UCD School of Electrical and Electronic Engineering

EEEN40280
Digital and Embedded Systems

Signal Measurements Report

Name: Anton Shmatov **Student Number: 14445402**

Working with:

I certify that ALL of the following are true:

1. I have read the *UCD Plagiarism Policy* and the *College of Engineering and Architecture Plagiarism Protocol*. (These documents are available on Blackboard, under Assignment Submission.)
2. I understand fully the definition of plagiarism and the consequences of plagiarism as discussed in the documents above.
3. 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.
4. I have not previously submitted this work, or any version of it, for assessment in any other module in this University, or any other institution.
5. I have not plagiarised any part of this report. The work described was done by the team, but this report is all my own original work, except where otherwise acknowledged in the report.

Signed: Anton Shmatov **Date:** 11 March 2019

Details of what is expected in the report are included in the assignment instructions.

Your report and program must be uploaded by 23:00 on Monday 11 March.

You should save your report as a pdf document, and upload it through the submission channel in Brightspace. One member of the team should upload the program through a separate channel in Brightspace. This should be the original source file in C or assembly language, as used in your uVision project. It should be well commented and fit to be compiled and run. If there is more than one source file, combine all the source files into one zip file and upload the zip file.

This assignment was completed in a team of two, where both team mates completed the work together. Some tasks were split up if possible, but due to the debugging phases during the labs, the majority of functions were implemented together.

1 Functional Blocks

The design was split up into four functional blocks: the display, ADC, frequency measurements and switches. These would get their own respective c and header files to make for a cleaner and more structured approach to programming. The display files would deal with only the SPI registers and should require a simple set on input values to display. The ADC files should be capable of providing any supported measurement from one function call. Similarly, frequency should be a one function call. The switches are then responsible for selecting which measurement is currently being displayed.

a 8-Digit Display

The 8-digit display was chosen over the 4-digit display to allow for a wider range of values to be displayed, especially when dealing with frequencies. The SPI interface was written by conforming to the specifications outlined, as seen in Listing 1.

After enabling SPI and matching the required settings on the MAX2719 driver, we set up the registers for normal operation. As *NUM_DIGITS* is typically defined as eight, we set up the driver for operation on all eight digits, with an intensity of eight and for the segment patterns to be decoded from their integer values.

The *write_dispi* function takes in a register address and the value to be written to that register, both in unsigned form. As shown in Listing 2, the transaction begins with the load pin being set to high as an OR operation. If this operation was a simple assignment, the communication between the PC and the board would be severed as some pins on LOAD_PIN=P3 are used for the UART connection.

The register address is then sent by writing to the SPIDAT SFR, which triggers the transfer in the ADuC hardware. A small delay is necessary here to avoid a hardware bug where immediately consequent transfers fail. This is handled by the function *delay_spi*. This function also clears the ISPI flag allowing another transfer to occur. A similar process is undertaken for the data value to be sent. The extra delay following this call could be avoided as there should be enough natural delay in the code to allow for the next transfer, however the ISPI flag must still be cleared.

The main function for writing to the eight-digit display takes in a 32-bit value to allow for up to eight digits, as well as a decimal point position. The 32 bit value is taken on the

assumption that the most significant digits are most important, hence if the value is any longer than eight digits it will be truncated from the back. In Listing 3 this operation begins on line 3 and ends on line 11. By dividing the maximum possible eight digit value into the current value we can infer the amount of extra digits present and remove them via division by ten.

The final portion of this function uses the previously mentioned *write_dispi* function to send a value into each of the eight registers on the driver. This value is sometimes combined with the decimal bit position, depending on the value of *decimal*.

b Frequency

To measure frequency, we decided to make use of the Schmitt Trigger circuit that was added to the development board. This circuit provides a single square pulse per period of an alternating signal. The rising edge of the square wave matches that of the input signal, following the zero crossing. With a clean square wave that matches the frequency of the input signal, any timer in counter mode now seemed like the perfect option to keep track of the pulses.

