**EE 478 Lab 2**

**Designing a High Reliability Microprocessor Based Remote Surgery System**

University of Washington – Department of Electrical Engineering

Ryan McDaniel Alyanna Castillo Patrick Ma

TABLE OF CONTENTS

[**I.** **ABSTRACT** 4](#_Toc387024558)

[**II.** **INTRODUCTION** 4](#_Toc387024559)

[**III.** **SYSTEM REQUIREMENTS** 4](#_Toc387024560)

[**a.** **Use Cases** 4](#_Toc387024561)

[**b.** **Requirements** 6](#_Toc387024562)

[System Description 6](#_Toc387024563)

[Specification of External Environment 6](#_Toc387024564)

[System Input and Output Specification 6](#_Toc387024565)

[User Interface 7](#_Toc387024566)

[System Functional Specification 7](#_Toc387024567)

[Operating Specifications 7](#_Toc387024568)

[Reliability and Safety Specification 7](#_Toc387024569)

[**IV.** **DESIGN SPECIFICATION** 8](#_Toc387024570)

[System Description 8](#_Toc387024571)

[Specification of External Environment 8](#_Toc387024572)

[System Input and Output Specification 8](#_Toc387024573)

[User Interface 8](#_Toc387024574)

[System Functional Specification 9](#_Toc387024575)

[Operating Specifications 9](#_Toc387024576)

[Reliability and Safety Specification 10](#_Toc387024577)

[**V.** **DESIGN PROCEDURE** 10](#_Toc387024578)

[**VI.** **SYSTEM DESCRIPTION** 10](#_Toc387024579)

[**VII.** **HARDWARE IMPLEMENTATION** 10](#_Toc387024580)

[a. Local Node 10](#_Toc387024581)

[**SRAM Interface** 10](#_Toc387024582)

[**External Clock** 11](#_Toc387024583)

[**RS232 Communications** 11](#_Toc387024584)

[**I2C Communication** 12](#_Toc387024585)

[b. Remote Node 12](#_Toc387024586)

[**SRAM Interface** 12](#_Toc387024587)

[**ADC Measurements** 12](#_Toc387024588)

[**PWM Output** 12](#_Toc387024589)

[**I2C Communication** 12](#_Toc387024590)

[**VIII.** **SOFTWARE IMPLEMENTATION** 13](#_Toc387024591)

[a. Local Node 13](#_Toc387024592)

[SRAM Control 16](#_Toc387024593)

[RS-232 Communication 17](#_Toc387024594)

[I2C Communication 17](#_Toc387024595)

[b. Remote Node 18](#_Toc387024596)

[SRAM Control 18](#_Toc387024597)

[ADC voltage measurement 18](#_Toc387024598)

[PWM output 19](#_Toc387024599)

[I2C Communication 19](#_Toc387024600)

[**IX.** **TESTING** 19](#_Toc387024601)

[a. Test Plan 19](#_Toc387024602)

[b. Test Specification 20](#_Toc387024603)

[**c.** Test Cases 20](#_Toc387024604)

[**X.** **RESULTS** 20](#_Toc387024605)

[a. Presentation 20](#_Toc387024606)

[b. Discussion 20](#_Toc387024607)

[c. Analysis 20](#_Toc387024608)

[**XI.** **ANALYSIS OF ERRORS** 20](#_Toc387024609)

[**XII.** **WHY PROJECT DIDN’T WORK** 20](#_Toc387024610)

[**XIII.** **SUMMARY** 20](#_Toc387024611)

[**XIV.** **CONCLUSION** 20](#_Toc387024612)

[**XV.** **APPENDIX** 20](#_Toc387024613)

[a. Charts and Tables 20](#_Toc387024614)

[b. Graphs 20](#_Toc387024615)

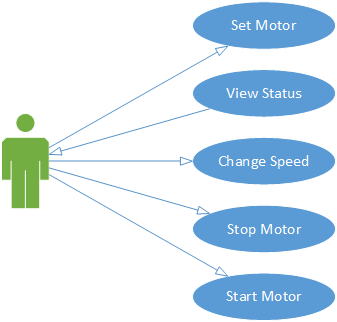
[c. Code 20](#_Toc387024616)

1. **ABSTRACT**

This lab provides experience in using the MPLabX development environment for programming the PIC18F25K22 microchip

1. **INTRODUCTION**
2. **SYSTEM REQUIREMENTS**
   1. **Use Cases**

The use cases are presented in Figure 1.



