1/4W 1% resistor	3	$0.48
2.4k ohm 1/4W 1% resistor	2	$0.35
1N4004-B Rectifier diode	1	$0.31
L7805CV Linear Voltage Regulators 5.0V 1.5A Positive	1	$1.01
LM336Z-2.5/LFT7 voltage regulator 2.5V	1	$1.57
40pin vertical shroud header	1	$3.87
Makerverse protoboard with shipping	1	$15.10
	Total =	**$65.05**

Building the board

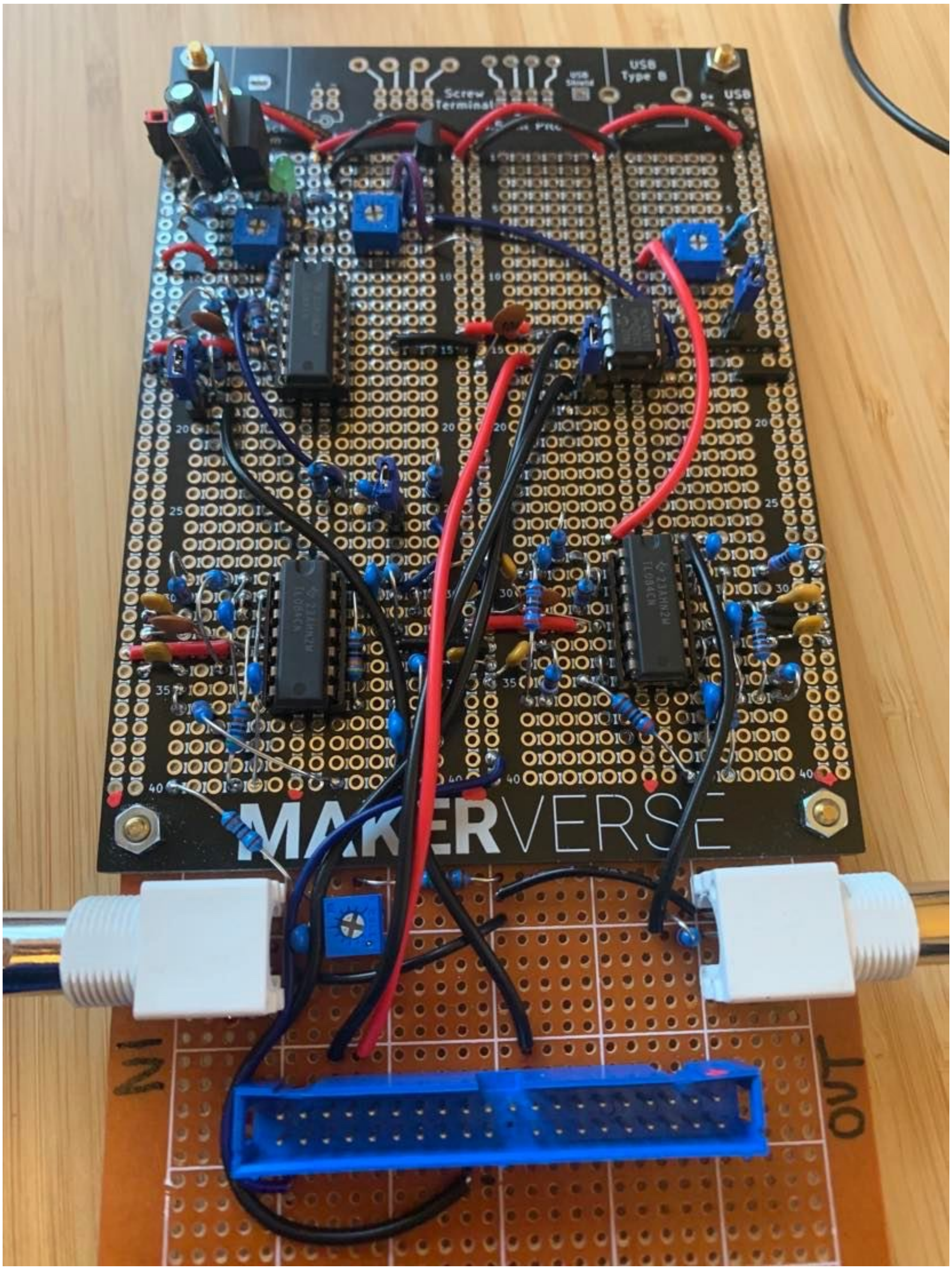
I built this using a Makerverse Protoboard, which has the layout of a solderless breadboard. I did this so that I could have power and ground rails laid out all the way down the board, and so that I could use the horizontal rows for extra components branching in/out of the IC's.
It also made planning it very simple, as I could do a full layout on the board without soldering.

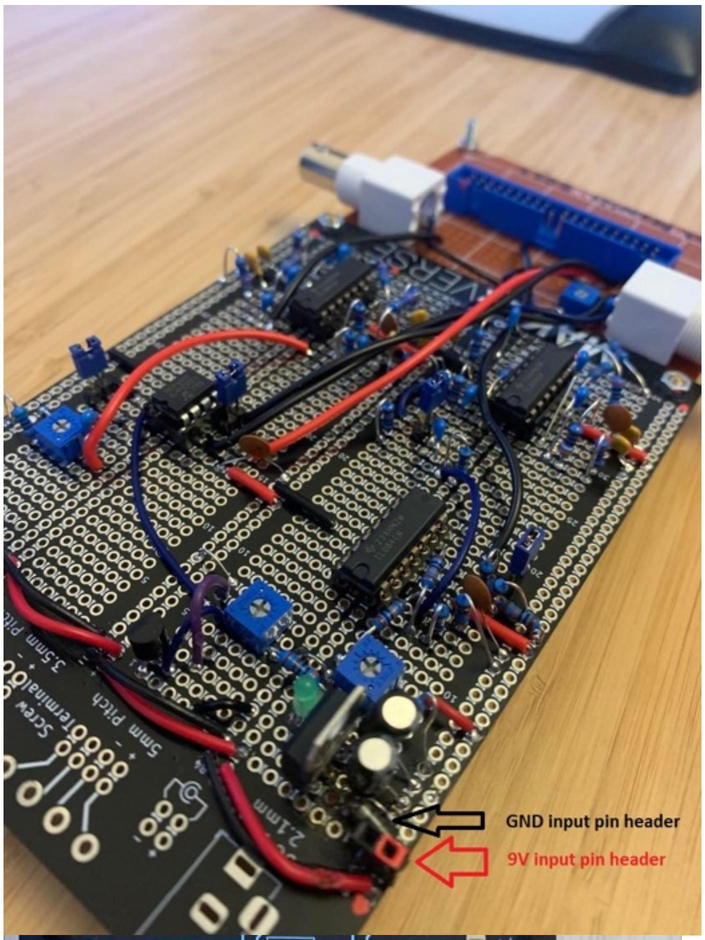
I also attached a standard perfboard to the bottom which functions as the I/O section. This was because the right angle BNC connectors had large mounts which did not fit through the holes of the protoboard, and the 40-pin connector requires individual connections to each pin (so it would not fit on a breadboard style board without cutting tracks), the simplest and safest option was to use the additional perfboard.

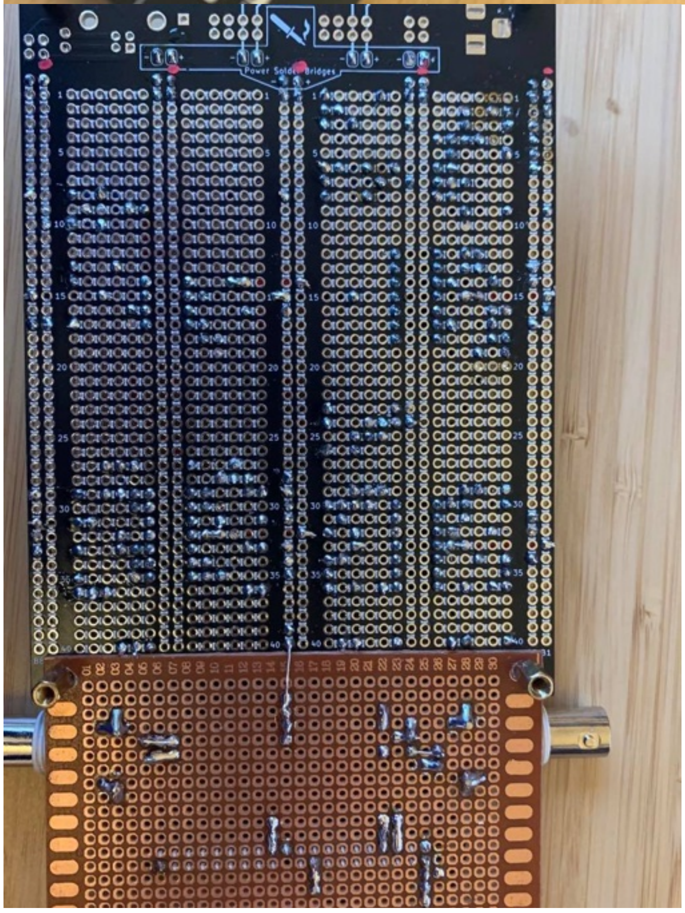
Initial layout of IC's and I/O board:

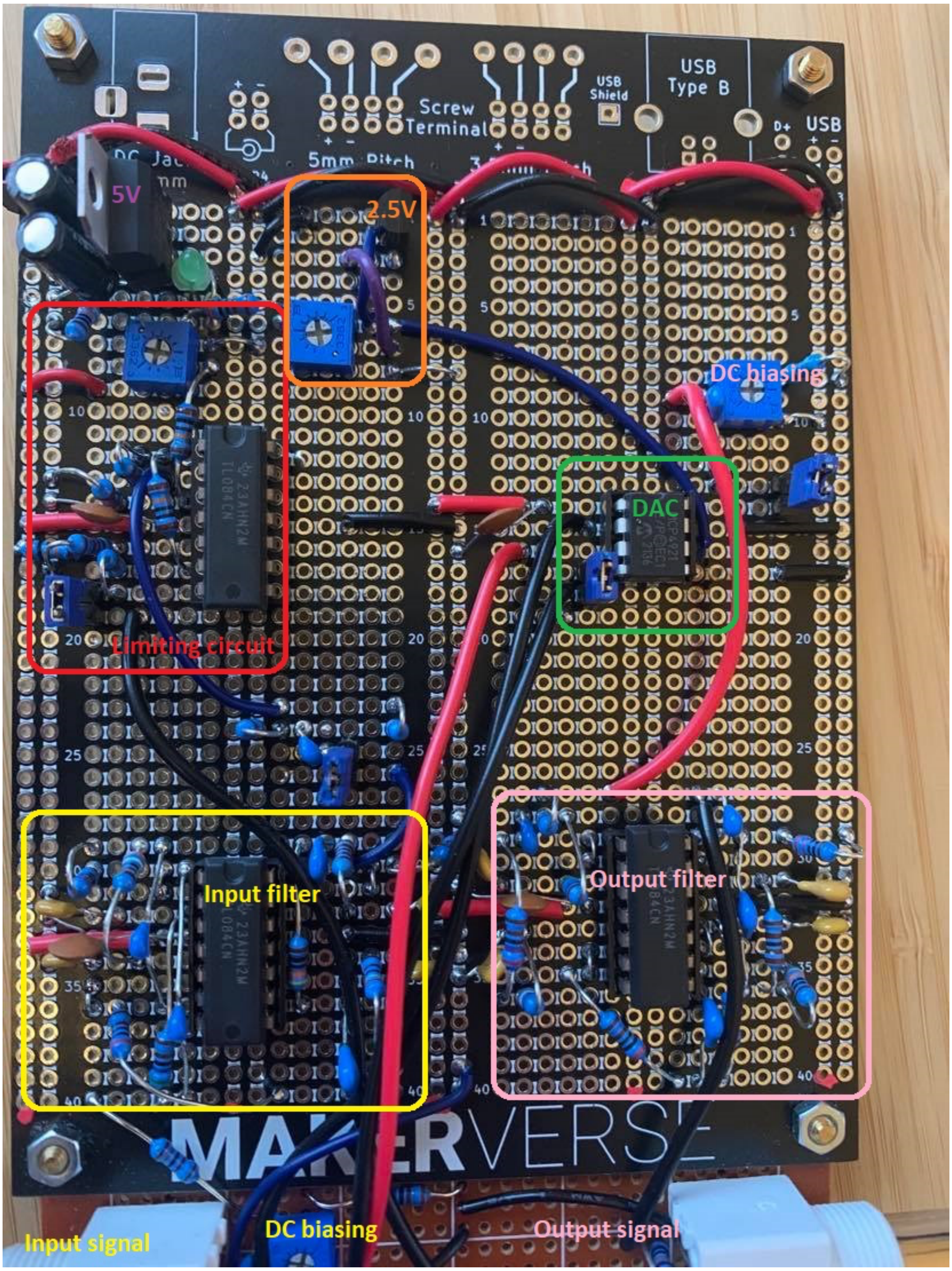


Finished build:




GND input pin header
9V input pin header
2.1mm
5mm Pitch
3.5mm Pitch
Screw Terminal


Power Solder Bridges


5V
2.5V
DC biasing
Limiting circuit
DAC
Input filter
Output filter
Input signal
DC biasing
Output signal
Screw Terminal
USB Type B
MAKERVERSE

Build errors and hardware troubleshooting

Testing the board step by step (documented later) resulted in a number of errors being found on the board, which needed to be fixed.
These errors, some by design of the original schematic, and some of my own build faults, are documented below:

1. Vcc of DAC (pin 1) was going to 9V, and should have been 5V. I powered up the unit once, before realising on the schematic that it should be 5V. This was of no consequence, and the DAC worked fine after moving power the 5V.

2. Stage 4 of the input filter was creating bad distortion. I swapped IC's from the output filter and the distortion remained. We also then tested Aldrin's board and found the same error. It was discovered that the 220 ohm resister on this stage of the filter was causing the issue, so this was removed. Note: this is removed in my schematic.

3. At the output of the limiting circuit, the signal was still biased above 0V, rather than having the signal have it's lowest value perfectly at 0V as intended.
In discussions about op-amp IC's with other students, I realised that I had used the same rail-to-rail quad op-amp IC for each op-amp in the circuit (the TL084CN).