Following this idea, we implemented Timer 1 to act as the edge counter, connecting the output of the Schmitt Trigger to the edge detector of T1 (P3.5). By detecting edges for a particular time period, we can calculate the frequency of the signal with some simple algebra. Timer 0 is used for timing with the system clock in mode 1, i.e. 16-bit timer.

$$f_{measured} = \frac{num_{edges}}{\frac{2^{16}}{11059200}} \tag{1}$$

However since the edge count can theoretically go as high as 2^{16} in the case of an 11.0592 MHz signal, we cannot multiply the edge count by the system clock as that would cause an overflow even in 32-bit storage. Similarly, we cannot divide the clock into 2^{16} as that would round to 0. Hence an approximation can be made, maintaining resolution but avoiding overflows. The number of edge can be at most 16 bits, while we have 32 bits to work with. We can multiply both top and bottom by 2^{16} and then use a single constant for the bottom line.

$$f_{measured} = \frac{num_{edges} << 16}{\frac{2^{16}<<16}{11059200}} \approx \frac{num_{edges} << 16}{388} \tag{2}$$

Thus the high frequency is found via a shift and 32-bit division. The problem with this method is then the resolution - for each edge counted, we increment roughly 169 Hz. This means that we cannot measure any low frequency signals below this value at all, with horrendous accuracy anywhere below 1.7 kHz. It was decided to only use this method if we count at least 128 edges for this reason, giving a lower bound of ≈ 21.6 kHz and a resolution of slightly less than 1%. This value can be easily changed in the header file however and

give a higher resolution at this point.

If we see that after one timer interval is completed (5.9 ms) we do not have at least 128 edges, we enter the low frequency measurement mode. Timer 1 is set to overflow once it has counted 128 edges in total, taking into account current edge count. Meanwhile, the amount of Timer 0 overflows will be counted by its ISR, allowing us to calculate a lower frequency once 128 edges have been detected. To avoid staying in this mode for too long however, we also implemented a timeout of 4 seconds. Hence we attempt to detect 128 edges within 4 seconds, but will perform the calculation regardless of the edge count at the end.

$$f_{measured} = \frac{11059200 * 128}{num_{overflows} * 65536 + T0} \tag{3}$$

$$f_{measured} = \frac{2^{16} - T1}{4} \tag{4}$$

Both Equations 3 and 4 give a fractional resolution, however we chose to put an integer limitation on the frequency hence our lowest measurable frequency is 1 Hz. Equation 3 is used for any signal that can aggregate 128 edges within 4 seconds (i.e. down to 32 Hz) and Equation 4 is used for any signal slower than 32 Hz. Thus our range in total spans from 1 Hz to 11.0592 MHz.

The math described in equations above was translated into C code and spread between functions and ISRs. Listing 4 begins the frequency measurement by ensuring that all interrupts are enabled and that all timers in their correct modes. After resetting initial values and variables, the timers are enabled and the program enters a while loop until an ISR sets the *measurement_ready* bit, where the measurement value will be returned. The ISR for T0 is responsible for high frequency calculations as well as activating the low frequency mode if necessary, while the ISR for T1 will calculate any frequency between 32 Hz and 21.6 kHz.

c ADC

To make use of the ADC we first need to enable it and configure the various parameters. The clock division was chosen to be 4, as with a division of 2 some problems in sample acquisition were observed, leading to a drifting voltage. This problem could have arisen as a result of extra capacitance from jumper cables. Switching to a slower clock frequency solved the issue. The sample acquisition time was arbitrarily chosen as 3 cycles, and the T2 overflow conversion bit was set, as seen in Listing 5. The ADC was now ready to begin sampling.

The interrupt used by the ADC described in Listing 6 is responsible for obtaining the sum of a set of samples and optionally, the squared sum of a set of samples. The normal sum will always be accumulated as this is needed when calculating both alternating signals and DC signals. The squared summation is activated whenever the measurement mode is set to RMS. Both sum variables are uint32 as only 16 normal samples will fit into a 16 bit sum,

while a squared value needs at least 24 bits. A sum pointed is also incremented at the end of the ISR, as this keeps track of how many samples have been accrued thus far. A conditional statement also checks for the end of the sampling by taking away one from the sum pointer and comparing against a defined value. We must take away one in this statement to allow for 256 samples, as the sum pointer is only uint8, for efficiency in addition. Block averaging was chosen over continuous averaging as we would be wasting a lot of calculation time and values due to slow display refresh rate.

