Project 2
Designing a Microprocessor Based Data Collection System

EE 475 Fall 2017

Denis Jivaikin
Sandy Lee
Jiayou Zhao

TABLE OF CONTENTS

ABSTRACT

The report presents the design a Remote data collection system that can be controlled by I2C protocols sent from a Local microcontroller that is then in turn controlled by a user on the PC through a RS232 connection. Test plans and specifications are then discussed, and results of the project are presented. The whole system consists of 2 major blocks: a Local Station and a Remote system. The system was designed so that multiple Remote Stations could be easily added and interfaced with the Local Station without much effort. Testing the system was made easy with the use of the Pickit 3 debugger tool and the logic analyzer. Overall, this project was a success and the requirements presented in the design specifications were met.

INTRODUCTION

The goal of the lab is to design an environmental data collecting system, which supports bi-directional communications, based on PIC18F25K22 microcontrollers. The purposes of the lab is to practice and get familiar with microcontrollers and utilize the knowledge of SRAM. At the same time, I2C protocol and RS232 protocol are designed to support the bi-directional communication feature between those electronic components.

DISCUSSION OF THE LAB

Design Specification

System Description

This specification describes and defines the design requirements and specifications for a environmental monitoring system. This system is to be able to measure temperature, carbon level, salinity, and flow rate of a location. It is to be interfaced with a Local PC with the ability to support Remote operations in the future. The system is to be low cost and reliable so that the data collected can be used in research with confidence.

Specification of External Environment

The monitoring system is to operate in an outdoors environment at a moderate temperature. It is to be durable to environmental factors such as water and wind.

System Input and Output Specification

System Inputs

The system shall support the measurement of the following datas:

- Salinity: 5.0 to 50.0 ppt
- Carbon level: from 10.0 to 350.0 ppm
- Flow rate: 100.0 to 1000.0 liters per second
- Temperature: 0 to 50 C

System Outputs

The system shall display the following measurement results using an LCD display:

- Salinity: 5.0 to 50.0 ppt ± 0.01
- Carbon level: from 10.0 to 350.0 ppm ± 0.1

- Flow rate: 100.0 to 1000.0 liters per second ± 0.1
- Temperature: -10 to 50 C ± 0.7 or displayed in Fahrenheit

User Interface

The user shall be able to select the following through the computer:

Measurements: the following measurements can be selected to initiate the measurement process

- Salinity
- Carbon level
- Flow rate
- Temperature

Display data: the follow data can be selected to be displayed onto the LCD in Figure 1.

- Salinity
- Carbon level
- Flow rate
- Temperature

Data size sent back: last measurement or 16 last measurements

Power: ON/OFF

Station: 1-16

Reset: the reset button will clear the display and end any ongoing measurements

The measurement results shall be presented on a 2 x 16 LCD display.

Figure 1. Front Panel Drawing

System Functional Specification

The system is intended to collect 4 different kinds of data: salinity, carbon level, flow rate and temperature, and display these data on the LCD display. For the temperature, the user will be able to select whether the data is to be displayed in Celsius or Fahrenheit.

The system will be designed with the final goal of turning one single system into a Remote collection Station in mind.

The system comprises 6 major blocks in block diagram in Figure 2.

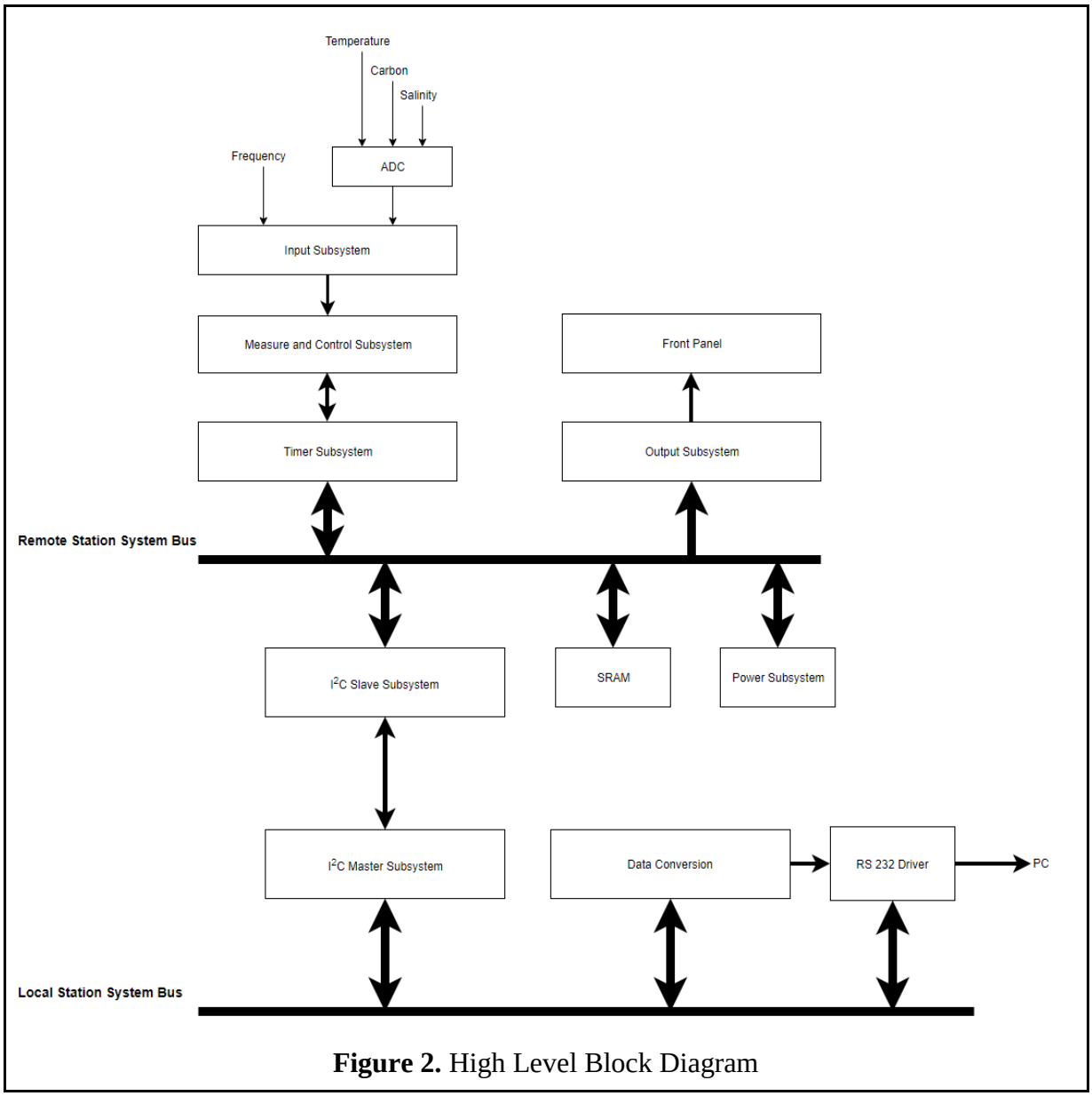

Figure 2. High Level Block Diagram

Input Subsystem – the input subsystem routes the input signals to the appropriate portion of the measurement and control subsystem. Inputs include frequency and and ADC value from the ADC system.

ADC – the Analog to Digital Converter (ADC) collects data from a particular analog input from the system, converts it into a digital measurements and sends it to the input subsystem. Measurement available to collect from ADC:

- Temperature
- Carbon
- Salinity

Measurement and Control Subsystem – the measurement and control subsystem provides the logic and control to convert sensor input to understandable data as well as deals with user input control logic

The flow rate is measured in frequency. A controlled gate signal will be fed through an AND gate with the to be measured frequency and the output of the gate will be used to clock a binary ripple counter. The controlled gate signal will be turned on for a known amount of time and the unknown frequency can be determined by the number of times the counter has incremented.

The rest of the data is collected in voltage, which can be read through ADC ports and translated into its corresponding units.

Last 16 measurements of unconverted raw sensor data is also saved in the Static Random Access Memory (SRAM) with ability to rewrite over old measurements.

Timer Subsystem – the timer subsystem is able to ask the Measurement and Control Subsystem to take a measurement of a specific sensor whenever necessary. The timer subsystem is also able to control when the measurement is taken, with the ability to pick the rate at which the measurements are made and when the timer system is on or off.

Output and Subsystem – the output subsystem provides the front panel display with the appropriate management and drive circuitry.

Power Supply Subsystem – the power supply subsystem will provide 5V upon the turn on of the system.

Front Panel – the front panel of the instrument shall display the results of the data on a 2 x 16 LCD.

SRAM Subsystem – the SRAM subsystem is responsible for storing the last 16 measurements is an octal format, where the oldest data gets overwritten by the newest measurements.

I2C Slave Subsystem – the I2C slave subsystem is responsible for receiving data from the I2C master when its address is specified, and translates the data into actions the slave has to perform

I2C Master Subsystem – the I2C master subsystem shall select the correct slave to send data to and receive data sent back from the slave that it selected.

Data Conversion – the data conversion subsystem is in charge of converting the measured ADC values into analog voltage values then into their corresponding sensor measurements. It also converts the frequency collected into its corresponding sensor measurement, flow rate.

RS 232 Driver – the RS 232 driver is responsible for receiving the user input from the PC and interpreting the commands. It is also responsible for sending measured and converted data to the PC through the RS 232 connection.

Operating Specifications

The system shall operate in an outdoors environment

Temperature Range 0-50C

Power 5 V DC

Reliability and Safety Specification

The MTBF will be a minimum of 10,000 hours

Design Procedure

RS232

The protocol is based on the receiving and transmitting feature of USART so that a USART channel should be configured with a baud rate of 9600 bps. Once the channel is configured and opened, the user should be able to send data through the RS232 cable, however the master PIC need to be programmed so that it can receive the data and write back to the PC. From the data sheet, the RX pin, which is used as incoming channel on the PIC, and the TX pin, which is used as outcoming channel, are defined as RC6 and RC7. Both of the pins should be enabled and the RX pin of the cable should be connected to the TX pin of the PIC and the TX pin of the cable should be connected to the RX pin of the PIC, so that the PIC should be able to get the data from the PC. However, master should be able to receive and transmit data in time. Since the arrival of data is arbitrary, interrupt with high priority should be set to control the communications.

I2C Communication

First for the connections, PIC18F25k22 provide two sets of SCL/SDA pins. One of them is defined as RC6 and RC7, another set is defined as RB1 and RB2. Either set is fine but need to make sure the SCL is connected to SCL and SDA is connected to SDA, unlike the connections for RS232.

For the configuration, all of the four pins should be enabled and even though I2C support both standard speed and high speed communication, standard speed at 100 kHz is much more reliable.