**Figure 1: Use Case Diagram for System**

Use Case #1: Set motor speed

User Actions:

- enters command to set motor speed

- enters a speed value

System:

Normal Operation:

- Processor converts user value to motor-relevant value

- Processor stores motor-relevant value to memory

- Processor/controller operate motor at set speed

Exception operation:

- user inputs inappropriate speed

- Stop motor

- indicate to user speed is not appropriate

- motor speed increases or decreases significantly

- see ERRORS

Use Case #2: Change speed

User Actions:

- enters command to increment or decrement speed

System:

Normal Operation:

- Processor calculates new speed

- New speed stored in memory

- processor operates motor at new speed (see normal operation of Use Case #1)

Exception Operation:

- Changing speed would put motor speed beyond acceptable range

- Do not change speed

- Inform user that system is at max or min of range

- Speed change causes motor to enter error range

- see ERRORS

Use Case #3: Start motor

User Actions:

- enters command to start motor

System:

Normal Operation:

- Processor calculates motor control values based on speed value stored in memory

- Processor sets motor status to "running"

- Controller determines motor feedback

- Processor calculates error, updates motor control value

Exception Operation:

- User has not defined a speed

- Alert user speed is not defined

- Stop motor

- Old speed in memory from previous usage; not intended user speed

- At system start up or reset, set initial speed to zero

- Between resets, assume input value is intended by user, continues as normal

Use Case #4: Stop Motor

User Action:

- enters command to stop motor

System:

Normal Operation:

- Processor sets motor status to "stopped"

- Processor turns off motor

Exception Operation:

- Sudden decrease in speed triggers ERROR situation

- ignore error, consider error normal

Use Case #5: View status

User Action:

- enters command to view motor and system status

System:

Normal Operation:

- Processor display motor set speed, current status and error state

Exception Operation:

- None expected

ERRORS:

- Level 0: +\- 5.0% difference between expected and actual motor speed (severe)

- Level 1: +\- 2.0% difference between expected and actual motor speed (moderate)

- Level 2: +\- 1.0% difference between expected and actual motor speed (mild)

* 1. **Requirements**

### System Description

This specification covers the requirements of the embedded circuitry for a microprocessor-based remote surgery system. The system is a prototype and proof of concept for a larger system. The circuitry must be able to communicate between a local node, connected to a computer, and a remote node, connected to a surgery robot motor. A remote node must be able to maintain motor speed, and the local node must be able to communicate with a PC to send information and receive commands. The product is meant to be developed in two phases. The first phase would be to implement the local and remote communication networks, and the user interface. The second phase will include adding support for the remote feedback channel through which warnings, data, and alarms are sent.

### Specification of External Environment

The unit is meant to be used in the medical environment. Concerns will be blood and corrosion, water, gasses, and sterilization. The system may have to operate in a frequency that does interfere with other medical devices, or produce any gasses that would contaminate a sterile environment. Finally, the system should not have a negative effect on the power draw of the environment.

### System Input and Output Specification

#### System Inputs

User commands from PC terminal output.

Set motor speed

Increment motor speed

Decrement motor speed

Start motor

Stop motor

Motor voltage level

Power On/Off

#### System Outputs

Voltage control to motor

Display to PC Terminal

Warnings

Alarms

Current motor speed

### User Interface

The user interface will be a command line presented through a terminal window. The user can enter the following commands:

Set speed:

Sets the speed of the motor to a specified value

Increment speed:

Increments the current motor speed.

Decrement speed:

Decrements the current motor speed.

In addition to the command line interface, the system will have an ON/OFF button for power.

### System Functional Specification

The system takes and executes user commands from the PC terminal with the intent of remotely operating motor speed through a network. These commands are turning the motor on/off, setting the motor speed, and increasing and decreasing the motor speed.

The system has two main parts – the local and remote node. Both nodes communicate with each other and have a separate memory to store data. The user interface (PC terminal) is connected to the local node, which processes user input. The commands are then sent to the remote node to control the motor, which in turn measures the current status and sends it back through the local node to the PC.

### Operating Specifications

The system shall operate in a sterile medical environment.

