**EE 478 Lab 1 Report:**

**Designing a Microprocessor Based Data Collection System**

**Lab Performed By:**

Rebecca Chu

Phongsakorn Liewsrisuk

Jian Ma

**Contribution**

(Oat) 125 hours |

(Jian) 125 hours |

(Rebecca) 125 hours|

**TABLE OF CONTENTS**

[**ABSTRACT** 3](#_Toc371192692)

[**INTRODUCTION** 3](#_Toc371192693)

[**REQUIREMENT DOCUMENTS** 3](#_Toc371192694)

[**DESIGN SPECIFICATION** 5](#_Toc371192695)

[**DESIGN PROCEDURE** 9](#_Toc371192696)

[**SYSTEM DESCRIPTION** 11](#_Toc371192697)

[**SOFTWARE IMPLEMENTATION** 24](#_Toc371192698)

[**HARDWARE IMPLEMENTATION** 28](#_Toc371192699)

[**TEST CASES** 34](#_Toc371192700)

[**TEST SPECIFICATION** 36](#_Toc371192701)

[**PRESENTATION, DISCUSSION, AND ANALYSIS OF THE RESULTS** 36](#_Toc371192702)

[**SUMMARY** 41](#_Toc371192703)

[**CONCLUSION** 41](#_Toc371192704)

# **ABSTRACT**

This lab report describes our process in designing and testing our microprocessor based data collection system. The whole design is to get have a local controller and remote controller that collect datas from sensors and stores in SRAM. The controllers communicate through I2C and with client through RS232. The The incorporates ripple counter, gal chips, SRAM, PIC, external crystal oscillator, and instrumentation amplifier to provide the needed function. The design is successful in accomplishing the I2C communication and responding the right requests from the PC and sending the right data. However, there are trouble with displaying the right data on hyperterminal.

# **INTRODUCTION**

For this lab, we are to design and implement a data collection system that involves varying sensors at remote locations and transmitting the results to the local collection sites upon requests from the client. To implement the design, we use two 18F25K22 PIC microprocessors to handle the data collection system. One at the local site and one to model six remote sites. The local site has four sensors that measure temperature with a thermocouple, carbon level, salinity level, and flow rate with external oscillator. The PICs are communicated through I2C. The local controller and the Master PC on the client side are communicated through RS-232 to display the returned data. The datas from the environmental sensors are stored in the SRAM accompanying the PIC. We used the Agilent Logic Analyzer, MPLABX IDE, and PICkit 3 to help us develop and debug our design.

.

# **REQUIREMENT DOCUMENTS**

**System Description**

This requirement describes and defines the basic requirement of the product, CheckIt-StoreIt. It will have the ability to perform many of the basic environmental measurements related to climate change. The instrument needs to be able to implement in any of the different monitoring sites around the world. It needs to be able to automatically collect data from as many of these tests as we can. Also, the products will be able to operate remotely from a single PC. Lastly, the batteries need to last as long as possible.

**Specification of External Environment**

The product need to be durable and energy efficient in order to last in the extreme climate in the different sites around the world when collection the data

**System Input and Output Specification**

**System Inputs**

The carbon and salinity sensors output 0 to 400 mV with 0.1 mV resolution

The carbon level sensor an output of 5.0 to 250.0 mV corresponds to 10.0 to 350.0

ppm

The salinity sensor, the range is 100.0 to 300.0 mV, which corresponds to 5.0 to

50.0 ppt.

The flow rate transducer produces an output frequency in the range of 1.00 KHz to 10.00

KHz that corresponds to the flow rate range of 100.0 to 1000.0 liters per second.

The J thermocouple outputs the range of -200 to 57360 microVolts

User Input: Command from RS232

**System Outputs**

Taking in different measurement inputs, from the carbon and salinity sensors, flow rate transducer and the J type thermocouple.

1. Temperature: Selectable F or C 15C -30C

2. Flow Rate: lps – 100.0 to 1000.0 liters per second.

3. Carbon Level: ppm – 10.0 to 350.0 ppm

4. Salinity Level: ppt – 5.0 to 50.0 ppt.

**User Interface**

Control and View :

Interface for Master PC to control local controller

Viewing Mode :

1. Temperature: Selectable F or C

2. Flow Rate: lps – Liters per second

3. Carbon Level: ppm – Parts per million

4. Salinity Level: ppt – Parts per thousand

Reset:

Will clear all the display as well as restart the whole system of the local controller

Power ON/OFF:

The user is able to turn on and off individual sensors.

Retrieve Data:

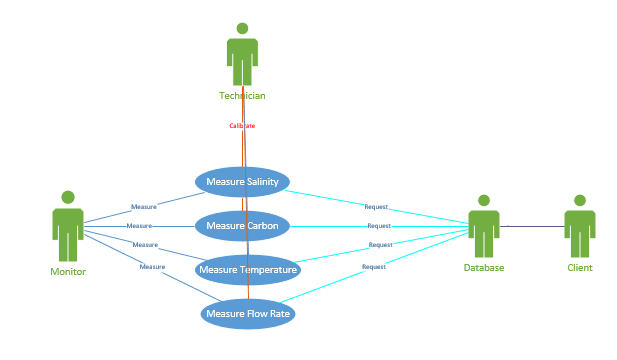
Monitoring: Live feed the data onto the PC screen

Database: The user could batch retrieve an array of data

Configure sensor:

The user could change the schedule and other parameter of the sensor remotely

**Use Cases**



**Figure 4.1 Use Case for the System**

Users send the request for the measurement data through RS232 to measure all four environmental readings (temperature, flow rate, carbon level, and salinity). These measurements are amplified with instrumentation amplifier. Therefore, the sensors can be calibrated with a technician. The controller will receive these sensor measurements (as seen as monitor above) and then send them back to client on the PC.

**Operating Specifications**

# **DESIGN SPECIFICATION**

**System Description**

This specification describes and defines the detailed design requirements for the product CheckIt-StoreIt. It will be implemented at a particular site to collect data through the sensor interfaces and drivers to obtain the environmental measurements. The system supports four measurements: Carbon level, Salinity level, temperature, and Flow rate. The collected measurements will be transferred to the local controller through bidirectional communication I2C serial channel. On top of that, the local controller at the site should be able to link to the Master PC that monitors these measurements through RS-232.

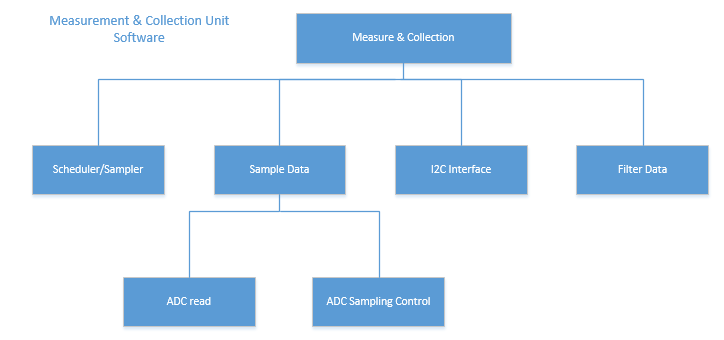
**Specification of the Environment**

The sensors for the system may be installed on varying local sites with various environmental conditions.

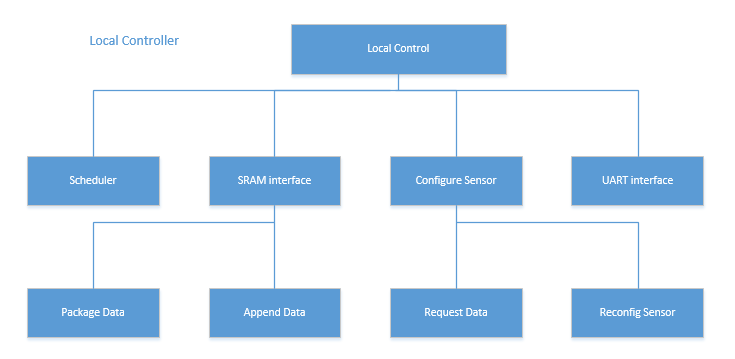
**System Functional Specification**

**FUNCTION DECOMPOSITION DIAGRAMS**

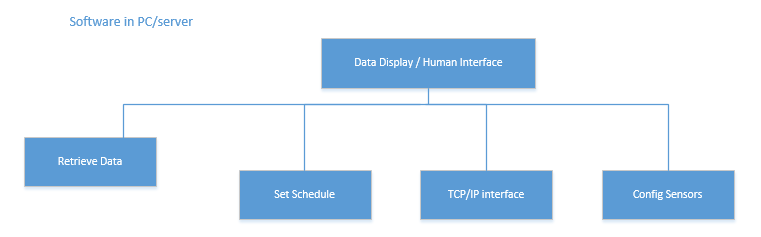
In terms of software, there are three software pieces. The firmware for measuring and collection data in the measurement and collection unit. The local controller software inside the local controller. The last one is the software/Interface in a server or PC.



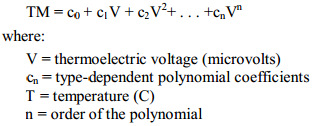
**Figure 5.1 Function Decomposition for the software in the measurement unit.**



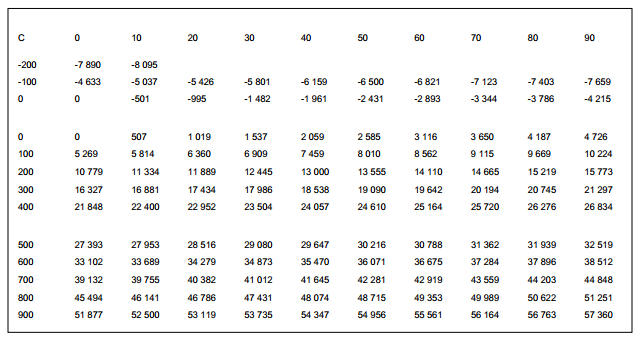
**Figure 5.2 Function Decomposition for the software in the local control unit.**

F**igure 5.3 Function Decomposition for the software in the PC/server.**

Temperature

For the input measurement like temperature, we receive the temperature from the thermocouple J and know the temperature by converting it to an equivalent temperature value in algebraic. Since the thermocouple is a non linear device, we have the following power series equation. 

The calculated voltage tells us the temperature.

This below table provides J type thermocouple’s output voltage for the specified temperature

**Figure 1.1 Table of J Thermocouple V vs T**

For the flow rate, carbon level, and salinity level, it is modeled by function generator to simulate inputs as these corresponding measurements.

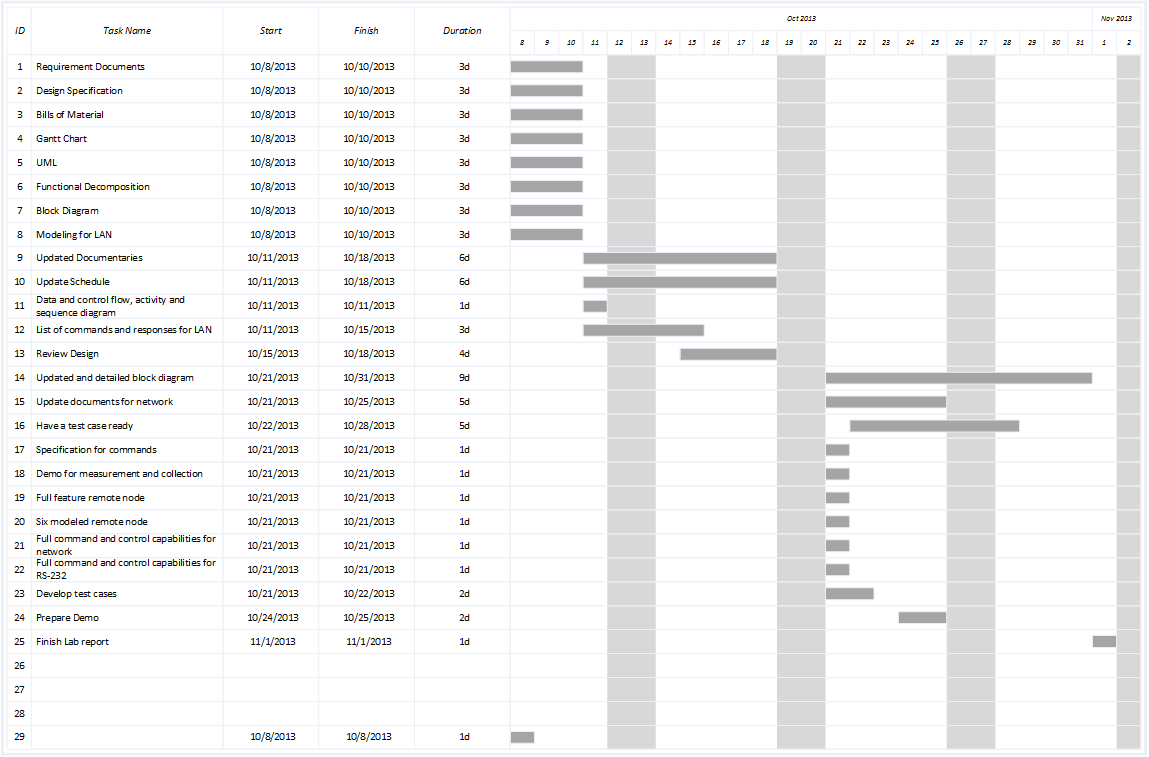
For connection between local controller and the measurement and collection system, the bidirectional communication is done through I2C protocol. The RS-232 that is built in to the PIC will be used as the connection between the local controller and the master PC. The PC will be able to send command to the instrument to make measurements. Additionally, the 16 most recent measurements will be stored in external SRAM of the local controller. From that, the PC screen will be able to retrieve the data and display it.

