PIC Based Test Instruments

Grant Wheatley, Nikola Dancejic, Tuan Nguyen

ABSTRACT:

Collins Radio requested a piece of testing equipment to measure three primary aspects of their radio production. The first testing system developed measures the amount of radio devices are produced per hour in the Collins Radio factory. The second system tests the frequency of the radio from 100Hz to 1MHz. The next system tests the frequency spectrum and displays the frequency of the highest amplitude noise. The final system measure the period of a signal.

INTRODUCTION:

The purpose of this project is to develop a piece of testing equipment for Collins radio in order to test its manufacturing process. The device is required to measure frequency, period, and count the amount of radios being produced per hour.

For our purposes a PIC microcontroller is being used to calculate these measurements. In order to send over the measurements to a computer and allow for control over the actions of the PIC, the RS-232 communication interface will also be used. In order for each of the testing module to communicate a LAN network using Master/Slave SPI communication interface will be the backbone of our design.

EE 475 Lab 2

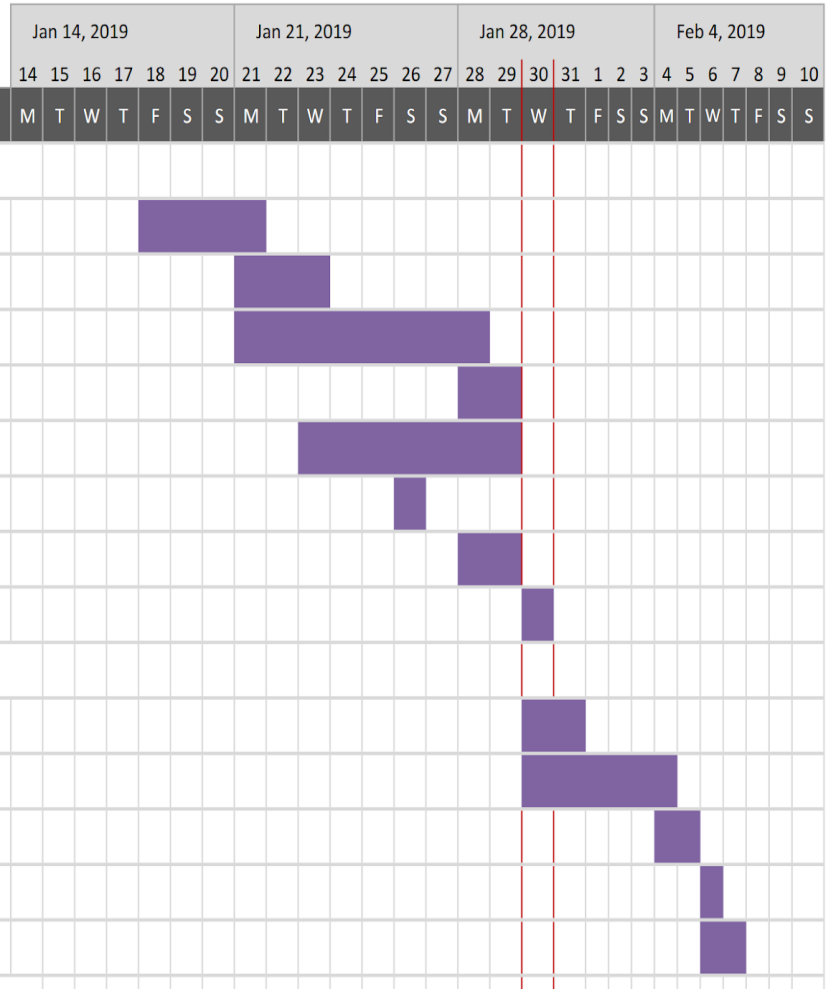
Group 3

Tuan Nguyen, Nikola Dancejic,
Grant Wheatley

Project Start: Fri, 1/18/2019

Display Week: 1

TASK	ASSIGNED TO	START	END
Phase 1 Measuremnents ad Display			
Familiarize with MPLAB and PIC	Tuan Nguyen, Nikola Dancejic, Grant Wheatley	1/18/19	1/21/19
Unit Counter	Tuan Nguyen	1/21/19	1/23/19
Frequencey Measurement	Tuan Nguyen and Grant Wheatley	1/21/19	1/28/19
Spectrum	Tuan Nguyen	1/28/19	1/29/19
Hardware wriring	Grant Wheatley	1/23/19	1/29/19
LCD Integration	Tuan Nguyen, Nikola Dancejic, Grant Wheatley	1/26/19	1/26/19
EIA-232 Sending Commands	Nikola Dancejic	1/28/19	1/29/19
Phase 1 Demo	Tuan Nguyen, Nikola Dancejic, Grant Wheatley	1/30/19	1/30/19
Phase 2: Master/Slave System			
Connect to PC using SPI	Nikola Dancejic, Grant Wheatley	1/30/19	1/31/19
Use SRAM to save 16 measuremnets	Tuan Nguyen	1/30/19	2/4/19
Displaying past measuremnenst on PC	Nikola Dancejic, Grant Wheatley	2/4/19	2/5/19
Demo	Tuan Nguyen, Nikola Dancejic, Grant Wheatley	2/6/19	2/6/19
Lab Report	Tuan Nguyen, Nikola Dancejic, Grant Wheatley	2/6/19	2/7/19



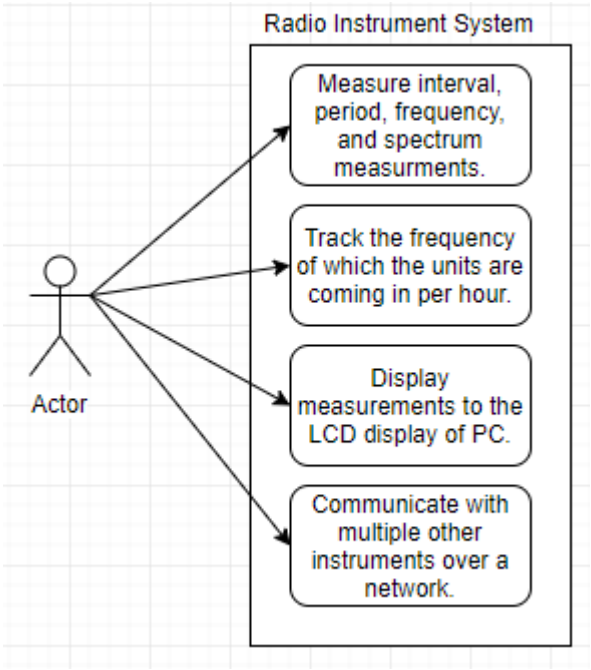
DISCUSSION:

Requirements Specifications:

According to the clients, the instruments should be able to measure signal spectrum, frequency, number of units coming in per hour. The client wants to also be able to accomodate for older models of the radio units. Once the measurements are taken, they will be displayed to the LCD screen of the PC.

All instruments will be connected by a network, which will all be controlled by a singular remote
instrument that is connected to the PC.

The clients also want to have a great number of these instruments lined up in their factories to make multiple measurements at the same time. Thus, a network would have to be designed and set up so that the measurements could be coordinated.



Use Cases Diagram (Figure 1)

System Specifications:

Interval, Period, Frequency, and Spectrum Measurements:

Frequency: **100Hz to 1MHz**

Noise Spectrum: **500Hz to 1000Hz**

Period: **10ms - 1s**

Units per Hour: **< 60 units**

Must be able measure interval and period of radio signals, frequency of radios, make several low frequency noise measurements and determine which frequencies the noise has the highest energy.

Possible Errors:
Internal clock of the instrument could be incorrect. The radio we are testing on could be from an older model that doesn't require frequency measurements. A/D pin from the PIC may be damaged.

Possible Errors:
The hardware counter of the this function may be broken, therefore making this entire system defective. Communication might produce inaccurate readings and translation to the other devices. Limitations in the computational ability of the PIC might lead in the inability for our system to measure a wide range of frequencies as requested.

