Digital and Embedded Systems: Laboratory 2 Report

Name: Dylan Boland
Student Number: 17734349
Working with: Cuan de Burca
Code submitted by: Dylan Boland

1. I have read the UCD Plagiarism Policy (available on Brightspace). I understand the definition of plagiarism and the consequences of plagiarism.
2. I recognise that any work that has been plagiarised (in whole or in part) may be subject to penalties, as outlined in the documents above.
3. I have not plagiarised any part of this report. The work described was done by the team, and this report is all my own original work, except where otherwise acknowledged in the report.

(Introduction)
In this assignment my teammate and I attempted to use the ADuC841 microconverter to perform three measurements. In order of priority, they were:

(1) The measurement of a DC input voltage.
(2) The measurement of an AC input's amplitude, either its peak or peak-to-peak value.
(3) The frequency of a periodic input signal.

(Design Features)
Upon completing the assignment, our design was able to:
(1) Display the value of a DC input voltage within the range of 0 to 2.5 V. The value was represented in mV on the display.
(2) Measure the amplitude of a periodic input signal, up to 2.5 V. The amplitude value was displayed in mV on the display, like with the first measurement.
(3) Compute the half period, in clock cycles, of a periodic input signal. Unfortunately, we did not have time to convert the number of clock cycles to a frequency value, although a comment on how this would be done comes nearer the end of this report.

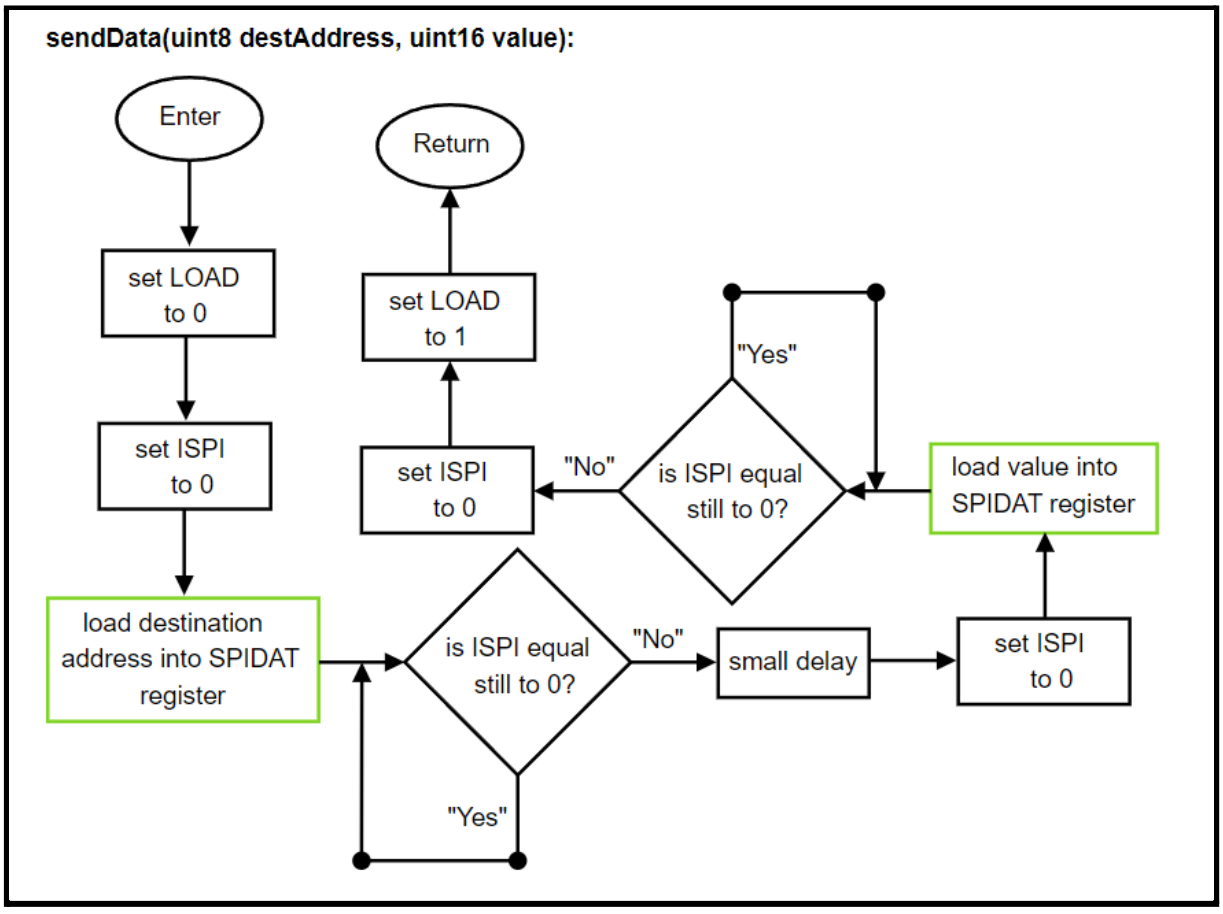
(Design Process)
To make the design process easier to begin, my teammate and I decided to focus on measurement 1. We thought about the core components that would be needed to implement the requirements. Two of these were the on-chip ADC and the external display driver circuit. We were aware that the ADC would be needed for measurement 1 and 2, and that the display device would be needed for all 3 measurements.

My teammate focused on how the ADC would need to be configured, and what functions would be needed to configure and read from it, while I focused on how we would interact with the display driver circuit.

I reasoned that there would be two principal functions needed relating to the display device:

(1) void setupDisplay(void)
(2) void sendData(uint8 destAddress, uint16 value)

The second function allows a value to be sent to a specified register on the display driver circuit. The first function is called during the setup phase, and it configures the display device by sending appropriate values, via the sendData() function, to the various control registers. Below is shown a flowchart which depicts in more detail the steps that the second function carries out:



Each time a value is written to the SPIDAT register, a transaction begins. This is highlighted in the above flowchart by the two process blocks outlined in green. When the transaction is done, the hardware sets ISPI to 1.

As for LOAD, I chose to use P2.7 for this signal:

```
// defining extra special function registers
sbit LOAD = 0xA7; // defining P2.7 to be "LOAD"; 1-bit special function register
```

Macros for the addresses of the control registers of the display driver circuit were defined at the top of the program:

```
#define DECODE_ADDR 9 // address of the decode mode register on the display device
#define INTENSITY_ADDR 10 // address of the intensity register
#define SCAN_LIMIT_ADDR 11 // address of the scan limit register
#define SHUTDOWN_ADDR 12 // address of the shutdown register
#define DISPLAY_ADDR 15 // address of the display test register
```

Inside the setupDisplay() function, the following values were sent to the various register addresses:

- 255 was sent to the decode mode register (DECODE_ADDR). This is because we wanted to display the standard patterns, instead of having direct control over each of the 8 segments.
- 1 was sent to the shutdown register (SHUTDOWN_ADDR). This allowed for the display device to turn on when power was supplied.
- 0 was written to the display test register (DISPLAY_ADDR) in order to select the standard operation of the segments.
- NUM_DIGITS was sent to the intensity register (INTENSITY_ADDR). NUM_DIGITS was a macro at the top of our program, and corresponded to the number of digits that we wanted to use on the display. Most of the time this was either 6 or 8.
- (NUM_DIGITS - 1) was written to the scan limit register (SCAN_LIMIT_ADDR).