Master

Master device should be able to send data to and read data from a slave device. For the master device, interrupts need to be enable for the reading process. However, interrupts are not necessary for writing process. Because of the scheme of I2C, both slave device address and data address is necessary besides the actual data. A slave device address, for example 0x20, should be determined at the very beginning. In the system, Master is expected to send a single 8-bit command, but read multiple readings each time. A reading buffer is necessary to store the readings and transmit them to the PC through a USART port. While writing or reading, the master need to provide a special instruction to the slave to make it execute properly. According to the scheme of I2C, 0 is for writing and 1 is for reading and the command bit is defined as the LSB of the data containing slave address. In order to keep the original slave address and include

the command bit, the slave address should be left shifted by 7 bits and mathematically added with the command bit before the data is actually transmitted to the slave.

Slave

To receive data properly from the master, the slave device should use the exact same address as defined earlier. In the slave system, both reading process and writing back process need to be built. For the reading process, global interrupts are necessary to receive the data arriving arbitrarily in time. However, interrupts are not necessary for writing back process. The current process, whether writing or reading, depends on the last bit of the device address sent by the master.

During each procedures of both the master and slave, responding of the target device and state of the SSPBUF (empty or full) should be checked step by step.

Sensors

Carbon and salinity sensors modelling

Carbon and salinity sensors measure voltages and convert these voltages to corresponding carbon and salinity values. In order to model varying voltage values, the best choice was to use a potentiometer to simulate changing input voltages. A 20-turn potentiometer was chosen so that a better control of different voltage values could be had.

For the carbon sensor, output voltages of 5.0 to 250.0 mV correspond to 10.0 to 350.0 ppm, therefore, the following equation was used to convert the measured voltage to the correct carbon value:

$$y = 1387.76\,x + 3.1$$
$$y: carbon\ (ppm)$$
$$x: voltage\ (V)$$

For the salinity sensors, output voltages of 100.0 mV to 300.0 mV corresponds to 5.0 to 50.0 ppt. As a result, the following equation was used to convert the measured voltage to the correct salinity value:

$$y = 225x - 17.5$$
$$y: salinity\ (ppt)$$
$$x: voltage\ (V)$$

Flow rate sensors modelling

Flow rate sensors measure the frequency of an input and convert the measured frequency to the corresponding flow rate value. In order to measure frequency in hardware, a counter is needed. Since the range of frequency is from 1K to 10K, a counter that can count to at least 10K is used. If the counter was clocked by the output of an AND gate that takes in two inputs -- the unknown frequency and a controlled gate that is turned on for a known amount of time, the counter will be

able to record how many times the unknown frequency goes high for the known amount of time. Since frequency is how many times a frequency is goes high in 1 second, frequency for the unknown input can then be calculated by dividing the count from the counter by the time the controlled gate is turned on. In order to simplify the math, the controlled gate was chosen to be turn on for 1 second, meaning that the counter output after the controlled gate is shut off is the frequency of the unknown input. However, the counter has parallel outputs while the PIC only has limited amount of pins, meaning that a parallel to serial shift register is needed for the design.

The flow rate sensor measures frequencies of 1K to 10K Hz that corresponds to 100.0 to 1000.0 liters per second, so the following equation is used to convert the frequency into flow rate:

$$y = 0.1x$$

$$y: flow\ rate\ (\frac{liters}{s})$$

$$x: voltage\ (V)$$

Thermocouple

According to the National Institute of Standards and Technology (NIST) website, at room temperature 25 ℃, the measured voltage between the two ends of the K type thermocouple is 1 mV, meaning that the voltage will barely be measured by the ADC module in the PIC. This problem can be solved by including an op amp in a non-inverting configuration in the circuit. A gain of 100 was chosen for easier computation purposes. The voltage to Celsius conversion provided on the website had extremely accurate coefficients and high order polynomials, which would slow down the conversion process and take up excessive resources on the PIC. In order to deal with this problem the curve for the thermocouple was plotted, and the range for which the system was to function at was linearized. The final linearized function is as follows:

$$y = 275x - 10.5$$

$$y: temperature\ (Celsius)$$

$$x: voltage\ from\ op\ amp\ (V)$$

LCD Front Panel

In order to program the LCD, at least 6 parallel inputs are expected. Like previously mentioned, to save pin usage on the PIC, a serial to parallel shift register is required to send commands to the LCD.

SRAM

The basic function of the SRAM in the system is to perform a write or read. The process can be controlled via ~WE and ~OE signals. When the SRAM performs a write process, it is expected to store the measuring results into the register, so the SRAM should be connected to the slave device. According to the requirements, 8 data input/output and 8 address pins are enough and the rest of the pins can be grounded. Meanwhile, the SRAM should be integrated with a slave PIC, multiple serial-to-parallel shifters are required to pass both actual data bits and address bits to the

PIC. When the SRAM performs a read process, an octal tri state buffer and a parallel-to-serial shifter is required to process the output data.

System Description

Overall this system prototype was compiled of two stations: Remote Station and Local Station. In the final implementation of such system, 15 more Remote Stations can be connected to the Local Station. This system is designed to be operated by a user, however it can also operate without the user after the system has been turned on.

Remote Station

Remote Station is a data collection station that is able to save the data to the Local SRAM, display the data on LCD, and follow the commands of the Local Station over I2C.

Inputs

- Sensor Readings - the readings can only be acquired one at a time from either the ADC Module or from the frequency input.
    - ADC - only one ADC module is used to collect the measurement from the analog inputs. Analog inputs are:
        - Temperature
        - Salinity
        - Carbon
    - Frequency - frequency is collected using a serial input.
- I2C Commands from Local Station - the Remote Station assumes the role of the slave in the I2C protocol and receives the following command from the user.
    - Start/Stop scanning - allows the system of scanning to be started or stopped.
    - Switch sensor - allows the user to switch the sensor from which the data is being read, as well as, display it on the LCD screen.
    - Convert temperature measurement from Celsius to Fahrenheit - allows the user to choose how the temperature measurement is displayed.
    - Send the most recent measurement - allows the Local Station to receive the most recent measurement from the selected sensor.
    - Send most recent 16 measurements - allows the Local Station to receive 16 most recent measurements from the selected sensor.

Outputs

- LCD Screen Display - displays the current sensor reading on the LCD screen with an option of Celsius/Fahrenheit for the temperature.
- Measured data to the I2C bus - the Remote Station is able to send the most recent data whenever the Local Station asks for it.
    - 16 most recent measurements - allows the Local Station to receive 16 most recent measurements from the selected sensor.

    - The most recent measurement - allows the Local Station to receive the most recent measurement from the selected sensor.

Local Station

Inputs

- UART User Input - allows the User to interface with each individual sensor and read data from the specific Remote Station. Commands able to send:
    - Start/Stop scanning - allows the system of scanning to be started or stopped.
    - Switch sensor - allows the user to switch the sensor from which the data is being read, as well as, display it on the LCD screen.
    - Convert temperature measurement from Celsius to Fahrenheit - allows the user to choose how the temperature measurement is displayed.
    - Send the most recent measurement - allows the user to receive the most recent measurement from the selected sensor.
    - Send most recent 16 measurements - allows the user to receive 16 most recent measurements from the selected sensor.
- I2C data from the Remote Station - able to receive the data from the I2C bus slave (Remote Station). Data received can be 16 or 1 most recent measurements.

Outputs

- UART Sending Data to the User - able to send the measurement back to the user over UART communication.
- I2C Commands to Local Station - Local Station assumes the role of the master on the I2C bus and can send and receive data to and from the Local Station. See the Remote Station System Description above for more detail on the signals.

Error Handling

- User error - the system is able to handle unknown inputs from the user by ignoring the commands entirely.

Software Implementation

The system consist of several individual parts and each of them has own functionalities. There are two main modules for two stations: Local Controller and Remote Station as seen in Figure 3. Local Controller is able to receive and send commands back and forth between the PC. Local Station is also in charge of the Remote System through the I2C bus. The Remote System is able to Receive Commands from the user through the Local Station, measure data through a variety of sensors, store the measured data, send data back to the Local Controller, as well as display that data on the LCD display.

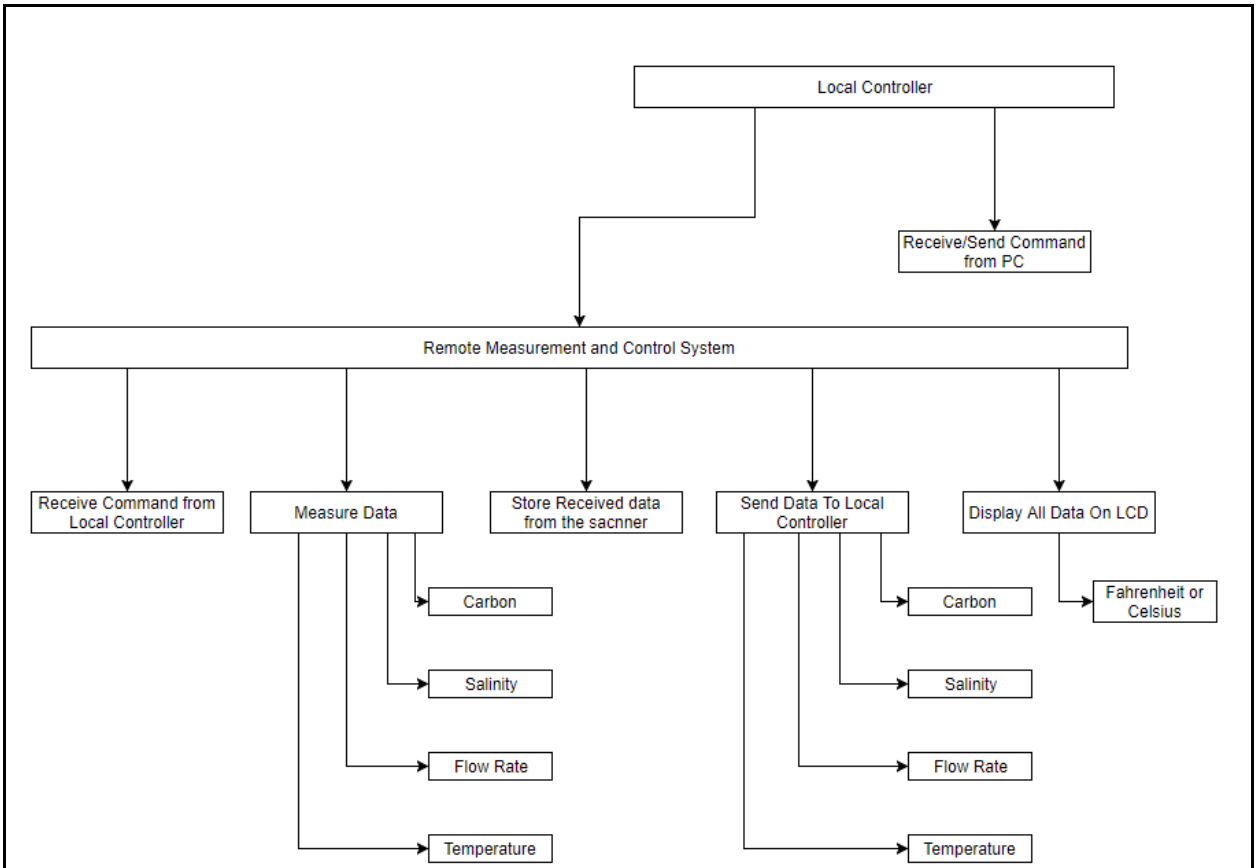

Figure 3. Functional Diagram of the Station System

As seen in control flow diagram in Figure 4 below, the system starts out with the user sending commands from a serial communication capable device to a Remote Station (Master device), which is able to communicate to all the other stations (in this case only one). After the command is sent, the Remote Station takes control and is able to communicate with the rest of the hardware, which includes:

- ADC
- SRAM
- LCD
- Frequency measurement

After the measurement is collected, it can transported back to the Master device, which in turn will be able to send it back to the user.