**BILLS OF MATERIAL**

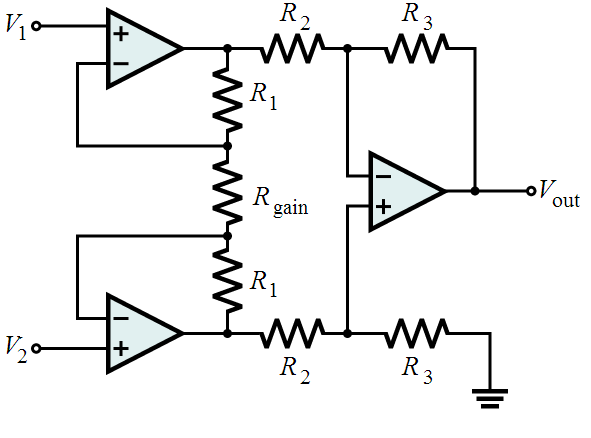


**Figure 2.1 Bill of materials for the design**

# **DESIGN PROCEDURE**



**Analog :** According to the design specification, we decided to use 3 instrumental amplifiers to amplifier the voltage to higher signals for a better resolutions. The voltage with potential meter are used to lower the voltage supply of 5 volts to a desirable range

Fig 2.0 Instrumental amplifier 

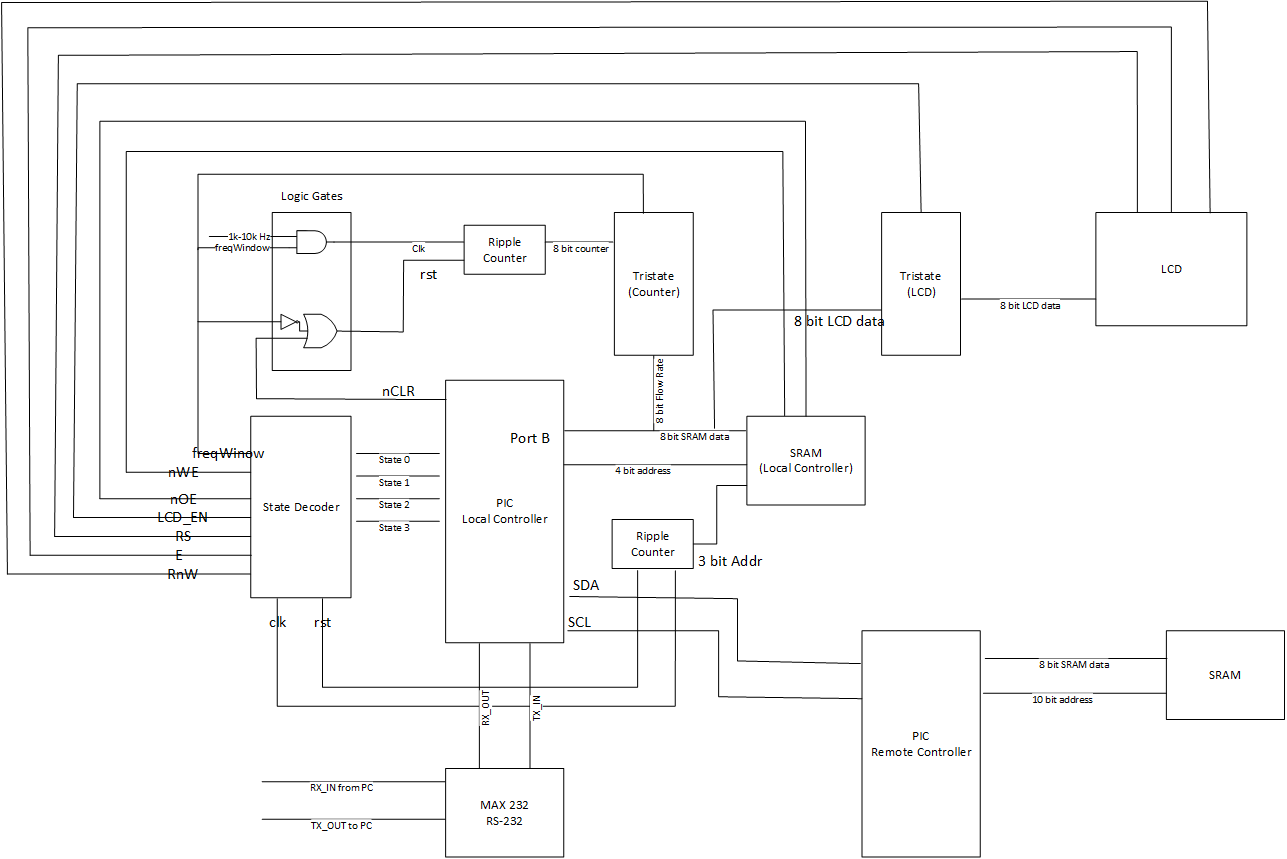
The voltage gain is calculated by the following equation.



**I2C** : I2C is implemented by using the waveform from the datasheet.

# **SYSTEM DESCRIPTION**

The system is as seen as below with each module’s input and output.



**State decoder:**

State decoder is our solution to the problem of insufficient available pins on the local controller PIC. It takes in 4 bits of states from the PIC and outputs 9 signals. Another benefit of the state decoder is that we are able to retain the state output to debug during on the logic analyzer to see which state the program is in. State decoder is programmed on the GALChip with verilog code. The output signals are different enable signals to control different modules. The signals are turned on and off depending on which state the main program in local controller PIC is in.

INPUT: State0, State1, State2, State3

OUTPUT: freqWindow, nWE, nOE, LCD\_EN, RS, E, RnW, clk, rst

The state decoder outputs the nWE and nOE signals for the read or write operation for the SRAM. There are also RS, E, and RnW for the LCD control bits. Additionally, there is the output bit freqWindow that goes into another GALChip named Logic Gates to control the clocking and reset for the ripple counter that simulates the flow rate measurement. The state decoder also provide clk and rst for another ripple counter to fulfill the 3 lower bits of the address that goes into the SRAM. For the details of the code implementation, please see the State Decoder section in Software Implementation. To see the pin outs of the GALChip, please see the state decoder section in Hardware Implementation.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **freqWindow** | **RS** | **E** | **nWE** | **nOE** | **LCD\_enable** |
|  | **States** |  | 0 | 0 | 1 | 1 | 0 |
| 0000 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0001 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0010 | 2 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0011 | 3 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0100 | 4 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0101 | 5 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0110 | 6 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0111 | 7 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1000 | 8 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1001 | 9 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1010 | 10 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1011 | 11 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1100 | 12 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1101 | 13 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1110 | 14 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1111 | 15 | 0 | 0 | 0 | 1 | 1 | 0 |

**Flow Rate Measurement (Logic Gates):**

Flow rate measurement of the system uses logic gate to control the measurements. The logic gate is used as the logic for controlling the flow rate measurement simulated by the ripple counter. The ripple counter should only count when the scheduler in the PIC requests for the flow rate measurement. Therefore, when the PIC asks for the flow rate measurement, the PIC will output the “flow rate start” state (state 2) to the state decoder. The state decoder will set the freqWindow bit to 1. Since we only want the counter to only start incrementing when the scheduler is at the flow rate measurment state. Therefore, the freqWindow bit anded with external oscillator clock allows us to have a clock window that only have counter clocked as specified time. The counter resets when either the scheduler is not asking for flow rate measurement (i.e. freqWindow is 0) or when there is master clear for the PIC. The equations for the logic gates can be seen as below:

INPUT: external oscillator frequency, freqWindow, master nCLR

OUTPUT: coutnerClk, counterRst

During the states that frequency for the flow rate measurement is not needed, the counter will not output any incrementation. The default value to the freqWindow is set to 0. Only when the scheduler calls state 2 that the counter will increment. The freqWindow is also the enable bit for the tristate allows the 8 bit result of the ripple counter to take the bus at PORTB to input into the PIC.

**Tristate**

The tristate in our system is used to control the 8 bit data bus at PORTB. Since there are very limited number of available pins on the PIC, we need to have the tristate to control the input and output coming out from and into the PIC port. We have two tristate gates in our data collection system. One of the is used for the ripple counter as mentioned in the “Flow Rate Measurement (Logic Gate) section. The other one is used for the databus for the LCD display.

INPUT: external oscillator frequency, freqWindow, master nCLR

OUTPUT: coutnerClk, counterRst

**SRAM (Local Controller)**

The SRAM at the local controller is one of the two SRAMs in our system. This SRAM stores all the measurements from the sensors. For each sensor (temperature, carbon, salinity, and , flow rate) the SRAM will store up to 16 measurements of the same sensor. For each individual reading from the sensor, the data will be 16 bits. However, due to the limitation of the output port of the PIC and the SRAM storing bits, each data is divided to lower byte and higher byte.

The address for the SRAM is provided by both the pic and the ripple counter. Since there are not enough pins to provide the 7 bit address needed to store 128 sets of data, we use an extra counter to provide the lower three bits of the SRAM data. The lower three bits of the SRAM address is provided by the ripple counter. For each measurement from 4 sensors, we need 8 spots in the SRAM. Therefore, the lower three bites of the SRAM address specifies which type of the sensor it is from. Additionally, if the last bit of the 3 bit addrL is 0, the data in SRAM is the lower byte of the sensor reading data. If the last bit of the 3 bit addrL is 1, the data in SRAM is the upper upper byte of the sensor reading data. The remaining four bits of the SRAM address is sent out from the PIC. It specifies which measurement it is. Each measurement is taken at different time. Therefore, different measurements gives us different reading at different times. Our design supports up to 16 different measurement. Therefore, the four bits are designed to tell us which measurement it is. An example of the address break down and what it stores can be seen as below:

1st Measurement

|  |  |  |
| --- | --- | --- |
| **AddrH**  **(Meas #)** | **AddrL**  **(Type)** | **Data** |
| 0000 | 000 | tempL |
| 0000 | 001 | tempH |
| 0000 | 010 | carbonL |
| 0000 | 011 | carbonH |
| 0000 | 100 | salinL |
| 0000 | 101 | salinH |
| 0000 | 110 | flowL |
| 0000 | 111 | flowH |

16th Measurement

|  |  |  |
| --- | --- | --- |
| **AddrH**  **(Meas #)** | **AddrL**  **(Type)** | **Data** |
| 1111 | 000 | tempL |
| 1111 | 001 | tempH |
| 1111 | 010 | carbonL |
| 1111 | 011 | carbonH |
| 1111 | 100 | salinL |
| 1111 | 101 | salinH |
| 1111 | 110 | flowL |
| 1111 | 111 | flowH |

For the operations of the SRAM, it needs the bits to determine whether to read or write. These bits are provided by the state decoder. When a write operation is requested, the PIC will update the “write state” (i.e. state 6). The state decoder will in turn set the nWE bit from 1 to 0. When a read operation is requested, the PIC will update the “read state” (i.e. state 7). The state decoder will in turn set the nWE bit from 1 to 0.

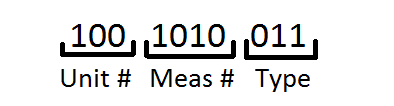
We designed the LCD display to be a banner constantly moving leftwards. Upon initialization, the microcontroller sends the following instructions to the LCD.

The entry mode is configured as shifting left, covering the entire display. The LCD is set as 8-bit mode, 1 line and font size of 11x5. The display on instruction is given as display on - cursor off - cursor position off. Display move is selected instead of cursor move and the direction is selected as shifting left. In the end the clear display instruction is executed for 2ms. When the initialization is finished, the bit stream is written to the LCD one by one. After finishing the whole sentence after each printf the LCD would be cleared and the data could be written again.

Instead of pulling data from the SRAM like the RS232 controller, the LCD controller only screenshots the current measurements once each printing is finished.

**SRAM (Remote Measurement Sites)**

The second SRAM accompanies the remote PIC. It stores the values for the remote pic that models 7 different units. Similar to the SRAM at the local controller, the address can also tell us which measurement number it is and what type of data it is. However, the address for SRAM is 10 bits. The extra 3 bits are to differentiate which unit the data originates from. The address breaks down as this:



The example as seen above tells us that it is from the 6th unit and it’s the 10th measurement of the unit. The particular data it stores is the upper byte of the carbon sensor reading data. Below shows the whole table of an example of the address and corresponding data.

Unit 1, Measurement 1 example

|  |  |  |  |
| --- | --- | --- | --- |
| **Addr**  **(Unit #)** | **Addr**  **(Meas #)** | **Addr**  **(Type)** | **Data** |
| 000 | 0000 | 000 | tempL |
| 000 | 0000 | 001 | tempH |
| 000 | 0000 | 010 | carbonL |
| 000 | 0000 | 011 | carbonH |
| 000 | 0000 | 100 | salinL |
| 000 | 0000 | 101 | salinH |
| 000 | 0000 | 110 | flowL |
| 000 | 0000 | 111 | flowH |

Since the PIC at the remote sites has a lower pin utilization. There are enough pins to support 8 bits of data and 10 bits of address. Unlike the SRAM at the local controller, we do not need an external ripple counter to support address bits.