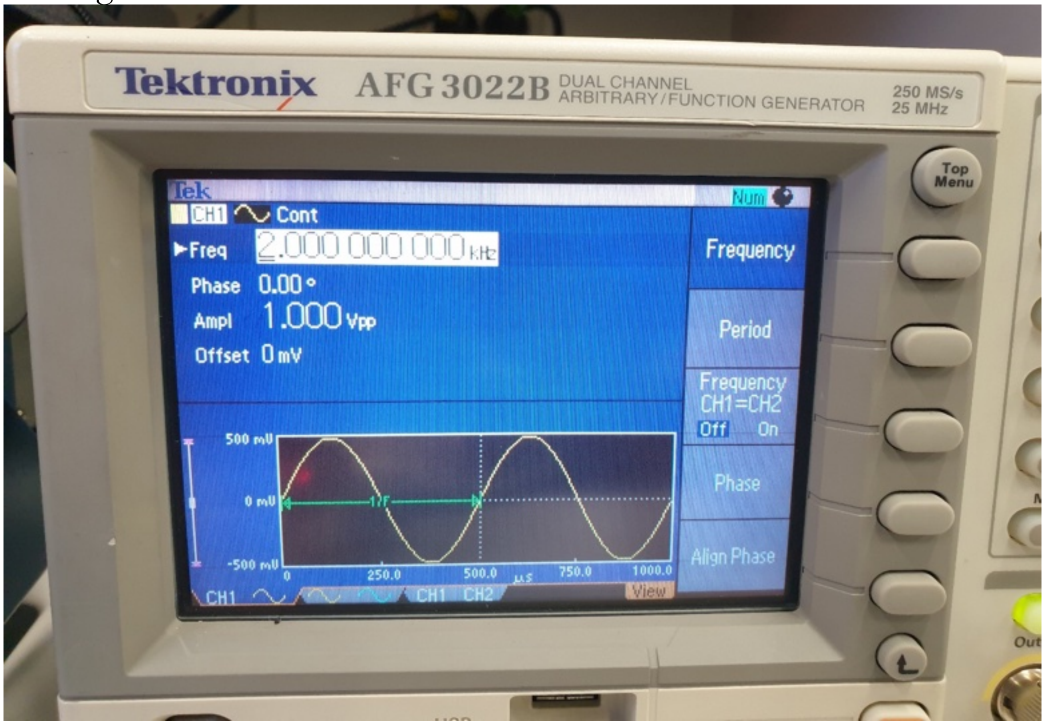
This *should* in theory be fine, if your rails are 0 and +Vcc, but in this case it was not biasing the signal from 0 to +Vcc, and it was raising it. The limiting circuit was operating at approx. 1-4.4V instead of 0-3.3V.
After swapping it out for an LM324, a single supply op-amp IC with the same pinout, the limiting circuit worked perfectly. I had used IC sockets in my build, so switching the IC's was no issue.

Sukhvir (without knowing I'd used a dual supply op-amp) and myself considered that it was the AC coupling capacitor after the first filter before the limiting circuit that was causing this, so I replaced the 1uF capacitor with a 0.01uF and noted there was still no difference.
I left this coupling capacitor in place, as the smaller value capacitance should make no difference to overall operations.

4. The RC high pass filter (coupling capacitor in series and resistor parallel to ground) at the output stage of the circuit where the BNC connector is, did not pass any signal. Final testing needed to be done with the DC biased signal before this final capacitor, and this issue was present on other students boards too. Lukas, Daniel and Aldrin were in my group and we all tested our boards with the DC biased output. The cause of this is still unknown and needs further investigation. I suspect that if the resistor was not going parallel to ground and instead was in series with the capacitor, it would create an impedance to draw current through and signal would pass properly, but the way it is built like a high pass filter, it is not providing the necessary impedance.

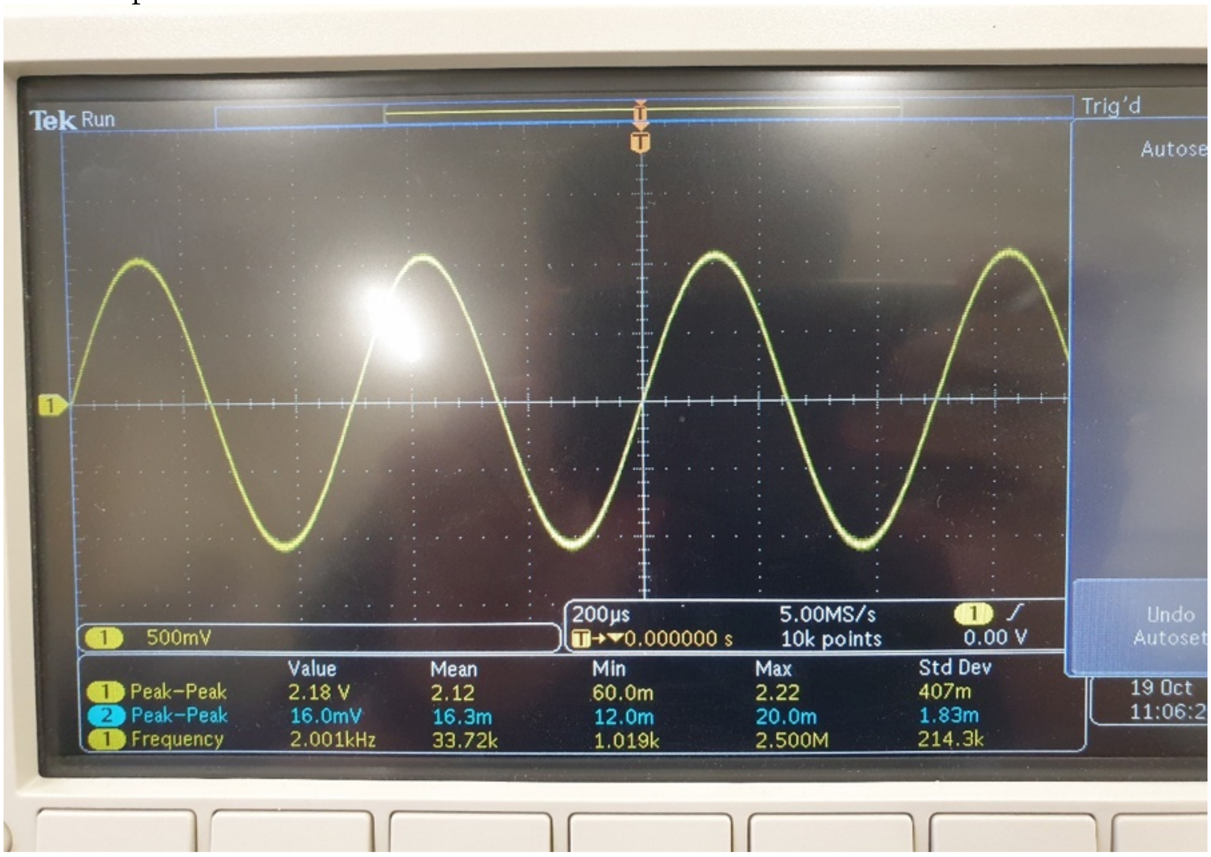
Testing input filter and limiting circuit

Testing with a 2Khz 1Vp-p signal. We increase frequency and amplitude to test filter and limiting circuit.



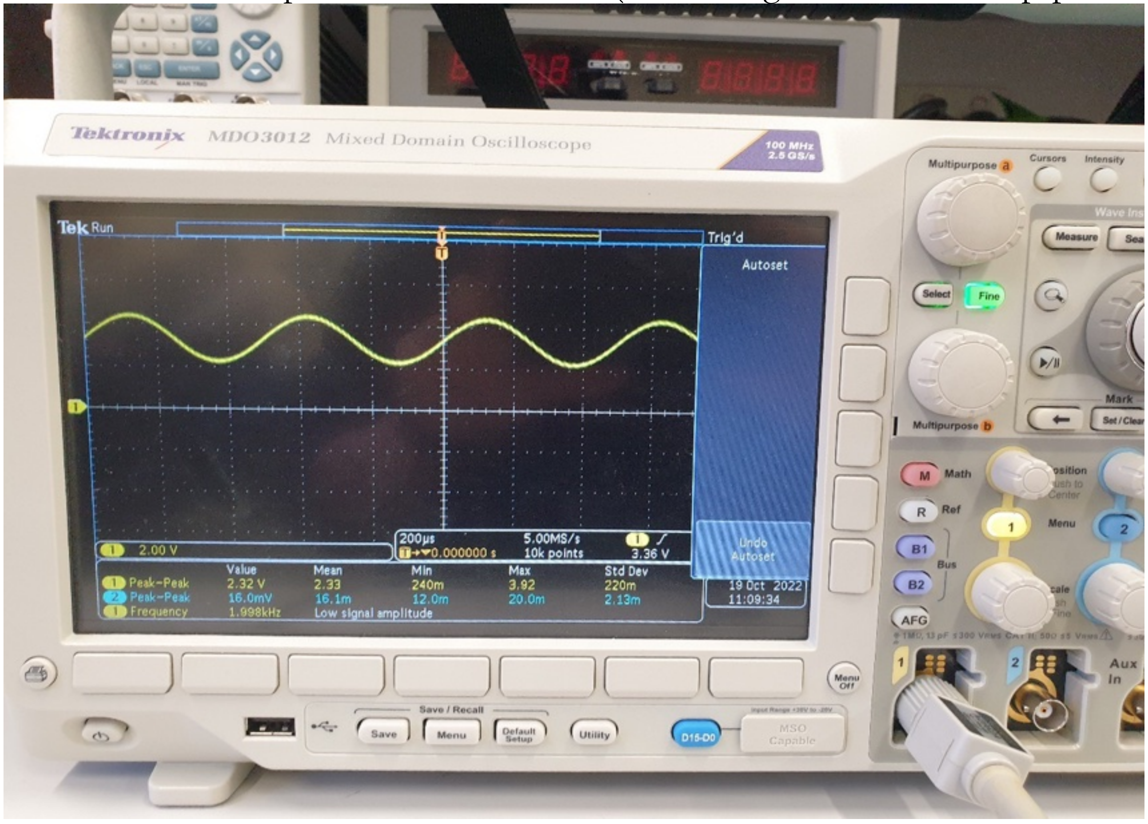
```
Tek
CH1  Cont
►Freq   2.000 000 000 kHz
 Phase  0.00 °
 Ampl   1.000 Vpp
 Offset 0 mV
```

At the input of the circuit:



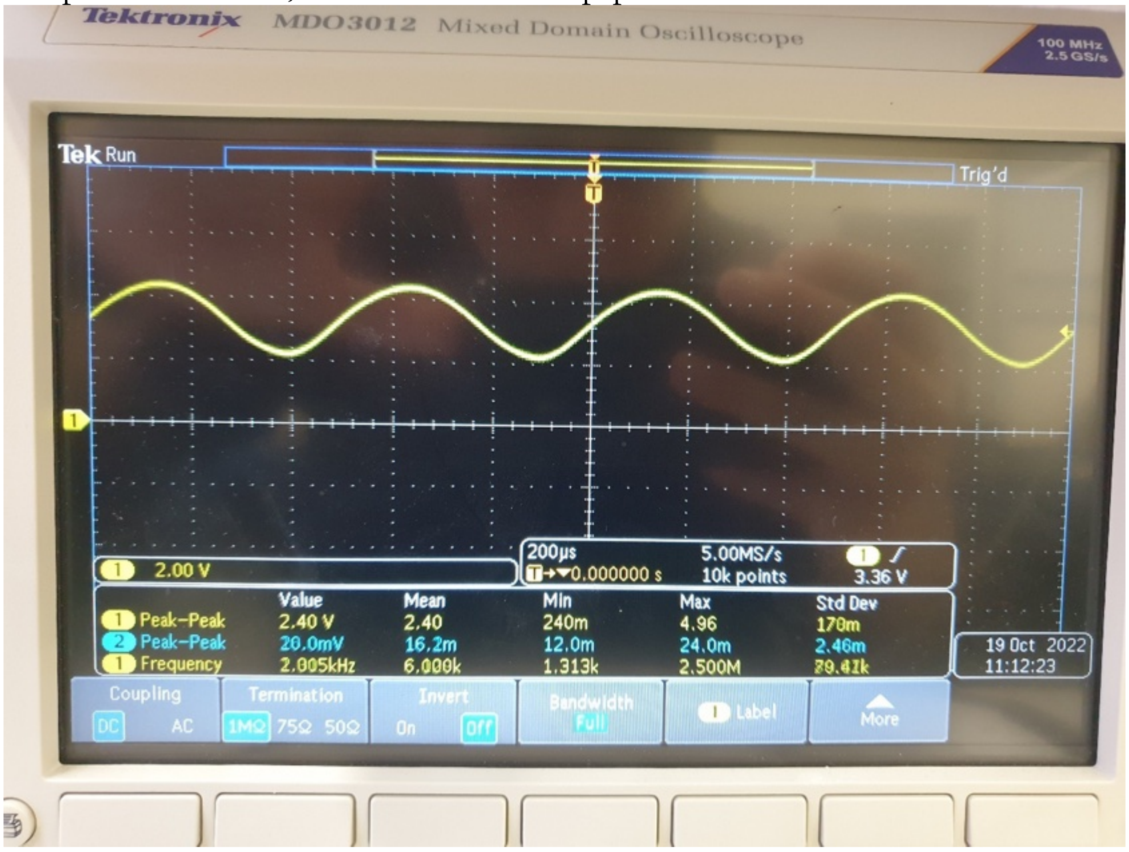
```
1  500mV        200µs            5.00MS/s     1  ∕
                T►0.000000 s     10k points   0.00 V

               Value      Mean     Min      Max      Std Dev
1 Peak-Peak    2.18 V     2.12     60.0m    2.22     407m
2 Peak-Peak    16.0mV     16.3m    12.0m    20.0m    1.83m
1 Frequency    2.001kHz   33.72k   1.019k   2.500M   214.3k

19 Oct
11:06:2
```

DC biased signal, at the input of the filter. We adjusted the DC bias trimpot to give the waveform a centre-point at around 1.65V (max voltage 3.92V − 2.32Vp-p = 1.6V):



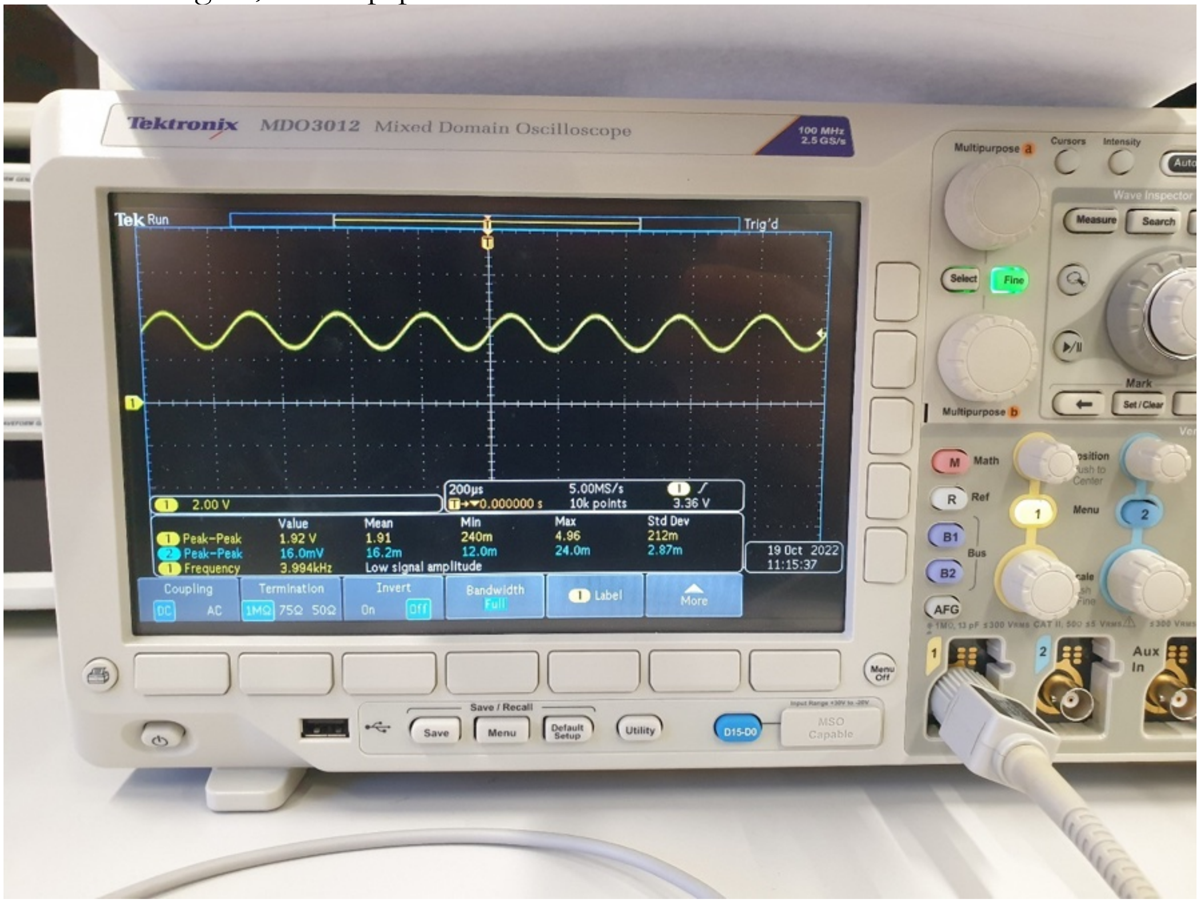
```
1  2.00 V          200µs            5.00MS/s      1 /
                   T→▼0.000000 s    10k points    3.36 V
                Value      Mean     Min      Max      Std Dev
1 Peak-Peak     2.32 V     2.33     240m     3.92     220m
2 Peak-Peak     16.0mV     16.1m    12.0m    20.0m    2.13m
1 Frequency     1.998kHz   Low signal amplitude
19 Oct 2022
11:09:34
```