Temperature Range

Humidity Range

Power

### Reliability and Safety Specification

The robot surgery system shall comply with the following safety standards

* Meet government regulations regarding medical devices as outlined by the FDA’s CFR (Code of Federal Regulations)
* Continue to safely function in the absence of power (blackout, etc.)
* All outer parts of must be sterilized before use

1. **DESIGN SPECIFICATION**

### System Description

This specification covers the embedded circuitry for a microprocessor based remote surgery system. The system is a prototype and proof of concept for a larger system. The circuitry must be able to communicate between a local node, connected to a computer, and a remote node, connected to the surgery robot motor, through an I2C connection. A remote node must be able to maintain motor speed by controlling the voltage applied to the motor. The local node must be able to communicate with a PC through an RS232 serial connection so that it can receive commands and send data to be displayed. The product is meant to be developed in two phases. The first phase would be to implement the local and remote communication networks, and the user interface. The second phase will include adding support for the remote feedback channel through which warnings, data, and alarms are sent.

### Specification of External Environment

The unit is meant to be used in the medical environment. Concerns will be blood and corrosion, water, gasses, and sterilization. The system must operate at a frequency that does interfere with other medical devices, and cannot produce any gasses that would contaminate a sterile environment. Finally, the system must not have a negative effect on the power draw of the environment.

### System Input and Output Specification

#### System Inputs

User commands from PC terminal output.

Set motor speed

Increment motor speed in 0.5% intervals

Decrement motor speed in 0.5% intervals

Motor voltage level with range from 0 – 5VDC

Power On/Off achieved with a SPDT reset switch

#### System Outputs

PWM voltage signal to motor with frequency of 20kHz, duty cycle variable from 0% -100%.

Display to PC Terminal

Warnings

Alarms

Current motor speed as a percentage

### User Interface

The user interface will be a command line presented through a terminal window. The user can enter the following commands:

Set speed (S):

Sets the speed of the motor to the specified value.

Increment speed (I):

Increments the current motor speed by 0.5%.

Decrement speed (D):

Decrements the current motor speed by 0.5%.

In addition to the command line interface, the system will have an ON/OFF button for power.

### System Functional Specification

The system takes and executes user commands from the PC terminal with the intent of remotely operating motor speed through a network. These commands are turning the motor on/off, setting the motor speed, and increasing and decreasing the motor speed in 0.5% increments.

The system has two main parts – the local and remote node. Both nodes communicate with each other through an I2C connection and have a separate SRAM to store data. The user interface (PC terminal) is connected to the local node.

User commands are processed in the local node and sent to the remote node. User input is received by the local node through an RS232 connection. Before being sent to the remote node, the local node checks that the user input is a valid command.

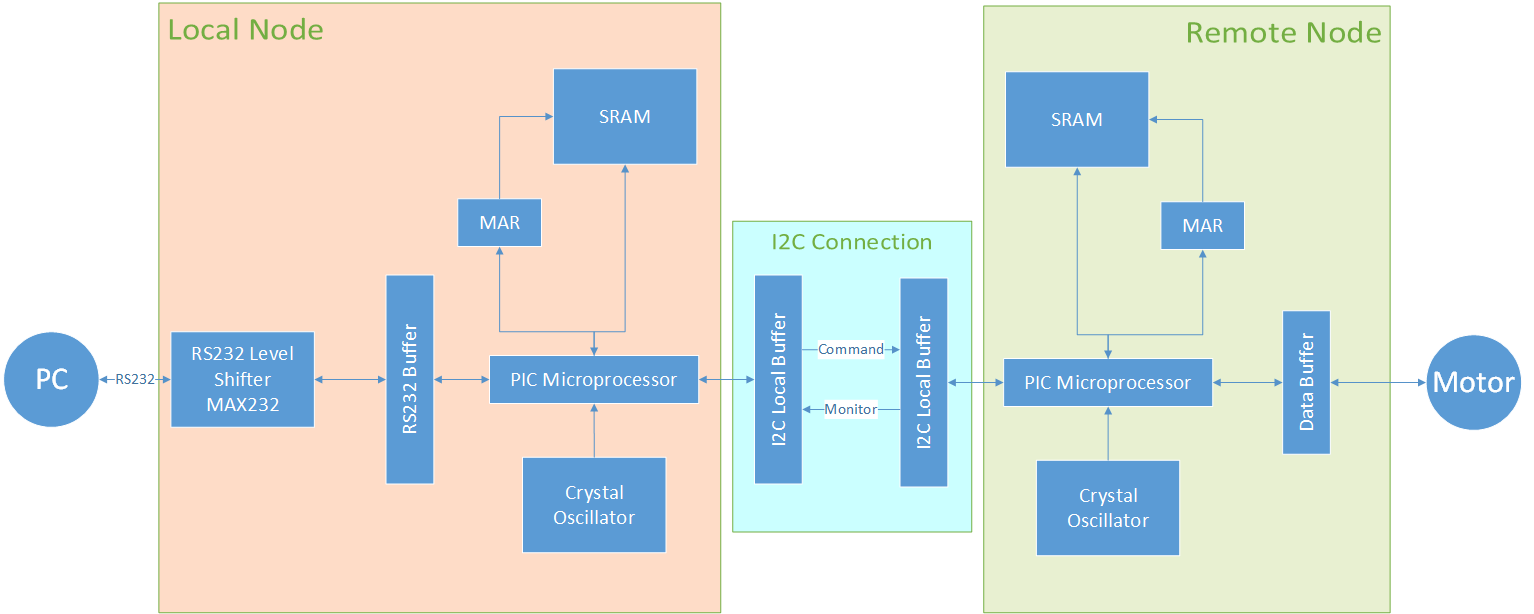
Valid commands are received by remote node through an I2C connection and executed. Depending on the command, the motor speed will be adjusted appropriately. The remote node also monitors the error state of the system and measures the current motor speed. The error state is calculated if any of the motor speeds go beyond the following ranges:

* ±5.0% - Level 0 - severe
* ±2.0% - Level 1 - moderate
* ±1.0% - Level 2 – of concern

The error state is sent back to the local node through the same I2C connection.

The local node then receives data on the current motor speed and error state from the remote node – this data is stored in the local node’s SRAM. The data is then read from the SRAM and outputted through an RS232 connection to the PC terminal, making it viewable to the user.

The following block diagram illustrates the high level functions of the system.

****

### Operating Specifications

The system shall operate in a sterile medical environment.

Temperature Range 19-23 C

Humidity Range is 20-60%

Power must have a supply voltage of 5V

### Reliability and Safety Specification

The robot surgery system shall comply with the following safety standards

* Meet government regulations regarding medical devices as outlined by the FDA’s CFR (Code of Federal Regulations)
* Continue to safely function in the absence of power (blackout, etc.)
* All outer parts of must be sterilized before use
* Properly alert personnel of any error states and take correct course of action

1. **DESIGN PROCEDURE**
2. **SYSTEM DESCRIPTION**
3. **HARDWARE IMPLEMENTATION**

## Local Node

The Local Node consists of the following hardware:

|  |  |  |  |
| --- | --- | --- | --- |
| Part Number | Description | Quantity | Price |
| PIC18F25K22 | 28-pin microcontroller | 1 |  |
| OSC-20MHz | Crystal oscillator | 1 |  |
| CY7C128A | SRAM | 1 |  |
| GAL22V10D | Programmable logic chip | 1 |  |
| MAX232 | RS232 level shifter | 1 |  |
| 1 uF Capacitors | ---------------------------- | 5 |  |

The PIC18 microcontroller is the central unit of the local node. Every other part in the node is connected to and controlled by the PIC18.

**SRAM Interface**

The following diagram shows the pin assignments for the SRAM interface.

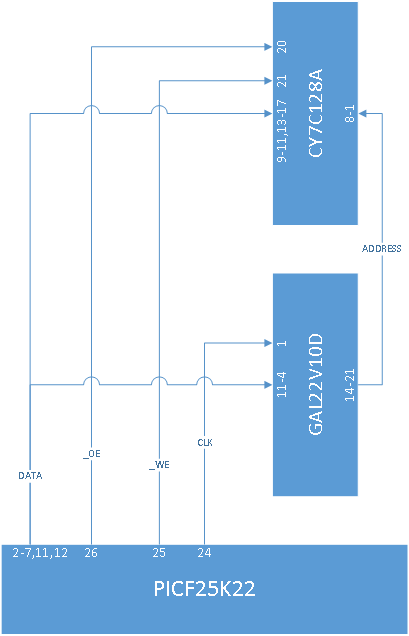


Figure 2 - Hardware block diagram for the SRAM interface

Pins 2-7, 11, and 12 of the PIC18 are connected to pins 11-4 of the GAL chip, and to the I/O ports of the SRAM on pins 9-11, and 13-17. Pin 2 on the PIC18 is the LSB of the data or address, while pin 12 is the MSB. Pins 14-21 on the GAL chip are connected to the address lines of the SRAM on pins 8-1. There are tristate drivers in both the SRAM and the PIC18 that control whether data is going into the SRAM and out of the PIC18, or if data is coming from the PIC18 and going to the GAL or SRAM. On the PIC18, these tristates are controlled by software and there is one for each I/O pin on the PIC18. On the SRAM, the tristates are controlled by the status of the Output Enable input signal.