Display the Measurements:
Take the measurements from the instrument and display it to the PC screen.

Possible Errors:
The measurements displayed to the screen may be older measurements, due to possible delays from the PIC to the PC.

Communicate with slave instruments over network:
As the title says, the host instrument should communicate and coordinate the actions of the slave instruments with the use of a network.

Possible Errors:
The connections of the slave instruments over the network could be incorrectly wired up in hardware, thereby making the system exhibit undefined behaviors.

DESIGN SPECIFICATION:

Every instrument should be able to make frequency and spectrum measurements. They should also be able to keep track of the number of radio units coming in per hour.

When making frequency measurements, the frequencies from the radio units are expected to be measured up to 1 MHz.

For the spectrum measurements, we have to make a few frequency measurements to determine at which frequency does the noise has the highest energy.

Measurements :

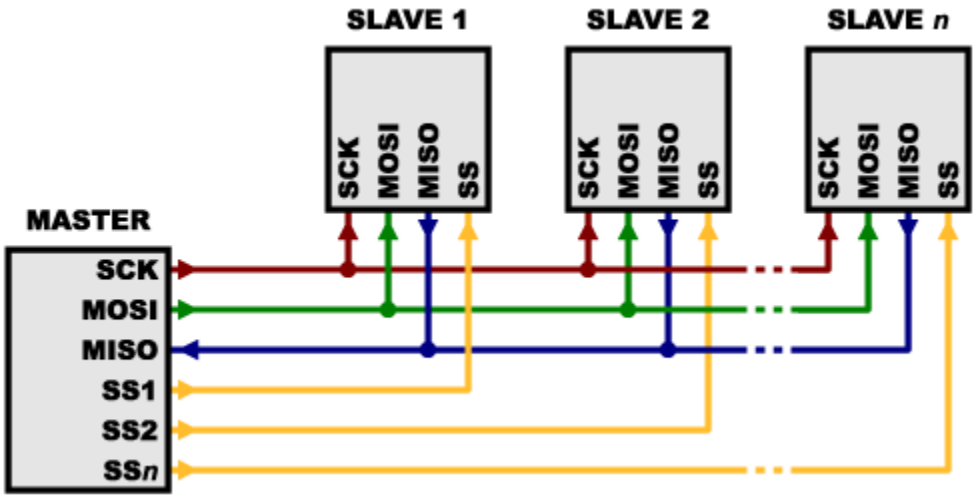
The noise Frequency in the range **500 to 1000 Hz.**
Frequency of **100 Hz to 1MHz with a tolerance of 0.1 Hz.**
Period measurements of: **10ms +- 0.01 ms to 1s +- 0.01 s**

Units per Hour: **0 to 60 units per hour**

When counting the number of units coming in per hour, there will be an IR beam that will be broken every time a unit passes through. In newer models, passing through the beam would create a negative 1 microsecond pulse (5 V to 0 V). However, to accomodate for older units, a positive pulse (0 V to 5 V) would also indicate an older unit passing through the IR beam.

There will be some instruments that will have an output signal that lasts 10.00 ms and other instruments that will last 1.000 second, both with a tolerance of 0.01 ms. It must be noted that these signals are not periodic.

Finally, the instrument should be able to send its measurements to the PC screen. The EIA-232 protocol will be utilized to enable such a communication between the PC and the instrument. This is only for the connection between the master and PC.



LAN Plan/Model (Figure 2)

We plan to have each slave follow the SPI communication model with the master. They will only send their data to the master when their “Slave select” signal is set to high. This will be controlled by the master. The model of the LAN communication can be best summed up in the diagram above. The slave select signals will be the GPIO pins on the master.

Failure Modes and Effects Analysis:

The follow table is a analysis of if a bus connecting two module in the Full System Block Diagram above where to fail.

Bus Connection:	Effect:
RS-232 Connecting PC to Master PIC	• The wrong or unknown command would be sent from the PC to the Master PIC

	• The data being sent from the Master PIC to the PC would be corrupted
Master to LCD Display	• The characters being sent from the Master PIC to the LCD would be corrupted thus displaying incorrect or unknown characters on the display
SPI LAN Lines Connecting Master PIC to Slave PICs	• The data being sent between the Master PIC and Slave PICs would be corrupted • A slave select line could be stuck on or stuck off meaning that two Slaves could be selected or one Slave could never be selected
Data Lines Connecting Slave PIC to SRAM	• The data being sent between the Slave PIC and SRAM would be corrupted
Address Lines Connecting Slave PIC to SRAM	• The address that the Slave is trying to access on the SRAM would be corrupted

Design Procedure:

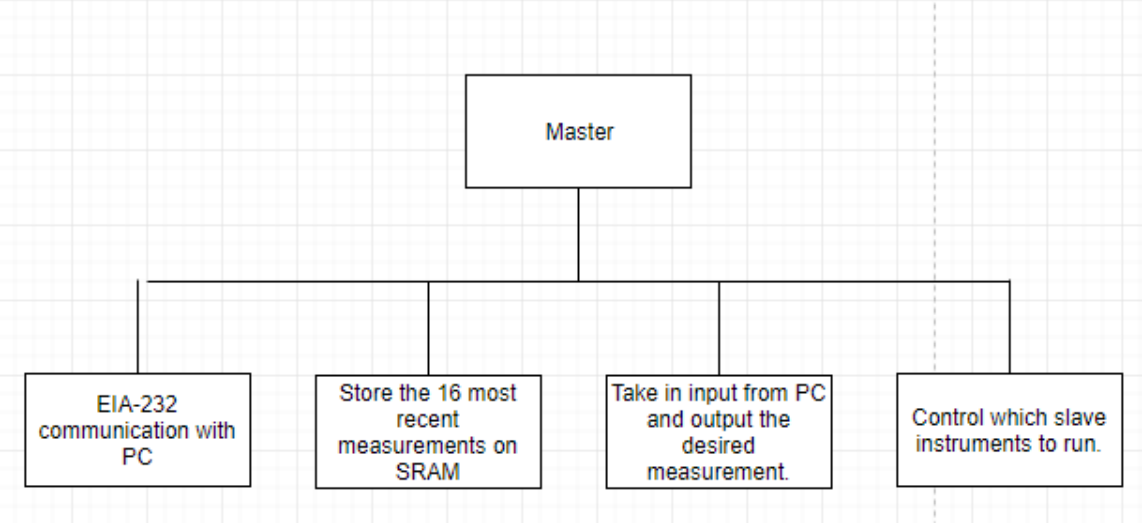
Bill of Materials:

Item:	Amount:	Cost per unit ($):
PIC18F452A Microcontroller	2	07.50
CD4040E 12-bit counter	1	00.40
CY7C128A SRAM	1	20.00
RS-232 Module	1	06.99
Ethernet Module	1	15.99
F5C-2 10Mhz Oscillator	1	3.00
TOTAL		**61.38**