**I2C :** By following the timing datasheet of pic18, the design is implemented accordingly.

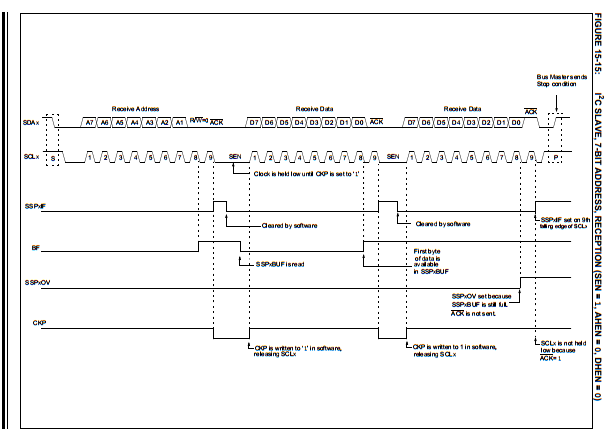


fig 2.1 Slave mode reception timing diagram

SSPIF interrupt is used after each state according to Fig2.1. The condition for each state can be determined using SSP1STAT.

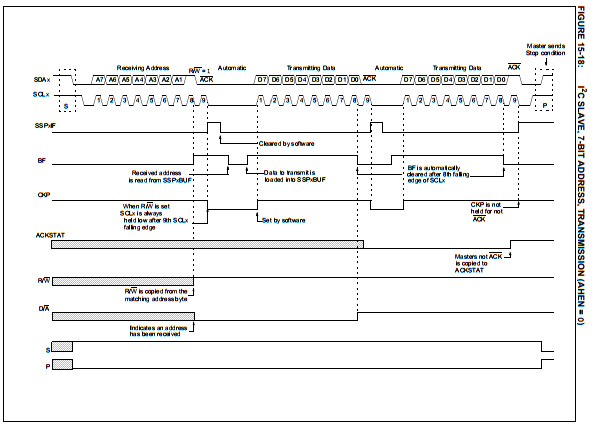


Fig 2.2 Slave mode transmission

Master Write (Slave reception)

This is the sequence for master write.

Master : START | ADDR+W | | DATA | | DATA | | ... | DATA | | STOP  
Slave : | | ACK | | ACK | | ACK | ... | | ACK |

Master Read (Slave Transmission)

However, the sequence for master read is a little different.

Master : START | ADDR+R | | | ACK | | ACK | ... | | NACK | STOP  
Slave : | | ACK | DATA | | DATA | | ... | DATA | |

Slave transmission was designed by using the with 5 different states with SSF1IF interrupt and SSP1STAT.

STATE 1: MASTER WRITE, LAST BYTE WAS

AN ADDRESS

• S = 1 (Start condition occurred last)

• R/W = 0 (Master writing data to the slave)

• D/A = 0 (Last byte was an address)

• BF = 1 (The buffer is full)

STATE 2: MASTER WRITE, LAST BYTE WAS

DATA

• S = 1 (Start condition occurred last)

• R/W = 0 (Master writing data to the slave)

• D/A = 1 (Last byte was a data byte)

• BF = 1 (The buffer is full)

STATE 3: MASTER READ, LAST BYTE WAS

AN ADDRESS

• S = 1 (Start condition occurred last)

• R/W = 1 (Master reading data from the slave)

• D/A = 0 (Last byte was an address)

STATE 4: MASTER READ, LAST BYTE WAS

DATA

• S = 1 (Start condition occurred last)

• R/W = 1 (Master reading data from the slave)

• D/A = 1 (Last byte sent was a data byte)

• BF = 0 (The buffer is empty)

STATE 5: MASTER NACK

• S = 1 (Start condition occurred last)

• D/A = 1 (Last byte sent was a data byte)

• BF = 0 (The buffer is empty)

• CKP = 1 (Clock is released)

**Command Protocol between I2C**

This protocol is sent from Master( local controller pic) to Slave (remote controller pic)

|  |  |  |  |
| --- | --- | --- | --- |
| **ID (3 bit)** | **TYPE (2bit )** | **Undefined (3bit)** | **Data (8bit)** |

|  |  |
| --- | --- |
| **secret Code (8bit)** | **secret code (8bit)** |

**ID -**1-6 (remote controller ID since there is six more remote controller)

**TYPE**

|  |  |
| --- | --- |
| 00 | temperature |
| 01 | Carbon |
| 10 | serenity |
| 11 | flow rate |

**Undefined**  is not used

**Data**  is the number of measurement 1-16

**Secret code** is generated with random function. After command is sent the local pic will decode the command and sent back the generated code. If the Code that is sent back to the master does not match, the code will not be displayed on Screen and send a fail signal to the screen

RS232 protocol

The RS232 protocol connects the system to local controller. The RS232 protocol serves as the primary user interface, user sends keyboard press commands and the RS232 protocol responds by updating the command line interface on the PC. Currently, we use hyperterminal as our user terminal, so software implementation on the client side will be omitted in this documentation.

|  |  |
| --- | --- |
| Command Byte in ascII | Meaning of the command |
| ‘r’ | Report current measurements of all types of data from one sensor. The number of samples is dependent on current settings |
| ‘a’ | Report types of data from all sensors, the number of samples is dependent on current settings |
| ‘s’ | initiate data scanning based on current setting, or stop scanning if the system is currently scanning. |
| ‘1’ | Report only the temperature from one sensor, the number of samples is dependent on current settings |
| ‘2’ | Report only the carbon level from one sensor, the number of samples is dependent on current settings |
| ‘3’ | Report only the salinity from one sensor, the number of samples is dependent on current settings |
| ‘4’ | Report only the flow rate from one sensor, the number of samples is dependent on current settings. |
| ‘t’ | Toggle the unit of temperature (metric system(C) or imperial system(F)) |
| ‘n’ | Left selection button for the number of samples. Press n to increment |
| ‘m’ | Right selection button for the number of samples, Press m to decrement |
| ‘j’ | Left selection button for selecting current device |
| ‘k’ | Right selection button for selecting current device |

The keyboard press commands are defined in the following table. When the software starts, the system send a welcome message to the PC

The uart has the following states, a RS232 receive interrupt would bring the uart system to one of the following states and each time the uart routine arrives to the program counter, the local controller will react accordingly.

**REPORT\_FLAG\_WELCOME**

**Local controller sends the welcome msg to the display and switch to wait state.**

**REPORT\_FLAG\_WAIT**

**Tell the user that the system is waiting for user inputs.**

**REPORT\_FLAG\_REPORT**

**Report data according to the global variable target ID and number of data requested.**

**REPORT\_FLAG\_IDLE**

**The system is busy preparing data. User interrupt will not be processed.**

**REPORT\_FLAG\_T**

**The same as report but the system will only output temperature data.**

**REPORT\_FLAG\_C**

**The same as report but the system will only output carbon level data.**

**REPORT\_FLAG\_S**

**The same as report but the system will only output salinity data.**

**REPORT\_FLAG\_F**

**The same as report but the system will only output flow rate data**

**REPORT\_FLAG\_SCAN**

**The system scans data to the terminal if flag\_scan is set.**

**Software module**

**master\_write\_I2C()**

**{**

**Wait until the bus is idle**

**Send START condition**

**Wait for the end of the START condition**

**Send address with R/W cleared for write**

**Wait for ACK**

**for (i = 0; i < buffer size; i++)**

**{**

**Write byte of data**

**Wait for ACK**

**}**

**Hang up, send STOP condition**

**}**

**// wait for device to reply, and read the data into RX buffer**

**master\_I2C\_read()**

**{**

**Wait until the bus is idle**

**change to receive mode**

**Wait until the bus is idle**

**Send START condition**

**Wait for the end of the START condition**

**Send address with R/W set for read**

**wait for data sent and acked**

**for(i=0; i< buffersize; i++)**

**{**

**Read first byte of data**

**send ACK**

**}**

**Read nth byte of data**

**Send NACK**

**Hang up, send STOP condition**

**}**

**I2C for PIC remote controller**

**if there is SSP interrupt**

**{**

**if last byte received is master write and last addr**

**{**

**clear buffer**

**take in new data**

**assign stuff to buffer**

**send Ack**

**make CKP high**

**} else if Master write and last data{**

**send ack**

**buffer = SSP1BUF;**

**take in new data**

**make CKP high**

**}**

**else if master read and last addr{**

**clear buffer**

**make CKP low**

**send new addr**

**make CKP high**

**}**

**else if(master read and last data){**

**clear buffer**

**make CKP low**

**send new data**

**make CKP high**

**}**

**else if(state is between master read and write){**

**clear buffer**

**make CKP low**

**send new data**

**decode the received data**

**make CKP high**

**}**

**else{**

**clear buffer**

**make CKP high**

**}**

**Clear the interrupt flag**

**}**

LCD display

The LCD display of analog s

# **SOFTWARE IMPLEMENTATION**

What is your design????

Present your design starting from a top level functional view and potentially block diagram or high level architecture. Refine that view to present and explain each of the modules that comprise the major functional blocks. Discuss the flow of control through the design. Identify and discuss the specific processes/tasks you have implemented in your design. Explain your design choices.

Pin expansion

The pics on the environmental monitoring system incorporates a pin multiplexing scheme. The microprocessor needs to receive 3 analog inputs, 2 serial lines, 2 I2C lines, 8-bit LCD data lines, 4-bit SRAM address lines, 8-bit SRAM data lines, 8-bit counter input lines. With only 28 pins, the PIC18f25k22 is not enough to supply all these pins needed. As a result, we multiplexed the 8-bit SRAM data, 8-bit counter input and the 8-bit LCD data to the PORB bus of the PIC18F25k22 and uses the state decoder to compress control signals. The control signals are indirectly controlled by the system states, or system status, which used to be only a debugging tool before.

Scheduler

The local and remote controller implements a real-time scheduler. All tasks are run through in order (round robin) each time a timer interrupt arrives. The rate at which timer interrupt occurs is the same as the sampling rate, which is by default, approximately 10 kHz. At the point of timer interrupt arrival, a global variable named current\_channel is incremented or reset to zero upon reaching a certain value(1000 by default). As a result, a full loop of tasks would be execute each 100ms. The scheduler, after resetting the timer shortly after and incrementing current\_channel, would execute a task based on the value of current\_channel.To modify the duration of the scheduling cycle, modify the value that the timer is reset to in firmware.

The timer used is timer0, which is a peripheral device of PIC18F25K22. Changing other field of SFRs (for example, PRESCALE or POSTSCALE)associated with timer0 would achieve other range of sampling/scheduling rate if modifying reset value no longer gives satisfying rates.

RS232 communication routine

The RS232 routines consists of functions for the receiver and the transmitter on PIC18F25K22. The receiver detects user keyboard inputs and send them to the environment monitoring system. The user input is set as interrupt. The local controller would report back the data it stored inside its internal memory, or from other units.

The RS232 Transmission routine prepares packets needed for the next transmission and writes them to the Transmit Shift Register (TSR) upon execution. The RS232 serial communication routines of this system is 8-bit data packets. Each timer the a user keyboard press is detected, the PC would sends the key press byte command to the local controller. The local controller would then capture and analyze the incoming command byte and switch states accordingly. The byte command table is design specification.

ADC routine

The analog to digital converting routine initiates an ADC conversion and filter the data. The conversion clock is set to be FOSC/32 to ensure accuracy and A/D acquisition time so that the charge holding capacitor discharges only when the conversion is done.

The ADC task is executed every time the timer0 interrupt arrives, i.e., at 10 kHz. For 100 such sampling cycles the ADC would measure one channel, and then move on to the next. In general, in each 100ms scheduler cycle, the ADC measures temperature for 10ms, carbon level for 10ms and salinity for 10ms.

Each 100μs when a result is obtained, it is filtered with an infinite impulse response (IIR) filter that is expressed with the following equation:

H[z] = 1/((N - (N-1)\*z^(-1))

N is expressed as . AVERAGE\_RANGE is defined in <project\_folder>/src/include/adc.h and is by default 3. Since multiplication is slow even though PIC18F25K22 comes with a decent 8x8 hardware multiplier. Multiplication is done using bit shift. As the A/D result is 10-bit, AVERAGE\_RANGE cannot go beyond 6, which will results in an overflow. So changing AVERAGE\_RANGE between 1 to 5 could modify the cut-off frequency of the digital filter if desired.

The corresponding transfer function of the IIR filter is:

Y[N] = Y[N-1] \* (N-1)/N + X[N] \* 1/N

By default N = 16(AVERAGE\_RANGE = 4) and sampling rate = 10kHz. Using the inverse of bilinear transform, H(s) = (20000+s)/(20000+31s), whose frequency response is bode-ploted below. So by default, the IIR filter has a cut-off frequency of approximately 100 Hz. As a result, the environment monitoring system could return samples of data flow accurately at a rate no greater than 200 Hz, which is the Nyquist rate of the filtered signal.



State Decoder

The state decoder uses behavioral code to assign each output signal 0 or 1 based on which state it is. The verilog code can be found in the Appendix.

Logic Gate

The logic gate uses structural code to AND and OR the inputs. The verilog code can be found in the Appendix.

Tristate

The tristate gate uses behavioral code for turnery. The enable for the tristate determines whether the databus should output the input to tristate or set high Z. The verilog code can be found in the Appendix.