(SPICON Register)

The SPICON register was configured as follows:

- The SPE (SPI Enable) bit was set to 1 to enable the SPI interface.
- The SPIM (SPI Mode) bit was set to 1 in order to assert that the microconverter device was the master device.
- The CPOL (Clock Polarity) bit was cleared to 0, as the display driver circuit required that the clock signal idle low.
- The CPHA (Clock Phase) bit was also 0. This is because the display driver circuit samples the data on the rising edge of the clock signal, meaning we wanted to transmit data on the falling edge of the clock signal.
- SPR1 and SPR0 were both cleared to 0. This selected a bit rate of half the oscillation frequency (f_{osc}), around 5.5 MHz.

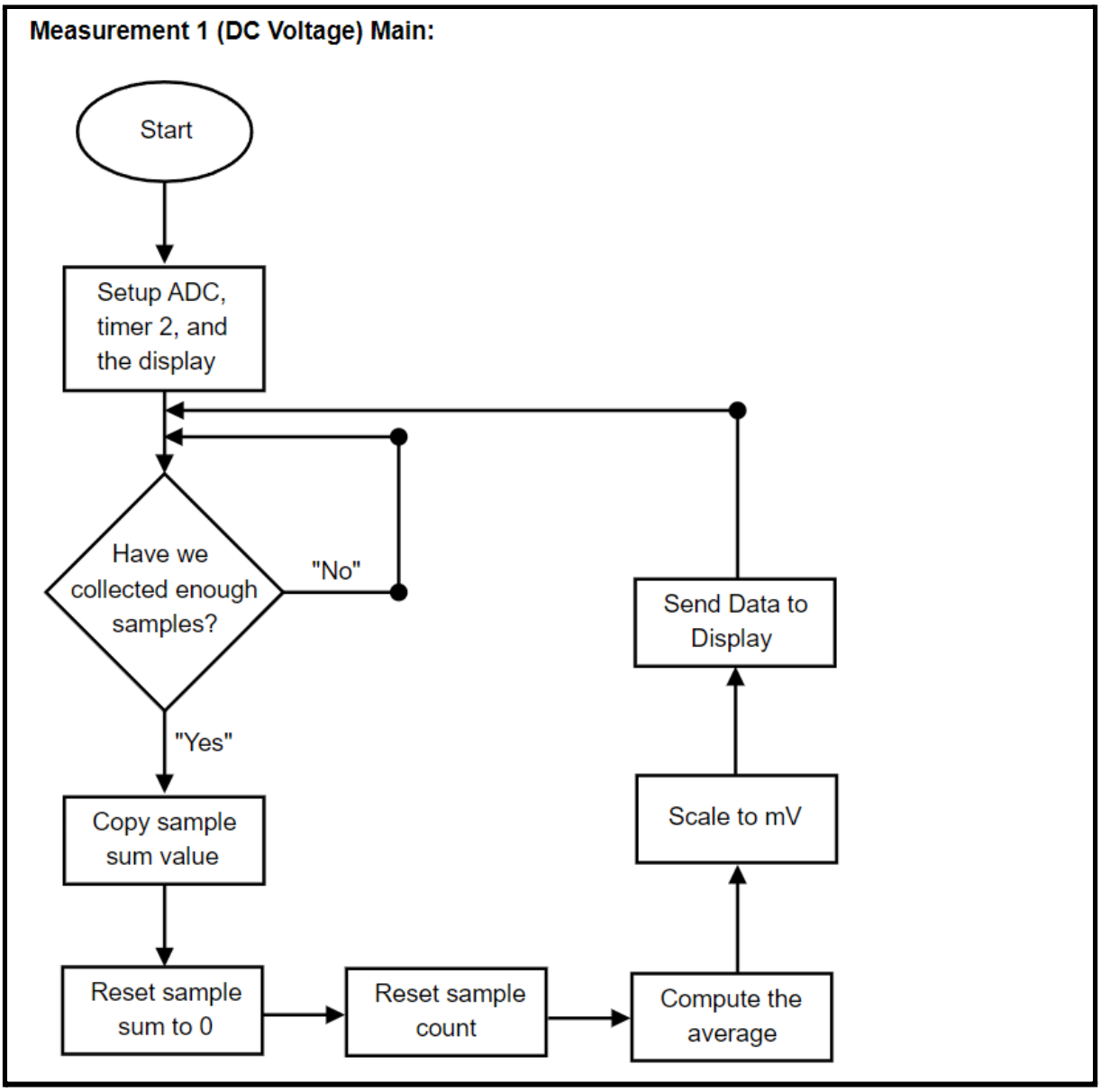
Before moving on to design other blocks that would be needed for measurement 1, we verified that we were able to display different numbers on the display. This helped us to test the setupDisplay and sendData functions, as well as displayNumber, a function that my teammate wrote.

(Measurement 1)

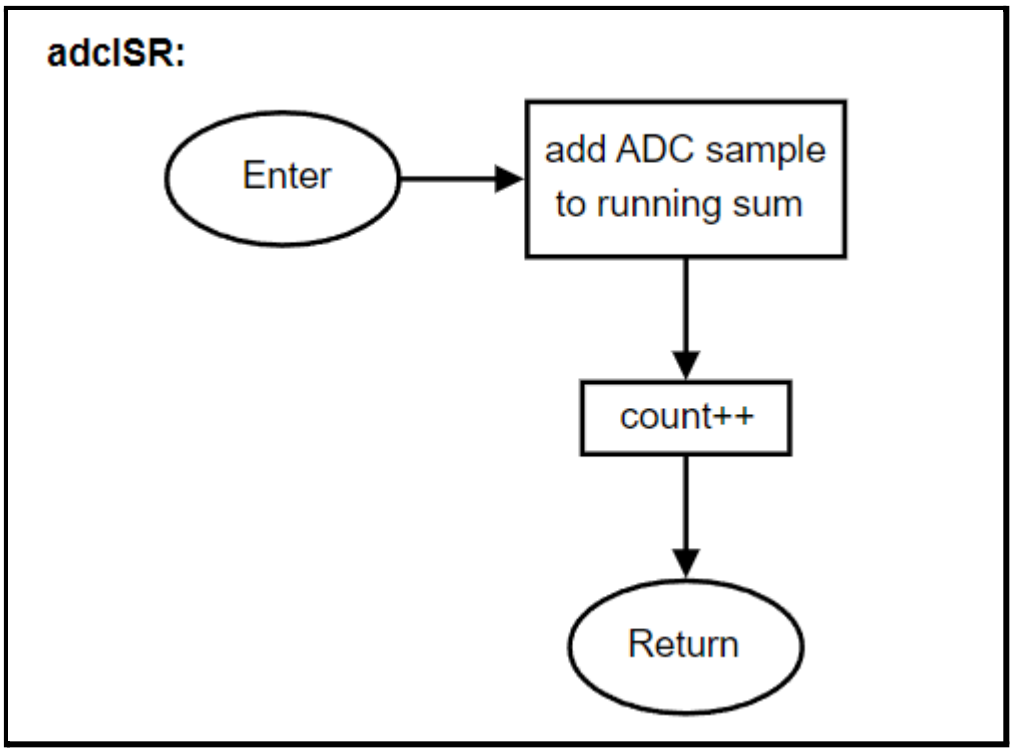
The need to debug various display-related issues set us back a little bit. As a result, my teammate and I decided to focus on different measurements in order to better meet the deadline. I focused on the main functionality of measurement 1, while he focused on the main functionality of measurement 2.