There are three different supported measurements within the ADC file, the first and simplest being the DC measurement. As Timer 2 is used to control the frequency of all conversions, the measurements are based around setting the timer up properly. The function *setup_timer2* takes in a single uint32 frequency and sets the respective reload values in the timer to create that conversion rate. As seen in Listing 7, the timer2 function first constrains the input frequency to the upper and lower limits, which are dictated by the 16 bit value in T2 and the speed of conversion in the ADC.

$$frequency_{upper} = \frac{11059200}{4 * 19} = 145,515 \tag{5}$$

When performing RMS or PEAK measurements, the upper limit is also affected by the speed of the ISR as the squaring of the ADC value takes a lot of cycles. At a clock divisor of 4 and 3 acquisition cycles, the ADC takes 19 cycles to complete the conversion which equates to a sampling frequency of roughly 145 kHz. As a minimum we want 4 samples per period, hence the highest measurable frequency with any certainty of accuracy is about 36.3 kHz, for RMS or PEAK measurements. We will like be able to measure signals that are much higher in frequency however we cannot be certain of where the sampling will occur and hence cannot be as confident in the resulting value. As long as our sampling frequency is not a harmonic of the signal frequency, we should obtain a reasonably accurate result.

After resetting some variables, Timer 2 is enabled in the *start_timer2* function, and sampling begins on channel 0. We then wait for the *sum_ready* bit to be set, after which a simple shift and multiplication performs the conversion.

c.1 DC Measurement

As seen in Listing 7:L34, the DC measurement is created by a division and multiplication. The division is equivalent ot the number of samples collected, i.e. 256 samples equals a right shift of 8, as it is in our case, to get the average value. The constant value ADC_UNIT_LEVEL is initialised as uint32, with value of 610. A unit level of our 12-bit ADC at 2.5V reference is roughly equivalent to 0.000610 V. Hence multiplying our averaged value by 610 results in a micro Volt value. While the ADC is not capable of producing values accurate to each μV, this is a good unit to work with given that we have 8 digits to output to. We thus have a resolution of roughly 0.1 mV with our DC measurements, although we will display all six decimal points.

c.2 RMS Measurement

The *measure_rms* function shown in Listing 8 similarly uses Timer 2 to obtain some samples, although this time in channel 1. The MEASUREMENT_MODE is also set to MEASURE_RMS, to inform the ISR that we need squared sums. This time we also attempt to measure the signals frequency and use that as the base for the sampling frequency, although multiplied by four to achieve a better sampling rate. This is still constrained by our prior limits.

The calculation of RMS begins at Listing 8:L24, where both DC and RMS sums are left shifted as before, to obtain their respective average values. As our ADC cannot inspect negative voltages, a DC offset was used as a level shifter. This DC offset is not needed for our RMS calculations however, so it should be removed. Ideally, the DC offset is known and can be subtracted from each sample, however we cannot be certain of its value. Hence, we can use an assumption and some algebra to take away the offset after the summation.

$$\sqrt{\frac{\sum x_i^2}{N}} = \sqrt{\frac{\sum (s_i + offset)^2}{N}} = \sqrt{\frac{\sum s_i^2 + 2s_i o + o^2}{N}} \tag{6}$$

assume $\sum s_i = 0$ as AC signal should be zero mean

$$RMS = \sqrt{\frac{\sum x_i^2}{N} - o^2} = \sqrt{\frac{\sum s_i^2}{N}} \tag{7}$$

The square root is implemented by flipping each bit of a 16-bit value and checking whether we are closer or further from the actual value. A point of note is a specific check on the least significant bit, as it can have a drastic effect on how close the square root is to the real value. Finally, we can take away the scaling of 2 provided by the level shifter and multiply once again by our ADC unit level to obtain the RMS voltage. It should be noted that the level shifter does not provide an exact half-scaling, hence the voltage displayed is slightly erroneous.

c.3 Peak Measurement

The peak measurement shown in Listing 9 is very similar to that of RMS, the only difference being another left shift before the square root to provide a multiplication of $\sqrt{2}$ as is required when obtaining the peak value of a sinusoid from its RMS measurement.

d Switches

The switches on the modded board serve to select which mode is currently being utilised by the micro-controller. This is then used together with the previously defined functions to display a measurement onto the screen. Listing 10 shows the general method in which this was achieved, where each case statement is responsible for generating a measurement and then displaying it. As we have a total of four functions, we utilised the least significant two

switches for the task. As such, we mask those two bits with ones to ensure that we read the desired state.