LCD display

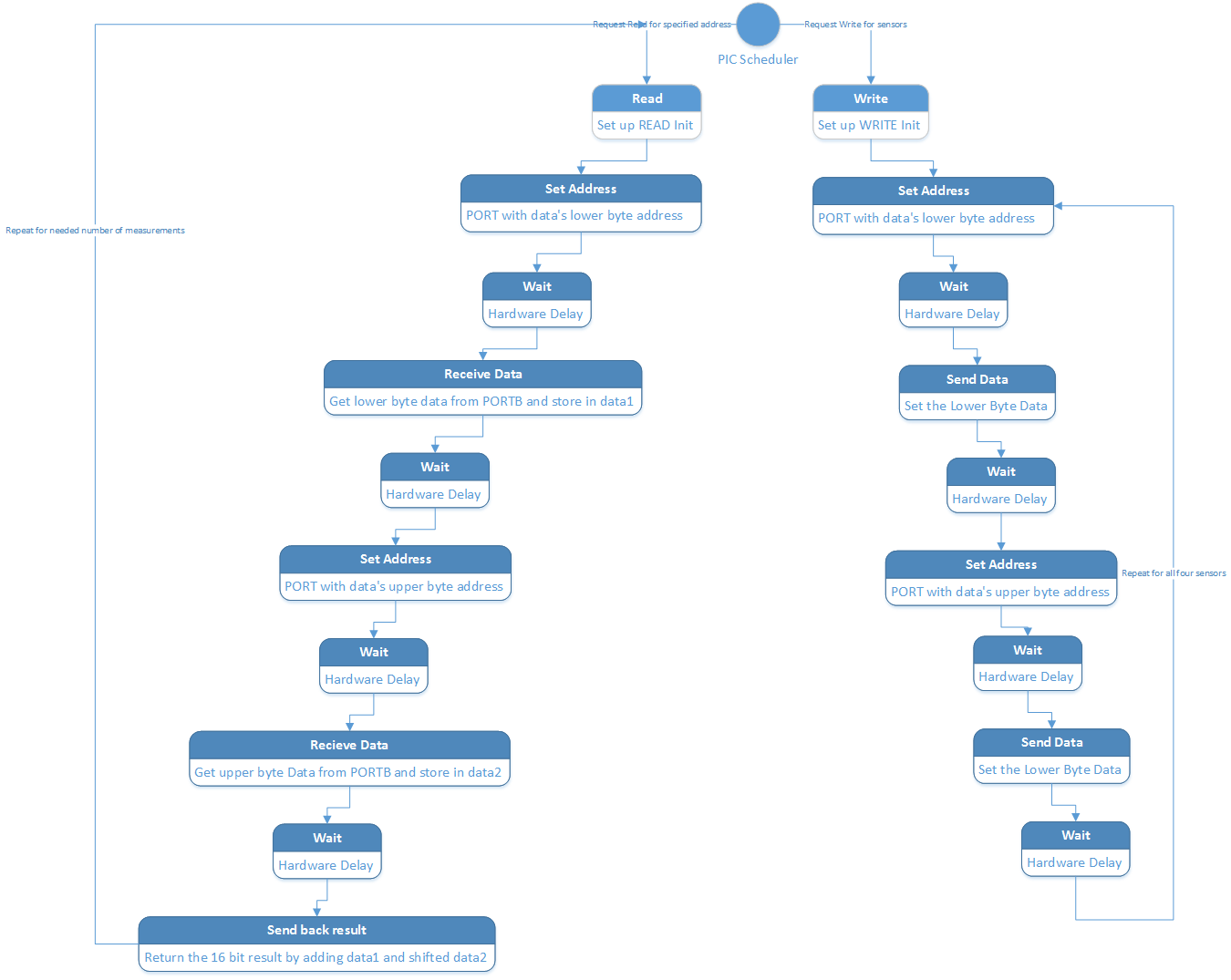
We designed the LCD display to be a banner constantly moving leftwards. Upon initialization, the microcontroller sends the following instructions to the LCD.

The entry mode is configured as shifting left, covering the entire display. The LCD is set as 8-bit mode, 1 line and font size of 11x5. The display on instruction is given as display on - cursor off - cursor position off. Display move is selected instead of cursor move and the direction is selected as shifting left. In the end the clear display instruction is executed for 2ms. When the initialization is finished, the bit stream is written to the LCD one by one. After finishing the whole sentence after each printf the LCD would be cleared and the data could be written again.

Instead of pulling data from the SRAM like the RS232 controller, the LCD controller only screenshots the current measurements once each printing is finished.

SRAM

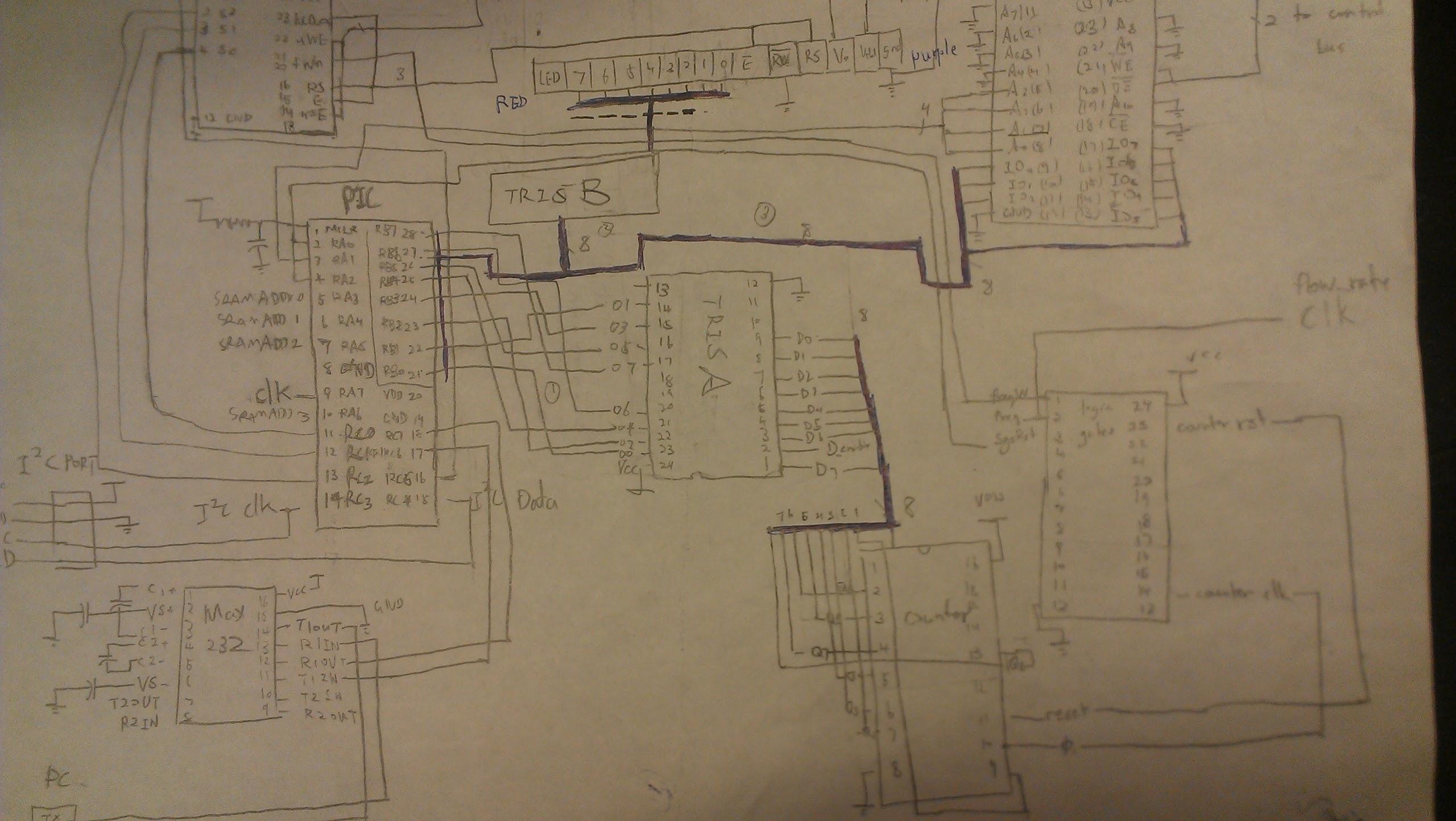
The flow of the SRAM code can be seen as the following. The SRAM operations are conducted by the PIC scheduler. The write operation is conducted every iteration after each new sets of measurements are received from the sensors. The read operation is only conducted when the client from the PC requested it. The SRAM can read from 1 to 16 measurements for each sensors. The loop for write operation will repeat 4 times, one each time for each sensor. The read operation depends on the how many measurements requested from the client. The code for the SRAM can be found under the appendix.