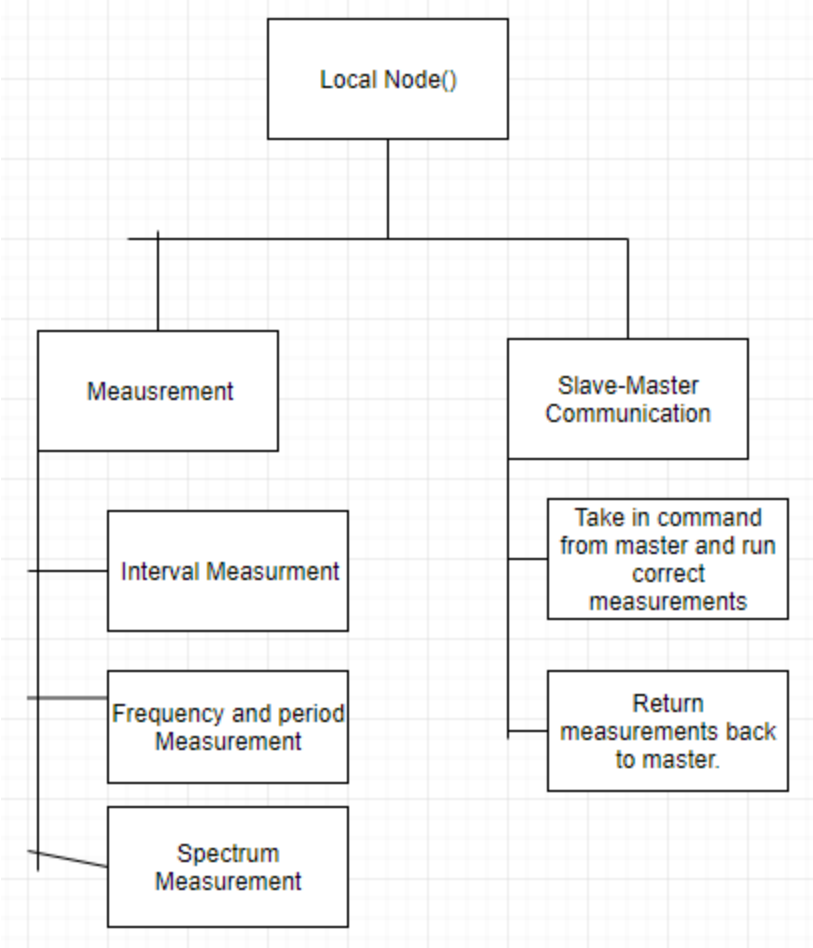
System Description:

The System would consist of a central PC which would handle user interfacing and communication with a central module for control of the remainder of the system. The central module would be a master PIC which would use SPI to command 6 peripheral slave PIC microcontrollers. The slave modules would be used to take measurements and store their last 16 measured values.

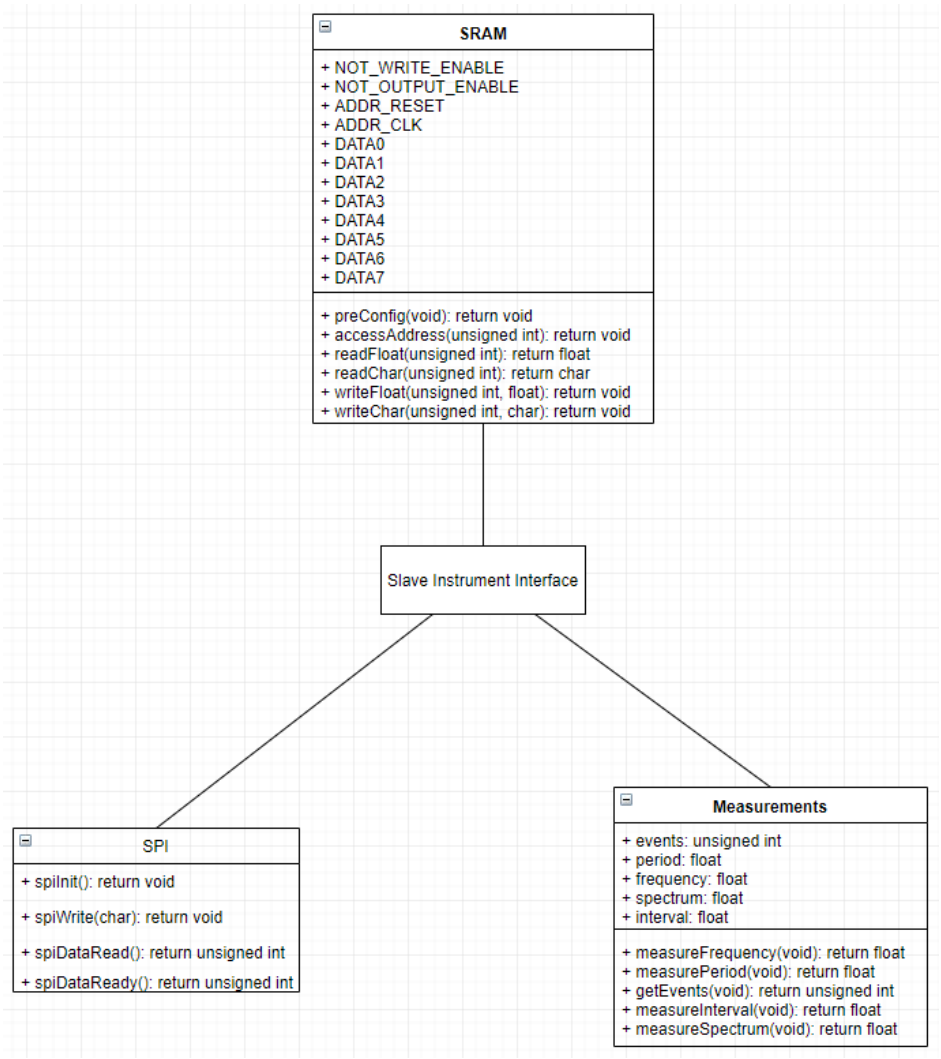
SOFTWARE IMPLEMENTATION:



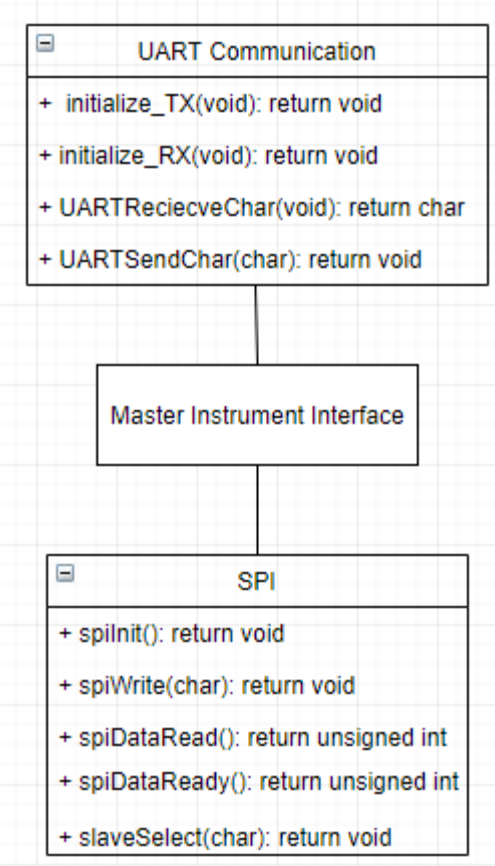
Functional Decomposition Master (Figure 3)



Functional Decomposition Slave (Figure 4)