2 Program Flow

The workflow of the program is very simple after set up; the switches are read and the respective mode is carried out with a single function to obtain a measurement. This measurement is then displayed on the screen, and the switches are read once more, repeating the cycle.

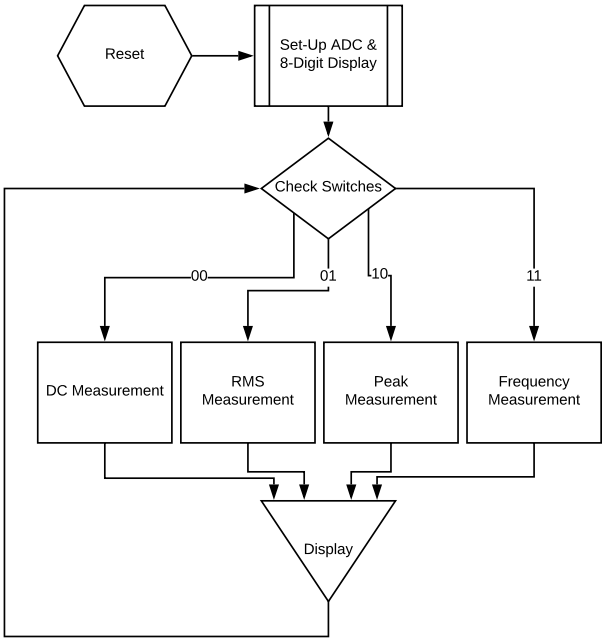

Figure 1: Program Workflow

From the generated listing files, the program is observed to be 2095 bytes in code and 40.3 bytes in data. The large code size resulted as an effort to make the code more structured and segmented, as many portions were separated into headers and functions creating more overhead. This still fits easily onto the micro-controller however hence is not an issue. All optimizations were enabled and code size was favoured to achieve these numbers.

3 Testing

Testing was performed on each individual section as they were completed, and a final test at the end with all parts included in the code.

The DC measurements and display were the first two sections to be tested. For this test, a continuous averaging method was used, similar to an FIR filter. The measurement was

clean and not too spurious, however we came upon some issues with the display. The first issue caused a several hour delay as we had mislabeled the address of the scan limit register and hence had only one digit showing up on the screen constantly. The second issue, also with the display, caused connectivity issues with the PC and was a result of writing to the UART pins while using SPI.

The frequency module was tested next, showing perfect results for higher frequencies but failing to produce any accurate results for lower frequencies. This module was then extended to use a different counting method for slower signals, and was thereafter working within margin of error for all signals. The slower signals in particular had fantastic results, matching the oscilloscope readings perfectly.

The RMS and Peak voltages were then put under test. After some minor difficulties of putting the input signal into the wrong pins, the RMS and Peak voltages appeared on the display and were fairly accurate. On average, an error of 0.05 V was observed and can be drawn to an inaccurate resistor scaling on the level shifter, as the measurements held steady and did not fluctuate too much. It was at this point that we also began using block averaging overing continuous averaging due to the lesser overhead. Additionally, our assumption with RMS measurements of a zero mean signal occasionally gives spurious results as we might not sample exact periods, but in general the value is very close to the expected.

Finally, a full test with all functions and switches integrated was performed. All functions were observed to be working well and the switches were successfully changing modes with no sign of faults.

The instrument was deemed to work as expected and hence the laboratory assignment was completed. A possible improvement that could have been made to the device are some error messages upon a fault in calculation, although this was never observed in testing. Optionally, an indicator with the LEDs or display to show which mode was currently being utilised.