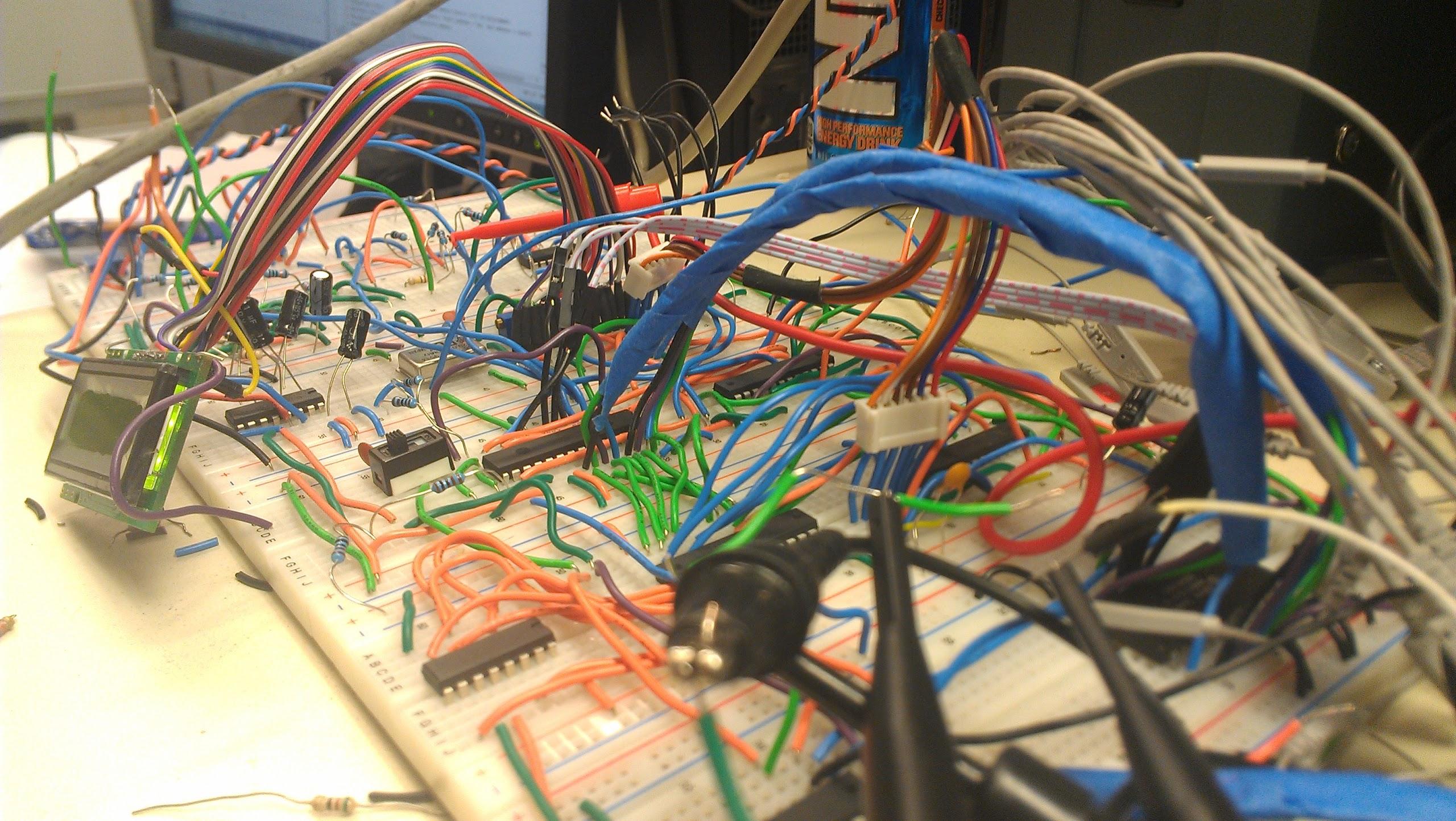


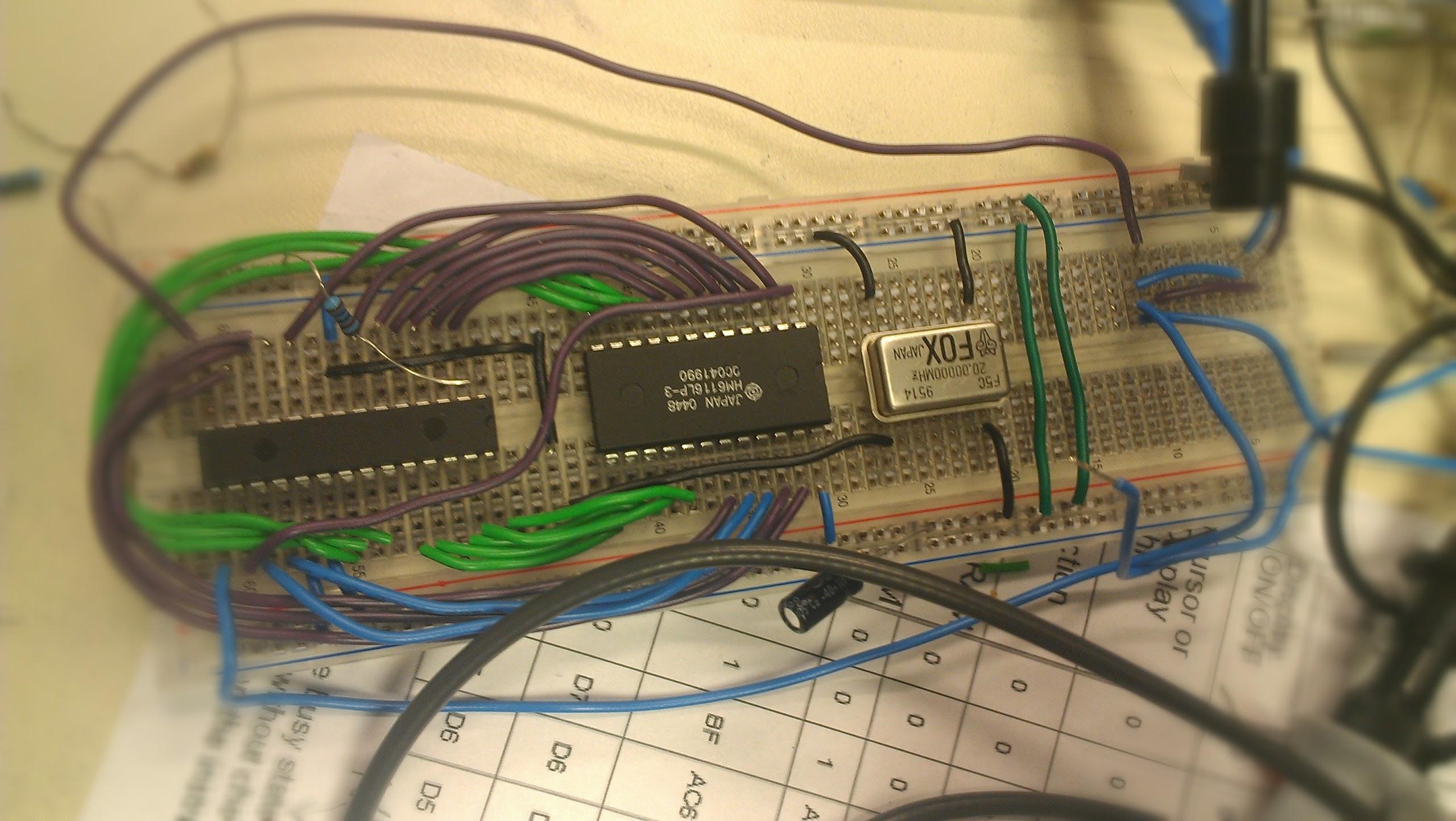
# **HARDWARE IMPLEMENTATION**

See software implementation. Now, draw the block diagrams of how you connected interconnected the major / minor modules in your system.

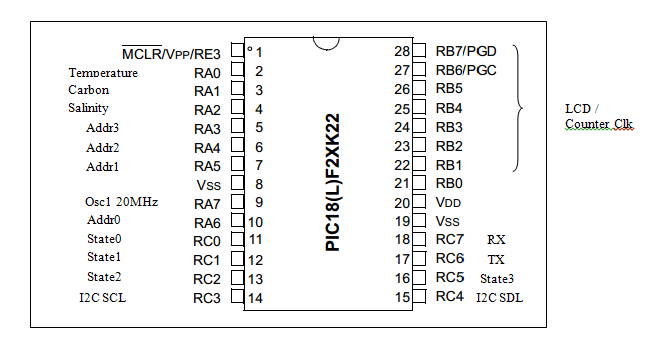
Include your logic equations as well.

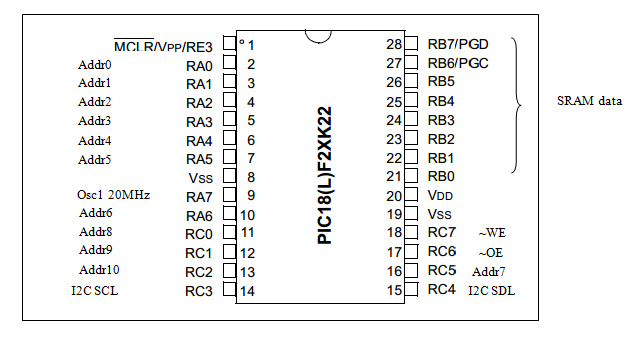






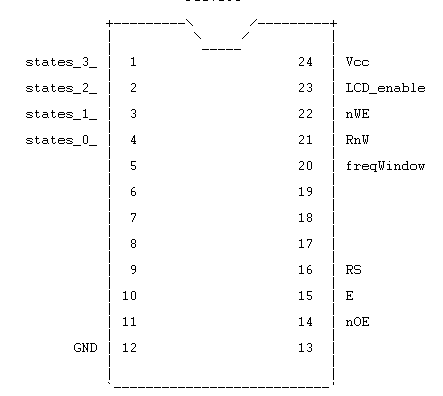
**PIC Local Controller:**



**PIC Remote Controller:**

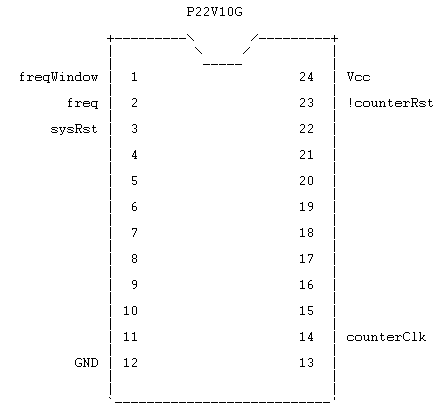
**State Decoder:**

The figure below shows the pinout for the state decoder on GALchip GAL22V10D. The chip is programed using ispLever with SuperPro. State0 to State3 are connected to the PIC at local controller RC0-RC2. The LCD\_enable, RnW, RS, and E are connected to the LCD. The freqWindow is connected for the flow rate measurement system. The nWE and nOE signals are connected to the SRAM.



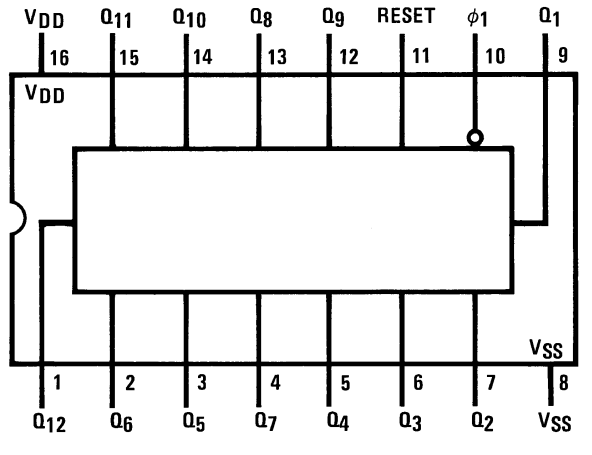
**Logic Gates:**

The figure below shows the pinout for the logic gates on GALchip GAL22V10D. The chip is programed using ispLever with SuperPro. The signals, freqWindow, freq, and sysRst, are input to the GALchip. The signal freqWindow comes from the other GALchip, state decoder. The signal for freq is from the external oscillator to generate various frequencies. The signal sysRst is from the nCLR from the PIC. The output from the GALchip are !counterRst and counterClk as seen below. Both of the signals goes the ripple counter that simulates the flow rate measurement.



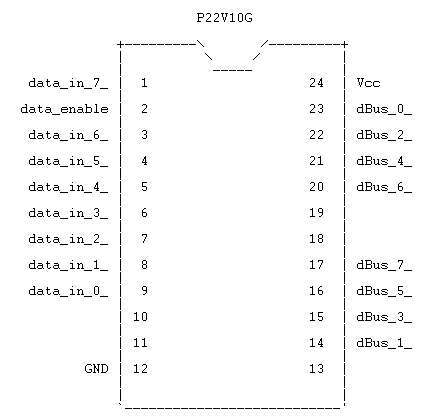
**Ripple Counter**

The ripple counter used in our system is Fairchild’s CD4040BC. It counts up to 12 bit. The output from the logicGate GALChip, counterRst and counter Clk goes to the RESET and respectively. Since we only need a resolution of 8 bits of flow rate measurement, we connected Q1 to Q8 of the counter to the tristate that when requested, will transmit to the PORTB of the PIC.



**Tristate**

The figure below shows the pinout for the tristate on GALchip GAL22V10D. The chip is programed using ispLever with SuperPro. The tristate controls the 8 bit bus for PORTB on the local controller PIC. The pins data\_in\_0 to data\_in\_7 are connected to the ripple counter for the 8 bit data. The pins dBUS\_0 to dBUS\_7 controls the output of the tristate that connects to the PIC. The pin data\_enable controls whether tristate should be input or output.

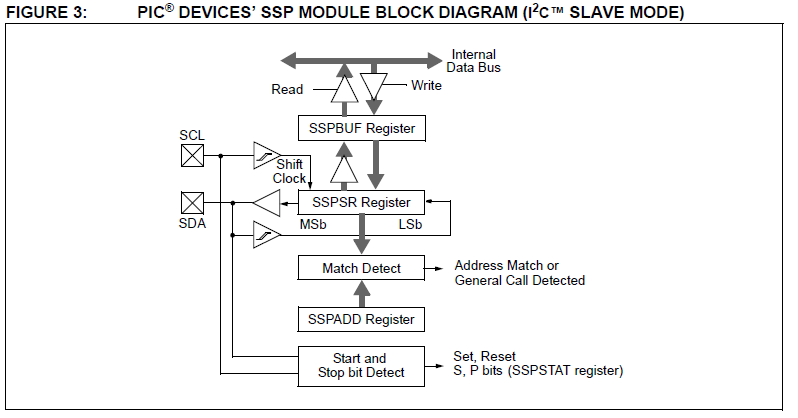


**Analogs:**

**Inputs :** The power supply of analog system is 5v.

**Output:** Amplified the voltage of 4 - 5 volts.

**I2C:**



**SSPBUF** is used to hold the data value transmitting and receiving the data.

**Test Plan**

In this project, there are a couple of major parts:

- four sensor analog inputs measurements with the thermocouple for temperature , external oscillator simulated flow rate and voltage simulated carbon level and salinity. Each sensor should have an instrumentation amplifier that amplifies the mV input to voltage of the desire range.

- RS-232 connection between the PC and the PIC for local controller being able to communicate. The PC should be able to send commands to the local controller and the MCU will send the corresponding data back

- I2C master and slave node that is able to communicate and transmit and receive measurement data

- The SRAM should be able to read and write

# **TEST CASES**

Measurement test

|  |  |  |
| --- | --- | --- |
| Amplifier test | Test description | Exit criteria |
| In-amp test | Use a small voltage input to test the in-amps and calibrate the gain for temperature, carbon level and salinity. | Given the maximum input voltage level specified in the specs. The corresponding maximum output  started to saturates to at 4.0V, which is our voltage reference. |
| Difference amp or reference offset tset | Input the lowest measurement to the amplifiers. Calibrate the potentiometer in the difference amp so that the output reaches exactly 0. | The minimum input has an output that is exactly 0V. |
| Flow rate test | Use the function generator to input frequencies to the system. Probe flow rate window, counter reset. Check the RS232 measurement output | Window covers the sampling period correctly and the measurement of frequencies is accurate. |
| LCD test | Integrate the LCD into measurement unit and check if it outputs the same data as in RS232. | The LCD output matches RS232 outputs. |

RS232 test

|  |  |  |
| --- | --- | --- |
| Test cases | Test description | Exit criteria |
| RS232 transmission | The character ‘u’is sent to the PC and the TTL and RS232 results are examined on the MSO. | The baud rate is measured as expected (19200). The world ‘u’can be seen on hyperterminal |
| RS232 reception | Implement the whole protocol for RS232. Input commands to the system and check the terminal response. | The system is able to respond the commands by echoing the data back. |

I2C test

|  |  |  |
| --- | --- | --- |
| Test cases | Test description | Exit criteria |
| Master transmission | The local controller sends to word “hello”to the I2C bus is only connected to the pull-up resistors. The SCL and SDL lines are probed with the MSO | Master is able to transmit the word “hello”on the SCL, SDL line if probed with the mixed signal oscilloscope. Due to the lack of acking from the slave devices, after each word there would a timeout delay of 300us. |
| Slave reception test | The bus is now connected to the slave. The slave is programmed to output the input char on port B on a real-time basis. | The slave is able to ack the word “hello”, acks would be set by the slave and timeout delays are eliminated. The slave is able to display to word hello on port B to the mixed signal scope. |
| Slave transmission test | The master performs a read operation after a write operation. The slave is programmed to send whatever it gets back to the SCL line. | The slave is able to send the word “hello”back to master device. |
| Master reception test | The master read the data in the read operation and display it via RS232 to the PC. | The master is able to receive the word “hello”sent from the slave device and each byte would be successfully acked |
| Data integrity test | The master sends a randomly generated secret code to the slave, the slave sends the it back to the master. The master verify the data to be the same. | The secret code matches, no error message is outputted from the system. |
| variable length transmission test | With the full protocol implemented, the master requests different lengths sensor data. | The master is able to read 0 - 16 previous data without crashing or having acking problems. |

SRAM test

|  |  |  |
| --- | --- | --- |
| Test cases | Test description | Exit criteria |
| SRAM read and write test | Probe the control signals for SRAM. Write a sequence of numbers into the SRAM and read them back | The data is correctly written and read. |
| SRAM integration | probe the bus control signals for SRAM and make sure there is no or address data conflict. | Sensor data could be read out from the SRAM and could be verified with the current measurements. |

# **TEST SPECIFICATION**

- four sensor analog inputs measurements with the thermocouple for temperature , external oscillator simulated flow rate and voltage simulated carbon level and salinity. Each sensor should have an instrumentation amplifier that amplifies the mV input to voltage of the desire range.