The SRAM and GAL chip are controlled by three signals coming out of the PIC18 on pins 24-26. Pin 24 is the clock signal for the GAL chip, which connects to pin 1 of the GAL. Pin 25 is the write enable signal for the SRAM, which is sent to the SRAM pin 21. Pin 26 of the PIC18 is the SRAM output enable signal, which connects to the SRAM pin 20. The last two signals for the SRAM and GAL are the GAL reset signal, which was connected to ground, and the SRAM chip enable signal, which is also connected to ground. The status of these signals is controlled by software on the PIC18.

**External Clock**

The 20MHz crystal oscillator is the primary clock for the PIC18 microcontroller. The output pin of the clock goes into pin 6 of the microcontroller, and the pin is configured with software to allow the clock to be used as the system clock.

**RS232 Communications**

Communication to a PC via a serial communications port is accomplished using the PIC18’s USART interface and a MAX232 level shifter. Pin 25 of the PIC18 is the RS232 transmit pin, and connects to pin 11 on the MAX232. Pin 26 of the PIC18 is the RS232 receive pin, and connects to pin 12 on the MAX232. The receive pin of the RS232 cable connects to pin 14 of the MAX232, and the transmit pin connects to pin 13 of the RS232. The MAX has capacitors connected between pins 1 and 3, 4 and 5, 16 and ground, 2 and ground, and ground and pin 6.

**I2C Communication**

An I2C connection is established between the local and remote nodes by connecting pins 11 and 12 on the local PIC18 to pins 19 and 20 on the remote PIC18. The local node is the master of the control channel, where the local node is sending information about the current motor speed to the remote node. The I2C protocol is controlled by software.

## Remote Node

The Remote Node consists of the following hardware:

|  |  |  |  |
| --- | --- | --- | --- |
| Part Number | Description | Quantity | Price |
| PIC18F25K22 | 28-pin microcontroller | 1 |  |
| OSC-20MHz | Crystal oscillator | 1 |  |
| CY7C128A | SRAM | 1 |  |
| GAL22V10D | Programmable logic chip | 1 |  |

The PIC18 microcontroller forms the central unit of the remote node. Every other part in the node is connected to and controlled by the PIC18.

**SRAM Interface**

Please see the local node section about the SRAM Interface, both the local node and remote node use the exact same interface.

**ADC Measurements**

Pin 10 of the PIC18 is used as input for the ADC which measures the actual motor voltage. The motor is simulated with an RC low-pass filter circuit consisting of a 10k Ohm resistor and a 220nF capacitor. The cutoff frequency is approximately 750Hz.

**PWM Output**

Pin 21 of the PIC18 is a PWM output that controls the motor. It is configured by software to output a 20 kHz modulated square wave.

**I2C Communication**

An I2C connection is established between the remote and local nodes by connecting pins 11 and 12 on the remote PIC18 to pins 19 and 20 on the local PIC18. The remote node is the master of the monitor channel, where the remote node is sending information about the current measured motor speed and warning state to the local node. The I2C protocol is controlled by software.

1. **SOFTWARE IMPLEMENTATION**

The software for this project was implemented on two different PIC microcontrollers, one for the local node, and one for the remote node. They both ran on flag-based schedulers with interrupts. This means that each node was running in an infinite loop and calling tasks to be run while in the loop. Those tasks would only run if certain flags were set to true. Interrupts, like keyboard input from the RS232 connection, could set flags as well. After running, each task would set its own flag to false to prevent itself from being run again.

## Local Node

**Scheduler**

The scheduler runs the following tasks in this order:

Display Front Panel, Process Input, SRAM, I2C Communications

Each of these tasks is controlled by a flag in the Global Data structure that is available at all times to each task. If that task’s particular flag is set, it will be run the next time the program reaches it in the loop. By default, all flags are false except for the flag for Display. Display will show an interface for a user to read through an RS232 connection, and instruct a user to enter a command (set point, increment, decrement). It will also display error information if the remote node is not in a safe state.