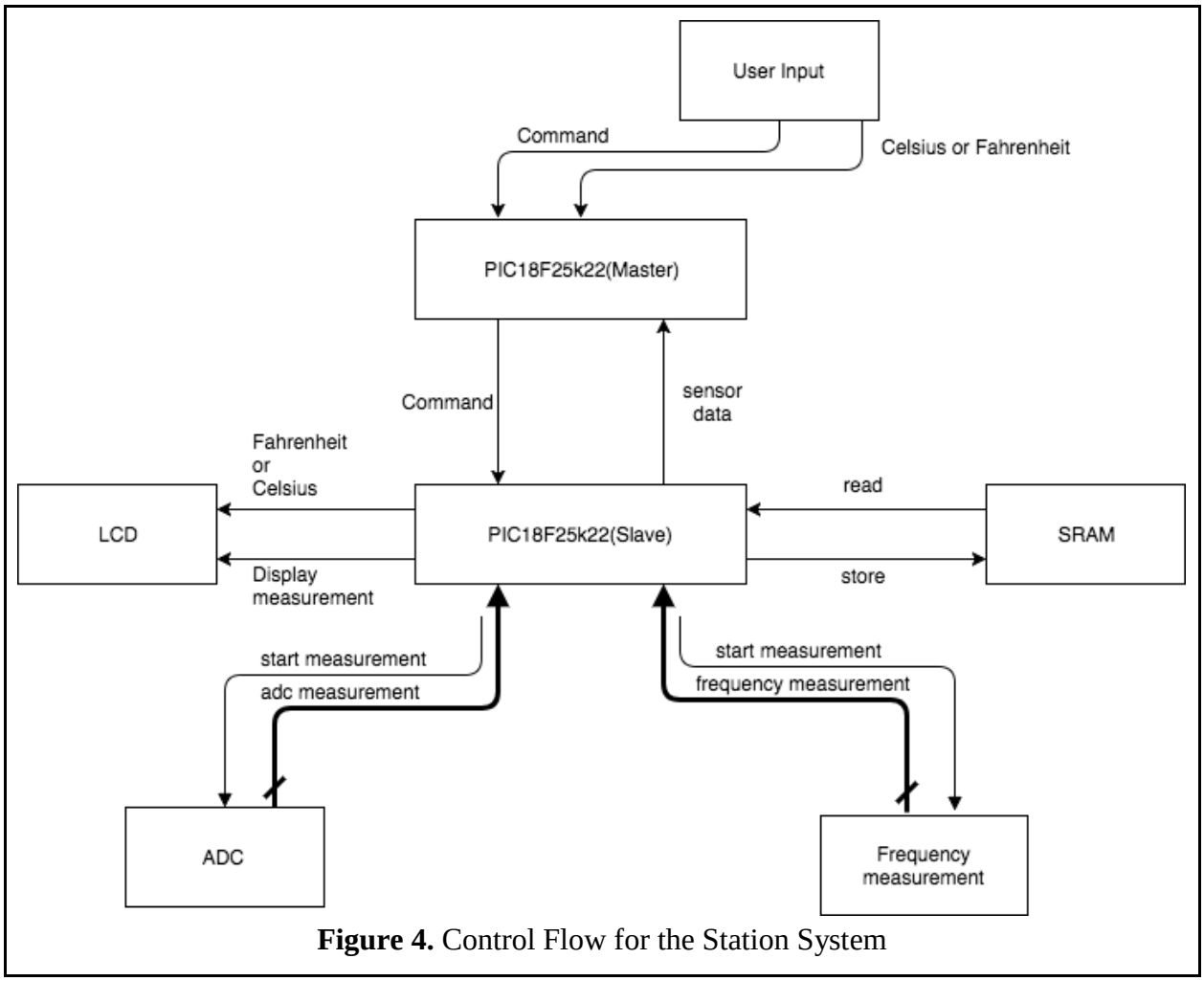

Figure 4. Control Flow for the Station System

User Input

This module is supported by RS232 cable. The user can send command to master PIC, which controls On/Off mode, sensor selection, Celsius or Fahrenheit conversion of the temperature sensor, device selection, and receive data back through USART ports. The USART channel is configured as COM5 with a baud rate of 9600 bps. The data length is expected to be 8-bits. The channel is controlled by interrupts seen in Figure 5. This is due to the arrival of data being arbitrary and it should have high priority. Every time data comes in, interrupt flags will be set and interrupt handler will start executing so that data can be received and transmitted in time.

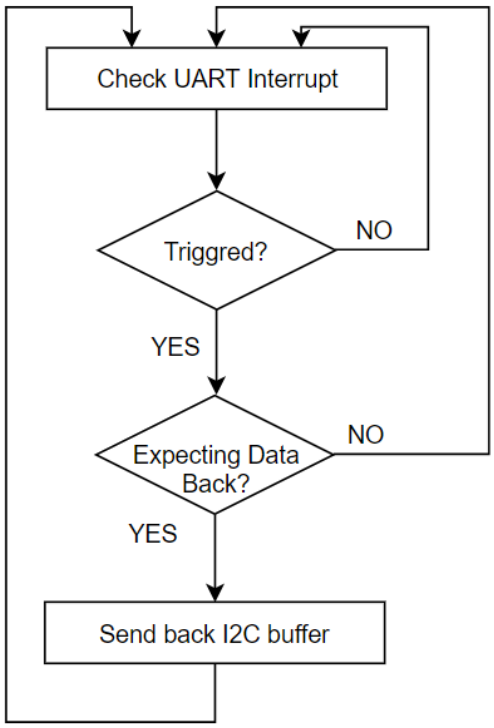

Figure 5. UART Interrupt check

PIC (Slave)

This module is the Remote Station Controller. The PIC can receive command from the master and activate specific sensors based on the command. At the same time, the PIC can collect data from sensors and send the data back to master.

The slave channel utilizes an interrupt-based scheme in Figure 6. This is due to the arrival of incoming data being arbitrary and it should have the highest priority. Incoming data is received through a second Master Synchronous Serial Port (MSSP2) that is configured as a slave device. While an address comes in with a writing command, the MSSP2 will compare the address with the defined slave address. If they matches, interrupt handler will start executing. If the received data is slave address, the handler will clear the buffer and get rid of that address. If the received data is actual information, then the handler will store the data into a specific location.

While an address comes in with a reading command and if the device address matches the received one, slave device will get the required data and send it to the master through I2C protocol.

The actual sensor measurements are taken with the help of the Timer interrupt seen in Figure 7. The timer interrupt will keep checking back to see if the timer has expired, and if it did a

measurement on a selected sensor will be taken. The Timer operates at a lower priority than the I2C interrupt, in case a message from the Master comes in before the measurement is over.

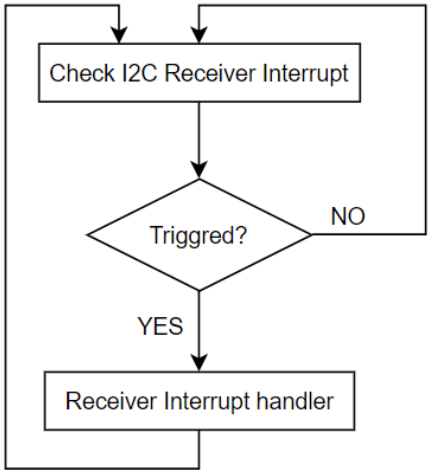

Figure 6. I2C Interrupt check

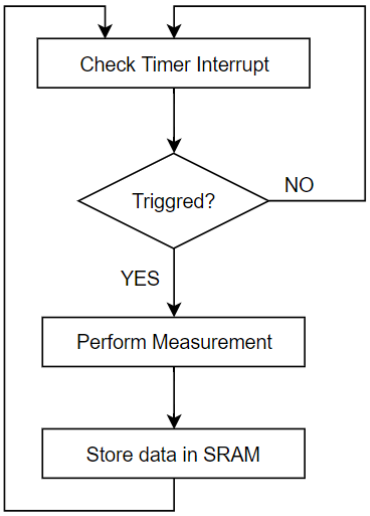

Figure 7. Timer Interrupt check

PIC (Master)

This module is the Local Controller. The PIC can read commands from the PC and transmit the commands to a slave PIC that performs as a Remote Station. If the slave PIC sends data back, the master PIC can also receive it.

The master channel sends the command defined by the user as an 8-bit character. The first Master Synchronous Serial Port (MSSP1) is configured as a master with a speed of 100kHz and

no slewing. All control of the MSSP1 is handled without interrupts. After waiting for an idle connection, the master begins communication according to the I2C protocol. The slave device is set to 0x20 and a writing command will be send to the slave. After that, normal data including data address and user commands will be transmitted to the salve.

The master channel can also send device address with a reading command to the slave. Once the command is received by the slave, the slave then is expecting for a specific data address. The master can then send the data address to the slave and expect for the desired value written back by the slave.

SRAM

This module is designed to store up to the last 16 readings for each of the sensors. The data gets passed in and collected to and from the SRAM using serial communication. A regular loop is used to send and collect data to and from the SRAM. Delay functions are used to make sure that all the shift registers and SRAM hardware has enough time to access and store the data.

LCD

This module is for displaying. The screen can display the measurements in Celsius or Fahrenheit based on the command received through RS232. The LCD display is controlled in the same way as the SRAM, where shift registers are controlled with additional delays for the hardware computation.

ADC

This module can measure analog signals and send the readings to the slave PIC. The module will execute based on the master command. ADC is a built in function in the PIC microcontroller with a digital value being available for reading anytime the device requires.

Frequency

This module can measure frequencies of signals and send the readings to the slave PIC. The module will execute based on the master command.

When the whole system is executing, every elements except for the PC is waiting for the master command. A user will sent a command to the Master PIC through the PC. The master will sent that signal to Slave PIC. The slave PIC will decode the received command and send it to the corresponding sensors/stations. Those ADC or frequency measurement sensors will be executed based on the commands. After measuring, results will be stored in a SRAM, displayed on the LCD screen, and sent back to master the PIC.