Output of the filter, note that it is 2.4V p-p:



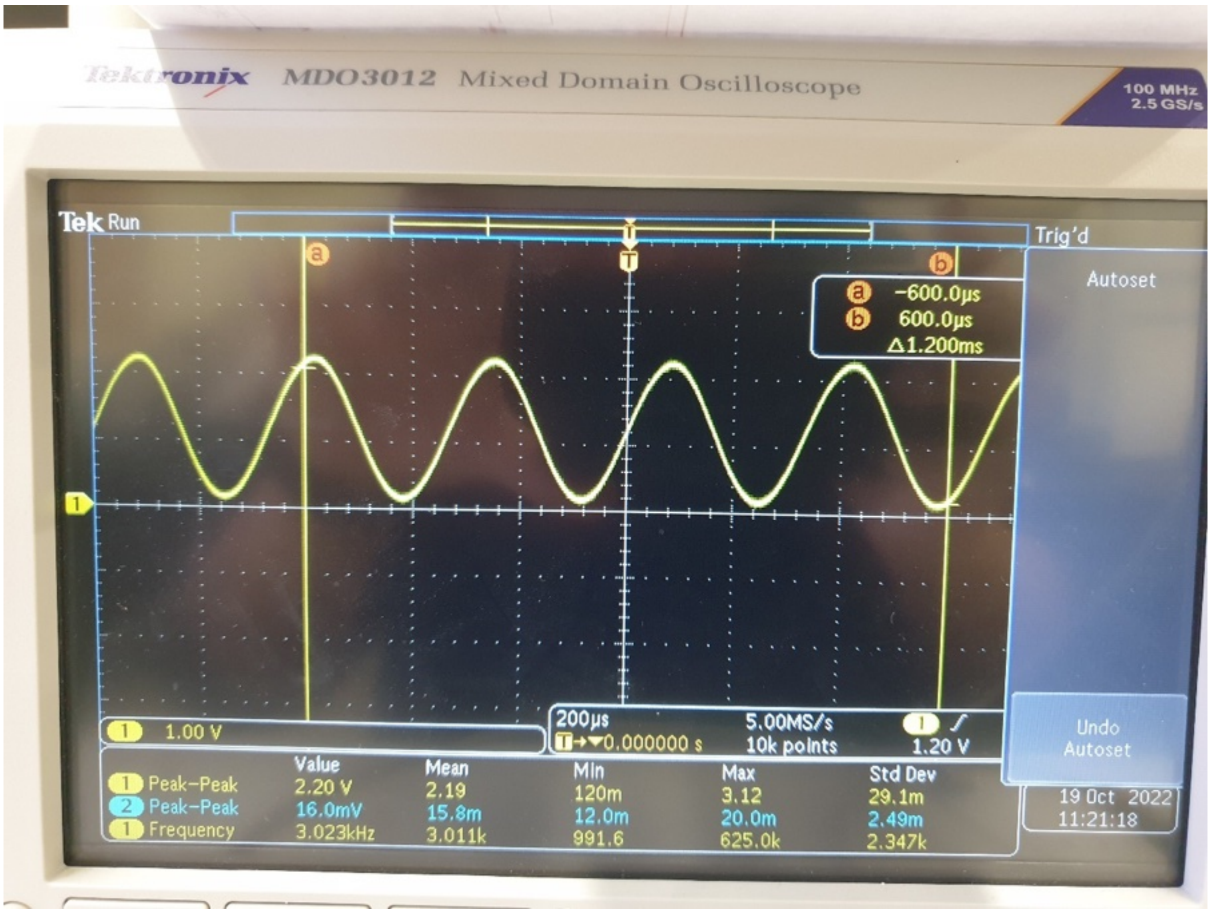
```
1  2.00 V          200µs            5.00MS/s      1 /
                   T→▼0.000000 s    10k points    3.36 V
                Value      Mean     Min      Max      Std Dev
1 Peak-Peak     2.40 V     2.40     240m     4.96     170m
2 Peak-Peak     20.0mV     16.2m    12.0m    24.0m    2.46m
1 Frequency     2.005kHz   6.000k   1.313k   2.500M   89.41k
19 Oct 2022
11:12:23
Coupling [✓ DC] AC   Termination [✓ 1MΩ] 75Ω 50Ω   Invert On [✓ Off]   Bandwidth Full   1 Label   More
```

Attenuated signal, 1.92V p-p at 4 kHz



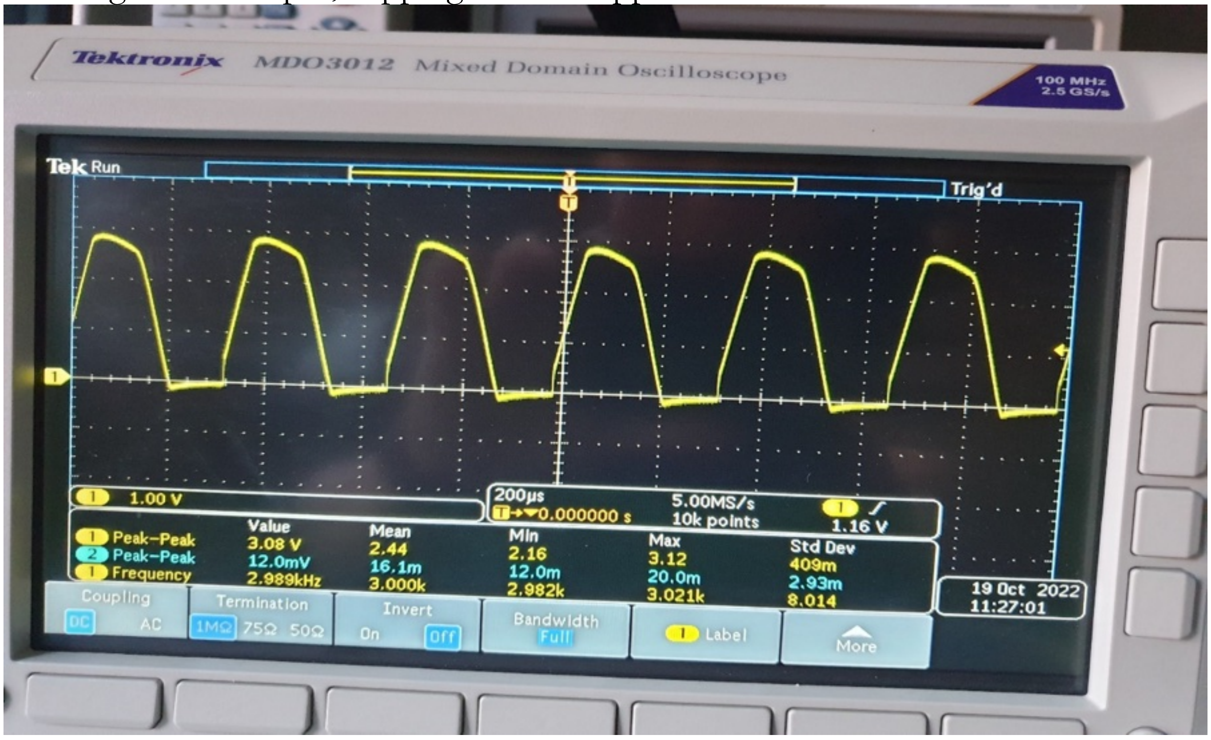
```
1  2.00 V            200µs              5.00MS/s     1 /
                     T→▼0.000000 s      10k points   3.36 V

                Value      Mean     Min      Max     Std Dev
1 Peak-Peak     1.92 V     1.91     240m     4.96    212m
2 Peak-Peak     16.0mV     16.2m    12.0m    24.0m   2.87m
1 Frequency     3.994kHz   Low signal amplitude

19 Oct 2022
11:15:37

Coupling      Termination       Invert     Bandwidth   1 Label   More
[✓ DC] AC     [✓ 1MΩ] 75Ω 50Ω   On [✓ Off] Full
```

Limiting circuit output, 2.2Vp-p giving undistorted sine wave, bottom of waveform at 0V:



```
a  -600.0µs
b   600.0µs
   Δ1.200ms

1  1.00 V        200µs            5.00MS/s      1  1.20 V
                 T→0.000000 s     10k points

                 Value      Mean    Min     Max     Std Dev
1 Peak-Peak      2.20 V     2.19    120m    3.12    29.1m
2 Peak-Peak      16.0mV     15.8m   12.0m   20.0m   2.49m
1 Frequency      3.023kHz   3.011k  991.6   625.0k  2.347k

19 Oct 2022
11:21:18
```

Limiting circuit output, clipping at 0 and approx 3.3V max

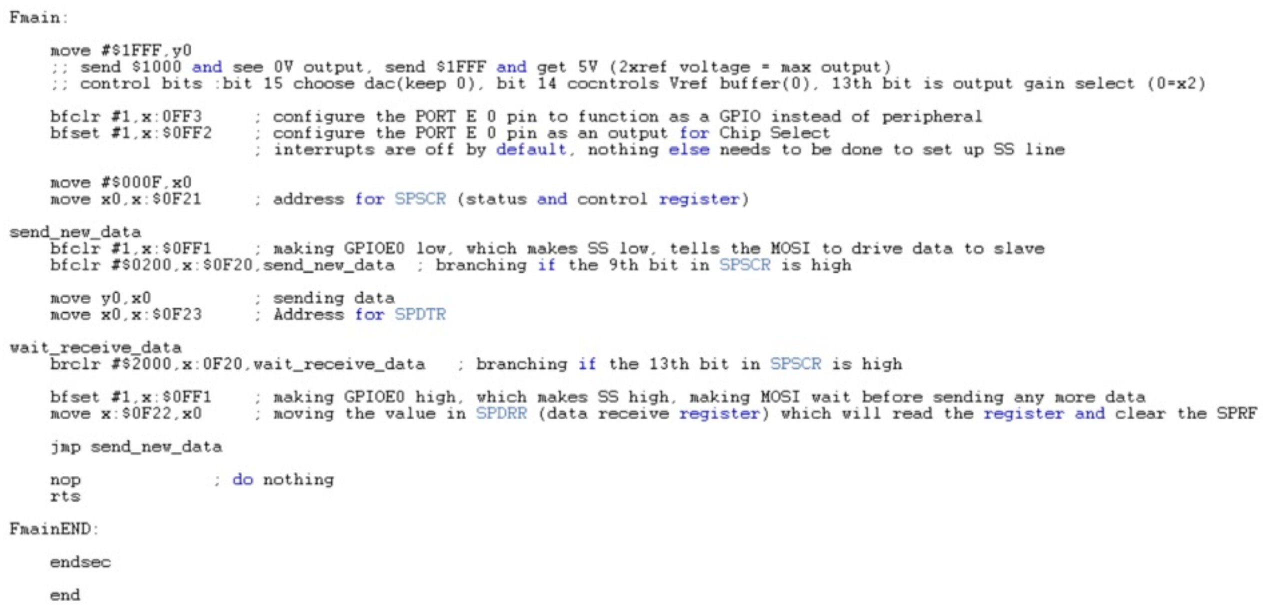


```
1  1.00 V        200µs            5.00MS/s      1  1.16 V
                 T→0.000000 s     10k points

                 Value      Mean    Min     Max     Std Dev
1 Peak-Peak      3.08 V     2.44    2.16    3.12    409m
2 Peak-Peak      12.0mV     16.1m   12.0m   20.0m   2.93m
1 Frequency      2.989kHz   3.000k  2.982k  3.021k  8.014

Coupling: DC AC   Termination: 1MΩ 75Ω 50Ω   Invert: On Off   Bandwidth: Full   1 Label

19 Oct 2022
11:27:01
```

<u>Testing DAC output</u>

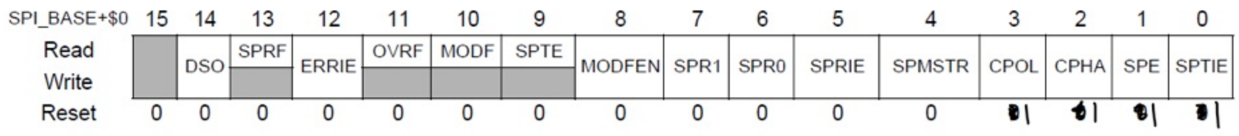
In this example, we are checking to see if the DAC will output the correct DC value when we load min and max values into a simple SPI program.