4 Appendix

```
1  SPICON =
2    (0 << ISPI_P) | // interrupt
3    (0 << WCOL_P) |
4    (1 << SPE_P)  | // enable SPI
5    (1 << SPIM_P) | // master
6    (0 << CPOL_P) | // SCLOCK idle low
7    (0 << CPHA_P) | // leading clock edge transmits
8    (3 << SPR_P);   // clock rate divisor, 2-16
9
10   write_dispi(SCAN_LIMIT_REG, NUM_DIGITS - 1); // all digits allow to display
```

```
11   write_dispi(INTENSITY_REG, NUM_DIGITS);       // set brightness 8
12   write_dispi(DECODE_REG, 0xFF); // decode ints to patterns on all digits
13   write_dispi(DISP_TEST_REG, OFF);// normal operation
14   write_dispi(SHUTDOWN_REG, ON);  // normal operation
```

Listing 1: SPI Setup

```
1  // write to the spi bus for the display
2  void write_dispi(uint8 reg, uint8 mdata) {
3   // load to on for accepting new data
4   LOAD_PIN |= LOAD_PIN_ON;
5
6   // send two sets of 8 bits
7   SPIDAT = reg;
8   delay_spi();
9
10  SPIDAT = mdata;
11  delay_spi();
12
13  // load to off
14  LOAD_PIN &= LOAD_PIN_OFF;
15 }
```

Listing 2: Write to Display

```
1  // write 32 bit value and decimal point to display
2  void write_8display(uint32 mdata, int8 decimal) {
3   uint8 tempd = mdata / DIG8_MAX; // will always be less than 255
4
5   // truncate the value to the number of digits currently being displayed and
        add the decimal point
6   while(tempd != 0) {
7    mdata /= 10;
8    tempd /= 10;
9   }
10
11  decimal++; // increment to account for register offset
12  // assign digits to array
13  for(tempd = 1; tempd <= NUM_DIGITS; tempd++) {
14   if(tempd < decimal) // normal digit or before decimal point
15    write_dispi(tempd, mdata % 10);
16   else if(tempd == decimal) // decimal point
17    write_dispi(tempd, (mdata % 10) | DECIMAL_POINT);
18   else if(mdata != 0) // after decimal but nonzero
19    write_dispi(tempd, mdata % 10);
20   else // else blank
21    write_dispi(tempd, BLANK);
22
23   // next digit
24   mdata /= 10;
25  }
```

```
26 }
```

Listing 3: Write to Display

```
1  // get the frequency of the input signal
2  uint32 get_frequency() {
3   // set timer modes
4   TMOD =
5     (0 << T1GATE_P) |  // Gate control, enable timer 1 when TR1 bit is set
6     (1 << T1CT_P) |    // select counter operation
7     (1 << T1MODE_P) |  // 16-bit counter
8     (0 << T0GATE_P) |  //
9     (0 << T0CT_P) |    // select Timer operation
10    (1 << T0MODE_P);   // 16-bit Timer
11
12   // enable all interrupts (IE SFR Bit)
13   EA = 1;
14   ET0 = 1;
15   ET1 = 1;
16
17   // High byte of Timer 0&1
18   TH0 = 0;
19   TH1 = 0;
20   // Low byte of Timer 0&1
21   TL0 = 0;
22   TL1 = 0;
23
24   // reset flags
25   measurement_ready = 0;
26   low_freq_measurement = 0;
27
28   // let timers run
29   TCON =
30    (0 << TF1_P) | // Counter 1 overflow flag is 0 in the beginning
31    (1 << TR1_P) | // Time 1 starts counting
32    (0 << TF0_P) | // Timer 0 overflow flag is 0 in the beginning
33    (1 << TR0_P);  // Timer 0 starts Timing
34
35   // wait for measurement to finish
36   while(!measurement_ready);
37
38   // return the measurement
39   return measured_frequency;
40 }
```

Listing 4: Get Frequency

```
1  EA = 1;     // enable all interrupt sources
2
3  // 17 cycles minimum for acquisition
4  ADCCON1 =
5   (1 << MD1_P) | // power up
```

```
6   (0 << REF_P) | // internal reference
7   (3 << DIV_P) | // division by 4, 32 - 2
8   (2 << ACQ_P) | // 3 clocks to acquire, 1 - 4
9   (1 << T2C_P) | // enable Timer2 overflow bit
10  (0 << EXC_P);  // disable external interrupt
```

Listing 5: ADC Set-Up

```
1  void adc() interrupt 6 {
2   // obtain adc value
3   uint16 adc_value = ((ADCDATAH & HIGH_BYTE_MASK) << 8) | ADCDATAL;
4
5   // sum the normal average
6   dc_average += adc_value;
7
8   // if RMS or peak, sum squared values
9   if(MEASUREMENT_MODE == MEASURE_RMS) // 0x01
10   rms_average += (uint32) adc_value * adc_value;
11
12  // increment counter
13  sum_pointer++;
14
15  // check if enough samples taken, -1 for 255 capability
16  if((sum_pointer - 1) == ((1 << SAMPLE_SHIFT)) - 1) //1000 0000
17   sum_ready = 1;
18 }
```