To start, I listed out the various steps that would need to be followed in order to average many samples from the ADC. From these steps or bullet points, I came up with various flowcharts for what the main part of the program would do and what the ADC ISR would do.

The final versions of each of these are shown below, with some comments on the choices that were made:



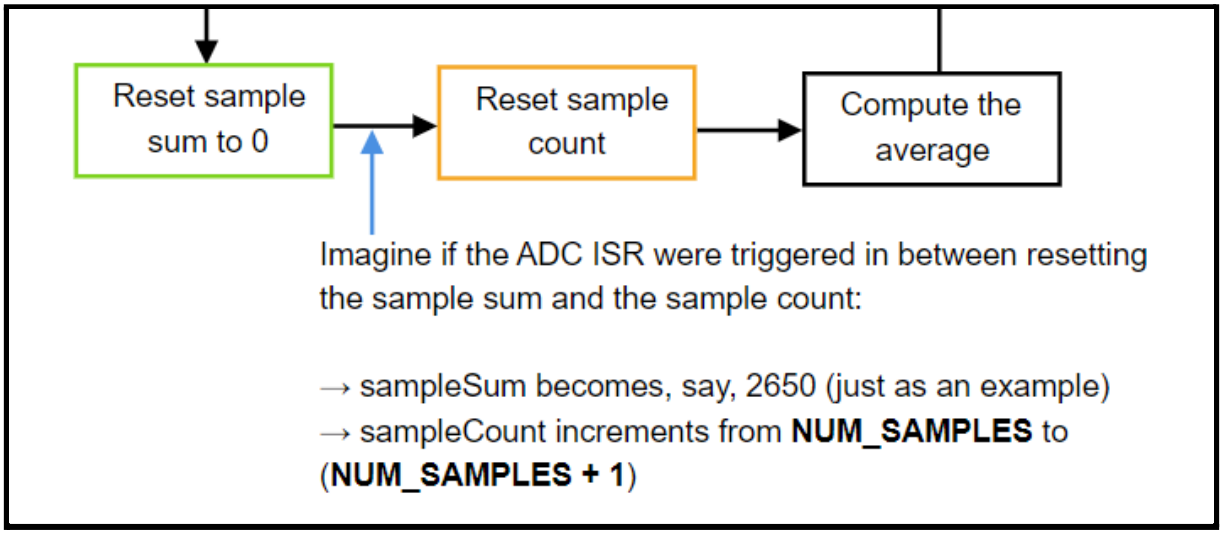
- We decided as a team to use timer 2 in order to start the ADC conversions. Timer 2 interrupts were disabled, but the ADC interrupt was enabled. This allowed us to be alerted of when a conversion was complete.
- The value of the running sum was copied before being quickly reset to 0, alongside the sample count. The main part of the program could then more safely compute the average of the samples with this copy of the running sum, as even if the ADC ISR was triggered, it would not be able to change this copy, but only the global running sum variable.
- We chose to display the value in mVs.



- My teammate had already written a function to return the ADC sample.
- The running sum variable (sampleSum) was global, so that both main and the ADC ISR could update it.
- The sample count variable (sampleCount) was also global, for the same reason as above.

(Resetting the sample count)
While writing the main part of the program for measurement 1, I thought about some problems that could arise if the ADC ISR were to be triggered just as main was changing the sample sum and the sample count. I thought of the following scenario:

1. Main clears the sample sum (but is yet to clear the sample count).
2. The ADC ISR is triggered:



As is shown above, sampleSum is updated and sampleCount is incremented.

3. Control returns to main, and sampleCount is reset to 0.

Now when sampleCount reaches 1024 (NUM_SAMPLES) again, sampleSum will actually consist of 1025 (NUM_SAMPLES+1) samples, and our average will be thrown off.

In order to help avoid this problem, I switched from using

```
sampleCount = 0;
```

to

```
sampleCount %= NUM_SAMPLES;
```

In this way sampleCount is assigned the correct value. If sampleCount were 1025 (NUM_SAMPLES+1), then after modulo division by 1024 (NUM_SAMPLES) it ends up getting assigned the value of 1, as needed.

(Measurement 3)

I also designed the main program for measurement 3, concerning the measurement of a periodic input signal's frequency. The periodic signal was input to the on-board Schmitt trigger. The corresponding square wave output then acted as an input on P3.2, and controlled the counting of timer 0.

In the laboratory, my teammate and I ended up altering the code so as to use external interrupt 0 instead of polling or checking the pulse's value on P3.2 in the software, as I originally had.

I wrote a function, setupT0(), which configured the TMOD and TCON registers for timer 0 as follows:

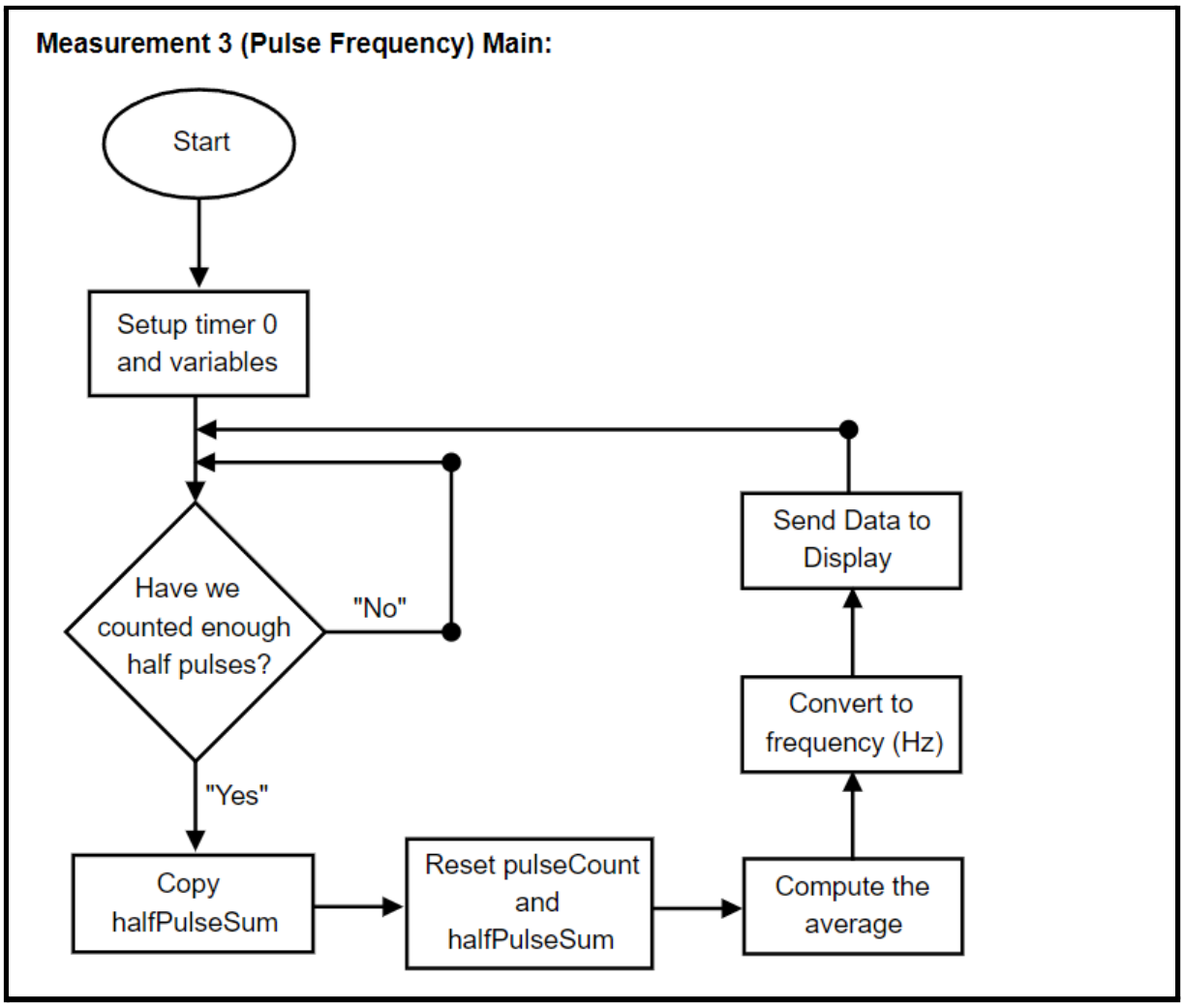
TMOD:

- The GATE bit was set to 1. This was done so that timer 0 only counted up when the square wave input to P3.2 was high.
- The C/T bit was cleared to 0 in order so that timer 0 would count clock cycles, thereby allowing time durations to be measured.
- M1 was cleared to 0, and M0 was set to 1. This configured timer 0 to be in 16-bit mode.

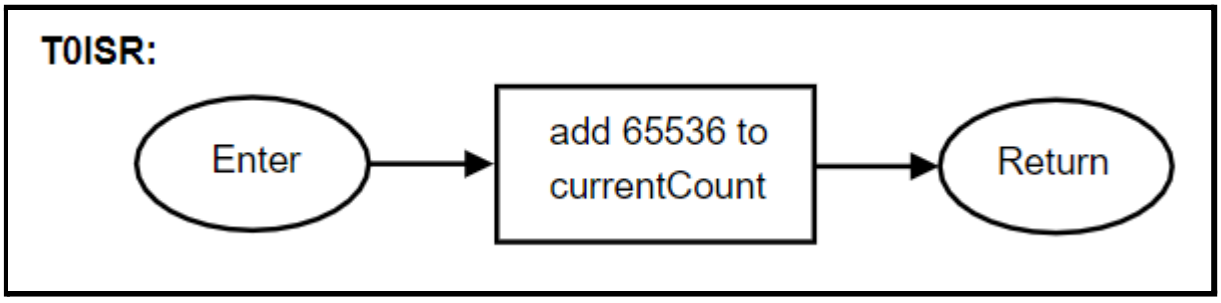
TCON:

- The TR0 (timer 0 run control bit) was set to 1.
- The IT0 (trigger type for external interrupt 0) bit was set to 1 so that the trigger type was edge-sensitive, meaning a high-to-low transition by the square wave on P3.2 would trigger an interrupt.

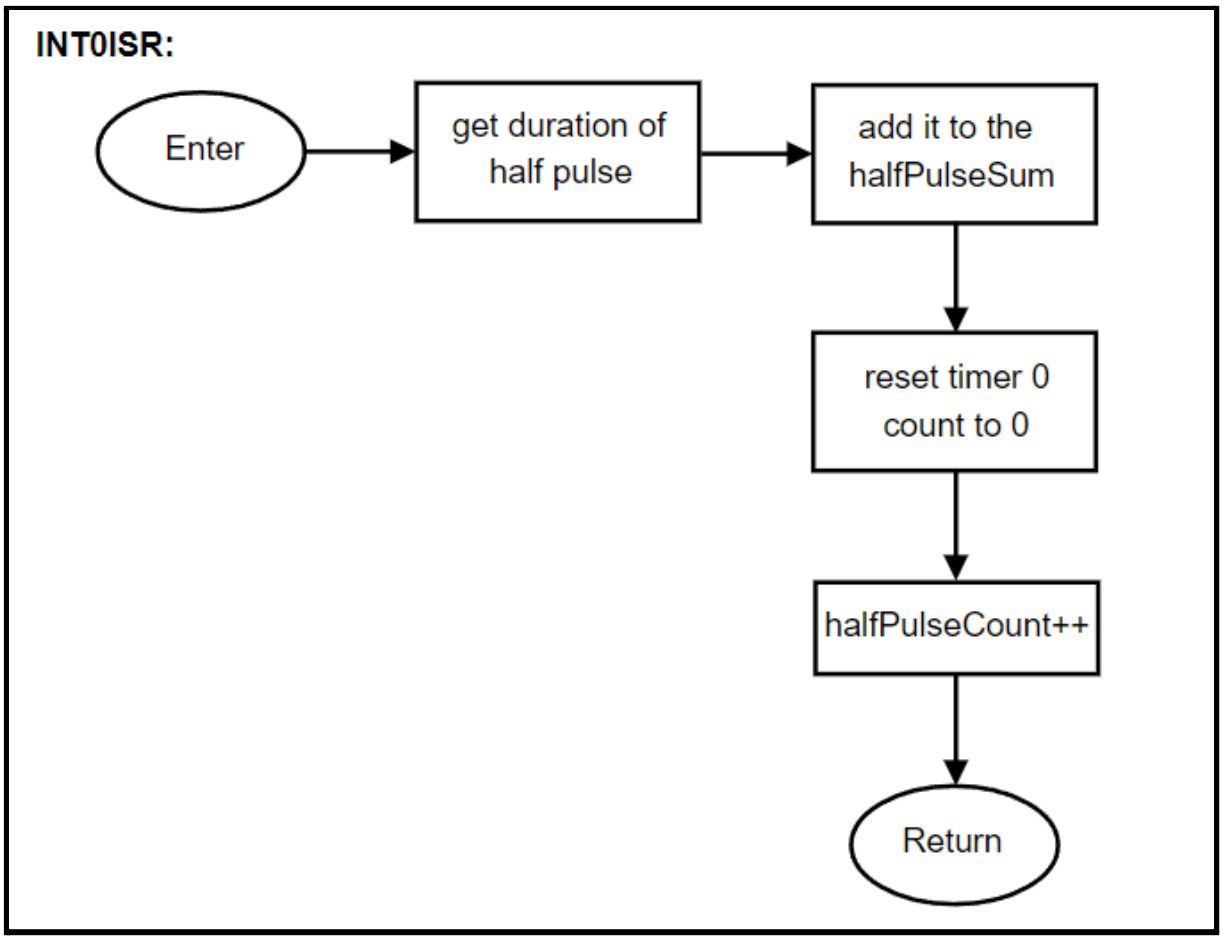
Below is shown the flowchart for the main part of the program:



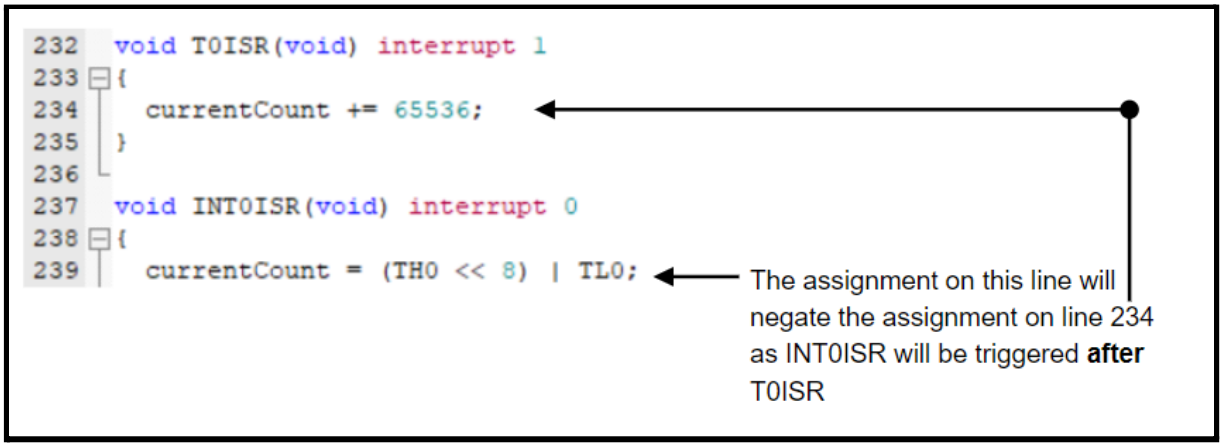
The main part of the logic got carried out inside the ISR for external interrupt 0 (INT0ISR). Since our design was using timer 0, there was also an ISR for it (T0ISR). Its objective was to handle overflows. Below are shown the two flowcharts for these ISRs, although on reflection there was a mistake in timer 0's which would require a change to be made in both it and INT0ISR - a solution is discussed below the two diagrams:



*currentCount is a variable to represent the clock cycle count of a half period. It is used by INT0ISR to update the variable halfPulseSum.



Firstly, the problem with the formulation of timer 0's ISR is that it increments the value of currentCount by 65536, but then the effect of this is negated by INT0ISR which assigns currentCount whatever value is in timer 0 when it stops counting (caused by the square wave input on P3.2 going low). This means if the square wave input has a period that is sufficiently long to cause timer 0 to overflow, then its duration will not be measured properly. Hopefully the diagram below makes this clearer:

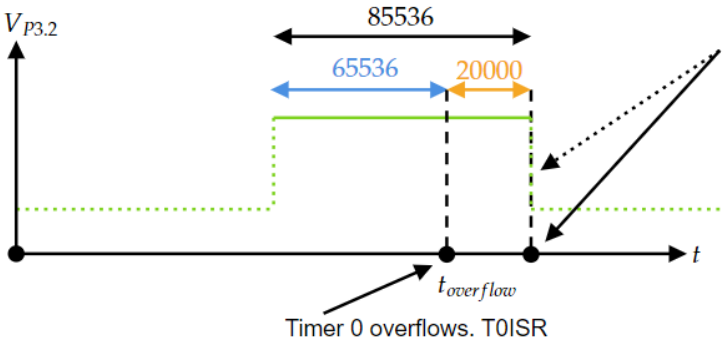


A better solution would have been to have T0ISR increment an overflow count variable. Then, inside INT0ISR, currentCount would have been assigned the value in timer 0, before being incremented by 65536 however many times timer 0 overflowed:

```
231  void T0ISR(void) interrupt 1
232  {
233    overflowCount++; // increment the overflow counter
234  }
235
236  void INT0ISR(void) interrupt 0
237  {
238    currentCount = (TH0 << 8) | TL0; // load in timer 0 value
239    currentCount += overflowCount*65536uL; // account for any overflows that might have occurred
240    overflowCount = 0; // reset the overflow counter for the next half pulse
```

```
192  volatile uint8 overflowCount = 0;
193
```

8 bits should be more than enough for the overflowCount variable. Unless the input frequency is very, very low, then this variable should not exceed its maximum value.



Timer 0 overflows. T0ISR is triggered, and the overflowCount variable increments to 1...