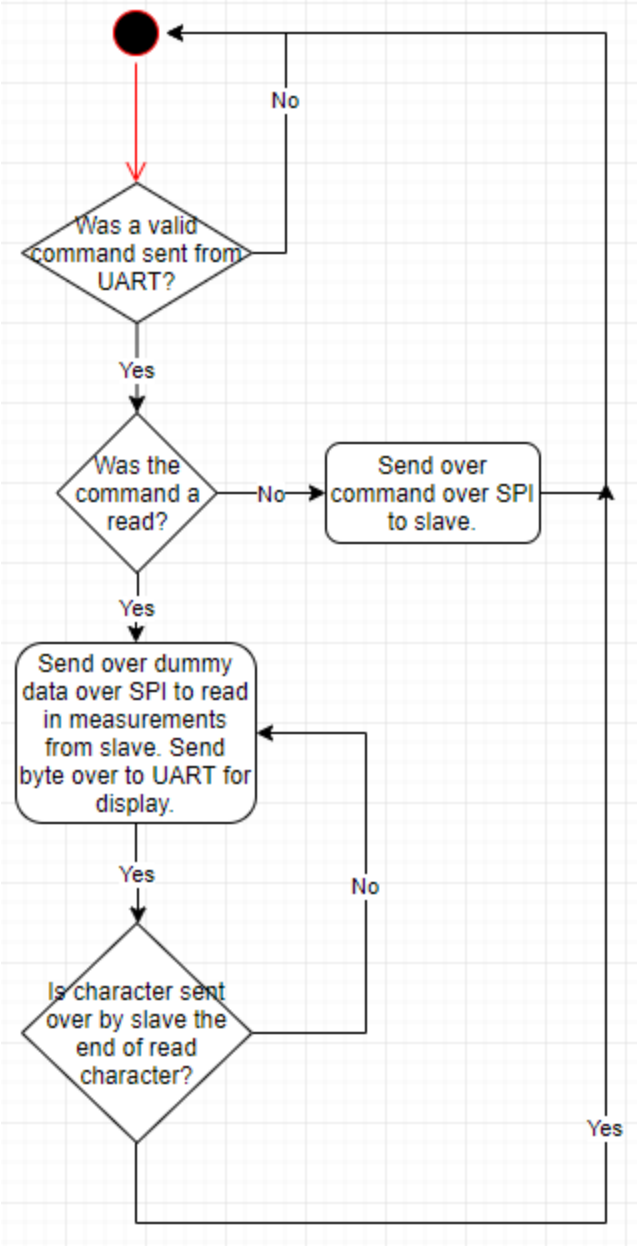
Class Diagram Slave (Figure 5)



Class Diagram Master (Figure 6)

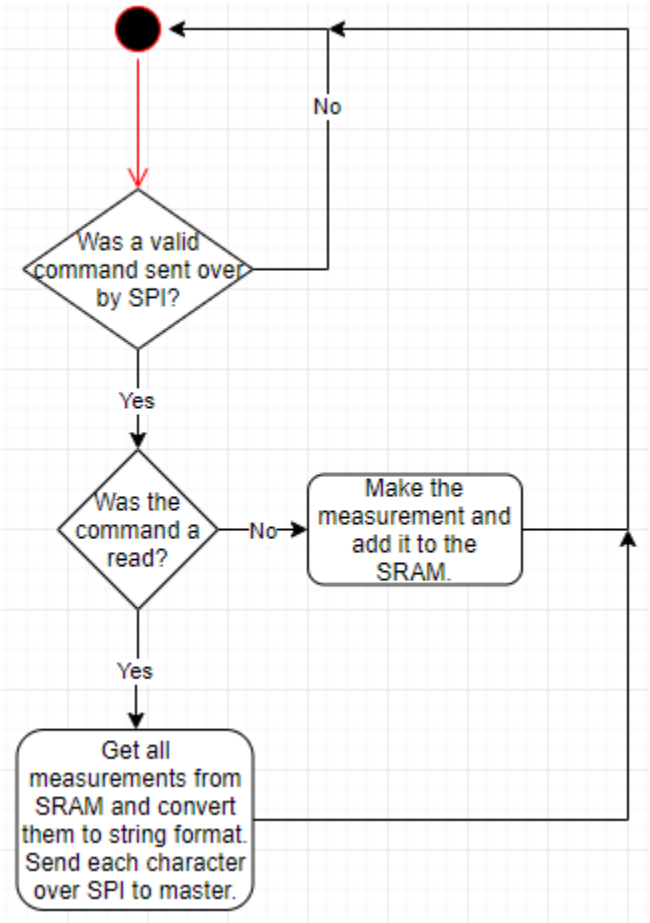
The software design consisted of three overall modules: the master PIC controller, the slave PIC controller and the computer for user interface. The master PIC controller consisted of a UART communication module, for interfacing with the PC, and an SPI which communicated with 6 different slaves. Each slave consisted of an SPI module for receiving commands from the master PIC, a module for storing each measurement into an SRAM, and a module for measuring frequency, intervals, spectrums, events, and period. The PC was used to communicate with the master PIC using an RS-232 interface.

Below is the expected activities process of the master unit:



Master Activities Diagram (Figure 7)

Below is the expected activities process of the slave unit:



Slave Activities Diagram (Figure 8)

UART/RS-232 Communication:

To set up the UART communication, it is all a matter of setting the bits correctly. To send a byte (or character), we just need to wait until the UART buffer is free, and then we set the transit register to our desired byte.

To receive a byte, we just wait until the receive flag is set to 1, and then we read in from the recieve register.

SPI:

For the master we have two different functions, selecting a slave and sending a command to the currently selected slave. The slave selection works by based on the slave requested over UART (A through F), the master pulls the one of it's slave selection pins (RB5 though RB0) to low to select the requested slave. Sending a command to a slave works by based on the command sent over UART, the master sends that command directly over SPI if it is a measurement command. If it is a read command however, the master sends that read command and then continues to send

“dummy data” (which we choose as a ‘@’ character) to the slave so that the SPI Clock is driven and thus allowing the slave to send data back to the master. The master sends the data received over SPI over UART if it is not the “dummy data” that we sent to the slave. The master continues to do this until we receive a stop character from the slave (which we chose as a ‘?’). We also have a ‘X’ command which will reset all the slave selection pins which we use to disconnect from a slave.

```
int selectSlave (void){
        set wasSlaveSelected = 1;
        char slaveChar = UARTRecieveChar();
        if slaveChar is 'X' then
                wasSlaveSelected = 0;
        if slaveChar is ‘A’ through ‘F’ then
                Set RB5 through RB0 accordingly
        return wasFunctionSelected;
}
```

Sudo Code for selecting a slave

```
int selectFunction (char slaveChar){
        set wasFunctionSelected = 1;
        Command = UARTRecieveChar();
        if Command is a measurement (‘F’, ‘P’, ‘I’, ‘S’, ‘E’) then
                spiWrite(Command);
        else if Command is a read (‘R’) then
                spiWrite(Command);
                spiWrite(dummy data);
                dataChar = spiRead();
                while dataChar is not stop character then
                        if dataChar is not dummy data then
                                UARTSendChar(dataChar);
                        Wait till the SPI buffer is full
                        spiWrite(dummy data);
                        dataChar = spiRead();
                UARTSendChar(stop character);
        else
                set wasFunctionSelected = 0;
        Deselect all slaves
        Select slave based on slaveChar
        return wasFunctionSelected;
}
```

Sudo Code for commanding the slave

For the slave, the slave waits till there is data ready to be read in it’s SPI buffer meaning that the master has sent a new command. If it is a measurement command then the slave conducts the measurement type requested and saves that measurement into it’s off board SRAM. If it is a read command, the slave retrieves measurements from the SRAM, uses sprintf to convert the float values into a array of characters, and sends each character of the array once the SPI buffer full bit

is true. We wait to send data until the buffer full bit is true to have the slave be in sync with the master sending "dummy data" to drive the SPI Clock. Once all the measurements have been sent we also send a stop character (which we chose as a '?') to let the master know that all measurements have been sent.

```
if there is new data in the SPI buffer then
        Command = spiRead();
        if Command is a measurement ('F', 'P', 'I', 'S', 'E') then
                conduct measurement and save result to SRAM
        if Command is a read ('R') then
                Retrieve measurements from SRAM
                Convert to char array
                for index in charArray then
                        Wait till the SPI buffer is full
                        spiRead(); //clears buffer full flag
                        spiWrite(charArray[index]);
                        Index++;
                Wait till the SPI buffer is full
                spiRead(); //clears buffer full flag
                spiWrite(stop character);
        else
                spiRead(); //clears buffer full flag
```

Sudo Code for Slave Function Selection

Note that our most of our SPI implementation is influenced by an electrosome post about implementing one-way PIC to PIC SPI communication which we were able to expand into allowing for two-way communication. The post can be found at the following address: https://electrosome.com/spi-pic-microcontroller-mplab-xc8/

SRAM:

Our implementation of the SRAM from the previous lab was sufficient for the implementation of the application for this project. In this case, the state machine from the last project was all implemented in the microchip. Before we read or write to an address, we must first traverse through the SRAM to that address. Before traversing the address, we have to reset the counter and set the NOT_WRITE_ENABLE and NOT_OUTPUT_ENABLE bits to true. Using the microchip, we sent clock signals to a counter until we get the address value we want. Of course, there would need to be a delay of a couple milliseconds afterwards to ensure that all the signals got to where they needed to be.