When the user enters an command through RS232, an interrupt is handled by the RS232 interrupt handler, rsISR(), which will set the Process Input flag to be true. The process Input flag can also be set if the local I2C interrupt, i2cISR(), receives an error from the remote node. The Process Input task determines if the user’s input is valid and interprets the command and data for the rest of the system to use. After running, it will disable its own flag and set the flag for the SRAM to run.

The SRAM task will write the current command into a fixed address in the SRAM. In the case that the user is setting a speed to run at, the speed that the user entered will also be written to the SRAM but in a different set address. After running, the I2C Communications flag is set. The I2C communications task will read the current speed from the SRAM and set the local debug PWM signal to that speed, as well as send it over the I2C connection to the remote node. After running, the flag is set to false and the Display task is set to be run again and await more user input.

**Interrupts**

The local node has two interrupt handlers, rcISR() and i2cISR(). rcISR() is run whenever there is input from the RS232 connection, and i2cISR() is run whenever there is data coming through the from the remote node through the system’s monitor channel.

rcISR() is a high priority interrupt, and it is set to run whenever the PIC’s RCIF flag in the PIR1 register is true. The interrupt will echo any valid characters that the user types back to the terminal so that the user sees what they are typing. Valid characters are all the characters of the alphabet, spaces, return carriages, new line characters, and backspaces.

If a regular alphabet character is typed, it is echoed to the terminal using the putc1USART() function, and placed in the global data’s myInput[] array. The current “spot” in the array is the incremented so that the character is not overwritten when something new is typed. If the backspace key is entered, the character in the current spot of the array is replaced with a null terminator, “\0”, and the current spot is decremented. The backspace character is printed to the terminal as well to show the user that they are typing over an old character now. Finally, if the input is a return carriage, new line, or the input is getting too large, myInput[] is terminated with a “\0”, the input spot is reset, and the global data’s Process Input flag is set to true. This occurs when the user pressed the enter key, or spams the console window. After interrupt completes, the RCIF flag is set to 0.

i2cISR() is a low priority interrupt that is set to run whenever the SSP2IF flag in the PIR3 register on the PIC is asserted. When running, the purpose of the interrupt is to store two values sent over the I2C connection. The first is the measured speed of the motor, and the second is the current error state. The rest of the software implementation for this interrupt handler is outlined in the Monitor channel of the I2C Communications section.

**Display Front Panel**

The display task takes a single input, a pointer to the global data structure called globalData. If the flag for display is true, then the task will be run, otherwise it will just skip to the end of the task. If the flag is true, then it is immediately set to false so that display is not run a second time.

The first thing the display task will do is clear the current screen by printing the character “0x0C” through the RS232 connection with the putc1USART() command. After that, the current motor speed is read out from the globalData structure and displayed. The integer is converted into a string by using the itoa() C-library function. The speed is stored as a value from 0-200, where each increment represents 0.5% speed, so the raw value is divided by 2 before it is converted. If the speed is an odd number, a 0.5 is displayed after the converted string. The string created by the sprintf() function is displayed on the PC terminal using the puts1USART() function. The measured data is displayed next in the very same way.

Next, the user is prompted to enter input. The putrs1USART() function is used to directly display strings onto the PC Terminal. After displaying the input prompt, the display task accesses the current error state and uses a switch-case statement to display an appropriate error. The error states range from 0-2 where 0 is the most severe. In each state, an appropriate message is printed and the Process Input flag is set to true. An error state of 3 means that the remote system is off and an appropriate message will be displayed. If the error state is 4, all systems are operating correctly. If the error state is anything other than 0-4, then there must be a problem with the connection between the remote and local nodes, and an appropriate message will be displayed.

**Process Input**

The Process Input task takes a single input, a pointer to the global data structure called globalData. If the flag for Process Input is true, then the task will be run, otherwise it will just skip to the end of the task. If the flag is true, then it is immediately set to false so that display is not run a second time.

The Process Input task will only accept the following commands:

“s [number in %]”, “i”, “d”.

These stand for Set Point (speed), increment, and decrement respectively. The number accepted in the set point command must be between 0 and 101, and the increment and decrement commands will do nothing if they attempt to move the current speed past those points. 101% is set as the high limit purely for testing the remote node’s response to errors, and would not be allowed in the final system.