Hardware Implementation

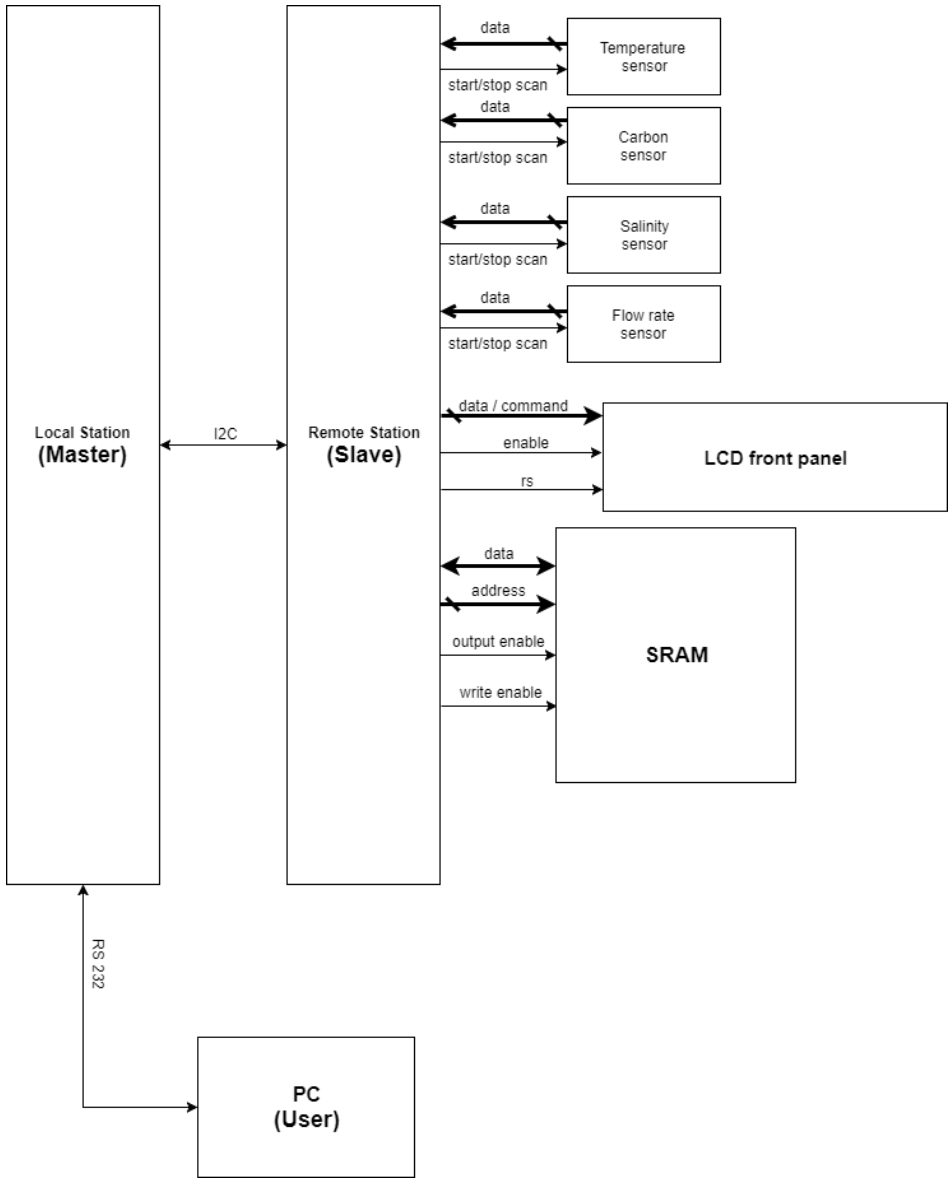

Figure 7. High level hardware block diagram

The whole system is implemented using two PIC18F25K22 microcontrollers - one Master and one Slave as seen in Figure 7. The Master PIC is connected to the PC with RS232 cable and to the Slave PIC a I2C connection. The Slave PIC acts like a “middleman” between the Master PIC and the rest of the hardware. This hardware includes:

- Sensors
    - ADC and serial connections
- LCD front panel
    - Serial output

- SRAM
    - Serial input and output

LCD Front Panel

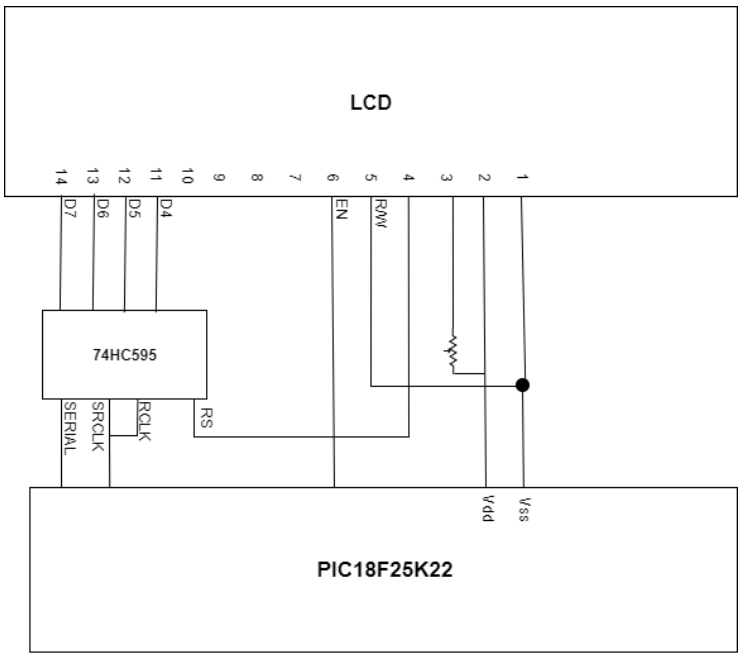

Figure 8. Hardware block diagram for LCD front panel

The LCD is controlled by the PIC by sending the majority of the commands through a serial to parallel shift register 74HC595 as seen in Figure 8. The use of the serial to parallel shift register reduces the pins used on the PIC. The clock signals on the shift registers are connected together, so srclk signal is one clock cycle behind, meaning that the PIC needs issue one extra clock cycle for all the commands to be shifted out of the register. The R/W pin of the LCD is grounded since only the writing to the LCD is performed. Note that the LCD is configured to function in 4 bit mode so as to reduce the PIC's pin usage.

Sensor modelling

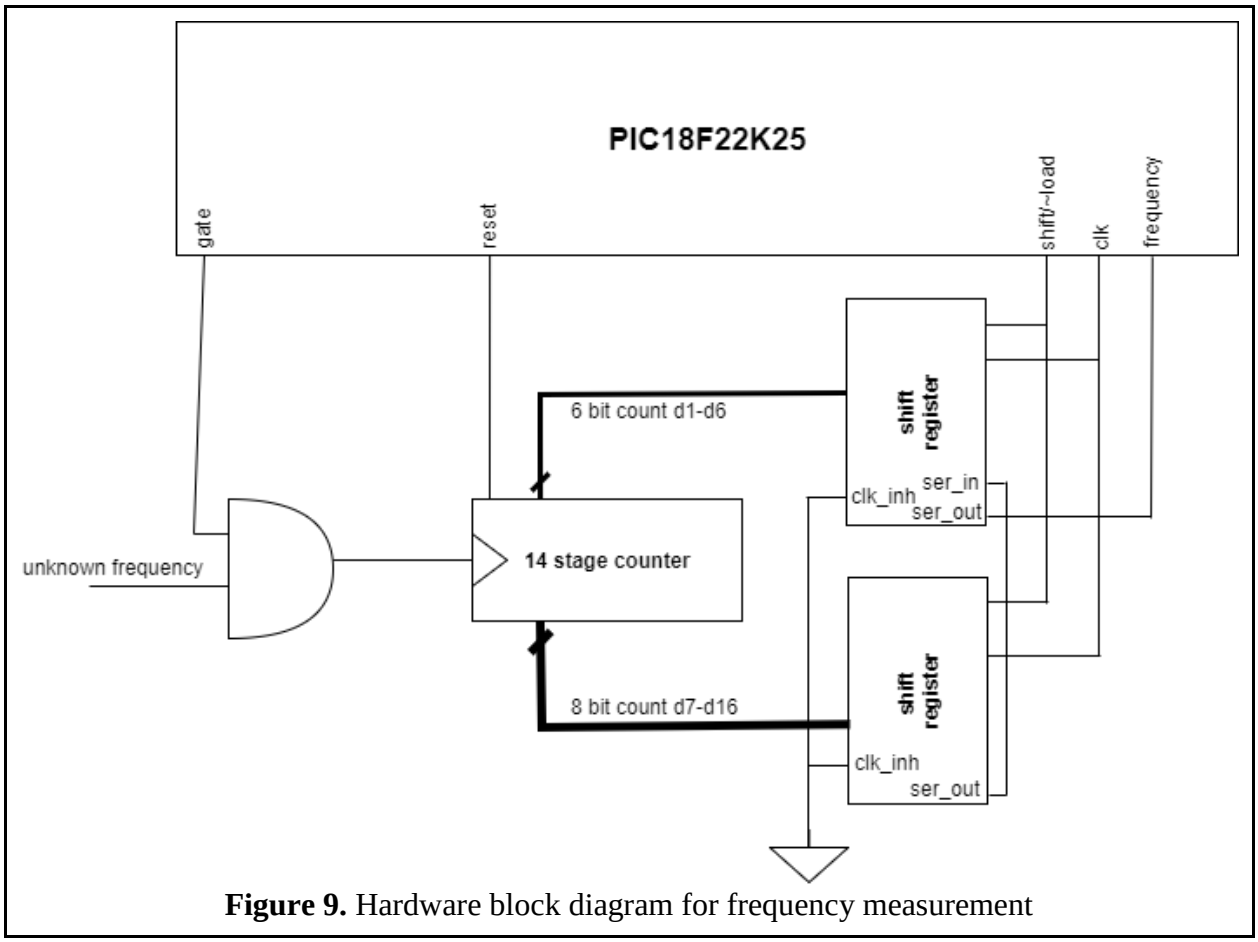

Figure 9. Hardware block diagram for frequency measurement

The sensor model is display in Figure 9. Since the flow rate sensor is measures frequency, the sensor is modelled by building hardware that measures frequency. This is achieved by feeding the unknown frequency into an AND gate with a gate that is opened for a known amount of time (1 s) and using the output from the AND gate to clock the counter. Two parallel to serial shift registers are daisy-chained to increase the input of the shift register to 14 bits instead of 8. The shift/~load, clk, and clk_inh signals on both registers are wired together to synchronize the process. In order to shift out serial outputs, the shift/~load has to be pulled low to load the parallel inputs then pulled high to shift out each bit at each positive edge of clock. Clock inhibit is wired to ground on both registers since the PIC will only read the serial output when the output is valid, so the shift registers do not need to be disabled.