- RS-232 connection between the PC and the PIC for local controller being able to communicate. The PC should be able to send commands to the local controller and the MCU will send the corresponding data back

- I2C master and slave node that is able to communicate and transmit and receive measurement data

- The SRAM should be able to read and write

# **PRESENTATION, DISCUSSION, AND ANALYSIS OF THE RESULTS**

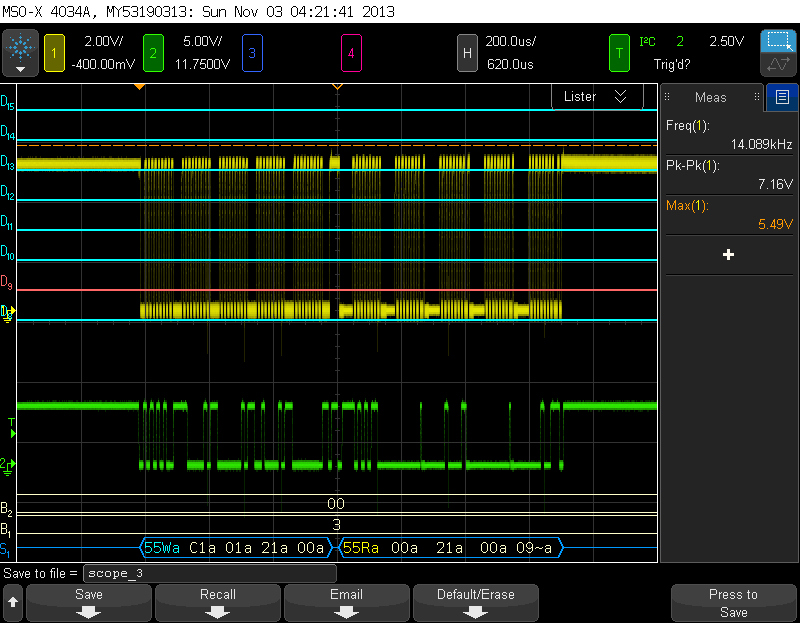


Fig 5.1 I2C result

Fig 5.1 show the result of I2C bus signal.

Master Write

0x55 The first byte represents the slave address. 0xC1 represent the ID and data Type 0x01 resent the number of measurement which is one and 0x21 , 0x00 represent the secret code.

Master Read

0x55 represent the matching slave address. Then, the slave will return the two secret code to the master. If the secret codes match the data will be displayed on the screen. 0x00 and 0x09 is the data of the measurement that is stored from the remote SRAM.

The yellow line represents I2c Clk SCL while the green line represents the I2C SDL.

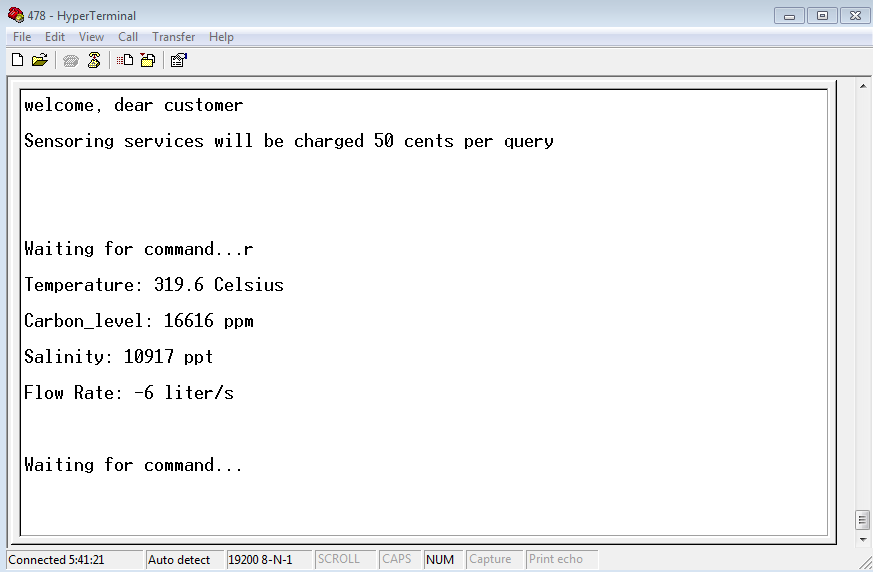


Fig 5.2 represents the result from RS232 when ‘r’ command is entered

The welcome message is shown in the beginning of the message. When command R (report current measurement) is enter, all the result from the current measurement is shown accordingly. Our result seems to be working right but due to the error in conversion from character arrays to integer, the result is displayed with error in presented results.

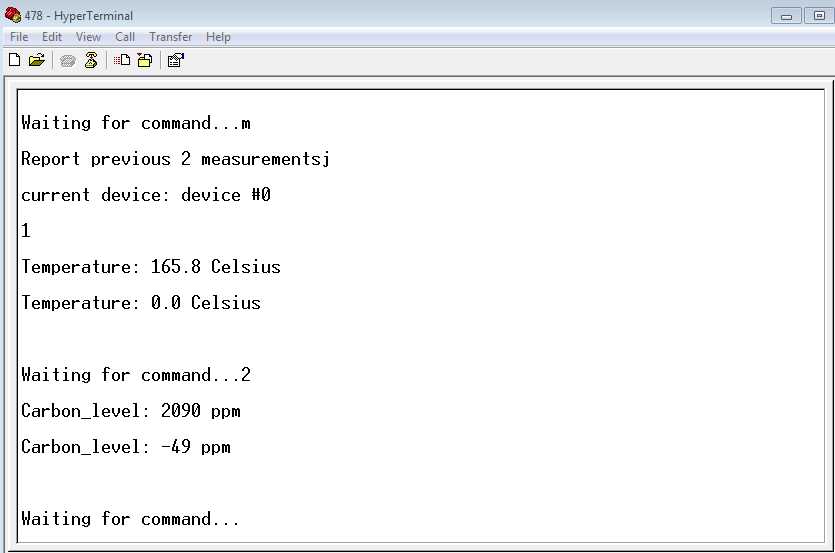


Fig 5.3 represents the command that is sent to through I2C

When command ‘m’ is entered, the number of measurement is incremented by 1 each time. When the command ‘1’ is entered, it selects the device to Temperature while ‘2’ means Carbon. For the list of command, it is under Design specification. The number of measure can go up to 16 measurement.

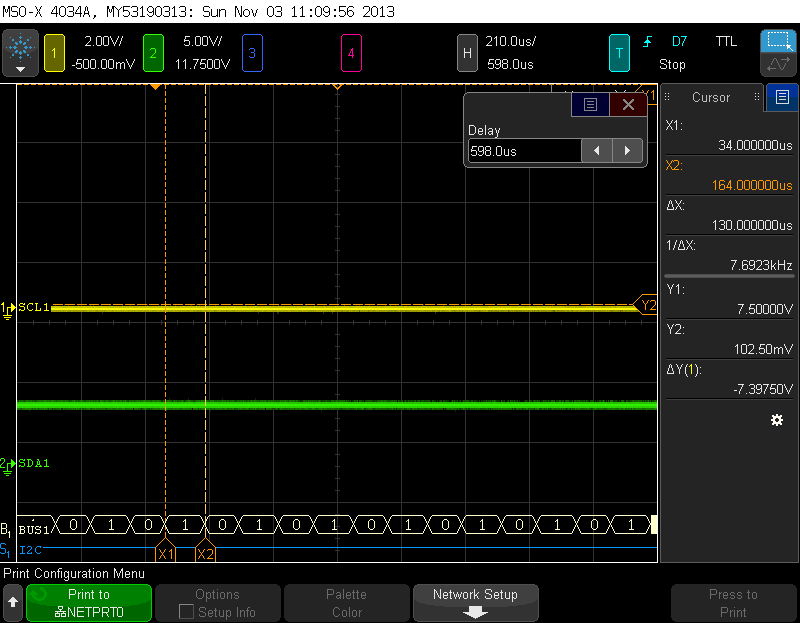


Fig 5.4 represents ADC states

Each ADC conversion is represented state1. When the conversion is done, the state goes to zero. This shows that we maximize the usage of ADC.

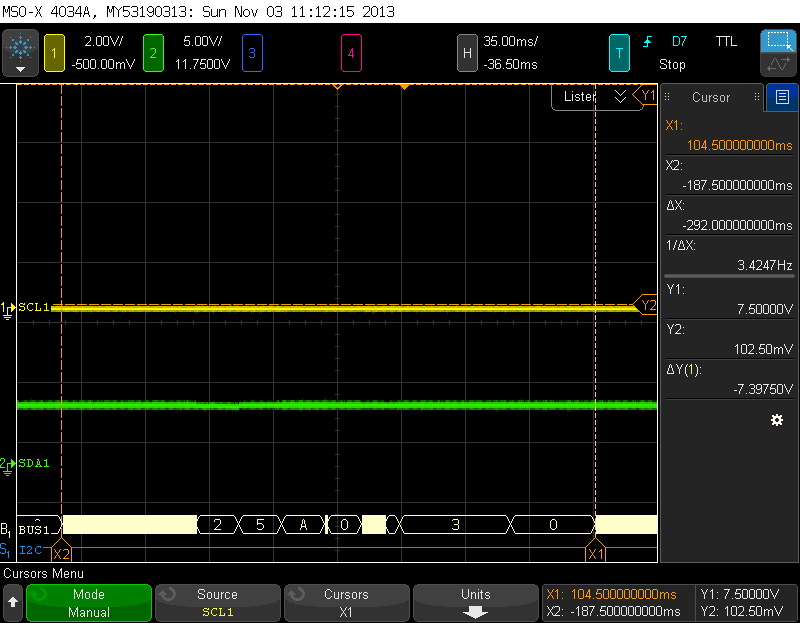


Fig 5.5 State changes for scheduler

SYSTEM\_IDLE 0x00

SYSTEM\_ADC 0x01

SYSTEM\_FREQ\_S 0x02

SYSTEM\_UART 0x03

SYSTEM\_LCD\_I\_FIRE 0x04

SYSTEM\_FREQ\_E 0x05

SYSTEM\_SRAM\_W 0x06

SYSTEM\_SRAM\_R 0x07

SYSTEM\_LCD\_INSTR 0x09

SYSTEM\_LCD\_DATA 0x0A

SYSTEM\_LCD\_D\_FIRE 0x0B

SYSTEM\_I2C 0x08

According to the Fig 5.5, the state changes from Frequency Start to Frequncy Ends. Then, it goes to LCD data and system Idle. The white part represents the ADC state and IDLE states.

Our power consumption is roughly 1 watts.

Analysis of any errors

Due to the error in the conversion from character Array to integer, we are not able to get the right display of the measurement from the RS 232. This is due to the complication of assignment of pointers and our UART display system which causes longer time than we expected. We have tried our best to investigate the root of the problem by debugging the result through the real time measurement which proves to be correct.

# **SUMMARY**

Overall, the general system meets the requirements from the client. The debugging process was implemented with the use of the Mixed Signal Oscilloscope logic analyzer, the MPLABX IDE, and PICKIT 3. The design contains the following parts to form the entire data collection system: ripple counters, gal chips, SRAMs, PICs, external crystal oscillators, max232, and instrumentation amplifiers. The entire project is completed two phases. The first phase sets up the four sensors with analog circuit to amplify the result. Additionally, the PIC of local controller is also set up to take in the readings from the sensors and displayed on hyperTerminal on the PC through the successful implementation of RS-232. In the second phase, the focus is on setting on PICs to store and read results from the SRAM and also setting up the remote controller and communicate the two systems with I2C.

# **CONCLUSION**

In conclusion, the second lab shows our ability to design, implement, and troubleshoot our data collection system. Although our microprocessor based data collection system does not have every detail entirely functioning as desired, we still have most of the system requirements. We have the I2C communicating between the local controller and remote controller. We have the SRAM reading and writing as expected. The RS-232 is transmitting and receiving as expected. However, due to the error in formatting the results, we are unable to display the results correctly. Overall, this lab report details our design in both software and hardware as well as the integration of both of these parts.