There are two flavors of reading from the address; character or float. To read a character, we would need to read only the 1 byte from the address. To read a float, we would need to read the next 8 bytes from this address. Each traversal of a byte would require another address traversal function call (described in the paragraph above). Before we begin reading a byte, we have to set the NOT_WRITE_ENABLE bit to true and NOT_OUTPUT_ENABLE bit to false. We then delay for a couple milliseconds to ensure that these bits were set. Then, we read in the data from the 8 data bits from our microchip (DATA0-DATA7). Afterwards, we reset the NOT_WRITE_ENABLE and NOT_OUTPUT_ENABLE bits to true and reset our counter. We then delay for a couple milliseconds to ensure that these bits are set.

Like reading, there are two flavors to writing; character or float. To write a character, we would need to write only the 1 byte from the address. To write a float, we would need to write the next 8 bytes from this address. Each traversal of a byte would require another address traversal function call (described in the paragraph above). Before we begin reading a byte, we have to set the NOT_WRITE_ENABLE bit to false and NOT_OUTPUT_ENABLE bit to true. We then delay for a couple milliseconds to ensure that these bits were set. Then, we read in the data from the 8 data bits from our microchip (DATA0-DATA7). Afterwards, we reset the NOT_WRITE_ENABLE and NOT_OUTPUT_ENABLE bits to true and reset our counter. We then delay for a couple milliseconds to ensure that these bits are set.

Below is the pseudo code for the SRAM functions described above:

```
void accessAddress(unsigned int addr) {
   NOT_OUTPUT_ENABLE = 1;
   NOT_WRITE_ENABLE = 1;

   Reset counter by setting reset bit.

   Delay to make sure all the signals get where they have to be.

   for (int i = 0; i < addr; i++) {
```

```
        ADDR_CLK = 1;
        ADDR_CLK = 0;
    }
}

float readFloat(unsigned int addr) {
    long long num = 0;

    for (int i = 0; i < 8; i++) {
        accessAddress(addr + i);
        NOT_OUTPUT_ENABLE = 0;
        NOT_WRITE_ENABLE = 1;
        delay_ms(10);
        char curByte = DATA0 + (DATA2 << 1) + … + (DATA7 << 7);
        num += curByte << (i * 8);
    }

    return *(float*)&num;
}

char readChar(unsigned int addr) {
    accessAddress(addr);
    NOT_OUTPUT_ENABLE = 0;
    NOT_WRITE_ENABLE = 1;
    char c = (char)(DATA0 + (DATA1 << 1) +...+ (DATA6 << 6) + (DATA7 << 7));
    NOT_OUTPUT_ENABLE = 1;
    NOT_WRITE_ENABLE = 1;
    return c;
}

void writeFloat(unsigned int addr, float f) {
    long long num = *(long long*)&f;
    unsigned int curAddr = addr;

    for (int i = 0; i < 8; i++) {
        accessAddress(addr+i);
        DATA0 = num % 2;
        DATA1 = (num >> 1) % 2;
        ...
        DATA7 = (num >> 7) % 2;
        Shift num down by 8 bits
        NOT_WRITE_ENABLE = 0;
        __delay_ms(5);
        NOT_WRITE_ENABLE = 1;
```

```
    }
}

void writeChar(unsigned int addr, char c) {
    accessAddress(addr);

    DATA0 = c & 1;
    DATA1 = (c >> 1) & 1;
    ...
    DATA7 = (c >> 7) & 1;
    NOT_WRITE_ENABLE = 0;
    __delay_ms(5);
    NOT_WRITE_ENABLE = 1;
}
```

Python Code:

The PC software for communicating with the master PIC and giving the user a basic user interface was done using a python script. The python code had a main menu which allowed the user to select what action to take; Read from a slave, Command a slave, Exit the program, Reset the master. The user then typed in a character from the options {R, C, E, X} which allowed them to enter secondary menus for each of these. Once there, they could select a slave A-F and proceed to select a command to read from the slaves data, or tell it to take one of the necessary measurements. If a measurement was selected, the corresponding character was sent to the master PIC, which transmit that character to the slave (the procedure for this is described in more detail in the SPI communication section). If a read command is given, the PC sends that command to the master PIC, which then proceeds to read the data from the slave (this procedure is also described in the SPI communication section) once the data is read, it is sent back to the PC which prints out any data read.

The python script works by using the pySerial libraries to initialize a connection to the RS-232 port with the master PIC. Once a connection is established, it prints out the name of the port used so that the user can confirm that the desired port connection was established. When a command was written to the master PIC, the function serial.write() was used to write a character. In order to ensure that the right encoding was used, the character had to be encoded using the “encode” function as shown: 'X'.encode('utf-8'). When the script read from the serial port, it used the function serial.read_until('?').decode('utf-8') to read the serial port and add each character to a string until the character ‘?’ is reached. This was our stop character which allowed us to signify that the data was sent over. Each individual measurement was split by a newline character ‘/n’ so the python method String.splitlines() was used to separate them into a list which was then printed out.

Measurement Implementation:

Frequency:

To measure the frequency of the input signal we use a offboard 12-bit up counter with the clock as the input signal in combination with a internal timer Timer0 with a prescaler of 1:256. When we want to measure the frequency, we reset the counter and the timer to zero and let them both start counting till the 8th bit pin of the offboard counter turns true (meaning the counter has reach the value of 512) which then we stop the internal timer from counting.

We then find the time it took for the internal timer to count to the value it got to by using the equation $time = \frac{frequency}{count}$since we know the frequency of the internal timer. That time is the time that it took the offboard counter to count to 512 so using the same equation we can find the frequency needed to count to 512 based on the time we found. This frequency is the frequency of the input signal.

The pseudo-code for the frequency measurement is shown below:

```
float measureFrequency() {
   Prepare the counter to take in the frequency input.
   Start timer
   Enable the counter to start counting
   while (8th bit of counter is still 0);
   Stop timer
   return 512 / time
}
```

Period:

To measure the period of the input signal we used two internal timers, Timer0 with a prescaler 1:64 and Timer1 which it's clock is driven by the input signal using the RC0/T1CKI pin. When we want to measure the period, we reset both timers to zero and turn on Timer1. Once Timer1 is not equal to zero meaning that it received a rising edge from the input signal, we start Timer0 and stop Timer0 when Timer1 is not equal to 1 meaning that it received another rising edge. Since we know the frequency of Timer0 and also the value it counted to, we can find the time between the two rising edges by using the equation $time = \frac{frequency}{count}$. This time is the period of the input signal.

Below is the pseudo-code for the function:

```
float measurePeriod() {
   Reset timers 0 and 1.
   while (TMR1L == 0);
   T0CONbits.TMR0ON = 1;
   while (TMR1L == 1);
   T0CONbits.TMR0ON = 0;
   T1CONbits.TMR1ON = 0;
   int lBits = TMR0L;
```

```
    int hBits = TMR0H;
    unsigned int time = (hBits << 8) + lBits;
    float ret = ((float)time / (float)39062.5);
    return ret;
}
```

Interval:

To measure the interval of the input signal we used a internal ADC reading the voltage value of the input signal and a internal timer Timer0 with no prescaler. When we want to measure the interval we first wait for the ADC to read a value greater than 1000 (meaning a logical high in the signal) and then wait for the ADC to read a value less than 1000 (meaning a logical low in the signal). We do this to make sure we start our Timer0 right after a falling edge of the signal. Then we start Timer0 and wait for the ADC to read a value greater than 1000 and then for the ADC to read a value less than 1000 to get one interval of the signal. We then stop our timer and return the value the Timer0 counted to as the interval of the input signal.