The program works in steps by:
- Loading data value into a register
- Configuring GPIO and chip select
- Set up the SPSCR (Status and Control Register) (full explanation below)
- Set the SPSCR to make the SPTE bit high, then data can be sent to SPDTR
- Move the data to the SPDTR, this is a write-only data register which then goes to the data transmit buffer and is sent to the output.



```
Fmain:

    move #$1FFF,y0
    ;; send $1000 and see 0V output, send $1FFF and get 5V (2xref voltage = max output)
    ;; control bits :bit 15 choose dac(keep 0), bit 14 cocntrols Vref buffer(0), 13th bit is output gain select (0=x2)

    bfclr #1,x:0FF3      ; configure the PORT E 0 pin to function as a GPIO instead of peripheral
    bfset #1,x:$0FF2     ; configure the PORT E 0 pin as an output for Chip Select
                         ; interrupts are off by default, nothing else needs to be done to set up SS line

    move #$000F,x0
    move x0,x:$0F21      ; address for SPSCR (status and control register)

send_new_data
    bfclr #1,x:$0FF1     ; making GPIOE0 low, which makes SS low, tells the MOSI to drive data to slave
    bfclr #$0200,x:$0F20,send_new_data  ; branching if the 9th bit in SPSCR is high

    move y0,x0           ; sending data
    move x0,x:$0F23      ; Address for SPDTR

wait_receive_data
    brclr #$2000,x:0F20,wait_receive_data   ; branching if the 13th bit in SPSCR is high

    bfset #1,x:$0FF1     ; making GPIOE0 high, which makes SS high, making MOSI wait before sending any more data
    move x:$0F22,x0      ; moving the value in SPDRR (data receive register) which will read the register and clear the SPRF

    jmp send_new_data

    nop                ; do nothing
    rts

FmainEND:

    endsec

    end
```

Setting the SPSCR with $000F:



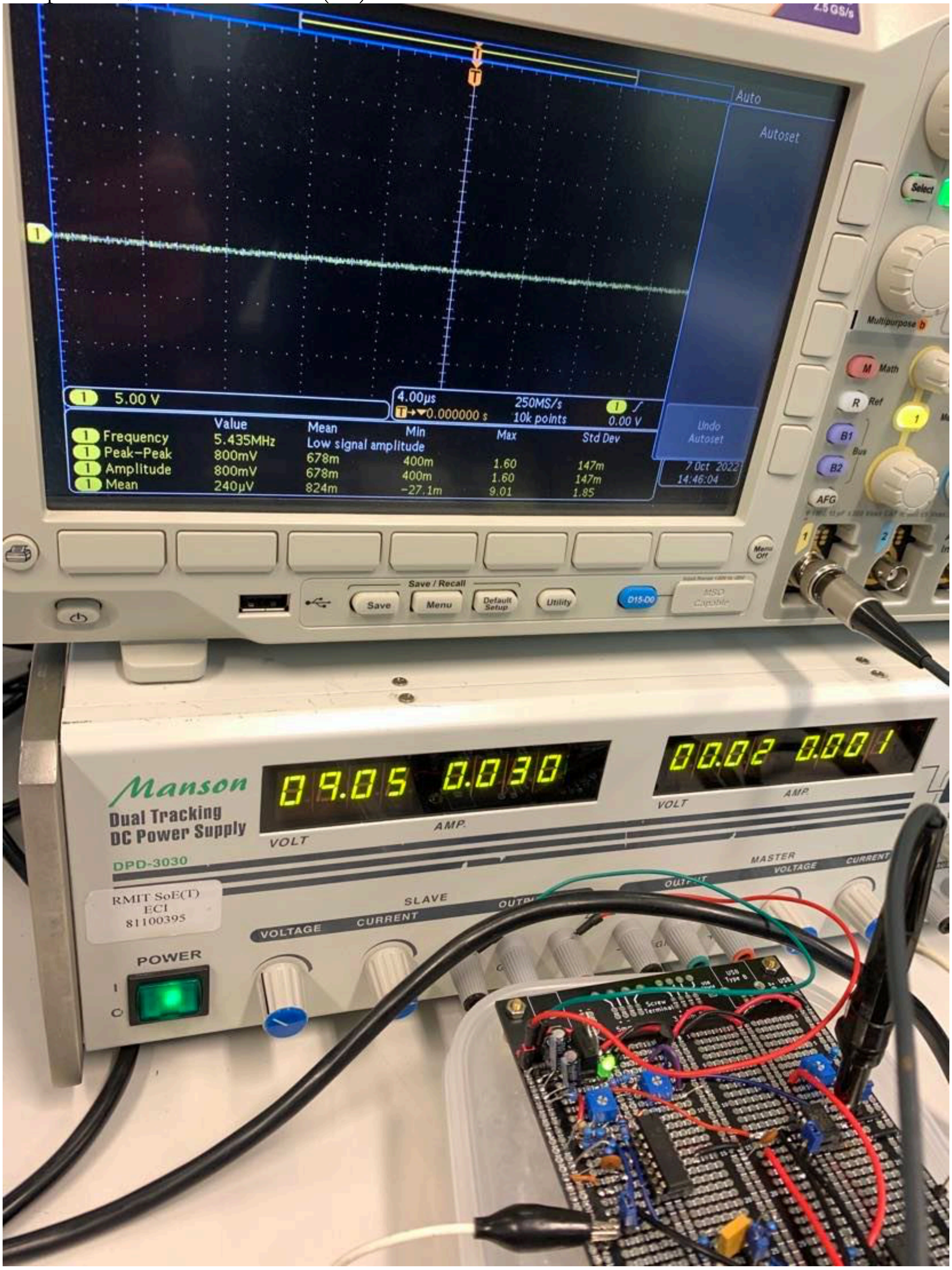
SPI_BASE+$0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	▨	DSO	SPRF	ERRIE	OVRF	MODF	SPTE	MODFEN	SPR1	SPR0	SPRIE	SPMSTR	CPOL	CPHA	SPE	SPTIE
Write	▨	DSO	▨	ERRIE	▨	▨	▨	MODFEN	SPR1	SPR0	SPRIE	SPMSTR	CPOL	CPHA	SPE	SPTIE
Reset	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Figure 13-12. SPI Status and Control Register (SPSCR)

See Programmer Sheet, Appendix C, on page C-104

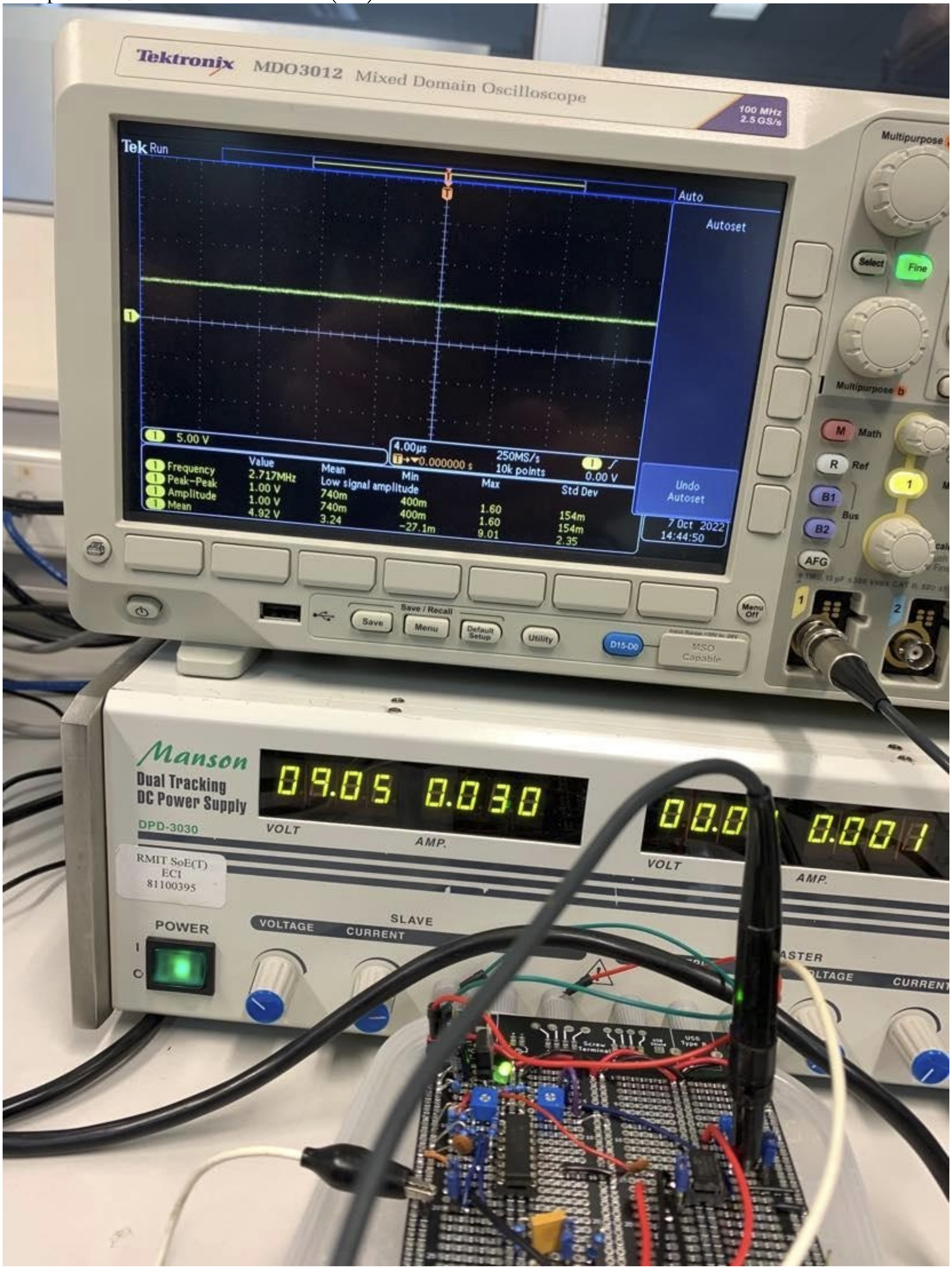
- We set the Clock Polarity Bit (CPOL) high so that the rising edge of SCLK starts transmission
- To send data between SPI modules, they need identical CPHA values, so we set the Clock Phase Bit high.
- SPI Enable (SPE) needs to be set high to enable SPI.
- SPI Transmit Interrupt Enable (SPTIE) enables interrupt requests generated by the SPTE bit, and we need to use interrupts in out program so this is enabled.

Output of $1000 from DAC (0V):



```
1  5.00 V                       4.00µs            250MS/s     1 /
                                T→▼0.000000 s     10k points  0.00 V
             Value      Mean                   Min       Max      Std Dev
1 Frequency  5.435MHz   Low signal amplitude
1 Peak-Peak  800mV      678m                   400m      1.60     147m
1 Amplitude  800mV      678m                   400m      1.60     147m
1 Mean       240µV      824m                   -27.1m    9.01     1.85
                                                                  7 Oct 2022
                                                                  14:46:04
```

Output of $1FFF from DAC (5V):

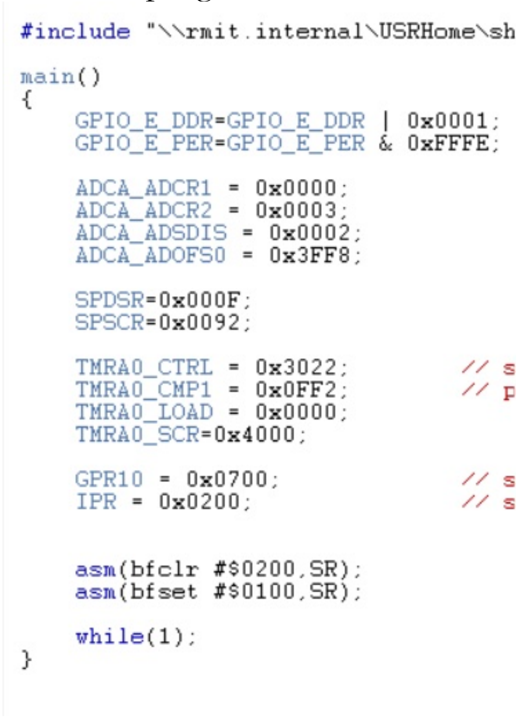


	Value	Mean	Min	Max	Std Dev
1 Frequency	2.717MHz	Low signal amplitude			
1 Peak–Peak	1.00 V	740m	400m	1.60	154m
1 Amplitude	1.00 V	740m	400m	1.60	154m
1 Mean	4.92 V	3.24	-27.1m	9.01	2.35

Testing input/output program

Using new program to test input signal, which gets sent to the program, and then back to the DAC. This is outlined in "Phase 1" of the project brief:
"*2. Use A/D subsystem of the DSP56803 processor to implement an interrupt driven 12 bit A/D conversion to have the output signal of the input analog filter, designed in the previous step, sampled at 9600Hz.*
3. Use SPI subsystem to transmit this data out at the same frequency (9.6KHz)"

This new program uses the following main C program to set up all of the registers.



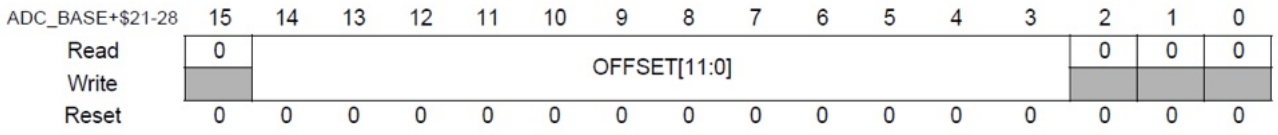
```
#include "\\rmit.internal\USRHome\sh

main()
{
    GPIO_E_DDR=GPIO_E_DDR | 0x0001;
    GPIO_E_PER=GPIO_E_PER & 0xFFFE;

    ADCA_ADCR1 = 0x0000;
    ADCA_ADCR2 = 0x0003;
    ADCA_ADSDIS = 0x0002;
    ADCA_ADOFS0 = 0x3FF8;

    SPDSR=0x000F;
    SPSCR=0x0092;

    TMRA0_CTRL = 0x3022;        // s
    TMRA0_CMP1 = 0x0FF2;        // p
    TMRA0_LOAD = 0x0000;
    TMRA0_SCR=0x4000;

    GPR10 = 0x0700;             // s
    IPR = 0x0200;               // s


    asm(bfclr #$0200,SR);
    asm(bfset #$0100,SR);

    while(1);
}
```

In this main program, we set up:
- ADC Control Register 1 (**ADCR1**) to all 0s, this makes the STOP bit have normal operation (no stop command). It puts a 0 in the START command which gives no action, we will later set this to 1 in the interrupt to begin program. All of the extra functions of ADR1 are not necessary, we leave the zero crossing interrupt enable off, the sync select is off (conversion will be initiated by the write bit, nothing external), and there are no hi-low limit interrupts needed.

- **ADCR2** sets the speed of the conversion. We load $0003 in to set the first and second bits (LSB). The clock divider circuit divides 40 MHz by N+1 where N is the value we put in the register. Our intended conversion speed is 5 MHz, so we put 3 in and the result is 5 MHz.

- ADC Sample Disable Register (**ADSDIS**) with $0002, because we want to only read from everything up to and not including the 2nd bit. We only read the first bit. Everything else is disabled.

- ADC Offset Register (**ADOFS0**) with 3FF8. This is to remove bias from the signal and bring the waveform data to be centred at the zero-point. Because we have given DC bias to make the waveform go from 0-3.3V, we subtract half of this (1.65V) to make it an AC signal. 1.65 = 7FF in hexadecimal. Putting the number in the ADC Offset register will make sure that this subtraction happens automatically to incoming data.
However, the register needs a 12 bit number loaded into it from bits 3-14 (shown below). So when we shift the number across by 1 bit it becomes 3FF8.



ADC_BASE+$21-28	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	0	OFFSET[11:0]												0	0	0
Write	▨													▨	▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

7FF = 0111 1111 1111 xxxx
3FF8 = 0011 1111 1111 1000
We will then need to add this digital "bias" back to the signal after processing has occurred, so that the output is 0-3.3V.

- Timer A Control Register(**TMRA0_CTRL**) with $3022, this makes the rising edge of primary source clock count, it counts until compare then reinitialise, and set flag on successful compare.