The Process Input task will take information stored in globalData’s myInput[] array, which is modified by the RS232 interrupt handler to store what the user typed in the terminal. If this array starts with the character “s” followed by a space, then the user wants to set the current speed of the motor. globalData’s myCommand field is set to “1” to represent the set point command, and the number that the user typed is processed using the stringToNum() helper function.

stringToNum() accepts a null terminated character array as an input, and will return an integer representation of that string. It does so by processing the string one character at a time until the null terminator character, “\0”, is reached. As it traverses through the string, each character has the ‘0’ character subtracted from it to turn the character into an integer. This works because the ASCII characters for ‘0’ through ‘9’ are represented in hex as 30-39. As each character is converted, it is added to a sum. When the next character is processed, that sum is multiplied by 10 before adding the new value to the sum.

The stringToNum() function will return an integer in percent that the user entered. However, that number is not precise enough to handle the 0.5% precision that increment and decrement both need. So, the percent that the user typed is multiplied by 2 in order to make each number represent 0.5%. After converting the user’s number, it is checked to be within the limits of 0 to 101% (or 0 to 202 in converted integers). If it is within the bounds, then globalData’s setSpeed field is set to the integer representation of the user’s input. If not, globalData’s myCommand field is set to 4 to represent an error.

If the first character of myInput[] is an ‘i’ or ‘d’ character, then myCommand is set to 2 or 3 to represent increment and decrement, respectively.

After processing the user’s input, the SRAM task’s flag is set to true so that the processed input can be stored into the SRAM.

### SRAM Control

If the SRAM flag is set to true, then the system will go into a switch-case statement based on the global data’s myCommand variable. The key for the myCommand variable is as follows:

|  |  |
| --- | --- |
| **myCommand** | **Meaning** |
| 1 | User entered set point as the command |
| 2 | User entered increment as the command |
| 3 | User entered decrement as the command |
| 4 | User entered an error as the command |

In any case, the current state of myCommand is written into address 0 on the SRAM using the writeData() function. If the user chose to set the speed, then the current value of the global data structure’s setSpeed variable is written into address 1 of the SRAM using the writeData() function. If the user decided to increment the current speed, then address 1 is read from the SRAM using readData(), and then the number 1 is added to it. Because the numbers are stored from 0 to 200, adding 1 increases the current value by 0.5%. The incremented number is stored back in the SRAM at address 1 using the writeData() function. The same thing is done for the decrement command, except 1 is subtracted from the current value instead.

In the case that the user wants to increment or decrement the current value, the result of adding or subtracting is checked against the bounds of 0-101% before writing back to the SRAM. If the value is out of range, then the current value will be rewritten into address 1 of the SRAM.

If the user entered a bad command for input, then a short message is printed to the terminal and the Display Front panel task is set to be run.

If there are no errors, the I2C flag in the global data structure is set to true.

The writeData() and readData() functions set various pins on the PIC to high and low in a sequence that allows reading and writing to the SRAM. Pins A0-A5, and C0-C1 are used to send out data and address information, and pins B3-B5 send out control information. The actual hardware pin numbers for these are in the hardware implementation section. B3 is the clock for the GAL chip programmed with a 10-bit register, B4 is the write enable signal for the SRAM, and B5 is the output enable signal for the SRAM.

The writeData() function takes an integer address as an input, along with an integer for data to write. To write data, first the address is set on pins A0-A5, and C0-C1. This is done by shifting the given integer for the address to the right 0 to 7 times, and using a bitwise and operation with the number 1. The result of this logic operation is stored on the output pins with pin A0 as the LSB and C1 as the MSB, where C1 is has the address shifted 7 times. That value is stored in the GAL register by setting RB3 to 0 and then back to 1. The same process for setting the address on the output pins is used to set the data. After that, the write enable pin on RB4 is set to 0 and then 1. After this, data has been successfully written to the given address on the SRAM.

The readData() function accepts only an address as input that designates where in the SRAM data is being read from. The address is set on the output pins in the same method that is described for writeData(). The tristate drivers are set up for input after the address is stored in the GAL register by using the setUpIn() function described in the Setup section. Next, pin RB5 is set to 0 to allow output from the SRAM. After a 10 clock cycle delay, the first 6 bits of port A (A0-A5) are combined with the first 2 bits of port C (C0-C1) using a bitwise or operation. The result is returned after disabling output from the SRAM.

### RS-232 Communication

Details of the RS-232 software implementation are provided in the Interrupts and Process Input sections above.

### I2C Communication

The hardware configuration creates two separate I2C communication channels, greatly simplifying the software implementation. Just like the hardware setup, the software is divided into an outgoing and incoming channel.