The pseudo-code for the interval measurement function is shown below:

```
unsigned int measureInterval() {
    while (signal is 0);
    Start timer
    while (signal is high);
    while (signal is 0);
    StopTimer();
    Return time;
}
```

Spectrum:

The spectrum is measured using Brent's FFT code and by following the instructions from Bin Wang's ffthelper document. However, we noticed that just following these instructions was not enough to get accurate results. So, we had to use a timer that counts in much smaller units to get more accurate results. We had a sampling rate of 4000 Hz.

What our code did was that it took 256 samples from the ADC is equally spaced intervals. Then, we plugged in the measured samples and the imaginary array to Brent's function and received an index. We then used Bin Wang's equations to get a rough frequency. We then multiplied that value by a percentage to get a more accurate frequency result.

The pseudo code for the spectrum measurement is shown below:

```
float measureSpectrum() {
    for (int i = 0; i < 256; i++) {
      realArr[i] = (ReadADC(0) / 1023.0) * 5 - 2.5;
      imgArr[i] = 0;
      __delay_us(250);
```

```
    }

    f = Do Bin Wang's formula
    return f * error percentage;
}
```

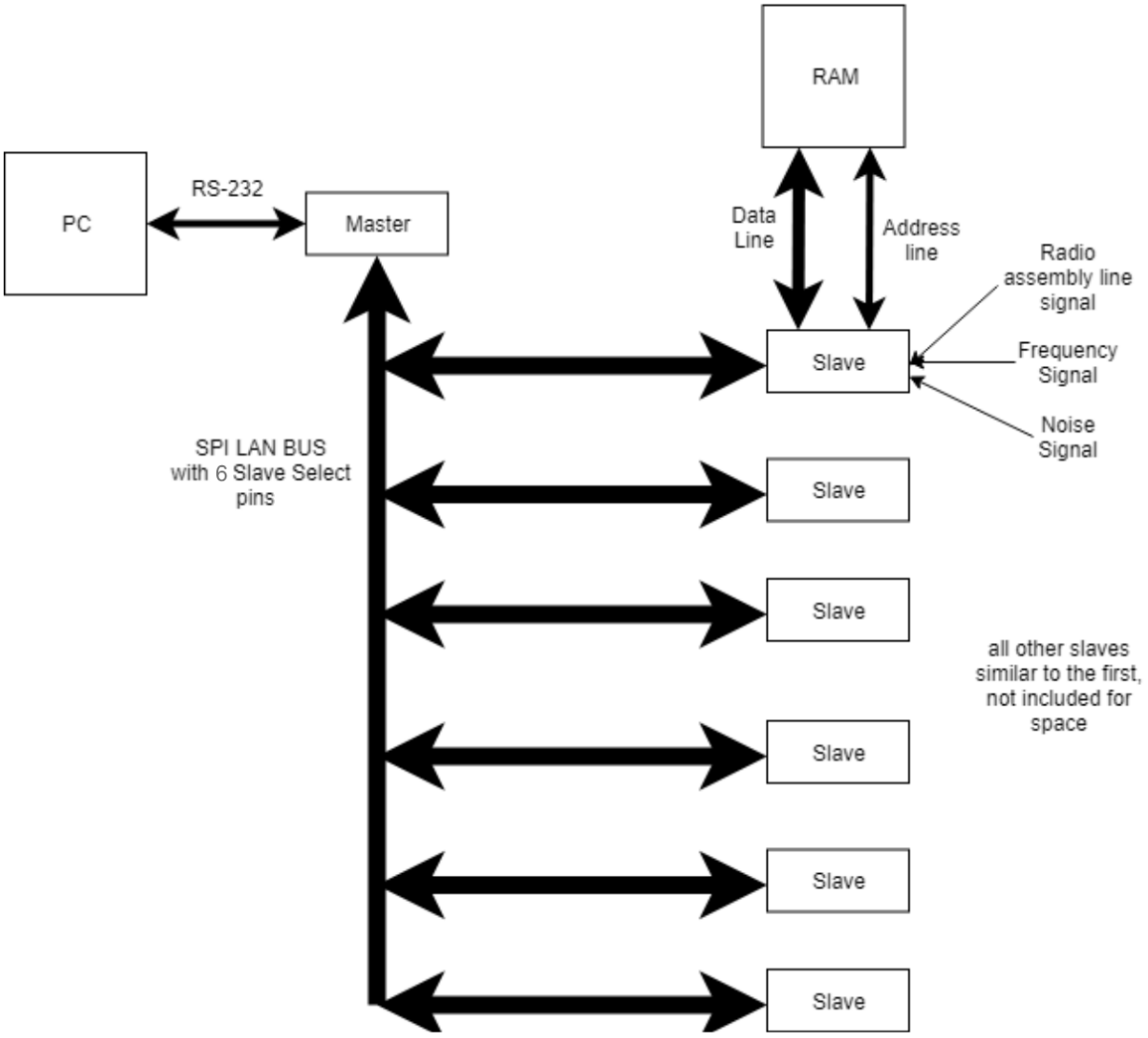
Events:

To measure the events we connected the event input to pin RB1/INT1 which is a interrupt pin. This means that for every rising edge of the event input, the Slave can interrupt and increase the internal count of how many events it has seen. When we want to measure how many events we have seen so far we can just return the event variable. This works for event inputs that are active high and active low since both only produce a rising edge once during one event happening. The pseudo-code for the function is shown below:

```
void __interrupt(void) eventsISR() {
        if (external interrupt flag is set to 1) {
                events++;
                Set external interrupt flag to 0;
        }
}

int getEvents() {
        return events;
}
```

HARDWARE IMPLEMENTATION:



Full System Block Diagram (Figure 9)

RS-232 / UART:

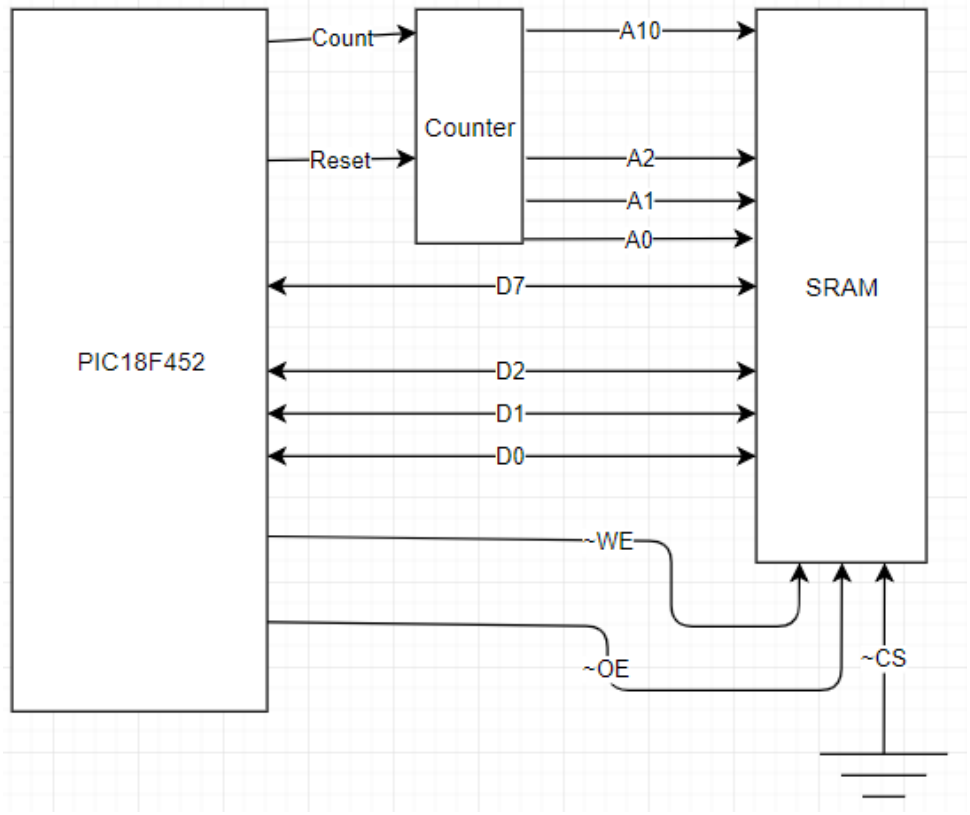
We used a RS-232 to TTL adapter to connect the USB to RS-232 cable to our breadboard. From the RS-232 to TTL adapter we connected the RX and TX pins to the master PIC TX and RX pins accordingly.