- Timer A Compare Register (**TMRA0_CMP1**) with $0FF2, which in decimal is 4082 and it sets how many clock sycles for Ts of sampling rate. Sampling rate is 9800Hz, so Ts is 1/9800 = 0.102mS.
So N for timer = 40 Mhz/9800 = 4082
→ 4082 clock cycles = Ts period of sample rate = 0.102mS

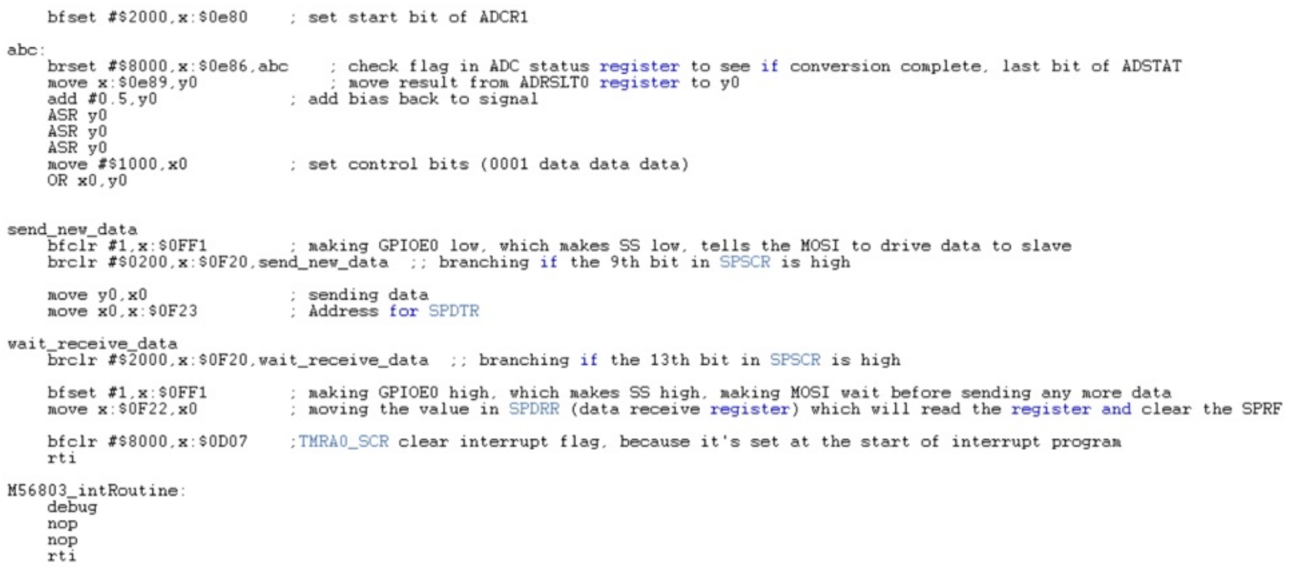
- **TMRA0_LOAD** with $0000 to set the initialise value of the counter.

- Timer A Status and Control Register (**TMRA0_SCR**) with $4000, the only bit we need to set is the Timer Compare Flag Interrupt enable, to enable the interrupt program at input edge.

- Priority of interrupt program, by selecting the priority of interrupt within group (**IPR**), and the setting priority of this group (**GPR10**).

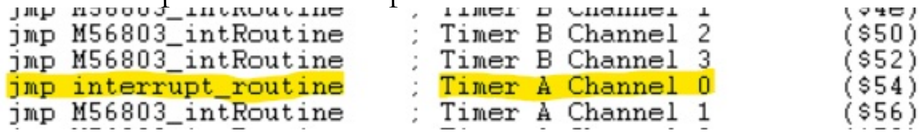
- Two lines of assembly code using the asm command, to make the Interrupt Mask in the Status Register read 01. This controls whether or not an interrupt can be processed or not, so we are permitting interrupts whether they have a Priority Level (IPL) of 0 or 1 and not masking any interrupts.

Then it uses an interrupt which uses assembly, which carries out all of the instructions to send/receive data while(1) in above main program.



```
    bfset #$2000,x:$0e80     ; set start bit of ADCR1

abc:
    brset #$8000,x:$0e86,abc      ; check flag in ADC status register to see if conversion complete, last bit of ADSTAT
    move x:$0e89,y0               ; move result from ADRSLT0 register to y0
    add #0.5,y0              ; add bias back to signal
    ASR y0
    ASR y0
    ASR y0
    move #$1000,x0           ; set control bits (0001 data data data)
    OR x0,y0


send_new_data
    bfclr #1,x:$0FF1         ; making GPIOE0 low, which makes SS low, tells the MOSI to drive data to slave
    brclr #$0200,x:$0F20,send_new_data  ;; branching if the 9th bit in SPSCR is high

    move y0,x0               ; sending data
    move x0,x:$0F23          ; Address for SPDTR

wait_receive_data
    brclr #$2000,x:$0F20,wait_receive_data  ;; branching if the 13th bit in SPSCR is high

    bfset #1,x:$0FF1         ; making GPIOE0 high, which makes SS high, making MOSI wait before sending any more data
    move x:$0F22,x0          ; moving the value in SPDRR (data receive register) which will read the register and clear the SPRF

    bfclr #$8000,x:$0D07     ;TMRA0_SCR clear interrupt flag, because it's set at the start of interrupt program
    rti

M56803_intRoutine:
    debug
    nop
    nop
    rti
```

At the bottom of the interrupt program, we have to change the name of the Timer A Channel 0 interrupt routine that we are using to our own custom name, which we also put at the top of the interrupt section.



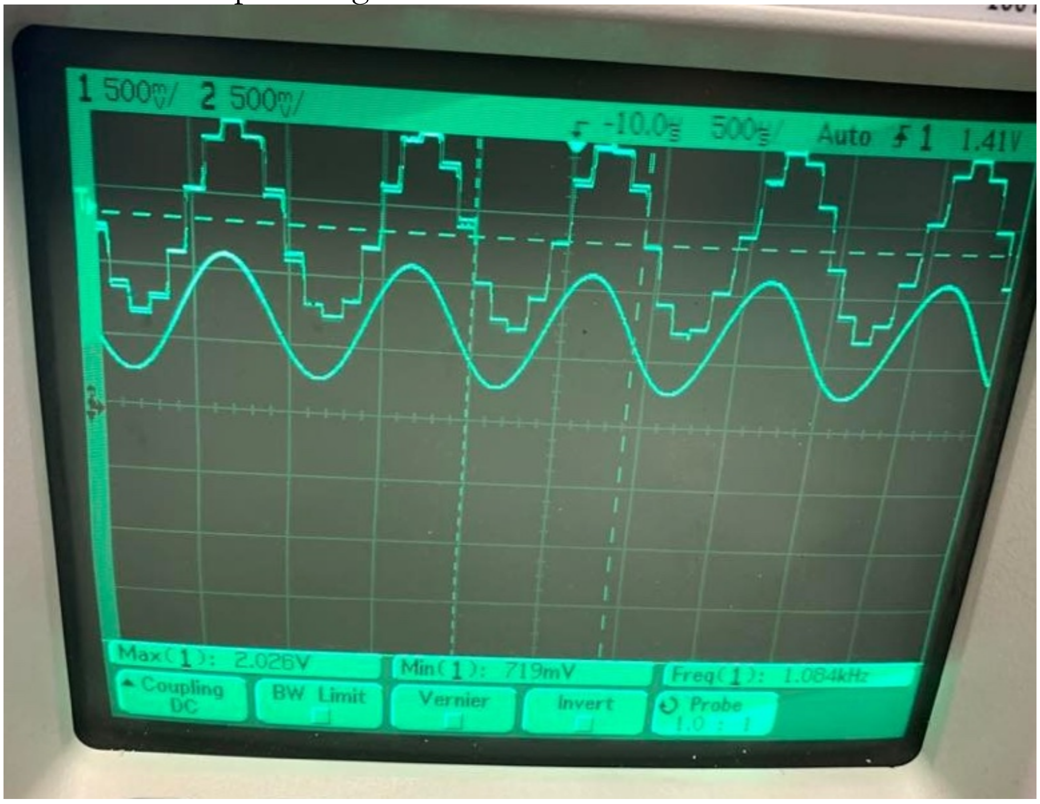
```
 jmp M56803_intRoutine     ; Timer B Channel 2        ($50)
 jmp M56803_intRoutine     ; Timer B Channel 3        ($52)
 jmp interrupt_routine     ; Timer A Channel 0        ($54)
 jmp M56803_intRoutine     ; Timer A Channel 1        ($56)
```

To check whether the program was going to the interrupt, we simply need to write "nop" as a line at the start of the interrupt, and then put a breakpoint on it. If the program stops on the breakpoint, then it is moving to the interrupt correctly.

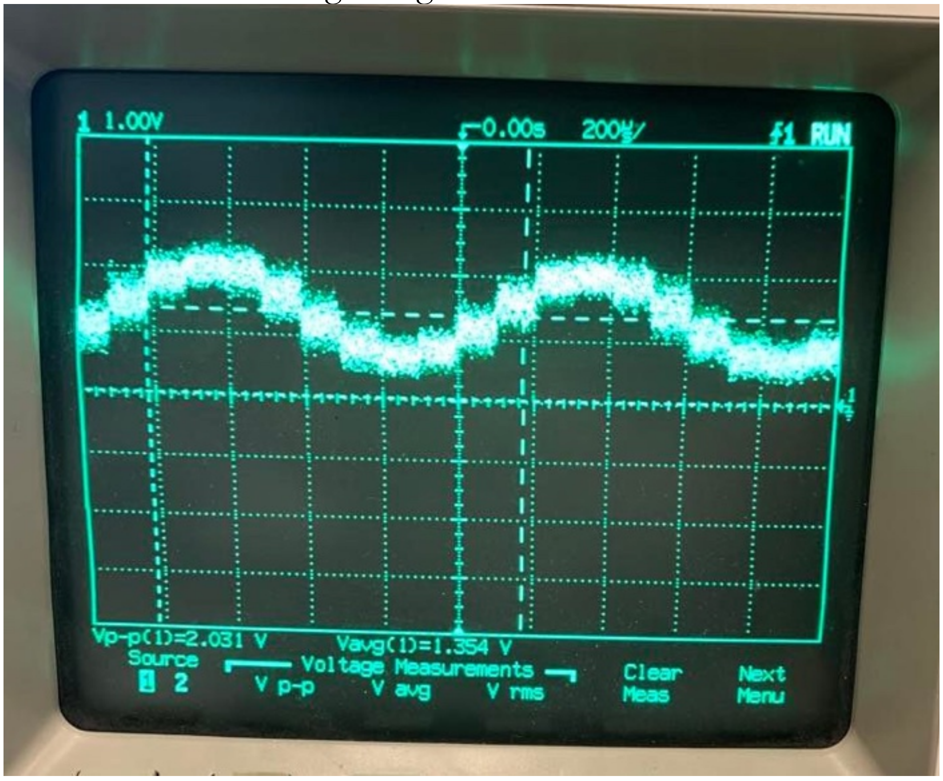
We also used a slightly modified version of this for testing, where at the start we manually load the data into the y0 register instead of taking an input. For example:
move #$1fff,y0
;; send $1000 and see 0V output, send $1FFF and get 5V (2xref voltage = max output)

We also tested whether the interrupt was working at the right speed by writing:
bfset $0001,x:GPIO_E_DR;
bfclr $0001,x:GPIOE_DR;
This makes it send a pulse once for every interrupt, and this was confirmed by probing with the oscilloscope.

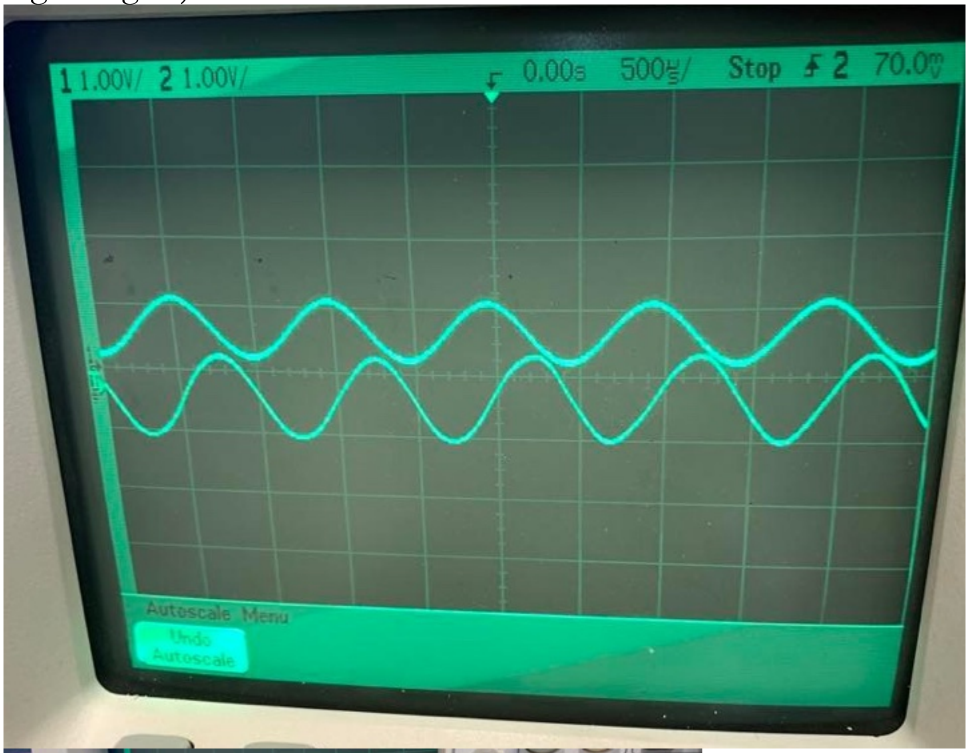
You can see the smooth sine wave below which is the input, then the stepped waveform which is the output straight from the DAC:

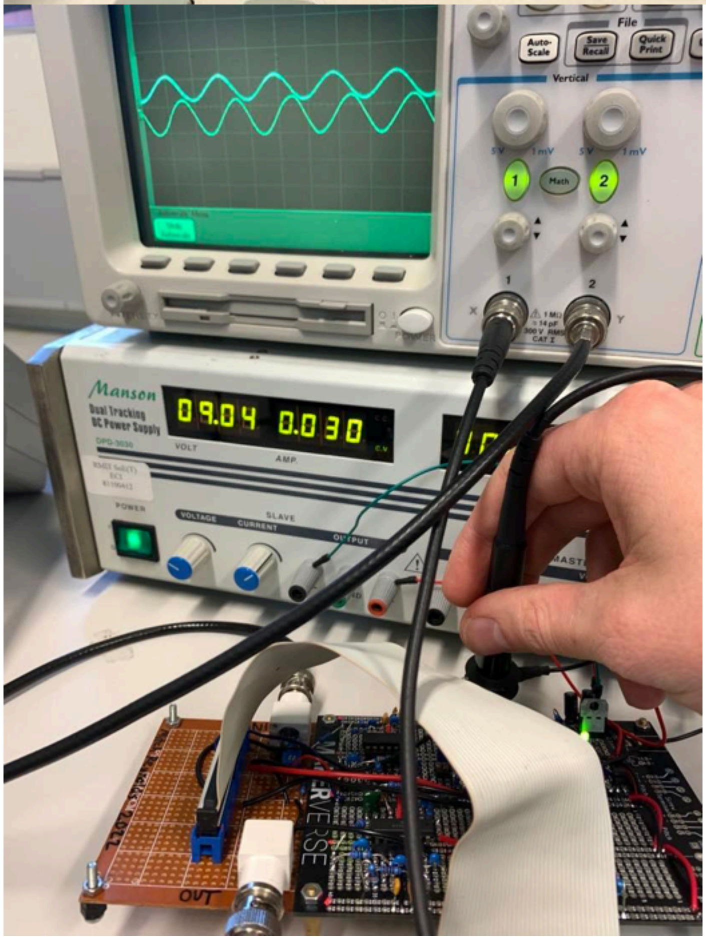


Another look at the digital signal:



Input and output both from BNC connectors (utilising the output filter to smooth the digital signal):