Signal on P3.2 makes a high-to-low transition, Timer 0 stops counting, and INT0ISR is triggered. Then currentCount is assigned timer 0's value before being incremented:

currentCount = 20000
currentCount += 1*65536

So currentCount = **85536**

The half period duration in clock cycles

(Converting to Frequency)

In the laboratory while we were testing our frequency measurement design, we simply displayed the average number of clock cycles associated with the half pulse of the periodic input signal. Had we time we would have added in code to convert the half period value from clock cycles to the frequency of the input signal using the following formula:

$$f_{input} = \frac{f_{clock}}{N_{clock\ cycles}}$$

And below is shown some code to implement the above formula:

```
#define FCLOCK 11059200u // clock frequency in Hz

/* To get the number of clock cycles in one full period, we multiply
halfPulseAverage by 2. This is the same as left shifting by 1.*/
N = (halfPulseAverage << 1); // number of clock cycles in 1 full period
frequency = FCLOCK/N; // the frequency in Hz; FCLOCK is the clock frequency in Hz
displayNumber(frequency); // send the data to the display
```

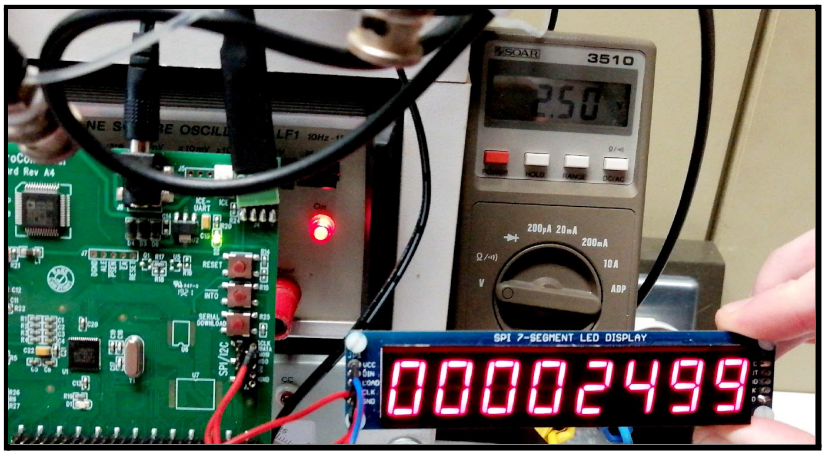
And an example output from a program written in CodeBlocks is shown below:

```
The half pulse average is: 2579 clock cycles
The total number of clock cycles in one full period is: 5158

Now to convert from clock cycles to frequency in Hz using: f = (Fclk/N)

The frequency of the input signal is: 2144 Hz

Process returned 0 (0x0)   execution time : 1.094 s
Press any key to continue.
```

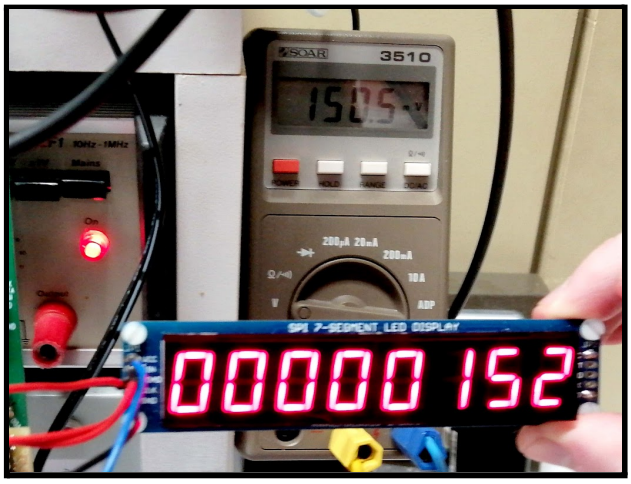
```
int main() {
    uint32 halfPulseSum = 1320448;
    uint16 halfPulseAverage = halfPulseSum >> 9; // dividing by 2^9, or 512
    uint32 frequency;
    printf("The half pulse average is: %u clock cycles", halfPulseAverage);
    uint16 N = (halfPulseAverage << 1); // Number of clock cycles in 1 full period
    printf("\nThe total number of clock cycles in one full period is: %u", N);
    printf("\n\nNow to convert from clock cycles to frequency in Hz using: f = (Fclk/N)");
    frequency = FCLOCK/N;
    printf("\n\nThe frequency of the input signal is: %u Hz\n", frequency);
}
```

(Testing and Results)
We tested each of our measurement designs separately.

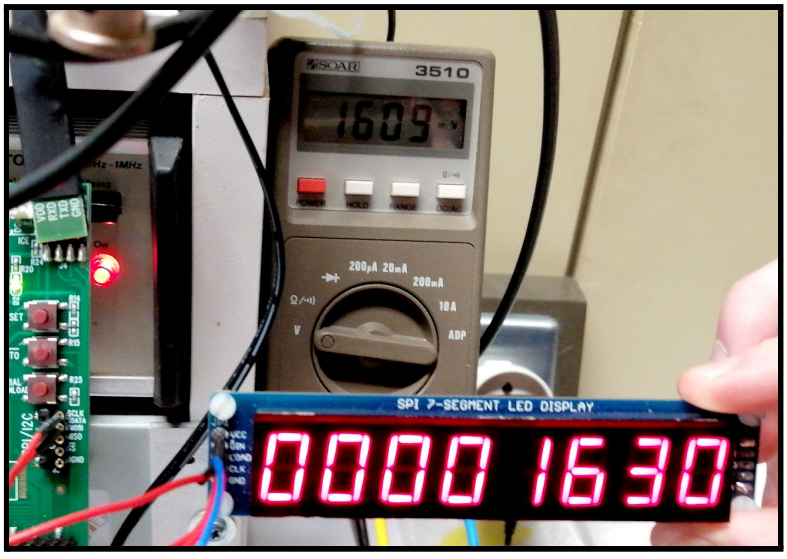
Measurement 1 DC Voltage:



Input was 2.5 V. Display showed 2499 mV.

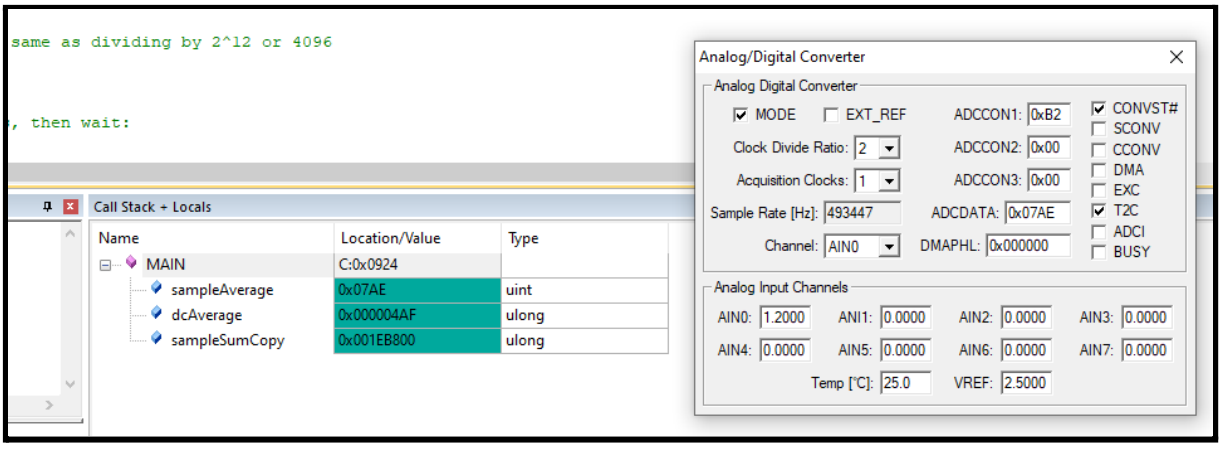


Input was 150.5 mV. Display showed 152 mV.