The carbon and salinity sensors measure voltages, then convert these voltages to the correct carbon or salinity values. These input voltages are modelled with two 20-turn 4k potentiometers so that input voltages can be changed easily. The output voltage from the potentiometers are fed into two separate ADC ports and represent carbon and salinity measurements respectively.

Thermocouple circuit

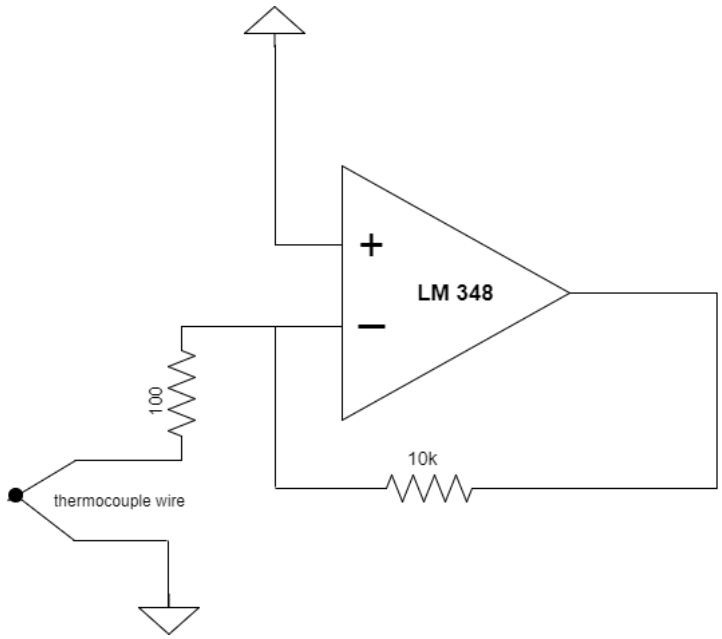

Figure 10. Hardware block diagram for thermocouple circuit

In order to measure temperature, the K type thermocouple is used. Since the voltage difference between the two ends of the thermocouple wire is extremely small and the ADC module in the PIC is not able to measure voltage with such fine accuracy, an op amp is used. In Figure 10, the LM 348 general purpose op amp is configured as an inverting amplifier with 100 gain. The output voltage is connected to an ADC input on the PIC.

Static Random Access Memory (SRAM)

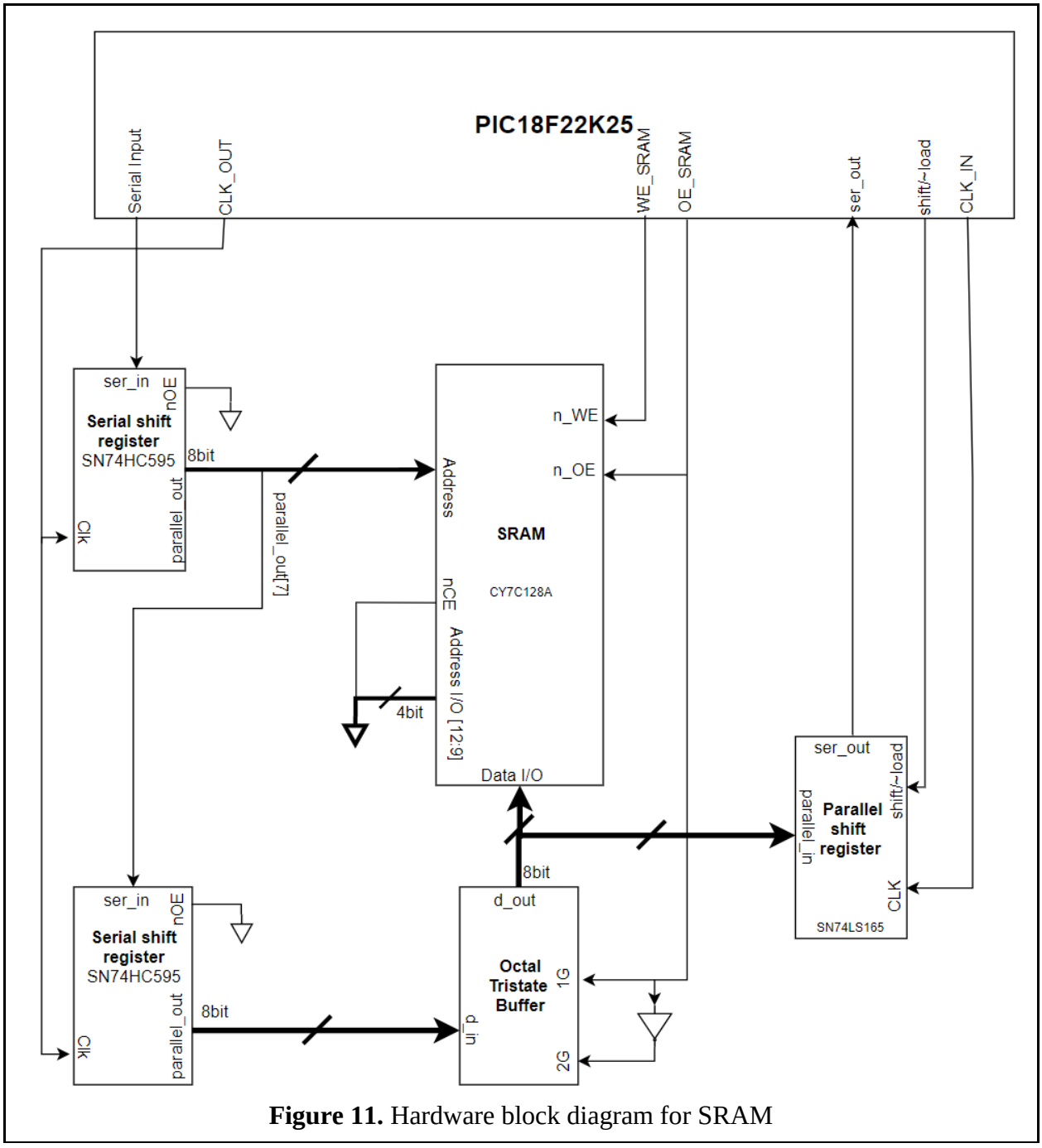

Figure 11. Hardware block diagram for SRAM

To operate the SRAM for storing and reading data purposes, the system would require:

- 8 Address pins - for the purposes of our application, only 8 connections are required for addressing. The other 4 pins can be connected to ground.
- 8 Data Input/ Output pin
- ~WE (Write Enable), ~OE (Output Enable), ~CE (Chip Enable)

Since the PIC can't support to handle all of the mentioned connections, 2 serial-to-parallel shifters, parallel-to-serial shifter, and an octal tristate buffer as seen in Figure 11. The two serial-to-parallel shifters are used to load the address and the input data from the PIC to the SRAM. The data that gets passed down from the serial input first includes the data to be written to the SRAM followed by the address bits. This is done by connecting the last bit of the first shifter to the serial input of the second shifter, as well as, clocking both counters at the same time.

The output of the data shifter feeds into an octal tri state buffer, which is controlled by the PIC to allow the data to be passed down to SRAM or not. The signal controlling the tri state buffer is also connected to the ~OE pin which tells the SRAM to perform the read at a specific address. This is done to make sure that when the read happens the data that comes out of SRAM is not interfering with the data coming from the data shifter.

SRAM is controlled with two signals: ~WE and ~ OE. These signals control whether the device is performing a write or a read. When a read is performed, and the tri state buffer is in the High-Z mode, the data flows to the parallel-to-serial shifter, which then sends the read data to the PIC.

TEST PLAN

Since the whole system consist of several single parts, testing each of them individually is the first thing that needs to be done before testing the system as a whole. First the RS232 feature should be tested to make sure the PIC can get command from the PC. After that, I2C communication should be tested so that the Local Controller can transmit data to Remote station(another PIC) and sensors. If the master PIC is able to transmit data, the next thing is to check is the receiving function of the slave PIC. Since only when the slave PIC receives commands from PC can the sensors perform correctly. Then, what needs to be tested is the write back function from the slave PIC to the master PIC. When the communications are tested, the whole system has its frame and things left are sensors and SRAM. The measurements made by the sensors then needs to be checked for their accuracy. Then SRAM should be tested to make sure data can be properly stored and read.
When all components of the system are functioning correctly, they were then combined together as a whole and the functions of the entire system wa then tested.

TEST SPECIFICATION

RS232

For the RS232 feature, the system should be able to read one or more byte(s) each time and echo the exact input back to the console.

I2C

For the I2C feature, the master should be able to send some specified values, for example, 0x0E to a register address of the slave and the slave should be able to correctly receive and store the data and send it back to master. Master should be able to receive the value written back by slave.

LCD

For the LCD, the screen should be able to present any received information include characters and numbers. If a character “c” or “1” is sent to the screen, then the LCD screen should display “c” or “1”.

Sensors

The testing of the sensors was made easy by a functioning LCD front panel. A known voltage or frequency is fed into the sensor modules, and the measured values were displayed on the LCD screen. If the displayed measured value is within percentage error of the actual value, then the sensors are deemed to be functioning correctly.

SRAM

Even though writing to the SRAM cannot be tested individually, it can be tested with the reading function. The SRAM module is said to be functioning if the data that was stored at a specified address can be read back with the data matching the data that was sent. This should be checked for all addresses for the test on the SRAM to pass.

TEST CASES

RS232

Using the “serial port utility” tool on the PC to send a char “n” to the master PIC, the PIC was programmed to receive any input from the PC and echo it back to the console. If a char “n” is send to the console, then the system can receive and transmit a single character. Then using the tool to send more characters, for example, “abc”, if “abc” is transmit to the console, then the system can receive and transmit multiple characters and the whole system should be fine.

I2C

There are several individual functions need to be checked in I2C protocol.

For configuration, the logic analyzer was a useful tool. By sending a value 0xAF from master to slave and observing the SCL and SDA signals through the oscilloscope, the configuration can be checked easily. If the configuration is correct, then the signals should appear in a particular pattern.

For the “master write”, oscilloscope is useful. By observing the signals, the writing feature can be tested. If the SDA signal matched what are sent by the master, then the writing feature should be fine.

For the “master read”, it can be tested together with the “slave write back”

For the “slave read”, LED is a useful tool. LED should light only when the value received by slave matches the value sent by master. In other word, if LED light, then the reading feature is fine.
For the “slave write back” and “master read”, LED is still useful. LED should light only when the value received by master matches the value sent by slave. In other word, if LED lights up, then the features are fine.

LCD
In order to test the LCD, the first step is to make sure the LCD can be initiated correctly. If the LCD is initiated correctly, the display will have a clear screen with a blinking cursor at the starting position. Next, writing to the LCD is tested by sending characters to the display. One case that needs to be test specifically is when the input string exceeds 16 characters, which is the line size for the LCD screen. If the LCD is able to write that correct data to the second line of the LCD, then the test for the LCD has passed.