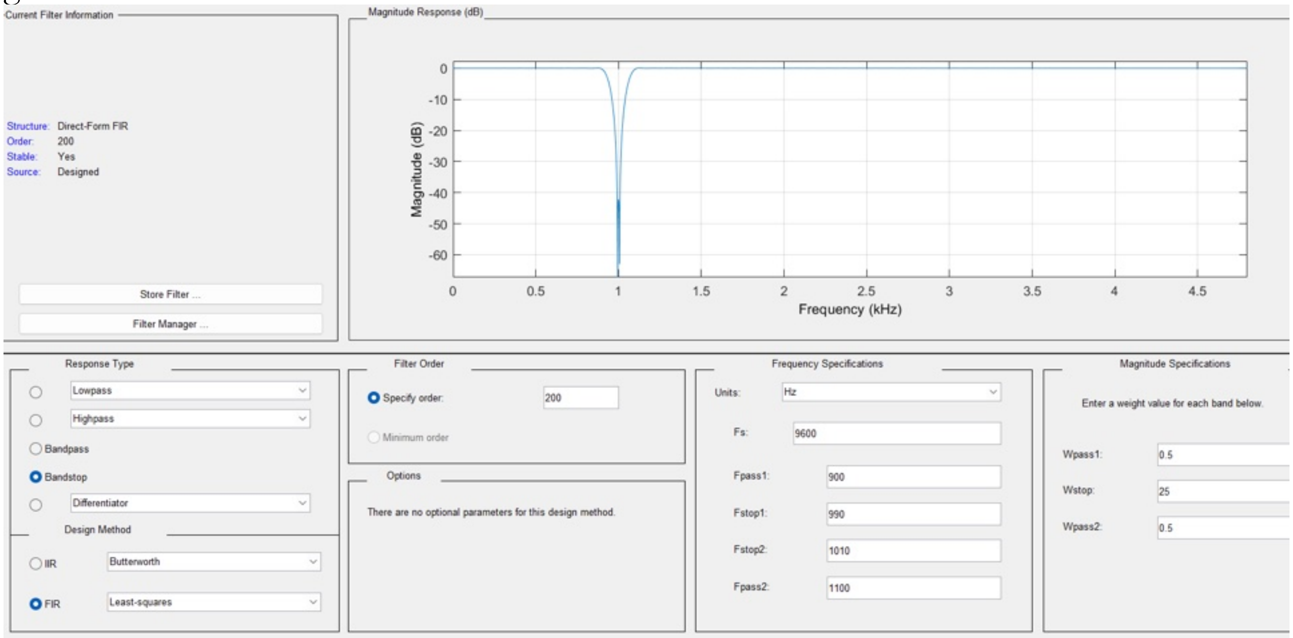




Final program and testing

For the final program we use everything already written with a few important additions.

Firstly, we use MATLAB's filterDesigner tool to enter our filter specifications and have it generate coefficients.



Current Filter Information

Structure: Direct-Form FIR
Order: 200
Stable: Yes
Source: Designed

Response Type

Lowpass
Highpass
Bandpass
[✓ Bandstop]
Differentiator

Design Method

IIR Butterworth
[✓ FIR] Least-squares

Filter Order

[✓ Specify order:] 200
Minimum order

Options

There are no optional parameters for this design method.

Frequency Specifications

Units: Hz
Fs: 9600
Fpass1: 900
Fstop1: 990
Fstop2: 1010
Fpass2: 1100

Magnitude Specifications

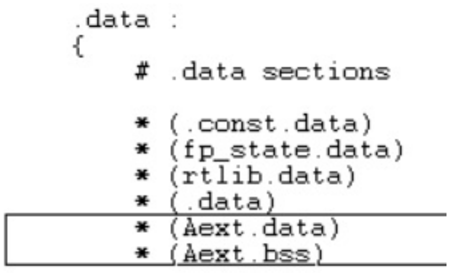
Enter a weight value for each band below.

Wpass1: 0.5
Wstop: 25
Wpass2: 0.5

We then get our 201 coefficients from a 200 order filter, export them as a .h file that we can open in Notepad and then create a .asm file linked to our main.c file. In the .asm file named "coeff" we enter them as constants:

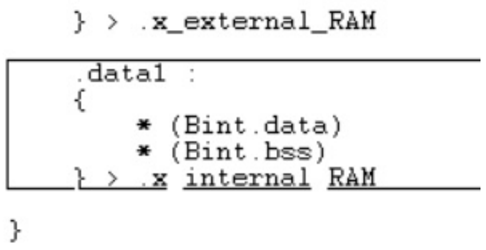
```
      section Bint
      org x:

coeff:
              dc -8,-4,2,5,6,4,1,0,0
              dc 2,3,1,-5,-11,-15,-11,0,16
              dc 29,32,19,-6,-34,-53,-51,-25,19
              dc 62,85,73,26,-41,-99,-121,-93,-18
              dc 74,145,161,108,0,-119,-200,-201,-114
              dc 31,176,259,237,110,-76,-244,-320,-264
              dc -90,136,320,379,279,54,-209,-399,-429
              dc -279,-1,292,476,467,259,-68,-381,-547
              dc -488,-220,150,471,605,488,162,-242,-556
              dc -646,-467,-88,337,630,666,424,0,-430
              dc -687,-662,-360,94,514,723,634,281,-190
              dc -584,32032,-584,-190,281,634,723,514,94
              dc -360,-662,-687,-430,0,424,666,630,337
              dc -88,-467,-646,-556,-242,162,488,605,471
              dc 150,-220,-488,-547,-381,-68,259,467,476
              dc 292,-1,-279,-429,-399,-209,54,279,379
              dc 320,136,-90,-264,-320,-244,-76,110,237
              dc 259,176,31,-114,-201,-200,-119,0,108
              dc 161,145,74,-18,-93,-121,-99,-41,26
              dc 73,85,62,19,-25,-51,-53,-34,-6
              dc 19,32,29,16,0,-11,-15,-11,-5
              dc 1,3,2,0,0,1,4,6,5
              dc 2,-4,-8

      endsec
```

We then need to make slight modifications to the linker file, and add the following declarations to link external memory and our internal data.



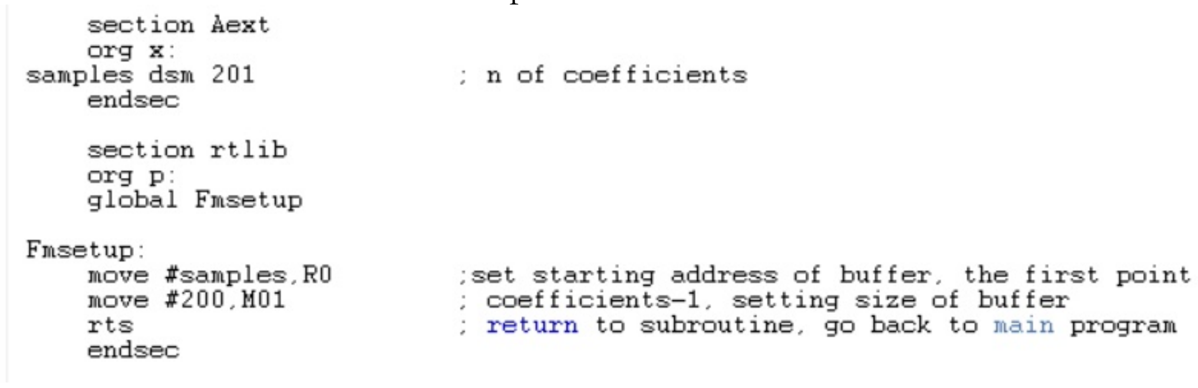
```
    .data :
    {
        # .data sections

        * (.const.data)
        * (fp_state.data)
        * (rtlib.data)
        * (.data)
        * (Aext.data)
        * (Aext.bss)
```

and further down the program:



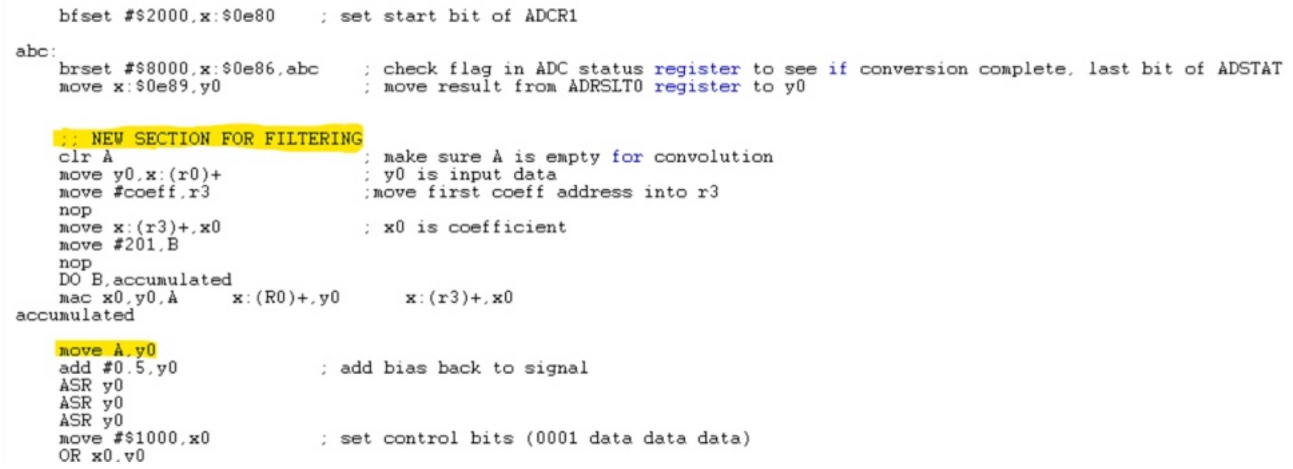
```
    } > .x_external_RAM

    .data1 :
    {
        * (Bint.data)
        * (Bint.bss)
    } > .x_internal_RAM

}
```

We create an additional linked .asm file, which contains the setup information for the circular buffer which contains the input data:



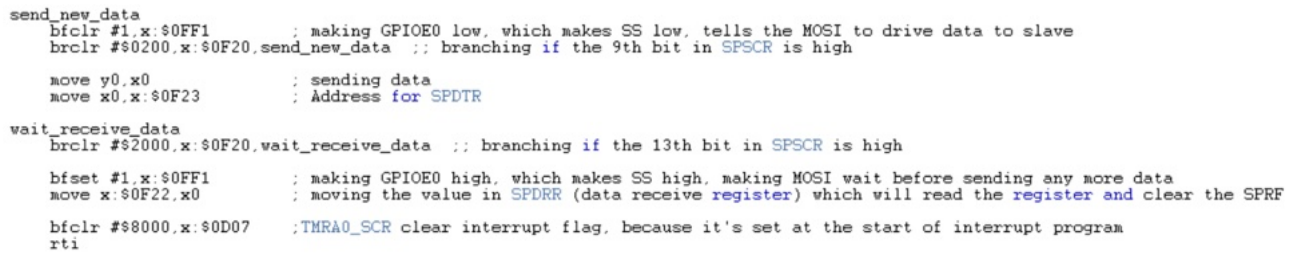
```
    section Aext
    org x:
samples dsm 201             ; n of coefficients
    endsec

    section rtlib
    org p:
    global Fmsetup

Fmsetup:
    move #samples,R0        ;set starting address of buffer, the first point
    move #200,M01           ; coefficients-1, setting size of buffer
    rts                     ; return to subroutine, go back to main program
    endsec
```

The new filtering section in the interrupt is as follows:



```
    bfset #$2000,x:$0e80      ; set start bit of ADCR1

abc:
    brset #$8000,x:$0e86,abc     ; check flag in ADC status register to see if conversion complete, last bit of ADSTAT
    move x:$0e89,y0              ; move result from ADRSLT0 register to y0


    ;; NEW SECTION FOR FILTERING
    clr A                        ; make sure A is empty for convolution
    move y0,x:(r0)+              ; y0 is input data
    move #coeff,r3               ;move first coeff address into r3
    nop
    move x:(r3)+,x0              ; x0 is coefficient
    move #201,B
    nop
    DO B,accumulated
    mac x0,y0,A      x:(R0)+,y0       x:(r3)+,x0
accumulated

    move A,y0
    add #0.5,y0               ; add bias back to signal
    ASR y0
    ASR y0
    ASR y0
    move #$1000,x0            ; set control bits (0001 data data data)
    OR x0,y0
```

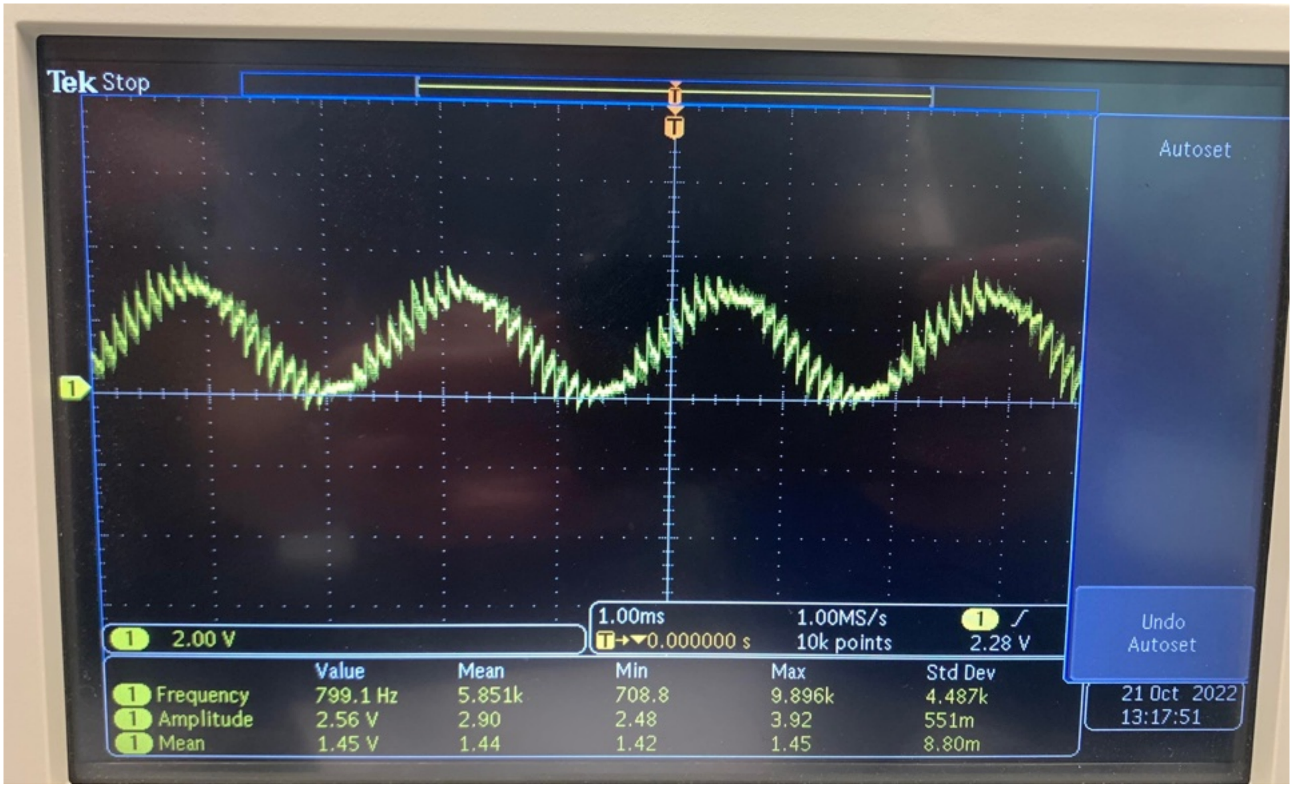
This uses the circular buffer set up in the asm file. In sequence, this program:
- Takes the input data and moves it to the memory location pointed to by R0, then increments R0
- The address of a coefficient is moved to R3, then a coefficient is moved to X0
- We set up a DO loop for 201 repeats, so that we multiply every coefficient
- In the DO loop, we use Multiply & Accumulate (MAC) which multiplies a coefficient with an input data, while retrieving the next coefficient and input data (in a dual parallel move).
- Then after 201 cycles, we move the accumulated result from A to y0, add the 0.5(1.65V) back to the signal, and continue with output program used previously:



```
send_new_data
    bfclr #1,x:$0FF1          ; making GPIOE0 low, which makes SS low, tells the MOSI to drive data to slave
    brclr #$0200,x:$0F20,send_new_data  ;; branching if the 9th bit in SPSCR is high

    move y0,x0                ; sending data
    move x0,x:$0F23           ; Address for SPDTR

wait_receive_data
    brclr #$2000,x:$0F20,wait_receive_data  ;; branching if the 13th bit in SPSCR is high

    bfset #1,x:$0FF1          ; making GPIOE0 high, which makes SS high, making MOSI wait before sending any more data
    move x:$0F22,x0           ; moving the value in SPDRR (data receive register) which will read the register and clear the SPRF

    bfclr #$8000,x:$0D07     ;TMRA0_SCR clear interrupt flag, because it's set at the start of interrupt program
    rti
```