**APPENDIX**

Local Controller  
#include "adc.h"

unsigned int prev\_result\_T;

unsigned int prev\_result\_C;

unsigned int prev\_result\_S;

unsigned int prev\_result\_F;

//Initial set up for the ADC module

void adc\_bus\_config(void) {

TRISB = 0x00;

PORTB = 0b01010101;

}

void adc\_config(void) {

// Configure AN2 as an analog channel

TRISAbits.TRISA0=1;

ANSELAbits.ANSA0=1;

// ADCON2

ADCON2bits.ADFM=1; // Results format 1= Right justified

ADCON2bits.ACQT=1; // Acquisition time 5 = 12TAD

ADCON2bits.ADCS=2; // Clock conversion bits 6= FOSC/64 2=FOSC/3

ADCON0 = 0b00000001;

//ADCON1 = 0b00001000;

ADCON1= 0b00001111;

ADCON0bits.ADON = 1;

VREFCON0 = 0b10010000;

//initialize measurement;

curr\_result\_T = 0;

curr\_result\_C = 0;

curr\_result\_S = 0;

}

int adc(int channel){

//adc\_channelSel(channel);

adc\_channelSel(channel);

if (channel == 0){

curr\_result\_T = adc\_read();

curr\_result\_T = IIR\_filter(prev\_result\_T, curr\_result\_T);

prev\_result\_T = curr\_result\_T;

}else if (channel == 1 ){

curr\_result\_C = adc\_read();

curr\_result\_C = IIR\_filter(prev\_result\_C, curr\_result\_C);

prev\_result\_C = curr\_result\_C;

}else if (channel == 2 ){

curr\_result\_S = adc\_read();

curr\_result\_S = IIR\_filter(prev\_result\_S, curr\_result\_S);

prev\_result\_S = curr\_result\_S;

}else if (channel == 3){

curr\_result\_F = adc\_read();

curr\_result\_F = IIR\_filter(prev\_result\_F, curr\_result\_F);

prev\_result\_F = curr\_result\_F;

}

return 1;

}

// Begin ADC conversion with the specified channel

void adc\_channelSel(int channel){

ADCON0bits.CHS = (channel == 1) ? 0:

(channel == 2) ? 1 :

(channel == 3) ? 2:

0b11111;

}

// Read the converted 10 bits ADC values from the registers

unsigned int adc\_read(void){

unsigned int lower;

unsigned int upper;

if (ADCON0bits.GO\_NOT\_DONE) {

error = 1;

} else {

upper = ADRESH;

lower = ADRESL;

ADCON0bits.GO\_NOT\_DONE = 1;

}

return upper << 8 | lower;

}

// 1st order IIR LPF for channel

unsigned int IIR\_filter(unsigned int past\_result, unsigned int new\_result)

{

// result = (past\*(N-1)/N + new/N)

return (((past\_result << AVERAGING\_RANGE) - past\_result) + new\_result) >> AVERAGING\_RANGE;

}

Freq Measurement

#include "freq\_meas.h"

// NOTES: bus configuration has to be compatible with

// ADC as they these two tasks are performaned in parallel.

// enable the correct data path for freq measurement

void frequency\_bus\_config\_S(void)

{

TRISB = 0x00;

PORTB = 0x55;

//TRISAbits.RA0 = 1; // input for thermometer

//TRISAbits.RA1 = 1; // input for carbon

//TRISAbits.RA2 = 1; // input for salinity

//ANSELAbits.ANSA0 = 1; //Analog input

//ANSELAbits.ANSA1 = 1; //Analog input

//ANSELAbits.ANSA2 = 1; //Analog input

}

void frequency\_bus\_config\_E(void)

{

TRISB = 0xFF;

//TRISAbits.RA0 = 1; // input for thermometer

//TRISAbits.RA1 = 1; // input for carbon

//TRISAbits.RA2 = 1; // input for salinity

//ANSELAbits.ANSA0 = 1; //Analog input

//ANSELAbits.ANSA1 = 1; //Analog input

//ANSELAbits.ANSA2 = 1; //Analog input

}

int freq\_meas(void)

{

int freqB;

freqB = PORTB;

//printf("%i,",freqB);

return freqB \* 4;

}

Interrupt

#include "interrupts.h"

void interrupt\_init(void)

{

// disable all interrupts except for external interrupt on INT2/RA2

// and timer0/1 interrupt

// disable interrupt priority

RCONbits.IPEN = 1;

INTCON = 0b00100000; // timer0 overflow interrupt

INTCON2 = 0b11000100; // Pull-ups disabled/Interrupt0 Rising edge

INTCON3 = 0b00000000; // external to zero

PIE1 = 0b00100000; // enable timer2

PIE2 = 0b00000000;

PIE3 = 0b00000000;

PIE4 = 0b00000000;

PIE5 = 0b00000000;

IPR1 = 0b00100000;

INTCONbits.GIE\_GIEH = 1; /\* Enable the interrupts \*/

INTCONbits.PEIE\_GIEL = 1; /\* Enable peripheral interrupts \*/

}

#pragma code high\_vector=0x08

void global\_interrupt(void) {

\_asm GOTO myISR \_endasm;

}

#pragma code

#pragma interrupt myISR

void myISR(void)

{

// check frame error

if (INTCONbits.T0IF) {

INTCONbits.T0IF = 0;

sampling\_flag = 1;

}

if (PIR1bits.RC1IF) {

PIR1bits.RC1IF = 0;

byte\_command = RCREG1;

byte\_command = RCREG1;

remaining\_buffer\_size--;

if (!remaining\_buffer\_size) {

RX\_buffer\_full = 1;

RCSTA1bits.CREN1 = 0;

PIE1bits.RC1IE = 0;

}

INTCONbits.GIE = 1;

}

}

Intra Com

#include "intra\_comm.h"

#include "i2c.h"

unsigned int TX\_buffer\_size;

unsigned int data\_buffer\_size;

unsigned int secret\_code;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Low-level intrasystem comm protocols

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void master\_write\_I2C(unsigned int device\_ID)

{

int i;

SSP2CON2bits.RCEN = 0;

IdleI2C(); // Wait until the bus is idle

StartI2C(); // Send START condition

IdleI2C(); // Wait for the end of the START condition

WriteI2C( device\_ID <<1 | 0x00 ); // Send address with R/W cleared for write

IdleI2C(); // Wait for ACK

check\_ack();

for (i = 0; i < TX\_buffer\_size; i++)

{

WriteI2C(I2C\_TX\_buffer\_ptr[i]); // Write first byte of data

IdleI2C(); // Wait for ACK

check\_ack();

}

StopI2C(); // Hang up, send STOP condition

}

// wait for device to reply, and read the data into RX buffer

void master\_I2C\_read(unsigned int device\_ID)

{

int i;

char a;

IdleI2C();

//receive mode

SSP2CON2bits.RCEN = 1;

IdleI2C(); // Wait until the bus is idle

StartI2C(); // Send START condition

IdleI2C(); // Wait for the end of the START condition

WriteI2C( device\_ID << 1 | 0x01 ); // Send address with R/W set for read

IdleI2C(); // wait for data sent and acked

for(i=0; i< data\_buffer\_size-1; i++)

{

I2C\_RX\_buffer\_ptr[i] = ReadI2C();

//a = ReadI2C();

// Read first byte of data

AckI2C(); //send ACK

}

I2C\_RX\_buffer\_ptr[data\_buffer\_size-1] = ReadI2C(); // Read nth byte of data

//a = ReadI2C();

NotAckI2C(); // Send NACK

StopI2C(); // Hang up, send STOP condition

}

void I2C\_master\_init(void)

{

//FOSC = 20MHz, I2C clock = 100kHz

SSP1ADDbits.SSP1ADD = 0x32;

//slew rate control desabled for 100kHz

SSP1STATbits.SMP = 0;

// disable SMbus input level

SSP1STATbits.CKE = 0;

// set RC4 and RC3 as open drain I2C ports

SSP1CON1bits.SSPEN = 1;

// I2C master mode, clock = FOSC/(4\*(SSP1ADD+1))

SSP1CON1bits.SSPM = 0b1000;

// enable receive mode for I2C master

SSP1MSK = 0xFF;

// enable general call

SSP1CON2bits.GCEN = 1;

// transmit mode

SSP2CON2bits.RCEN = 0;

// disable data/addr hold

SSP1CON3bits.AHEN = 0;

SSP1CON3bits.DHEN = 0;

// enable start/stop condition interrupt enable

SSP1CON3bits.PCIE = 1;

SSP1CON3bits.SCIE = 1;

// 100ns hold time

SSP1CON3bits.SDAHT = 0;

// disable bus collision

SSP1CON3bits.SBCDE = 0;

//OpenI2C(MASTER, SLEW\_OFF);

}

// wait until ack arrived or ack timeout

void check\_ack(void)

{

int time\_out = 0;

int ack = SSPCON2bits.ACKSTAT;

while (ack && time\_out < TIMEOUT) {

ack = SSPCON2bits.ACKSTAT;

if (sampling\_flag) {

timer\_rst();

time\_out++;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

High-level intrasystem comm protocols

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// functions for masters

void I2C\_test(char\* buffer\_ptr)

{

//send hello to device5 - simulator

sprintf(buffer\_ptr, "Hello");

TX\_buffer\_size = 5;

master\_write\_I2C(0x55);

}

// write the request msg to TX\_buffer, update TX\_buffer\_size, update RX buffer size

unsigned int I2C\_request\_data(char\* buffer\_ptr, unsigned int device\_ID,

unsigned int data\_size, unsigned int type)

{

request\_msg\* new\_request;

new\_request = (request\_msg\*)buffer\_ptr;

new\_request->ID = device\_ID;

new\_request->secret\_code = secret\_code;

new\_request->type = type;

new\_request->data\_size = data\_size;

TX\_buffer\_size = 4;

data\_buffer\_size = data\_size\*2 + 2;

return 1;

}

// write a cmd msg to TX\_buffer, update TX\_buffer\_size

unsigned int I2C\_config\_sensor(char\* buffer\_ptr, unsigned int device\_ID,

unsigned int parameter, unsigned int new\_value)

{

config\_msg\* new\_command;

new\_command = (config\_msg\*)buffer\_ptr;

new\_command->PACKETSIZE = 8;

new\_command->request = 0;

new\_command->parameter = parameter;

new\_command->new\_value= new\_value;

new\_command->secret\_code = secret\_code;

TX\_buffer\_size = 6;

return 1;

}

// change a configuration parameter for all sensors

unsigned int I2C\_all\_config\_sensor(char\* buffer\_ptr, unsigned int parameter,

unsigned int new\_value)

{

config\_msg\* new\_command;

new\_command = (config\_msg\*)buffer\_ptr;

new\_command->PACKETSIZE = 8;

new\_command->request = 0;

new\_command->parameter = parameter;

new\_command->new\_value= new\_value;

new\_command->secret\_code = secret\_code;

TX\_buffer\_size = 6;

return 1;

}

// return a pointer to an unsigned interger array that stores N data

// update data\_buffer\_size

char\* master\_request\_data(unsigned int device\_ID,

unsigned int data\_size, unsigned int type)

{

// compose the command

I2C\_request\_data(I2C\_TX\_buffer\_ptr, device\_ID, data\_size, type);

// go to TX mode and send data to slave

master\_write\_I2C(0x55);

master\_I2C\_read(0x55);

//printf("%d%d%d%d%d\n\r", I2C\_RX\_buffer\_ptr[0], I2C\_RX\_buffer\_ptr[1],

//I2C\_RX\_buffer\_ptr[2], I2C\_RX\_buffer\_ptr[3], I2C\_RX\_buffer\_ptr[4]);

return I2C\_RX\_buffer\_ptr;

}

// check the first integer of RX\_buffer

// return 1 if data is valid

unsigned int verify\_data(char\* buffer)

{

unsigned int result;

//code = bytes2int(buffer[0], buffer[1]);

//printf("\n\r%d vs %d\n\r", secret\_code, buffer[1]);

//update secret code

result = (buffer[1] == (char)secret\_code) ? 1:0;

secret\_code = (secret\_code + 271) % 51;

return result;

}

LCD Controller

#include "LCD\_controller.h"

#include <delays.h>

char temp\_text[30] = "Temperature: 0C/F";

char carbon\_text[30] = "Carbon Level: 0 unit";

char salinity\_text[30] = "Temperature: 1 unit";

char flow\_text[30] = "Flow rate: 1 unit";

void LCD\_init (void)

{

LCD\_init\_flag = 0;

LCD\_str\_number = &temp\_text[0];

LCD\_char\_posi = &temp\_text[0];

LCD\_str\_size = 14;

}

int LCD\_task(void) {

LCD\_bus\_config();

if (LCD\_init\_flag < 20){

LCD\_init\_flag++;

update\_status(SYSTEM\_LCD\_INSTR);

} else {

update\_status(SYSTEM\_LCD\_DATA);

//LCD\_update\_screen();

}

// check if initialization

if (LCD\_init\_flag == 1) {

//entry mode = shift left, entire display

LCD\_instruction(ENTRY\_MODE\_SET | 0b11);

} else if (LCD\_init\_flag == 2){

//set 8-bit mode, 1 line and font size of 11x5

LCD\_instruction(CSR\_DSP\_SHIFT | 0b1000);

} else if (LCD\_init\_flag == 3) {

//display on - cursor off - cursor position off

LCD\_instruction(DISPLAY\_ON | 0b100);

} else if (LCD\_init\_flag == 4) {

//display move & shift left

LCD\_instruction(FUNCTION\_SET | 0b10100);

} else if (LCD\_init\_flag == 5) {

//clear display

LCD\_instruction(CLEAR\_DISPLAY);

}

return 1;

}

void LCD\_update\_screen(void)

{

//initialization is done, perform actual task

if (LCD\_str\_number == &temp\_text[0]) {

if (LCD\_char\_posi == LCD\_str\_number) {

LCD\_str\_size = update\_temp\_value();

}

LCD\_put\_C(\*LCD\_char\_posi++);

if (LCD\_char\_posi - LCD\_str\_number == LCD\_str\_size) {

LCD\_str\_number = &carbon\_text[0];

LCD\_str\_size = update\_temp\_value();

LCD\_char\_posi = &LCD\_str\_number[0];

}

} else if (LCD\_str\_number == &carbon\_text[0]) {

LCD\_put\_C(\*LCD\_char\_posi++);

if (LCD\_char\_posi - LCD\_str\_number == LCD\_str\_size) {

LCD\_str\_number = &salinity\_text[0];

LCD\_str\_size = update\_temp\_value();

LCD\_char\_posi = &LCD\_str\_number[0];

}

} else if (LCD\_str\_number == &salinity\_text[0]) {

LCD\_put\_C(\*LCD\_char\_posi++);

if (LCD\_char\_posi - LCD\_str\_number == LCD\_str\_size) {

LCD\_str\_number = &flow\_text[0];

LCD\_str\_size = update\_temp\_value();