SRAM
Testing the SRAM was made easy by lighting an LED. First, different data were written to each of the 2K addresses in the SRAM, then these data were then read back from the SRAM. Data stored at each address were made to match the address, meaning that 0 would be stored at address 0, 1 at address 1 and so on. The test program was written so that the PIC would light up the LED if each data read from each address matches the address itself. If the LED lights up after the program is done running, it means the SRAM passed the test successfully

Sensors
Each sensors were to be tested individually but the test procedure were the same for each sensor. Since each sensor had a range of voltage/frequency measurements, the minimum and maximum measurements were tested and so were a few points in between. The three sensor models were tested by applying a voltage across the potentiometer and turning the potentiometer to create varying voltages. Each sensor was said to have passed the test if the voltage from the potentiometer measured by a multimeter matched the measurement sent to and displayed on the LCD screen. The same method was used for testing the flow rate model, which measured frequency, except that a square wave with a known frequency was fed into the circuit.

FAILURE MODE ANALYSIS

Flow rate modelling -- frequency measurement

Signal		Result/Impact to system
gate	SA1	The AND gate that has the gate signal as an input will always be stuck at 1, which means the counter will always be off. This results in the frequency measured will always be 0 no matter what the actual frequency is. The flow rate data stored in the SRAM will be all 0’s and

		when the user on the PC end tries to request flow rate data, the user will always receive 0 values on the screen.
	SA0	The AND gate that has the gate signal as an input will always be stuck at 0, meaning that the counter will always be off. This results in the frequency measured will always be 0 no matter what the actual frequency is. The flow rate data stored in the SRAM will be all 0's and when the user on the PC end tries to request flow rate data, the user will always receive 0 values on the screen.
reset	SA1	The counter will be constantly be reset. The counter will always have an output of 0, causing the flow rate measured to always be 0. The SRAM will store all 0 for the flow rate and when the user requests the flow rate on the PC end, the user will always receive 0.
	SA0	The counter will never be reset. Every time the gate turns on for a flow rate measurement, the counter will start counting from the previous number it stopped at. This means all the flow rate measurements except for the first one will lose their integrity and the wrong values will be stored in the SRAM. When the user requests the flow rate, the user will get back incorrect measurements.
shift/~load	SA1	The outputs from the counter will never be properly shifted into the shift registers. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.
	SA0	The outputs will constantly be loaded into the counter, but the parallel inputs will never be shifted out. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.
clk	SA1	The values in the shift register will never be shifted out properly. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.
	SA0	The values in the shift register will never be shifted out properly. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.

LCD front panel

Signal		Result
en	SA1	The commands sent to the LCD will always be recognized by the LCD even when the output from the shift register is not ready yet. The LCD

		cannot be initialized correctly, thus the data cannot be displayed.
	SA0	The commands sent to the LCD will never be recognized by the LCD so the display will never be configured or turned on. The data will not be displayed on the LCD.
clk	SA1	The data will never be shifted into/out of the shift register. This means the LCD will never be configured to display any data.
	SA0	The data will never be shifted into/out of the shift register. This means the LCD will never be configured to display any data.
serial	SA1	The data shifted out of the shift register will always be 0. The LCD will not receive the proper commands to be initialized. Data will not be displayed on the LCD.
	SA0	The data shifted out of the shift register will always be 0. The LCD will not receive the proper commands to be initialized. Data will not be displayed on the LCD.

SRAM

Signal		Result
Serial input	SA1	SRAM will always write to or read from address 0xFFF. The desired data will never be available, and data will never be stored at the correct address. Address at 0xFFF will always be overwritten on every scan from the sensors.
	SA0	SRAM will always write to or read from address 0x000. The desired data will never be available, and data will never be stored at the correct address. Address at 0x000 will always be overwritten on every scan from the sensors.
clk_out	SA1	The address/data will never be shifted out of the registers. The correct data will never be stored at the correct address. When the user requests data, the correct data will never be read from the correct address.
	SA0	The address/data will never be shifted out of the registers. The correct data will never be stored at the correct address. When the user requests data, the correct data will never be read from the correct address.
we_sram	SA1	The data will always be read from the SRAM and new data will never be stored into the SRAM. When the user tries to request data from the PC end, the user will get back data that was stored in the SRAM from startup.
	SA0	The data will always be written to the SRAM and no data can

		requested. The user will never be able to get data from the sensors when the user requests data on the PC end.
oe_sram	SA1	The SRAM will never produce an output upon a read request. On the user end, the user will never be able to get the data.
	SA0	The SRAM will always be producing outputs even on a write request. This error should be undetected since the output data will not be handled during write requests.
shift/~load	SA1	The outputs from the counter will never be properly shifted into the shift registers. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.
	SA0	The outputs will constantly be loaded into the counter but the parallel inputs will never be shifted out. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.
clk_in	SA1	The values in the shift register will never be shifted out properly. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.
	SA0	The values in the shift register will never be shifted out properly. The frequency measured will lose integrity, causing the SRAM to store the wrong data and the user will get back the wrong data on the PC end.

I2C Master

If the data sent from the master is stuck at 1, the correct station will never be activated and the master will be unable to communicate with the Remote Station.

If the data sent from the master is stuck at 0, the correct station with never be activated and the master will be unable to communicate with the Remote Station.

I2C Slave

If the data sent from the slave is stuck at 1, the data received by the station will be stuck at 1 as well. This will be interpreted as the data read from the sensors are encoded in all 1's, meaning that the data from the sensors has lost their integrity.

If the data sent from the slave is stuck at 0, the data received by the station will be stuck at 0 as well. This will be interpreted as the data read from the sensors are encoded in all 0's, meaning that the data from the sensors has lost their integrity.

RS232

If the data transmitted through RS232 is stuck at 1, the data received by the serial port on the PC will receive 1, meaning the sensor reading has lost its integrity during the transmission and is not correct.

If the data transmitted through RS232 is stuck at 0, the data received by the serial port on the PC will receive 1, meaning the sensor reading has lost its integrity during the transmission and is not correct.

PRESENTATION, DISCUSSION, ANALYSIS OF RESULTS

The expectation was that the Remote Station would be able to collect data from three different ADC sensors and one serial input sensor. The Remote Station would then display such data onto the LCD display and keep a collection of 16 most recent measurements in the memory. Another expectation was that the Local Station would be able to communicate with the user from the PC and then interface with the Remote Stations. It was expected of the system to be able to scan and stop at a user's request as well as to send the data back from the Remote Station back to the user.

The results of our design met all of the expectations. On the hardware level the design was implemented using two PIC microcontrollers with addition of SRAM, LCD, and sensors. The hardware was integrated with shift registers to conserve the amount of digital inputs and outputs used. On the software level, an interrupt-based programming approach was used, where it was possible to run the system without integrating a lot of code in the main method.

The project was started with development of sensors and LCD display. This phase of the design was important to implement first to make sure that the integral part of the hardware is functioning correctly. The measured values from the sensor were compared to the calculated values to make sure that they were properly calibrated. For the purposes of debugging an LCD was used to display all of the data.

The second phase of the design included adding an SRAM and a Local Station through I2C. SRAM was tested independently in a previous laboratory so the testing operation of the PIC on the SRAM was brief. However, I2C was more difficult to calibrate. There has been a lot of debugging in the integration of I2C. The problems included writing to the slave, reading from the slave, catching the I2C interrupt as a slave. The communication problems were solved quickly with the help of the Pickit debugger. It was possible to stop the iteration of code wherever needed and read the value assignments. The debugger allowed the I2C communication to be tested properly by receiving expected data from another PIC.

Overall the project was finalized after the I2C communication was integrated, since after that the user could communicate with the Remote Station through the Local Station and from there the

only possible error could have been in setting up the User Interaction with the program. For this purpose, user testing had to be performed.

ANALYSIS OF ERRORS

Several errors that surfaced during the debugging phase and their solutions are as follow:

- When sending the data from the slave to the master, the first entry in the array received by the master was always a value of hexadecimal 62, even if that entry was hardcoded into something else in the slave program
    - Reason: the reason behind this problem was not found
    - Solution: the first entry in the array (0th position) was skipped when the slave as writing and when the master was reading.
- The PIC was not reading the right values shifted out by the shift register in the flow rate model even though the output from the shift register was verified through the logic analyzer
    - Reason: the PIC was reading in the values at the wrong time
    - Solution: the PIC was programmed to read the values from the shift register after the clock is low instead of when it was high
- The output from the op amp in the thermocouple circuit was always stuck at Vdd
    - Reason: the -Vdd pin on the op amp was wired to ground of the power supply
    - Solution: provide double rail power supply to the op amp
- When flow rate measurements were taken, the scanning process could not be stopped
    - Reason: the flow rate measurement takes the longest among all measurements, and the I2C interrupt could not be triggered when the system was already in a timer interrupt (used to take measurements at a set rate)
    - Solution: enable the high and low priority interrupts on the PIC. This allows for context switches when the system is already in an interrupt

SUMMARY AND CONCLUSION

In this lab, two microcontrollers were integrated together to create a Local and Remote Stations system capable of collecting data from the carbon, salinity, flow rate, and temperature sensors. This data could then be sent to the user for inspection. The user would be able to control up to 16 different stations by controlling the on/off mode, data size transportation, selection of scanner, as well as temperature conversion.

In conclusion this project, of building a self-sustaining scanner system with user help, was successful. This success was achieved by integrating a lot of provided tools, such as Pickit debugger, Logic Analyzer, and Microchip Code Configurator. If this project was to be redone again, Code Configurator would have been used earlier to save time debugging legacy peripheral libraries.