SPI:

We connected our master PIC to our slave PICs by using SPI. The SPI Clock, SPI In, and SPI Out pins are connected from the master to each slave with SPI In and SPI Out being swapped for the slave connection. For the Slave Selection for each slave, we used RB5 through RB0 pins on the master to be able to select which slave we wish to command. RB5 refers to slave A with RB0 referring to slave F.

SRAM:

The SRAM's connection is very similar to the connection from the previous lab (shown again below). However, we did not use pull-up resistors or registers. Because of how the PIC18F452 is designed, it is able to control all of the signals going into the counter and the SRAM. So, it is able to control both the counter and the enable signals of the SRAM. With this, we do not have to worry about "keeping" the values in the counter to a register. Also, this simplistic model is much easier for us to wire up than the previous model of lab 1. The only drawback of this model is that it uses up a lot of the pins of the PIC, which somewhat restricts the number of applications that can be done on the PIC. The chip select pin of the SRAM is always grounded, since we are always using this SRAM. The reason why we chose not to connect it to the PIC is so that we could have more pins freed.



SRAM Connection to Slave (Figure 10)

TEST PLAN:

For testing purposes, we needed to show that our system met the specifications outlined in the design and requirement specification sections. We needed to test the ranges of frequency, spectrum, interval, period, and events. We also needed to test the ability to communicate between the PC and master PIC and the master PIC to the slave PIC.

For each of frequency, spectrum, and period, we needed an input signal which we could control and be able to compare with our system's measurements. For events and interval, we needed a much slower signal in order to accurately test our system. For communications, we needed to test the ability of each communication separately and then as a larger system.

TEST SPECIFICATION:

Our system should be able to accurately make certain measurements within a specified bound. The measurement specifications are specified below:

- Frequency: 100 ± 0.1 Hz to 1 ± 0.001 MHz
- Period: 10 ± 0.01 ms to 1 ± 0.01 s
- Interval: 10 ± 0.01 ms to 1 ± 0.01 s
- Spectrum: 500 Hz to 1000 Hz
- Events: Up to 9999 events per hour

The outliers and middle values of the ranges will have to be tested to ensure that the measurements are made correctly.

The UART communication between the Master unit and the PC should be functioning correctly. That means that characters sent over from the Master unit should be seen by the serial monitor of the PC. The reverse should also withhold. The UART communication should be bidirectional.

The SRAM should be able to accurate keep track of the last 16 measurements. To do this, the SRAM must accurately access and write/read.

The SPI communication between the master and slave must be bidirectional, like the UART. Bytes sent between the two units must not be corrupt. In the context of this project, the master should be able to send instructions to the slave to execute. Also, whenever the master requests a read, the slave should be able to send over the last 16 measurements it has made over SPI, without any loss of accuracy.

TEST CASES:

To test the events, we used a wire which we set to high and low for each radio that passed the sensor. We tested the RS-232 and SPI communication by printing on the PC or in the case of PIC to PIC communication, turning on an LED when a specific character was input. For the remaining frequencies we used a function generator to simulate different signals across the required range of frequencies and periods.

For frequency measurements, we needed to try the following testing signals:

100Hz to 1MHz with a 1, 2, 5 pattern of testing (100Hz, 200Hz, 500Hz, 1KHz, etc.)

For noise spectrum measurements we needed to try the following testing signals:

500Hz to 1000Hz with 100Hz testing intervals between the two

For period measurements we needed signals with the following periods:

10ms - 1s testing 10ms, then 100 ms, followed by 200ms, up until 1s

For interval measurements, we needed the following interval:

10ms - 1s testing 10ms, then 100 ms, followed by 200ms, up until 1s

For units per hour, we needed to test the units with a very slow signal, so we used a wire which we toggled high and low in order to test the units: **< 60 units**

For communication testing, we needed to be able to both send and receive comprehensible characters between the PC and master, as well as between the master and slave. For PC to master communication we tested by sending a character over RS-232 to the master which would send a different character back to the PC. This meant that we could send commands to the master and also receive data from the master. For master to slave communication we tested by sending a character over SPI, and then sending "dummy data" to the slave to drive the SPI Clock for the slave to return a array of characters which then we sent over RS-232 to the PC. This meant that we could send a command to the master and the slave would be able to perform the command and also return data to the master to send to the PC. To test our slave selection we tried connecting the slave selection pin of the slave to our different slave selection pins on the master and see if we could still control the slave. This meant that we could have multiple slaves connected to the master while being able to select which one we want to command.

RESULTS:

Frequency results: our system an accurately measure the frequency up to 800 KHz with < %10 error. The higher ranges up to 1MHz are more error prone with an error of up to %20.
Interval Results: our system can accurately measure the Interval from 10ms to 1s with < %10 error.
Period Results: our system can accurately measure the Period from 10ms to 1s with < %10 error.
Spectrum Results: our system an accurately measure the Spectrum noise from 500Hz to 1000Hz with < %10 error.
Event Results: our system can accurately measure events with controlled impulses.

ERRORS:

Errors in our testing showed that the higher ranges of frequency are more difficult to measure accurately, we got a large error on the very high frequencies such as 1 MHz with an error of up to 20% the desired value. When testing events, we would sometimes trigger more events than we intended which led to larger error margins. The rest of the measurements had error associated with their values, but these were within the specification requirements and as such were not considered serious errors that require immediate fixing.

SOLUTION/EXPLANATION OF ERRORS:

Event testing errors were likely the result of the test case which was used. When toggling a wire high or low, debouncing can easily occur, especially when moving the wire in and out of a breadboard. The error in our results was likely due to this fact and can be fixed by changing the test case to use a software defined signal which would be much more accurate.

Frequency errors were likely due to many factors which are difficult to fix without a complete redesign of our system. One of the key reasons why we believe the frequency measurement was inaccurate when reaching higher frequencies is a rounding error when dividing using the PIC. Another error likely came from the method that we were using to measure the frequency. (this is described in detail above) We used a counter to increment at the rate of the frequency, once the counter reached a certain value, we divided it by the time it took. At high frequencies, this would sometimes occur too fast to get a readable time. To fix this, we could count to a higher value for the higher frequency ranges.

The communication was fairly accurate, allowing the measurements to be transferred from the slave, to the master, and finally to the PC.The remaining measurements were fairly accurate and would only need minor adjustments to lower the error.

SUMMARY:

The final design of the system matches the specifications that were requested in the initial design and requirement specifications. It also has a logical user interface and easy control of measurements and data collection. The final design consists of a PC console which communicates with the master. The master can command up to 6 slave modules, each of which can store up to 16 of its measurements and return that data to the master. The slaves have the capability to measure frequency, intervals, period, noise spectrum, and count events as they occur.

CONCLUSION:

The final system and its design match the specifications given for this product. It is able to measure and communicate to the computer as specified. However, as with all systems, there are improvements which could potentially be made in order to improve the effectiveness of the design. Some of these changes include changing designs such as where components are connected and how, using components in different ways to achieve more accurate results, and to use more precise components in general.