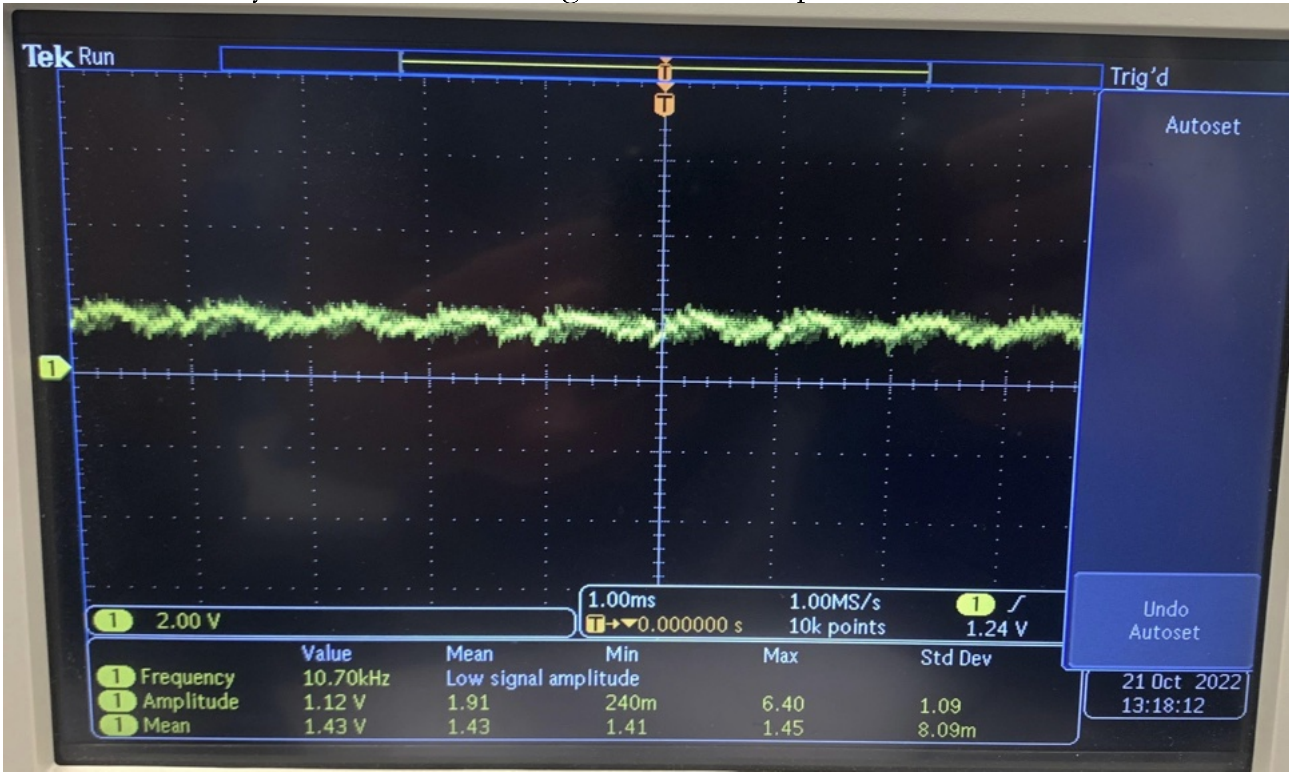
Photos of testing the final program:

Sine wave output from DAC, before the output filter. Note, the sharper high frequency sections which will be filtered out.



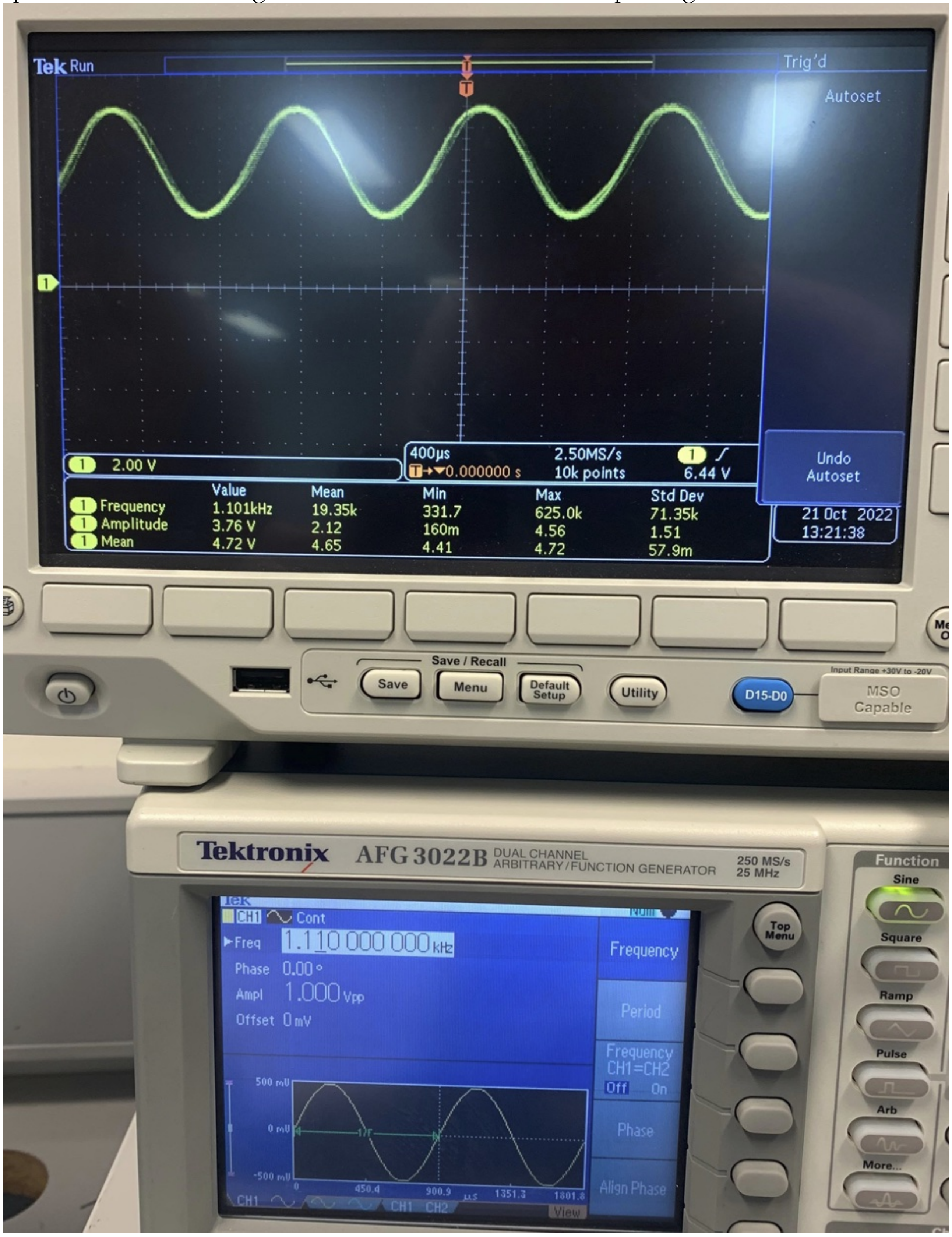
	Value	Mean	Min	Max	Std Dev
1 Frequency	799.1 Hz	5.851k	708.8	9.896k	4.487k
1 Amplitude	2.56 V	2.90	2.48	3.92	551m
1 Mean	1.45 V	1.44	1.42	1.45	8.80m

Above 4kHz, only DC and noise, no signal due to low-pass filter:



	Value	Mean	Min	Max	Std Dev
1 Frequency	10.70kHz	Low signal amplitude			
1 Amplitude	1.12 V	1.91	240m	6.40	1.09
1 Mean	1.43 V	1.43	1.41	1.45	8.09m

1.1kHz input, receiving a smooth filtered sine wave at final output. Note: all this testing was done without AC coupling at end due to circuit issue. The filter works as per specifications but testing needed to be done with DC coupled signal.



1 2.00 V

400µs T→0.000000 s 2.50MS/s 10k points 1 6.44 V

21 Oct 2022 13:21:38

	Value	Mean	Min	Max	Std Dev
1 Frequency	1.101kHz	19.35k	331.7	625.0k	71.35k
1 Amplitude	3.76 V	2.12	160m	4.56	1.51
1 Mean	4.72 V	4.65	4.41	4.72	57.9m