LCD\_char\_posi = &LCD\_str\_number[0];

}

} else if (LCD\_str\_number == &flow\_text[0]) {

LCD\_put\_C(\*LCD\_char\_posi++);

if (LCD\_char\_posi - LCD\_str\_number == LCD\_str\_size) {

//sequence ended, init everything

LCD\_init\_flag = 0;

}

}

}

// common bus confugure for LCD operations

void LCD\_bus\_config(void)

{

TRISB = 0x00;

PORTB = 0x55;

}

// send out a LCD instruction

void LCD\_instruction(char instruction)

{

PORTB= 0x0F;

LCD\_fire();

}

// stream one bit to LCD

void LCD\_put\_C (char data)

{

//PORTB = data;

//for debugging

PORTB = 0xF0 ;

LCD\_fire();

}

void LCD\_fire(void)

{

if (LCD\_init\_flag < 20) {

update\_status(SYSTEM\_LCD\_I\_FIRE);

timer0\_delay(10);

update\_status(SYSTEM\_LCD\_INSTR);

} else {

update\_status(SYSTEM\_LCD\_D\_FIRE);

timer0\_delay(10);

update\_status(SYSTEM\_LCD\_DATA);

}

}

// Update the temp\_value\_string, return new size

int update\_temp\_value(void)

{

if (Temp\_Unit == 1) {

return sprintf(temp\_text, "Temperature: %d C", curr\_result\_T);

} else {

return sprintf(temp\_text, "Temperature: %d F", curr\_result\_T);

}

}

// Update the temp\_value\_string, return new size

int update\_carbon\_value(void)

{

return sprintf(carbon\_text, "Carbon Level: %d unit", curr\_result\_C);

}

// Update the temp\_value\_string, return new size

int update\_salinity\_value(void)

{

return sprintf(salinity\_text, "Salinity: %d unit", curr\_result\_S);

}

// Update the temp\_value\_string, return new size

int update\_flow\_value(void)

{

return sprintf(flow\_text, "Flow rate: %d unit", curr\_result\_F);

}

Remote Code

Measurement\_Unit

// EE478 Lab2

// Measurement Unit

/\* Compile options: -ml (Large code model) \*/

// PIC18F25K22 Configuration Bit Settings

// CONFIG1H

//#pragma config FOSC = ECHP

#pragma config FOSC = ECHPIO6 // Oscillator Selection bits (EC oscillator (high power, >16 MHz))

#pragma config PLLCFG = OFF // 4X PLL Enable (Oscillator used directly)

#pragma config PRICLKEN = ON // Primary clock enable bit (Primary clock enabled)

#pragma config FCMEN = OFF // Fail-Safe Clock Monitor Enable bit (Fail-Safe Clock Monitor disabled)

#pragma config IESO = OFF // Internal/External Oscillator Switch-over bit (Oscillator Switch-over mode disabled)

// CONFIG2L

#pragma config PWRTEN = OFF

// Power-up Timer Enable bit (Power up timer disabled)

#pragma config BOREN = OFF // Brown-out Reset Enable bits (Brown-out Reset enabled in hardware only (SBOREN is disabled))

#pragma config BORV = 190 // Brown Out Reset Voltage bits (VBOR set to 1.90 V nominal)

// CONFIG2H

#pragma config WDTEN = OFF // Watchdog Timer Enable bits (Watch dog timer is always disabled. SWDTEN has no effect.)

#pragma config WDTPS = 32768 // Watchdog Timer Post-scale Select bits (1:32768)

// CONFIG3H

#pragma config CCP2MX = PORTC1 // CCP2 MUX bit (CCP2 input/output is multiplexed with RC1)

#pragma config PBADEN = OFF // PORTB A/D Enable bit (PORTB<5:0> pins are configured as analog input channels on Reset)

#pragma config CCP3MX = PORTB5 // P3A/CCP3 Mux bit (P3A/CCP3 input/output is multiplexed with RB5)

#pragma config HFOFST = ON // HFINTOSC Fast Start-up (HFINTOSC output and ready status are not delayed by the oscillator stable status)

#pragma config T3CMX = PORTC0 // Timer3 Clock input mux bit (T3CKI is on RC0)

#pragma config P2BMX = PORTB5 // ECCP2 B output mux bit (P2B is on RB5)

#pragma config MCLRE = EXTMCLR // MCLR Pin Enable bit (MCLR pin enabled, RE3 input pin disabled)

// CONFIG4L

#pragma config STVREN = ON // Stack Full/Underflow Reset Enable bit (Stack full/underflow will cause Reset)

#pragma config LVP = OFF // Single-Supply ICSP Enable bit (Single-Supply ICSP enabled if MCLRE is also 1)

#pragma config XINST = OFF // Extended Instruction Set Enable bit (Instruction set extension and Indexed Addressing mode disabled (Legacy mode))

// CONFIG5L

#pragma config CP0 = OFF // Code Protection Block 0 (Block 0 (000800-001FFFh) not code-protected)

#pragma config CP1 = OFF // Code Protection Block 1 (Block 1 (002000-003FFFh) not code-protected)

#pragma config CP2 = OFF // Code Protection Block 2 (Block 2 (004000-005FFFh) not code-protected)

#pragma config CP3 = OFF // Code Protection Block 3 (Block 3 (006000-007FFFh) not code-protected)

// CONFIG5H

#pragma config CPB = OFF // Boot Block Code Protection bit (Boot block (000000-0007FFh) not code-protected)

#pragma config CPD = OFF // Data EEPROM Code Protection bit (Data EEPROM not code-protected)

// CONFIG6L

#pragma config WRT0 = OFF // Write Protection Block 0 (Block 0 (000800-001FFFh) not write-protected)

#pragma config WRT1 = OFF // Write Protection Block 1 (Block 1 (002000-003FFFh) not write-protected)

#pragma config WRT2 = OFF // Write Protection Block 2 (Block 2 (004000-005FFFh) not write-protected)

#pragma config WRT3 = OFF // Write Protection Block 3 (Block 3 (006000-007FFFh) not write-protected)

// CONFIG6H

#pragma config WRTC = OFF // Configuration Register Write Protection bit (Configuration registers (300000-3000FFh) not write-protected)

#pragma config WRTB = OFF // Boot Block Write Protection bit (Boot Block (000000-0007FFh) not write-protected)

#pragma config WRTD = OFF // Data EEPROM Write Protection bit (Data EEPROM not write-protected)

// CONFIG7L

#pragma config EBTR0 = OFF // Table Read Protection Block 0 (Block 0 (000800-001FFFh) not protected from table reads executed in other blocks)

#pragma config EBTR1 = OFF // Table Read Protection Block 1 (Block 1 (002000-003FFFh) not protected from table reads executed in other blocks)

#pragma config EBTR2 = OFF // Table Read Protection Block 2 (Block 2 (004000-005FFFh) not protected from table reads executed in other blocks)

#pragma config EBTR3 = OFF // Table Read Protection Block 3 (Block 3 (006000-007FFFh) not protected from table reads executed in other blocks)

// CONFIG7H

#pragma config EBTRB = OFF // Boot Block Table Read Protection bit (Boot Block (000000-0007FFh) not protected from table reads executed in other blocks)

// ISR to reset other parts

#include "routines.h"

// global variables for scheduler

unsigned int curr\_channel;

unsigned int system\_status;

unsigned int sampling\_flag;

unsigned int error;

int no;

unsigned int SSP\_flag = 1;

char data = 0x00;

char command[4];

char send\_data[34];

void system\_init(void);

char \*command\_ptr;

char \*data\_ptr;

int ne;

char temp\_ID;

char temp\_type;

char dataNum;

char encrypt1;

char encrypt2;

void main(void)

{

// system init

system\_init();

command\_ptr= &(command[0]);

data\_ptr = &(send\_data[0]);

// redirect output stream to \_H\_USER

// initialize peripherals

timer\_init();

I2C\_slave\_init();

interrupt\_init();

// initialize global variables

done = 0;

curr\_channel = 0;

sampling\_flag = 0;

// start measurements

start\_scheduler();

}

// power up initialization

void system\_init(void)

{

// system clk set-up

OSCCON = 0b11111000;

OSCCON2 = 0b00000100;

//low-frequency internal osc sourced from LFINTOSC

OSCTUNEbits.INTSRC = 0;

//disable PLL

OSCTUNEbits.PLLEN = 0;

// IO ports initialization

// see document for exact assignments

// TRISA = 0b11000111;

TRISB = 0x00;

TRISCbits.RC3 = 1;

TRISCbits.RC4 = 1;

ANSELCbits.ANSC3 = 0;

ANSELCbits.ANSC4 = 0;

// TRISC = 0b10011000;

// ANSELA = 0b00000111;

// ANSELB = 0x00;

// ANSELC = 0x00;

// WPUB = 0x00;

// IOCB = 0x00;

}

Intra\_Comm

// schedule the sensor module to achieve a certain sampling rate and low power consumption

#ifndef RESET\_H

#define RESET\_H

#include "routines.h"

void timer\_init(void);

void start\_scheduler(void);

void update\_status(int status);

int check\_status(void);

void timer\_rst(void);

void decoder(void);

extern unsigned int done;

extern unsigned int curr\_channel;

extern unsigned int freqCounter;

extern char \*command\_ptr;

extern char \*data\_ptr;

extern int no;

extern int ne;

extern char temp\_ID;

extern char temp\_type;

extern char dataNum;

extern char encrypt1;

extern char encrypt2;

#endif

Interrupt

#include "interrupts.h"

#include "i2c.h"

void interrupt\_init(void)

{

// disable all interrupts except for external interrupt on INT2/RA2

// and timer0/1 interrupt

// disable interrupt priority

RCONbits.IPEN = 1;

INTCON = 0b00100000; // timer0 overflow interrupt

INTCON2 = 0b11000100; // Pull-ups disabled/Interrupt0 Rising edge

INTCON3 = 0b00000000; // external to zero

PIE1 = 0b00001000; // enable timer2

PIE2 = 0b00000000;

PIE3 = 0b00000000;

PIE4 = 0b00000000;

PIE5 = 0b00000000;

IPR1 = 0b00100000;

PIR1bits.SSP1IF = 0;

IPR1bits.SSP1IP = 1;

INTCONbits.GIE\_GIEH = 1; /\* Enable the interrupts \*/

INTCONbits.PEIE\_GIEL = 1; /\* Enable peripheral interrupts \*/

// //extern no

// int no = 0;

// char T1 [] = {0x0000,0x0018,0x0018,0x0018,0x0018,0x0018,0x0018,0x0018,0x0018,0x0018,

// 0x0018,0x0018,0x0018,0x0018,0x0018,0x0018,0x0018};

// char C1 [] = {0x0000,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,

// 0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015};

// char S1 [] = {0x0000,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,

// 0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015};

// char F1 [] = {0x0000,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,

// 0x0015,0x0015,0x0015,0x0015,0x0015,0x0015,0x0015};

}

#pragma code high\_vector=0x08

void global\_interrupt(void) {

\_asm GOTO myISR \_endasm;

}

#pragma code

#pragma interrupt myISR

void myISR(void)

{

char count;

char buffer;

if (INTCONbits.T0IF) {

sampling\_flag = 1;

INTCONbits.T0IF = 0;

}

if (PIR1bits.SSP1IF) { // Check for SSP interrupt

count = (SSP1STAT & 0x2d);

//PORTB = count;

// last byte received is master write addr

if (count == 0x09){

no = 0;

ne = 0;

buffer = SSP1BUF;

//\*(command\_ptr + no) = buffer;

//no++;

SSP1CON2bits.ACKDT = 0; //ack

SSP1CON1bits.CKP = 1;

//state2, write/last date

} else if(count == 0x29){

SSP1CON2bits.ACKDT = 0; //ack

buffer = SSP1BUF;

//data[no] = buffer;

\*(command\_ptr + no) = buffer;

no++;

SSP1CON1bits.CKP =1;

}

//state3, master read, last byte addr

else if(count == 0x0D){

buffer = SSP1BUF;

SSP1CON1bits.CKP = 0;

SSP1BUF = data\_ptr[0];

// timer0\_delay(200);

SSP1CON1bits.CKP = 1;

ne++;

}

//state4, master read, last byte data

else if(count == 0x2C){

buffer = SSP1BUF;

SSP1CON1bits.CKP = 0;

SSP1BUF = data\_ptr[ne];

//SSP1BUF = 'H';

SSP1CON1bits.CKP = 1;

ne++;

// no++;

//WriteI2C(data);

}

else if ((count == 0x21) ||(count == 0x20))

{

buffer = SSP1BUF;

SSP1CON1bits.CKP = 0;

SSP1BUF = data\_ptr[ne];

decoder();

SSP1CON1bits.CKP = 1;

}

else{

buffer = SSP1BUF;

SSP1CON1bits.CKP = 1;

}

PIR1bits.SSP1IF = 0; // Clear the interrupt flag

}

INTCONbits.T0IF = 0;

INTCONbits.GIE = 1;

}

void decoder(void){

int i =0;

temp\_ID = command\_ptr[0] & 0x07; // mask for ID

temp\_type = command\_ptr[0] & 0x18>>3; // mask for type

dataNum = command\_ptr[1];

encrypt1 = command\_ptr[2];

encrypt2 = command\_ptr[3];

data\_ptr[0] = encrypt2;

data\_ptr[1] = encrypt1;

for (i = 1; i < dataNum+1; i++)

{

data\_ptr[2\*i] = 0x00;

data\_ptr[2\*i+1] = 0x09;

}

//PORTB = data\_ptr[2];

}