Listing 6: ADC ISR

```
1  void setup_timer2(uint32 frequency) {
2   // apply frequency range limits
3   if(frequency > ADC_FREQ_LIM)
4    frequency = ADC_FREQ_LIM;
5   else if(frequency < ADC_LFREQ_LIM)
6    frequency = ADC_LFREQ_LIM;
7
8   frequency <<= 2;
9   frequency = ADUC_CLOCK / frequency;
10  frequency = COUNTER_16_MAX - frequency;
11
12  T2CON = 0; // make sure timer is not running
13  RCAP2H = frequency >> 8;
14  RCAP2L = frequency;
15 }
16
17 uint32 measure_dc() {
18  setup_timer2(ADC_FREQ_LIM >> 5); // go at maxfreq / 32 (arbitrary)
19
20  MEASUREMENT_MODE = MEASURE_DC;
21
22  ADCCON2 =
23   (0 << ADCI_P) | // interrupt
```

```
24     (0 << DMA_P) |
25     (0 << CCONV_P) |
26     (0 << SCONV_P) |
27     (0 << CHAN_P); // channel select 1
28
29   start_timer2();
30
31   await_measurement();
32
33   return ((dc_average >> SAMPLE_SHIFT) * ADC_UNIT_LEVEL);
34 }
```

Listing 7: Measure DC

```
1  void measure_at_frequency(uint32 frequency) {
2   setup_timer2(frequency);
3
4   MEASUREMENT_MODE = MEASURE_RMS;
5
6   ADCCON2 =
7     (0 << ADCI_P) | // interrupt
8     (0 << DMA_P) |
9     (0 << CCONV_P) |
10    (0 << SCONV_P) |
11    (1 << CHAN_P); // channel select 0
12
13  start_timer2();
14
15  await_measurement();
16 }
17
18 uint32 measure_rms() {
19  // try and get the frequency of the incoming signal before measuring
20  uint32 freq = get_frequency();
21  measure_at_frequency(freq);
22
23  // shift and scale the measured values to calculate the rms
24  dc_average >>= SAMPLE_SHIFT;
25  rms_average >>= SAMPLE_SHIFT;
26
27  rms_average -= ((uint32) dc_average * dc_average);
28  rms_average = sqrt(rms_average) << LEVEL_SHIFT_SCALE;
29
30  return rms_average * ADC_UNIT_LEVEL;
31 }
```

Listing 8: Measure RMS

```
1  uint32 measure_peak() {
2   // try and get the frequency of the incoming signal before measuring
3   uint32 freq = get_frequency();
4   measure_at_frequency(freq);
```

```
5
6   // shift and scale the measured values to calculate the peak
7   dc_average >>= SAMPLE_SHIFT;
8   rms_average >>= SAMPLE_SHIFT;
9
10  rms_average -= ((uint32) dc_average * dc_average);
11  rms_average = sqrt(rms_average << ROOT_2) << LEVEL_SHIFT_SCALE;
12
13  return rms_average * ADC_UNIT_LEVEL;
14 }
```

Listing 9: Measure Peak

```
1  uint8 mode;
2  uint32 measurement;
3  SWITCH_PORT = SWITCH_PORT_MASK;
4  mode = SWITCH_PORT & SWITCH_PORT_MASK;
5
6  //setup hardware based on new mode
7  switch(mode){
8   case DC_MODE:
9    // measure DC voltage
10   measurement = measure_dc();
11   write_8display(measurement, 6);
12   break;
13  case RMS_MODE:
14   // measure RMS
15   measurement = measure_rms();
16   write_8display(measurement, 6);
17   break;
18  case PEAK_MODE:
19   // measure Peak Value
20   measurement = measure_peak();
21   write_8display(measurement, 6);
22   break;
23  case FREQUENCY_MODE:
24   // measure frequency
25   measurement = get_frequency();
26   write_8display(measurement, NO_DECIMAL);
27   break;
28 }
```

Listing 10: Switch Modes