#### Outgoing –Control Channel

The outgoing channel sends the speed defined by the user as an 8-bit word. The first Master Synchronous Serial Port (MSSP1) is configured as a master with a baud rate of 400 kHz and no slewing. All control of the MSSP1 is done using a simple polling scheme with no interrupts. After waiting for an idle connection, the local node begins communication according to the I2C protocol. The slave address (set to 0x00 for convenience) and a write instruction are sent followed by the speed. The local node then closes the connection with a STOP signal.

#### Incoming – Monitor Channel

The incoming channel utilizes an interrupt-based scheme because the arrival of incoming data is arbitrary. Incoming data is received through Master Synchronous Serial Port 2 (MSSP2), which is configured as a slave device. When an address is sent on the monitor channel, the MSSP2 module compares it to the slave address (0x00 chosen again for convenience). If the address matches, an interrupt flag is asserted and the interrupt handler runs. The interrupt handler takes one of three actions depending on the state of the communication. If the data is an address, the interrupt handler clears the address from the MSSP2 data buffer, deasserts the interrupt flag and exits. If the data is actual information, the interrupt handler checks which bye of the sequence it is expecting. If it is the first byte, the MSSP2 data buffer contents correspond to the motor speed; otherwise the ISR assumes the data is an error state. For each option, the data is saved to the appropriate global variable, the byte counter adjusted appropriately, the interrupt flag is deasserted, and the display flag is set.

## Remote Node

### SRAM Control

The remote node SRAM control software is identical to the local node software. See the local node SRAM control section for details.

### ADC voltage measurement

The analog to digital module measures the actual voltage across the motor. The setup for the ADC involves configuring the ADC clock source, input channel, and acquisition time. These parameters are defined as follows.

* ADC clock source is 1/64th the primary oscillator frequency
* ADC acquisition time is 12 time periods
* Input channel is channel 14 (Pin 10)
* ADC interrupt is disabled
* ADC result is right justified
* ADC voltage references are the internal voltage references (5V and GND)

Since no interrupts are used, the microcontroller polls the ADC status flag until a computation is complete. The result is read into a temporary register and converted to a zero to 200 scale by multiplying the result by 50, bit shifting to the right by 8 and adding 1. To reduce the unnecessary memory writes, and because the precision of the ADC is greater than the noise from the input signal, the current motor speed is only updated if the new measurement is 1% greater or less than the value for the actual speed held in memory. If this is the case, a flag is set to send this value to the local node.

### PWM output

The pulse width modulation (PWM) output has only a minimal software implementation –the majority of the work is handled by the hardware module. Within software, the PWM has two primary components: setup and adjustment of the duty cycle. Setup occurs once each time the microcontroller is reset but adjustment occurs continuously while the PIC18 is running. When PWM setup occurs, the following registers are set:

* PR4 is set for 20kHz frequency
* T4CON is set for a 1:1 pre-scaling factor
* CCP4CON is set to select PWM mode
* CCP4CON and ccPR4L are set to an initial 50% duty cycle
* The Timer4 interrupt flag is cleared
* The GPIO pin B4 (pin 21) is configured as an output

To adjust the PWM duty cycle, the PIC18 calls the compiler function SetDCPWM4(). To pass an appropriately scaled value to the PWM module, the global control variable is multiplied by 5, giving a range from 0 to 1023. The PWM duty cycle is adjusted each time the main program loop is executed.

### I2C Communication

Much of the remote node I2C software operates using the same code as the local node. For a general description of operation refer to the local node section. Specifics for the remote node are provided below.

#### Outgoing –Monitor Channel

The outgoing channel sends two bytes of data to the local node. The first byte encodes the actual motor speed following the coding convention used throughout the system. The second byte contains the error state encoded as an integer. The MSSP1 module is configured just like the local node using the same polling scheme.

#### Incoming – Control Channel

The incoming channel utilizes an interrupt-based scheme in the same manner as the local node. However, since the remote only receives a single byte containing the set speed, the interrupt service routine merely check whether the MSSP data buffer contains an address or actual data: actual data is stored to the set speed global variable; an address is simply cleared from the buffer and discarded.

1. **TESTING**
   1. Test Plan
   2. Test Specification
   3. Test Cases
2. **RESULTS**
   1. Presentation
   2. Discussion
   3. Analysis
3. **ANALYSIS OF ERRORS**
4. **WHY PROJECT DIDN’T WORK**
5. **SUMMARY**
6. **CONCLUSION**
7. **APPENDIX**
   1. Charts and Tables
   2. Graphs
   3. Code