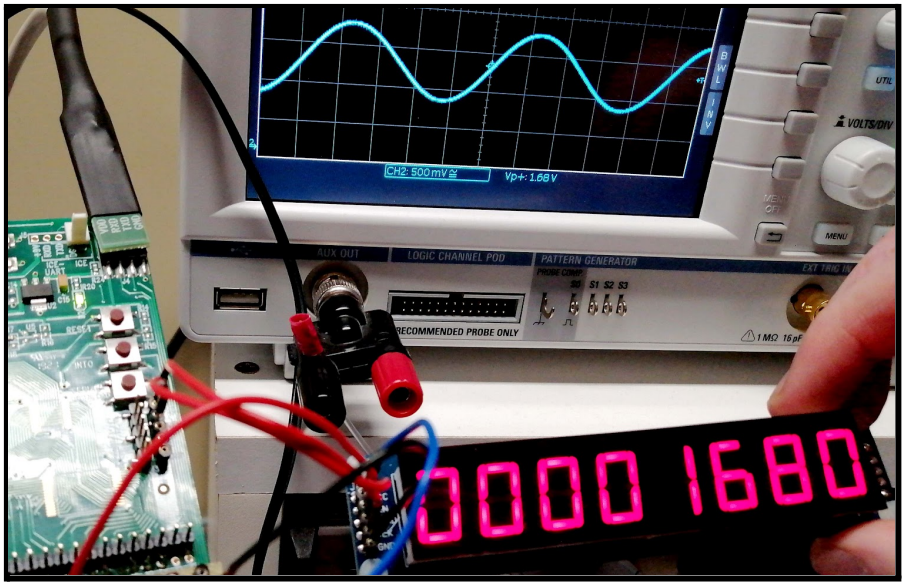
Input was 1609 mV. Display showed 1630 mV.

We ran our program in the simulator in order to try to gain insight into why there was a small discrepancy between the calculated DC voltage and the value that was actually at the input. A value of 1.2 V was assigned or input to the first analog input channel (AIN0). The program was then run. Shown below is a snapshot from the simulator. It shows the value of dcAverage, a 32-bit unsigned integer that was used to store the result of the sample average, after it had been converted to a voltage in units of mV.

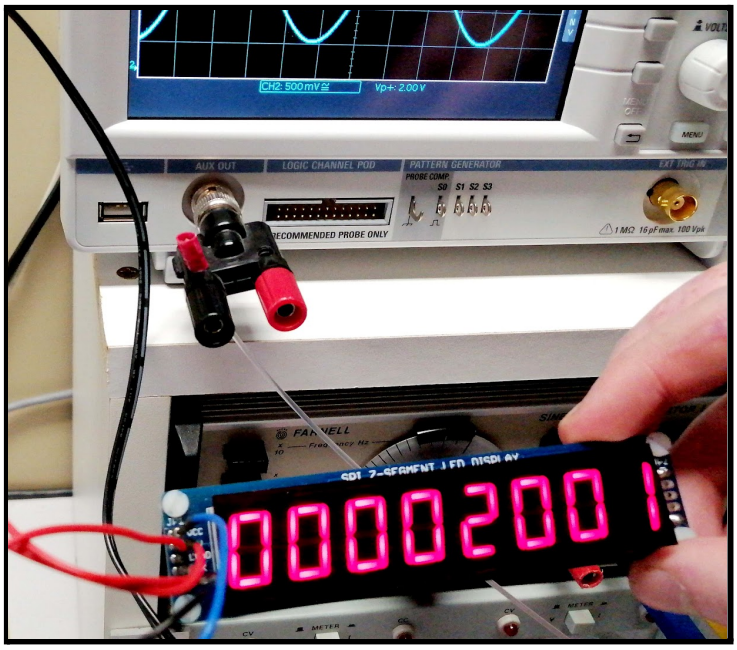


The value of the variable dcAverage, as can be seen above, is 0x000004AF. This is equivalent to 1199 in decimal. This indicates that the discrepancy between the voltage that was being shown on the display and the voltage that was input to the microconverter might not have been related to the algorithm or the software, but instead the result of a calibration error with the on-chip ADC.

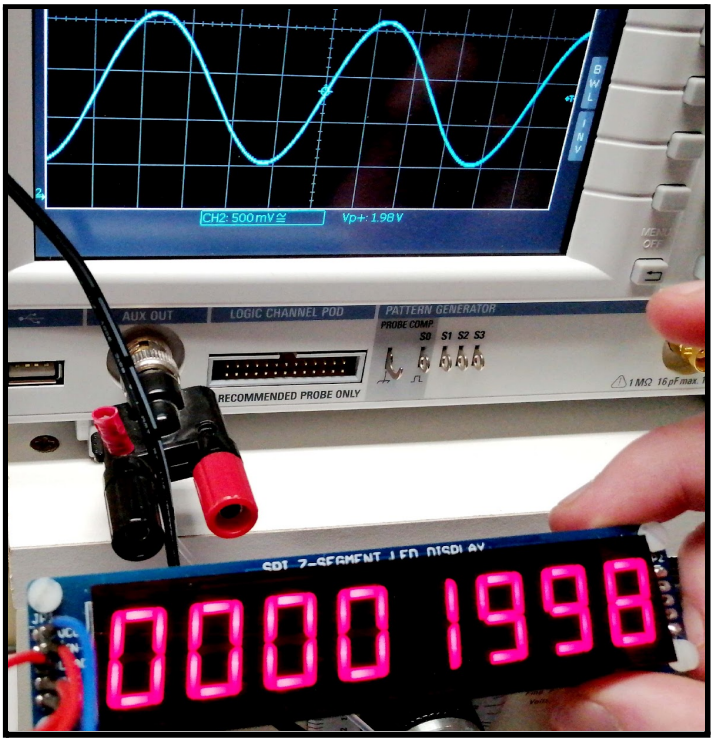
Measurement 2 Amplitude Estimation:



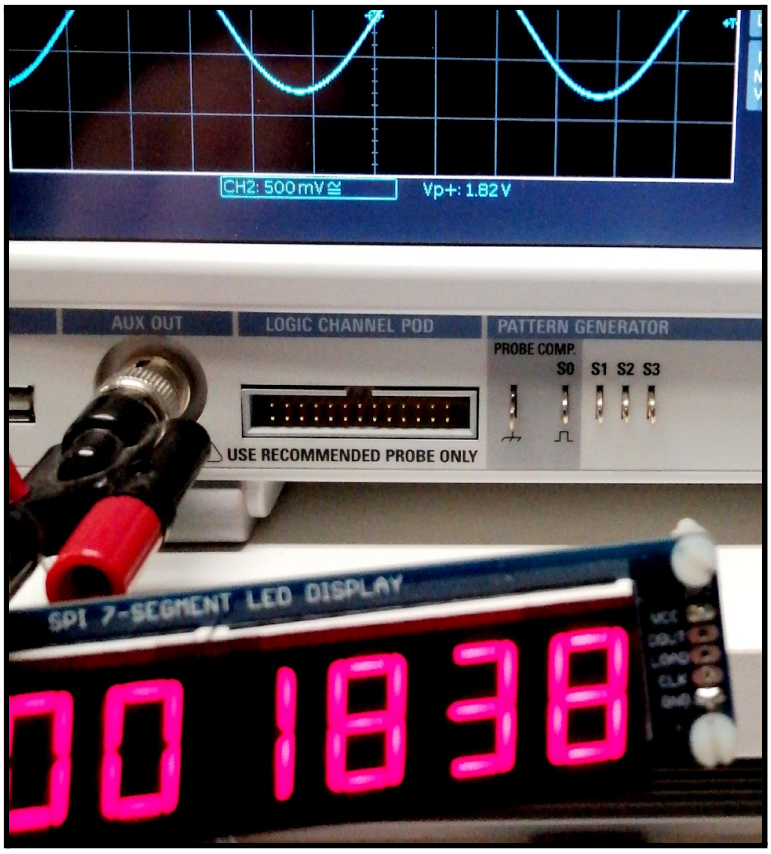
Input amplitude was 1.68 V. Display showed 1680 mV.



Input amplitude was 2 V. Display showed 2001 mV.

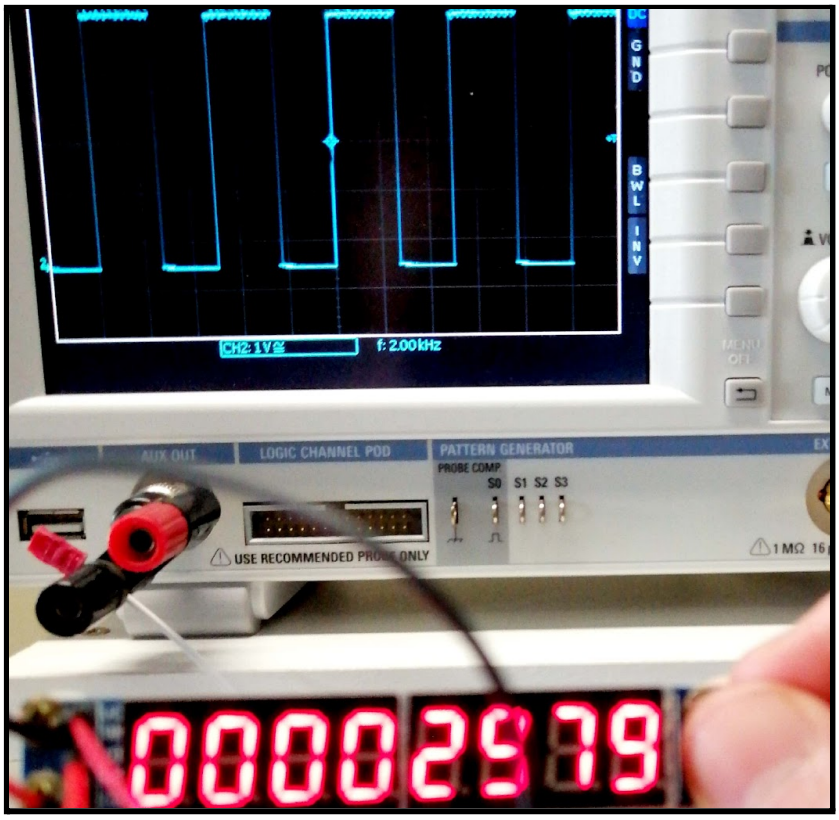


Input amplitude was 1.98 V. Display showed 1998 mV.

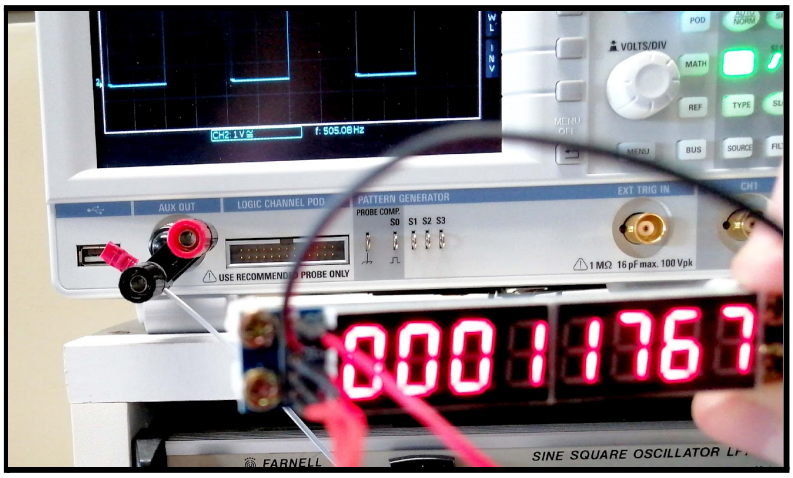


Input amplitude was 1.82 V. Display showed 1838 mV.

Measurement 3 Frequency Counter:



Input frequency was 2 kHz. Display shows 2579 clock cycles for the half period, which gives a frequency of 2.144 kHz.



Input frequency was 505.08 Hz. Display shows 11767 clock cycles for the half period, which gives a frequency of 469.92 Hz.

As can be seen from the two photos above, our results were close, but a little off. We didn't have sufficient time at the end of the laboratory session in order to inspect what might have been leading to the discrepancy.

(Program Size)

dcVoltage.c (inspecting the listing file after compilation)

```
MODULE INFORMATION:   STATIC OVERLAYABLE
    CODE SIZE         =    385     ----
    CONSTANT SIZE     =   ----     ----
    XDATA SIZE        =   ----     ----
    PDATA SIZE        =   ----     ----
    DATA SIZE         =     10       11
```

frequencyCounter.c (inspecting the listing file after compilation)

```
MODULE INFORMATION:   STATIC OVERLAYABLE
    CODE SIZE         =    410     ----
    CONSTANT SIZE     =   ----     ----
    XDATA SIZE        =   ----     ----
    PDATA SIZE        =   ----     ----
    DATA SIZE         =     10        9
```

amplitudeMeasurement.c: 313 Bytes

Lab2Combined.c: 964 Bytes

My teammate wrote the Lab2Combined.c file. It combines the three measurement programs into one, and uses the states of the port 0 pins in order to decide which measurement to perform.

(Conclusion)
In this laboratory session I and my teammate became more familiar with the ADuC841 microconverter. We also got more exposure to writing software for an embedded system. We learned the importance of reading the data sheet closely, as well as planning a design out on paper or on a whiteboard before programming anything.