APPENDICES

Local Station

Main module:

```
#include "mcc_generated_files/mcc.h"
#include <string.h>
#include <stdio.h>
#include <math.h>
#define SLAVE_I2C_GENERIC_RETRY_MAX     100
/*
                         Main application
 */

// UART send function
void sendUART(char * val) {
    int i = 0;
    while (val[i] != '\0') {
        EUSART1_Write(val[i]);
        i++;
    }
    EUSART1_Write('\n');
}

// Bit to double salinity calculation
double send_sal(uint16_t val) {
    double converted = 0;
    for (int i = 0; i < 10; ++i) {
        unsigned int bit0 = (val >> i) & 1;
        converted += pow(2,i) * bit0;
    }
    converted = converted * 0.001;

    double salinity = 0;
    salinity = 225 * converted - 17.5;
    return salinity;
}

// Bit to double temperature calculation
double send_temp(uint16_t val) {
    double converted = 0;
    for (int i = 0; i < 10; ++i) {
        unsigned int bit0 = (val >> i) & 1;
        converted += pow(2,i) * bit0;
    }
    converted = converted * 0.001;

    double temperature = 300 * converted - 40;
    return temperature;
}

// Bit to double carbon calculation
double send_car(uint16_t val) {
    double converted = 0;
    for (int i = 0; i < 10; ++i) {
        unsigned int bit0 = (val >> i) & 1;
        converted += pow(2,i) * bit0;
    }
    converted = converted * 0.001;
```

```
    double carbon = 0;
    carbon = 1387.76 * converted + 3.1;
    return carbon;
}

// Bit to double flow rate calculation
double send_flow(uint16_t val) {
    int converted = 0;
    for (int i = 0; i < 12; ++i) {
        unsigned int bit0 = (val >> i) & 1;
        converted += pow(2,i) * bit0;
    }
    double flowr = 0;
    flowr = converted * 0.1;
    return flowr;
}

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    // define onstants
    uint16_t dataAddress = 0x10;
    uint8_t         writeBuffer[3];
    uint16_t        timeOut;
    I2C1_MESSAGE_STATUS status;
    uint8_t slave_out[33];
    int size_buf = 0;
    bool read_i2c  = false;
    bool tempC = true;
    uint8_t last_sensor = 0;
    uint8_t station = 0x18;
    while (1)
    {
        // check for station
        status = I2C1_MESSAGE_PENDING;
        if (uart_in >= 0x31 && uart_in <= 0x3F) {
            station = 0x18 + (uart_in - 0x31);
            uart_in = 0;
        }
        // check for input of uart
        if (uart_in != 0) {
            writeBuffer[0] = (dataAddress >> 8);                // high address
            writeBuffer[1] = (uint8_t)(dataAddress);            // low low address

            // data to be written
            writeBuffer[2] = uart_in;
            if (uart_in == 'o') {
                size_buf = 3;
                read_i2c  = true;
            } else if (uart_in == 'm') {
                size_buf = 33;
                read_i2c = true;
```

```
            } else {
                read_i2c = false;
                if (uart_in == 'C') {
                    tempC = true;
                } else if (uart_in == 'F') {
                    tempC = false;
                } else if (uart_in == 's' || uart_in == 't' || uart_in == 'c' || uart_in ==
'f') {
                    last_sensor = uart_in;
                }
            }


            // Now it is possible that the slave device will be slow.
            // As a work around on these slaves, the application can
            // retry sending the transaction
            timeOut = 0;
            while(status != I2C1_MESSAGE_FAIL)
            {
                // write one byte to EEPROM (3 is the number of bytes to write)
                I2C1_MasterWrite(  writeBuffer,
                                   3,
                                   station,
                                   &status);

                // wait for the message to be sent or status has changed.
                while(status == I2C1_MESSAGE_PENDING);

                if (status == I2C1_MESSAGE_COMPLETE)
                    break;

                // if status is  I2C1_MESSAGE_ADDRESS_NO_ACK,
                //               or I2C1_DATA_NO_ACK,
                // The device may be busy and needs more time for the last
                // write so we can retry writing the data, this is why we
                // use a while loop here

                // check for max retry and skip this byte
                if (timeOut == SLAVE_I2C_GENERIC_RETRY_MAX)
                    break;
                else
                    timeOut++;
            }

            if (status == I2C1_MESSAGE_COMPLETE && read_i2c)
                {

                    // this portion will read the byte from the memory location.
                    timeOut = 0;
                    while(status != I2C1_MESSAGE_FAIL)
                    {
                        // write one byte to EEPROM (2 is the count of bytes to write)
                        I2C1_MasterRead(    slave_out,
                                            size_buf,
                                            station,
                                            &status);

                        // wait for the message to be sent or status has changed.
                        while(status == I2C1_MESSAGE_PENDING);
```

```
                        if (status == I2C1_MESSAGE_COMPLETE)
                            break;

                        // if status is  I2C1_MESSAGE_ADDRESS_NO_ACK,
                        //               or I2C1_DATA_NO_ACK,
                        // The device may be busy and needs more time for the last
                        // write so we can retry writing the data, this is why we
                        // use a while loop here

                        // check for max retry and skip this byte
                        if (timeOut == SLAVE_I2C_GENERIC_RETRY_MAX)
                            break;
                        else
                            timeOut++;
                    }
                    uint16_t data = 0;
                    double val;
                    char output[50];
                    // parse the data from the slave
                    for (int i = 1; i<size_buf; i++) {
                        if (!(i%2)) {
                            data = data + ((uint16_t)slave_out[i] << 8);
                            if (last_sensor == 'c') {
                                val = send_car(data);
                            } else if (last_sensor == 's') {
                                val = send_sal(data);
                            } else if (last_sensor == 'f') {
                                val = send_flow(data);
                            } else {
                                val = send_temp(data);
                                if (tempC == false) {
                                    val = val * (9.0/5) + 32.0;
                                }
                            } // send to UART
                            sprintf(output, "%f", val);
                            sendUART(output);
                            data = 0;data = (uint16_t) slave_out[i];
                        } else {
                            data = (uint16_t) slave_out[i];
                        }

                    }
                    EUSART1_Write('\n');
                }

            //if (test_in == 1) {
                TEST_LED_Toggle();
            //}
            uart_in = 0;
        }
    }
}
```

Interrupt manager sample:

```
void interrupt INTERRUPT_InterruptManager (void)
{
    // interrupt handler
    if(INTCONbits.PEIE == 1 && PIE2bits.BCL1IE == 1 && PIR2bits.BCL1IF == 1)
```

```
    { // bus collision interrupt
        I2C1_BusCollisionISR();
    }
    else if(INTCONbits.PEIE == 1 && PIE1bits.SSP1IE == 1 && PIR1bits.SSP1IF == 1)
    { // i2c interrupt
        I2C1_ISR();
    }
    else if(INTCONbits.PEIE == 1 && PIE1bits.TX1IE == 1 && PIR1bits.TX1IF == 1)
    { // uart interrupt
        EUSART1_Transmit_ISR();
    }
    else if(INTCONbits.PEIE == 1 && PIE1bits.RC1IE == 1 && PIR1bits.RC1IF == 1)
    { //uart interrupt
        uart_in = RCREG1;
        if (uart_in == 'c') {
            TEST_LED_Toggle();
        }
    }
}
```

Remote Station

Main module:

```
/**
  Generated Main Source File

  Company:
    Microchip Technology Inc.

  File Name:
    main.c

  Summary:
    This is the main file generated using PIC10 / PIC12 / PIC16 / PIC18 MCUs

  Description:
    This header file provides implementations for driver APIs for all modules selected in
the GUI.
    Generation Information :
        Product Revision  :  PIC10 / PIC12 / PIC16 / PIC18 MCUs  - 1.45
        Device            :  PIC18F25K22
        Driver Version    :  2.00
    The generated drivers are tested against the following:
        Compiler          :  XC8 1.35
        MPLAB             :  MPLAB X 3.40
*/

/*
    (c) 2016 Microchip Technology Inc. and its subsidiaries. You may use this
    software and any derivatives exclusively with Microchip products.

    THIS SOFTWARE IS SUPPLIED BY MICROCHIP "AS IS". NO WARRANTIES, WHETHER
    EXPRESS, IMPLIED OR STATUTORY, APPLY TO THIS SOFTWARE, INCLUDING ANY IMPLIED
    WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A
    PARTICULAR PURPOSE, OR ITS INTERACTION WITH MICROCHIP PRODUCTS, COMBINATION
    WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.

    IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE,
```

```
    INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND
    WHATSOEVER RELATED TO THE SOFTWARE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS
    BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE
    FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN
    ANY WAY RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY,
    THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THIS SOFTWARE.

    MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR ACCEPTANCE OF THESE
    TERMS.
*/

#include "mcc_generated_files/mcc.h"
#include "lcd_drivers.h"
#include "sensors.h"
#include "frequency.h"
#include "sram.h"
#include "uart_interrupts.h"
//#include "usart.h"
//#include <stdbool.h>
/*
                         Main application
 */

/*
//interrupt XC8
void EUSART1_Receive_ISR(void) {
    unsigned char input;
    input = getc1USART();
    if(input >= ' ' && input <= '~') {
        if(SIZE == index || input == 'x') {
            for(int k = 0; k < SIZE; k++) {
                buffer[k] = NULL;
            }
            //if(index < SIZE) {
              //  buffer[index] = '\0';
            //}
            index = 0;
            //puts1USART(finish);
            //finish = 1
        } else if(input == 'z' && index > 0) {
            index--;
            buffer[index] = NULL;
        } else {
            buffer[index] = input;
            index = 0;
            //if(index == SIZE) {
              //  index = 0;
            //}
            //puts1USART(message);
            puts1USART(buffer);
            RS_SetLow();
            write_lcd(0x01);
            write_lcd(0x02);
            RS_SetHigh();
            write_lcd((unsigned char) buffer[i]);


        }
    }
}*/
```

```
void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    // If using interrupts in PIC18 High/Low Priority Mode you need to enable the Global
High and Low Interrupts
    // If using interrupts in PIC Mid-Range Compatibility Mode you need to enable the Global
and Peripheral Interrupts
    // Use the following macros to:

    // Enable high priority global interrupts
    INTERRUPT_GlobalInterruptHighEnable();

    // Enable low priority global interrupts.
    INTERRUPT_GlobalInterruptLowEnable();

    // setup
    master_in = 0;
    lcd_init();
    _delay(500000);
    sram_init();
    __delay_ms(1000);

    while(1) {
        // check the input from i2c global variable and change the lcd screen
        if (master_in == 's') {
            sendString(sal_buf);
            screen_shift = 0;
        } else if (master_in == 'f') {
            sendString(flow_buf);
            screen_shift = 1;
        } else if (master_in == 'c') {
            sendString(car_buf);
            screen_shift = 2;
        } else if (master_in = 't') {
            sendString(temp_buf);
            //TEST_LED_SetHigh();
            screen_shift = 3;
        } else if (master_in = 'y') {
            0();
        } else if (master_in = 'n') {
            TMR0_StopTimer();
        }
        master_in = 0;


    }
}
/**
 End of File
*/
```

Frequency module:

```c
#include "frequency.h"

void getFrequency() {
    FREQ_RST_SetHigh();
    __delay_us(10);
    //NCLR_SetLow();
    FREQ_RST_SetLow();
    //NCLR_SetHigh();

    GATE_SetHigh();
    __delay_ms(1000);
    GATE_SetLow();

    //SHIFT_SetHigh();

    SHIFT_SetLow();
    __delay_ms(1);
    SHIFT_SetHigh();
    FREQ_INH_SetLow();

    unsigned int freq1[8];
    unsigned int freq2[8];
    //SHIFT_SetHigh();
    for (int i = 0; i < 8; i++) {

        FREQ_CLK_SetHigh();
        __delay_ms(1);
        FREQ_CLK_SetLow();
        freq1[i] = FREQ1_GetValue();
        freq2[i] = FREQ2_GetValue();

    }
    FREQ_INH_SetHigh();
    unsigned int frequency = 0;

    for (int i = 4; i < 8; i++) {
        frequency += freq1[i] * pow(2,(i - 4));
    }

    for (int i = 0; i < 8; i++) {
        frequency += freq2[i] * pow(2,(i + 4));
    }

    double converted = frequency * 0.1;

    sprintf(flow_buf+16, "%f", converted);
    strcat(flow_buf,"\n");
}
```