The design of our system could be improved in a few ways. Once change which could be made is with the location of the SRAM. At the moment, the SRAM is planned to be on each slave, however this could just as easily be used on the master. The way this could be done is that the slave takes measurements and stores small amounts of data on its main memory. When the master requests measurements from the slave, it could store them on the SDRAM for faster reading of the data in the future rather than requesting new data each time.

Component use is another aspect of the design which could be changed. Currently, our counter only counts to 512. As mentioned in the error section, this leads to inaccurate division in the higher frequency range. One way that this can be improved is if we have two settings, one for the lower range of frequencies, and one for the higher. When in the lower range, the counter counts up until 512. This has shown to be accurate for these purposes. When in the higher range however, we can use the higher bits on the counter to give the clock more time to run before the timer overflows.

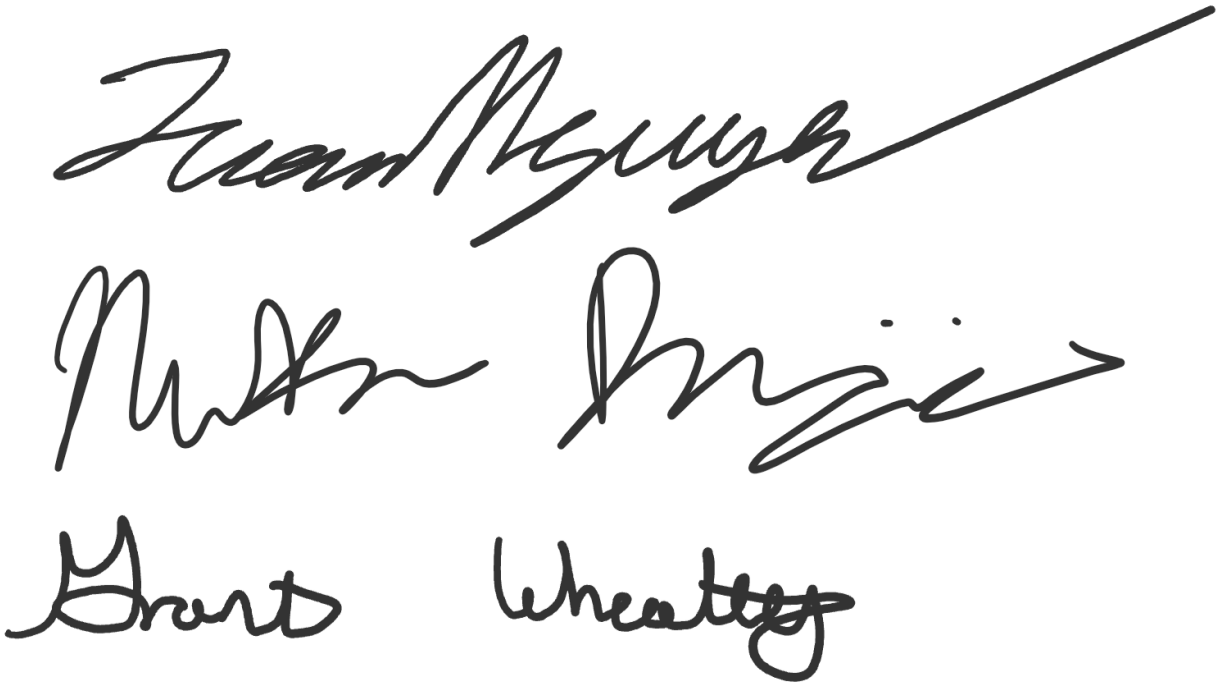
Finally as in all designs, component precision and design matters. This of course has to be balanced with cost and power consumption, but each component can be adjusted according to this balance. A deeper search on components which might be better suited for our purposes could improve performance and accuracy.

Our system design is just a prototype. As a prototype, it has proven its effectiveness and is ready to continue to the next phase of a design. The next step here would be to improve the individual parts of the design and encapsulate it into a nicer looking design for production. These steps would complete this project.

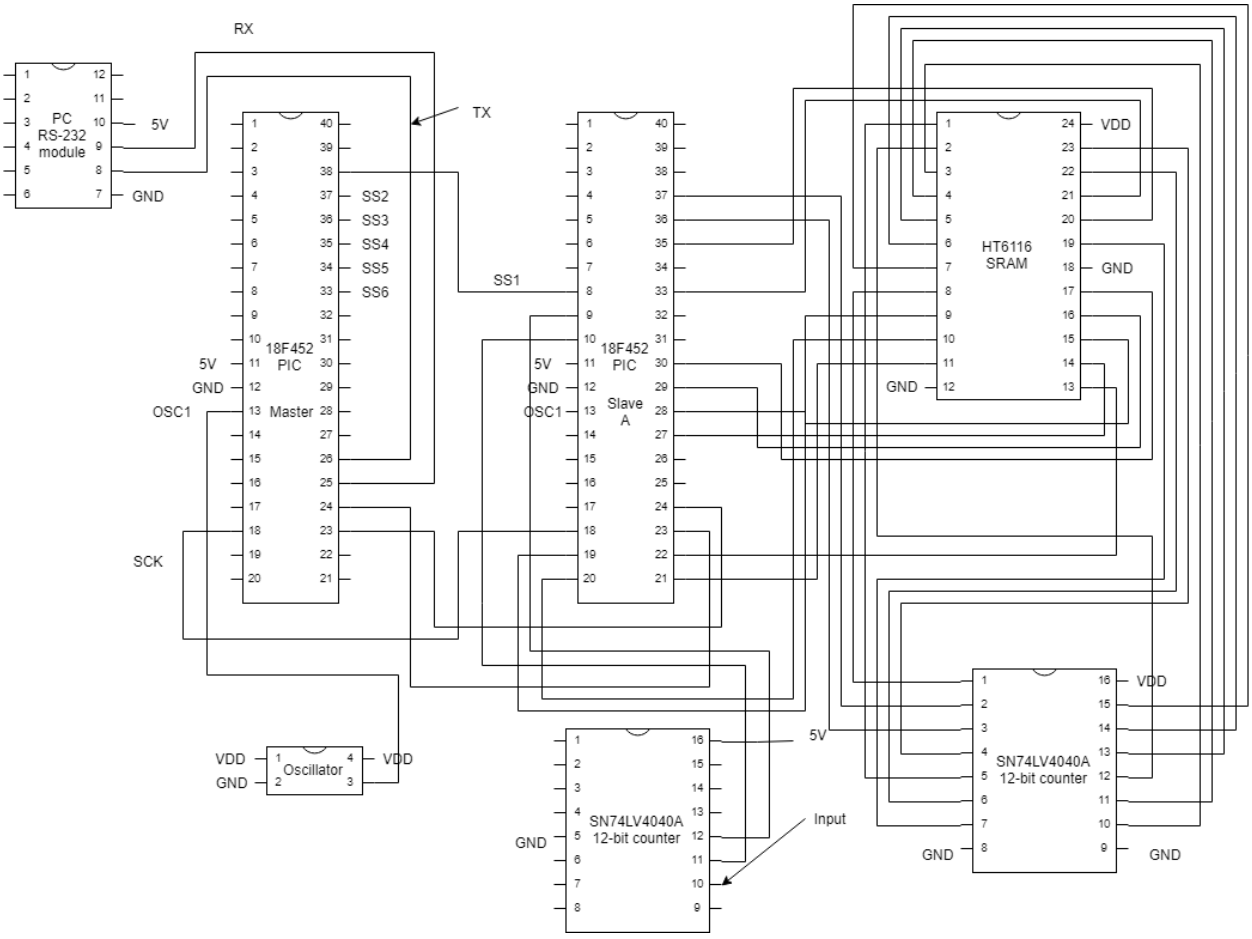
INDIVIDUAL CONTRIBUTIONS:

Group Member:	Workload:
Grant Wheatley	SPI, RS-232 connection, wiring
Nikola Dancejic	PC interface, debugging
Tuan Nguyen	SRAM, Measurements

SIGNATURES:



APPENDICES:



System Schematic (Figure 11)