CH1 Cont

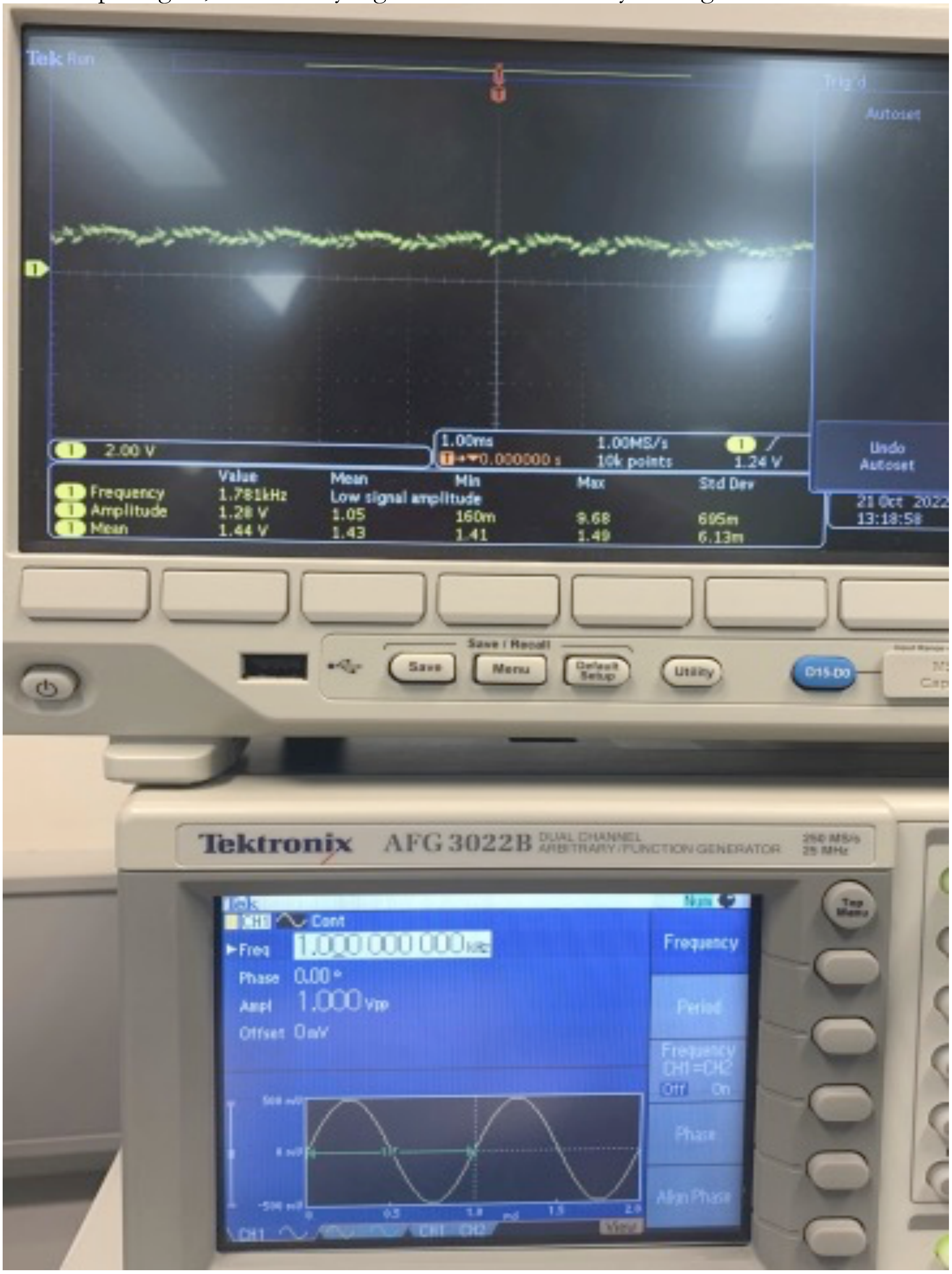
Freq 1.110 000 000 kHz

Phase 0.00 °

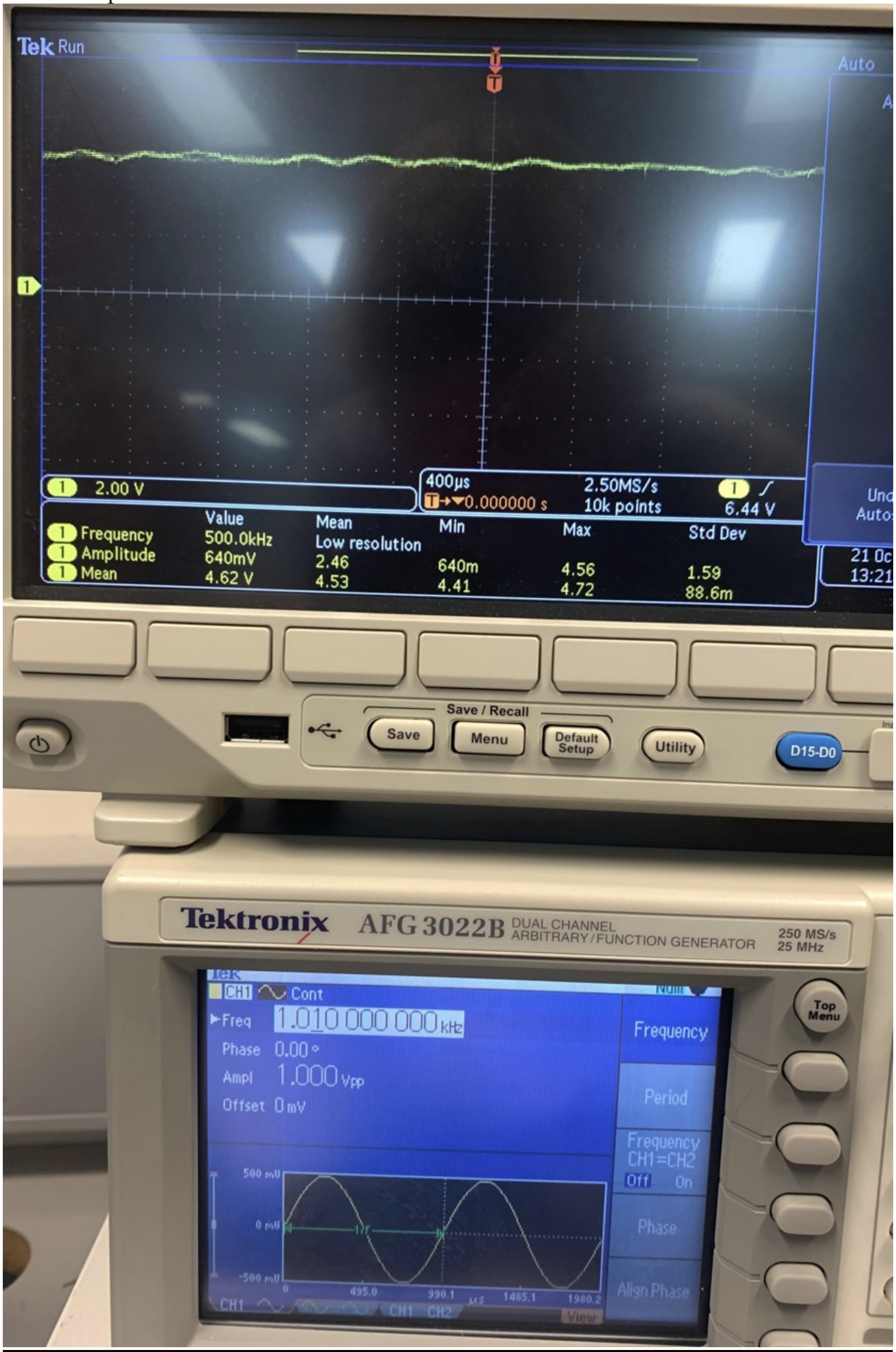
Ampl 1.000 Vpp

Offset 0 mV

1 kHz input signal, removed by digital FIR filter and only DC signal+noise remains:


```
 1  2.00 V                 1.00ms            1.00MS/s      1  /
                           T→▼0.000000 s     10k points      1.24 V
              Value      Mean       Min       Max       Std Dev
1 Frequency   1.781kHz   Low signal amplitude
1 Amplitude   1.28 V     1.05       160m      9.68      695m
1 Mean        1.44 V     1.43       1.41      1.49      6.13m
                                                        21 Oct 2022
                                                        13:18:58
```
```
CH1  Cont
Freq    1.000 000 000 kHz
Phase   0.00 °
Ampl    1.000 Vpp
Offset  0mV
```

Another photo of 1010 Hz removed:



	Value	Mean	Min	Max	Std Dev
1 Frequency	500.0kHz	Low resolution			
1 Amplitude	640mV	2.46	640m	4.56	1.59
1 Mean	4.62 V	4.53	4.41	4.72	88.6m

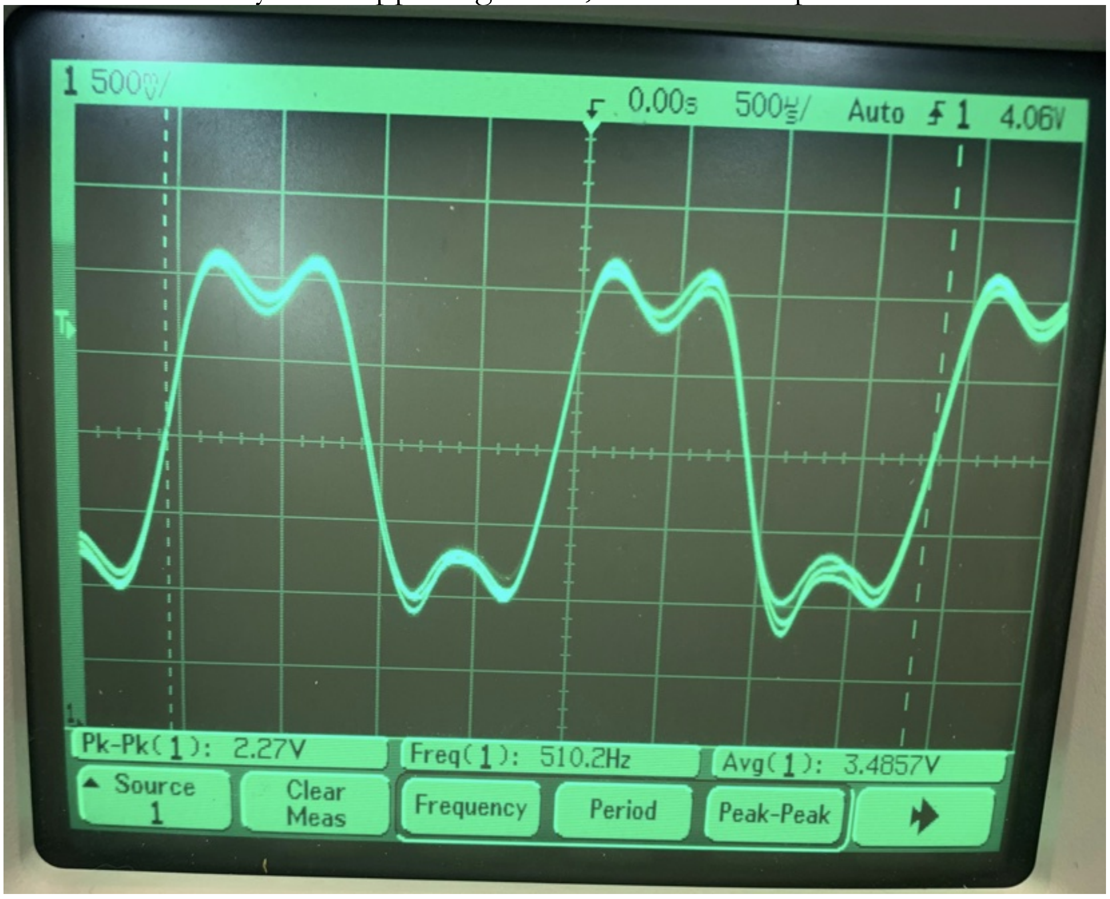
CH1 Cont
Freq 1.010 000 000 kHz
Phase 0.00°
Ampl 1.000 Vpp
Offset 0 mV

Square wave harmonics demonstration

Square waves are a complex waveform consisting of sine waves of a fundamental frequency and then its odd harmonics (3rd, 5th, 7th, 9th, 11th, 13th…).
These harmonics are calculated by N x Fundamental, so for a fundamental of 500Hz the 3rd harmonic is 1500Hz, the 5th is 2500Hz etc.

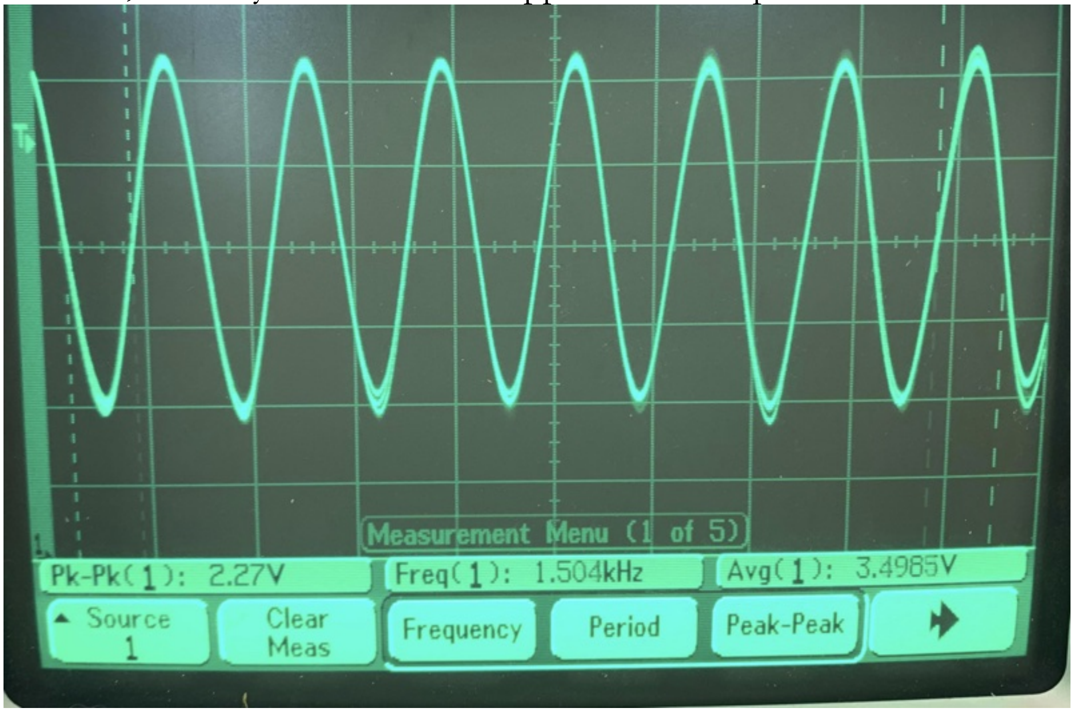
The harmonics decrease in amplitude and are equal to 1/N where N is the harmonic. Amplitude is less relevant in this example, but it is important to consider.

With the previous harmonic calculations in mind, we can expect that a low-pass filter with a cutoff frequency of 2000Hz filtering a 500Hz square wave would keep its 3rd harmonic of 1500Hz and then cutoff any further harmonics.
We can see exactly this happening below, note the two peaks on the waveform:

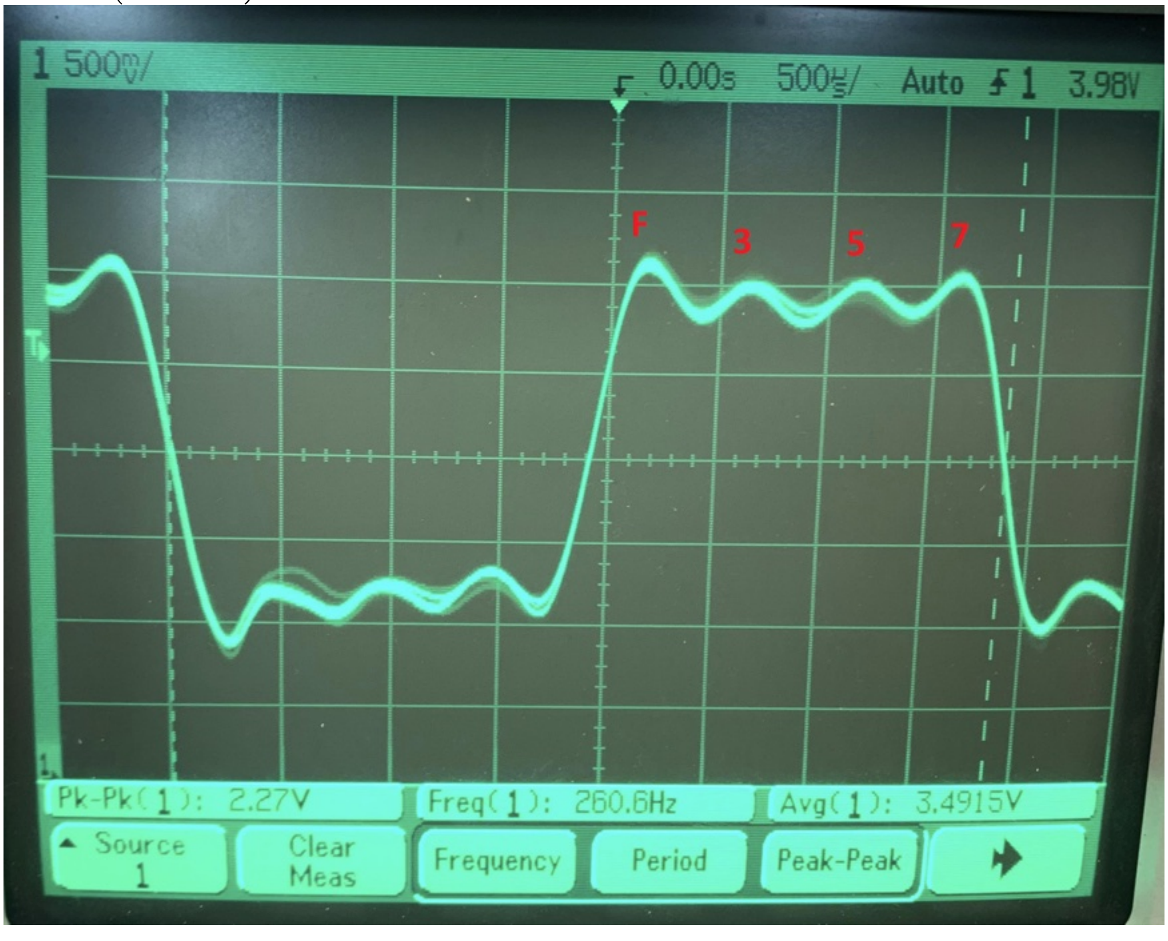


We used exactly the same programming, but used MATLAB's filterDesigner to create a low-pass filter, and then placed these coefficients into the .asm file.
The program works exactly the same for whatever FIR filter we would like, all we need to change is the coefficients.
Then, we can make the following observations based on the Fourier series of a square wave.

A square wave with a fundamental of 1500 Hz will have all of its harmonics cut off by the filter, and only a sine wave will appear at the output:



A square wave with a fundamental of 260 Hz will have its 3rd (780 Hz), 5th (1300 Hz) and 7th (1820 Hz):



Reflections

Overall, this was a very challenging project. The hardware was easier to build, as I am very experienced with building circuit boards and DIY projects in my spare time, but perhaps I may have been able to plan it better and laid things out a bit clearer in hindsight. When debugging the board I needed to constantly refer to the schematic as the layout was not very intuitive.

If I also had proper tools, perhaps I would have not needed the separate vero board for the I/O section. But because I was unable to cut tracks on my board it was necessary. In the future, perhaps something like this would be more suited to designing in Eagle, where I'd made the schematic, and then manufacturing a PCB.
This would reduce the possibility of troubleshooting though, and in my case quite a few modifications were made after soldering.
I could now confidently create a PCB layout and have it manufactured exactly to spec.

The programming was also very difficult, as I am much more confident with hardware. At a certain point though it began to be very intuitive, once we memorised where data was being stored, which memory locations we were using, and the certain commands used in C.

Conceptually, it was easy to understand, but the implementation of the program proved very difficult with the coefficients needing to be multiplied in a circular buffer and the linker file connecting different files within the same program – I have never done anything like this before and learned so much from the project.

I would like to thank my fellow students for their assistance, especially Aldrin, Daniel and Dimitri who all either gave advice with the coding or helped inspect my board when I was troubleshooting issues and testing.
I would also like to thank Sukhvir Judge for assisting us with the project along the way and offering advice for troubleshooting which will be very useful in future projects.