LCD driver:

```c
#include "lcd_drivers.h"

void clockToggle() {
    LCD_CLK_SetLow();
    __delay_us(100);
    LCD_CLK_SetHigh();
    __delay_us(100);
    LCD_CLK_SetLow();
```

```
}

void enableToggle() {
    LCD_CLK_SetLow();
    __delay_us(100);
    LCD_EN_SetHigh();
    __delay_us(100);
    LCD_EN_SetLow();
    __delay_us(100);
}

void clear_screen() {
    sendCommand(0x00);
    __delay_us(150);
    sendCommand(0x01); //clear display
    __delay_us(17);
}

void temp_screen() {
    temperature();
    sendString(temp_buf);
}

void flow_screen() {
    getFrequency();
    sendString(flow_buf);
}

void sal_screen() {
    salinity();
    sendString(sal_buf);
}

void car_screen() {
    carbon();
    sendString(car_buf);
}

void sendCommand(unsigned char command) {
    unsigned char mask = 16;
    unsigned char pinout;

    for (int i = 0; i < 5; i++) {
        pinout = command & mask;
        if (pinout == 0) {
            LCD_SER_SetLow();
        } else {
            LCD_SER_SetHigh();
        }
        clockToggle();
        mask = mask >> 1;
    }

    clockToggle();
    enableToggle();
}

void lcd_init() {
    __delay_ms(16); //sleep 15 ms after power on
    sendCommand(0x03); //function set
```

```
    __delay_ms(5); //sleep more than 4.1 ms
    sendCommand(0x03); //function set
    __delay_us(150);
    sendCommand(0x03); //function set
    __delay_us(150);
    sendCommand(0x02); //function set
    __delay_us(150);

    sendCommand(0x02);
    __delay_us(150);
    sendCommand(0x08); //function set command:two lines
    __delay_us(150);

    sendCommand(0x00);
    __delay_us(150);
    sendCommand(0x08);
    __delay_us(150);

    sendCommand(0x00);
    __delay_us(150);
    sendCommand(0x01); //clear display
    __delay_us(17);

    sendCommand(0x00);
    __delay_us(150);
    sendCommand(0x06); //entry mode: increment mode off, shift operation on
    __delay_us(150);

    sendCommand(0x00);
    __delay_us(150);
    sendCommand(0x0F); // display on: yes cursor yes blinking

}

void temp_convert(unsigned char sign) {
    if (sign == 'C' & temp_buf[13] == 'F') {
        temp = temp * (9/5) + 32;
    } else if (sign == 'F' & temp_buf[13] == 'C') {
        temp = (temp - 32) * (5/9);
    }
    sprintf(temp_buf+16, "%6f", temp);
    sendString(temp_buf);
}

void sendChar(unsigned char letter) {
    LCD_SER_SetHigh();
    clockToggle();

    unsigned char mask = 128;
    unsigned char pinout;

    for (int i = 0; i < 4; i++) {
        pinout = mask & letter;
        if (pinout == 0) {
            LCD_SER_SetLow();
        } else {
            LCD_SER_SetHigh();
        }
        clockToggle();
        mask = mask >> 1;
```

```
    }
    clockToggle();
    enableToggle();

    LCD_SER_SetHigh();
    clockToggle();

    for (int i = 0; i < 4; i++) {
        pinout = mask & letter;
        if (pinout == 0) {
            LCD_SER_SetLow();
        } else {
            LCD_SER_SetHigh();
        }
        clockToggle();
        mask = mask >> 1;
    }
    clockToggle();
    enableToggle();
}

void sendString(char* string) {
    int i = 0;
    while ((string[i] != '\n')) {

        if (i == 16) {
            //shift cursor to next line
            sendCommand(0xc);
            __delay_us(150);
            sendCommand(0x0);
            __delay_us(150);
        }

        sendChar(string[i]);
        i++;
    }
}
```

Sensor module:

```
#include "sensors.h"


void salinity() {
    double converted = 0;

    ADC_Initialize();
    uint16_t val = ADC_GetConversion(SALINITY);
    write_sram_sal(val);
    //uint16_t val = 1024;
       //resolution of adc = 0.4/(2^10 - 1) = 0.00039
       for (int i = 0; i < 10; ++i) {
              unsigned int bit0 = (val >> i) & 1;
              converted += pow(2,i) * bit0;
       }
    converted = converted * 0.001;

    double salinity = 0;
    salinity = 225 * converted - 17.5;
```

```
    sprintf(sal_buf+16, "%f", salinity);
}

void carbon () {
    double converted = 0;

    ADC_Initialize();
    uint16_t val = ADC_GetConversion(CARBON);
    write_sram_car(val);
    for (int i = 0; i < 10; ++i) {
        unsigned int bit0 = (val >> i) & 1;
        converted += pow(2,i) * bit0;
    }
    converted = converted * 0.001;

    double carbon = 0;
    carbon = 1387.76 * converted + 3.1;

    sprintf(car_buf+16, "%6f", carbon);
}


void temperature () {
    double converted = 0;

    ADC_Initialize();
    uint16_t val = ADC_GetConversion(TEMPERATURE);
    write_sram_temp(val);
    for (int i = 0; i < 10; ++i) {
        unsigned int bit0 = (val >> i) & 1;
        converted += pow(2,i) * bit0;
    }
    converted = converted * 0.001;

    double temp = 275 * converted - 10.75;
    if (temp_buf[13] == 'F') {
        temp = temp * (9.0/5) + 32.0;
    }
    sprintf((void*)(&temp_buf+16), "%6f", temp);
    temp_buf[24] = ' ';
    temp_buf[25] = ' ';
    temp_buf[26] = ' ';
    temp_buf[27] = ' ';
    temp_buf[28] = ' ';
    temp_buf[29] = ' ';
}
```

SRAM module:

```
#include "sram.h"

void read_sram_temp(uint16_t * val) {
    read_sram_sensor(temp_ptr, val);
}

void read_sram_sal(uint16_t * val) {
    read_sram_sensor(sal_ptr, val);
}
```

```
void read_sram_car(uint16_t * val) {
    read_sram_sensor(car_ptr, val);
}

void read_sram_freq(uint16_t * val) {
    read_sram_sensor(freq_ptr, val);
}

void read_sram_sensor(uint8_t strAddr, uint16_t * val) {
    for (int i=0; i < 16; i++) {
        *(val+i) = read_sram_16(strAddr);
        strAddr = (strAddr & HEAD_MASK) | ((strAddr + 0x02) & (~HEAD_MASK));
    }
}

void write_sram_temp(uint16_t data) {
    write_sram_16(temp_ptr, data);
    temp_ptr = (temp_ptr & HEAD_MASK) | ((temp_ptr + 0x02) & (~HEAD_MASK));
}

void write_sram_car(uint16_t data) {
    write_sram_16(car_ptr, data);
    car_ptr = (car_ptr & HEAD_MASK) | ((car_ptr + 0x02) & (~HEAD_MASK));
}

void write_sram_sal(uint16_t data) {
    write_sram_16(sal_ptr, data);
    sal_ptr = (sal_ptr & HEAD_MASK) | ((sal_ptr + 0x02) & (~HEAD_MASK));
}

void write_sram_freq(uint16_t data) {
    write_sram_16(freq_ptr, data);
    freq_ptr = (freq_ptr & HEAD_MASK) | ((freq_ptr + 0x02) & (~HEAD_MASK));
}

uint16_t read_sram_16(uint8_t addr) {
    uint16_t val = 0;
    for (int i=1; i >= 0; i--) {
        val = val | ((uint16_t)read_sram(addr + (1 - i)) << (i * 8));
    }
    return val;
}

// write starting with most significant 8 bits
void write_sram_16(uint8_t addr, uint16_t data) {
    uint16_t mask = 0xFF00;
    for (int i=1; i >= 0; i--) {
        write_sram(addr + (1 - i), (uint8_t)(((uint16_t)data & mask) >> (i * 8)));
        mask = mask >> 8;
    }
}

void write_sram(uint8_t addr, uint8_t data) {
    uint16_t ser_out = ((uint16_t)data << 8) | addr;
    uint16_t mask = 0x8000;

    OE_SRAM_SetHigh();

    // Delay 1s
```

```
    __delay_ms(1);
    for (int i=0; i < 16; i++) {
        if (ser_out & mask) {
            S_ADR_SetHigh();
        } else {
            S_ADR_SetLow();
        }
        CLK_OUT_pulse();
        mask = mask >> 1;
    }
    CLK_OUT_pulse();
    WE_SRAM_pulse();
}

uint8_t read_sram(uint8_t addr) {
    uint8_t mask = 0x80;
    OE_SRAM_SetLow();
    CLK_INH_SetHigh();
    for (int i=0; i < 8; i++) {
        if (addr & mask) {
            S_ADR_SetHigh();
        } else {
            S_ADR_SetLow();
        }
        CLK_OUT_pulse();
        mask = mask >> 1;
    }
    CLK_OUT_pulse();

    __delay_ms(1);
    SH_LD_pulse();

    uint8_t val = 0x00;

    CLK_IN_SetHigh();
    CLK_INH_SetLow();
    for (int i=7; i >= 0; i--) {
        CLK_IN_SetHigh();
        __delay_ms(1);
        CLK_IN_SetLow();
        val = ((uint8_t)S_DI_GetValue() << i) | val;
        __delay_ms(1);
    }
    CLK_INH_SetHigh();
    return val;
}

void WE_SRAM_pulse() {
    WE_SRAM_SetHigh();
       __delay_ms(1);
       WE_SRAM_SetLow();
       __delay_ms(1);
       WE_SRAM_SetHigh();
       __delay_ms(1);
}

void CLK_OUT_pulse() {
    CLK_OUT_SetLow();
       __delay_ms(1);
       CLK_OUT_SetHigh();
```

```
        __delay_ms(1);
        CLK_OUT_SetLow();
        __delay_ms(1);
}

void SH_LD_pulse() {
    SH_LD_SetHigh();
        __delay_ms(1);
        SH_LD_SetLow();
        __delay_ms(1);
        SH_LD_SetHigh();
        __delay_ms(1);
}

void sram_init() {
    SH_LD_SetHigh();
    OE_SRAM_SetHigh();
    WE_SRAM_SetHigh();
    CLK_INH_SetLow();
}
```

This report and its contents are solely work of the participants below.

Denis Jivaikin

Sandy Lee

Jiayou Zhao