EE 475 Lab 2

Katie Neff <1168464>
Vishesh Sood <1239599>
Sean Happenny <1229085>

September 6, 2017

We affirm that this report and its contents are solely the work of Katie Neff, Vishesh Sood, and Sean Happenny.

Signature	Date
Signature	Date
Signature	Date

Contents

List of Figures

List of Tables

List of Listings

1 ABSTRACT

This lab involved developing a data collection rover system using two PIC18F25K22 microcontrollers. The rover collects data from three discrete sensor packages and stores the data in buffers implemented in SRAM. The rover sends this collected data to a surface ship over a synchronous connection, which temporarily stores it before sending it to a land station over an asynchronous connection. The land station is able to send commands to the rover via the surface ship to start or stop data collection. Additionally, the data uploads to the surface ship and land station only begin once the ship or station has given permission to upload.

2 INTRODUCTION

In this lab, we learned how to use a PIC18F25K22 microcontroller, Serial Peripheral Interface Bus (SPI), and RS232 serial communications over a Universal Synchronous/Asynchronous Receiver/Transmitter. Additionally, we gained more practice with our oscilloscope/logic analyzer and reading and writing a CY7C128A 2KB SRAM. The logic analyzer was especially useful as we discovered how to use its built-in SPI and UART communications monitoring tool to make debugging these systems much easier. On the PIC, we learned to use its SPI and USART modules, Analog to Digital Converter (ADC), and General Purpose Input/Output (GPIO). Further, we learned how to program the PIC using a PICKit 3 programming interface and the MPLAB-X Integrated Development Environment (IDE) and compiler.

3 REQUIREMENTS SPECIFICATION

3.1 System Description

This specification describes and defines the high-level requirements and constraints of an undersea earthquake data rover and a surface communication ship. The rover will collect data on earthquake hazards, ground shaking, geographic faults, and plate motion and stress. It will upload this information asynchronously to the surface ship, which will relay that information synchronously to one or more ground stations around the world. This specification covers the full system involving the rover, surface ship, and land station development that occured over Phase 1 and Phase 2.

3.2 Specification of External Environment

The rover will operate on the sea floor and will face significant pressures, low temperatures, and saltwater. The surface ship will operate on the ocean surface and will face varying weather conditions and saltwater. Additionally, the rover should not be too heavy so that it can be lowered from the ship and pulled back up to it.

3.3 System Input and Output Specification

3.3.1 System Inputs

Rover

- Data inputs from sensors–stored in three 1KB buffers
- Synchronous connection with surface ship

  Receives commands from surface ship and from land station

Surface Ship

- Asynchronous connection to land station

  Receives commands from land station to rover

- Synchronous connection with rover

  Receives 1KB of data at a time

  Receives command confirmations and requests to upload

Land Station

- Asynchronous connection to surface ship

  Receives data and command confirmations from surface ship

- Terminal input

  Receives user-provided rover control commands

3.3.2 System Outputs

Rover

- Synchronous connection to surface ship

  Sends 1KB of collected data at a time

  Sends acknowledgment of received commands

Surface Ship

- Asynchronous connection to land station

  Sends 1KB of collected data at a time

- Synchronous connection to rover

  Sends commands to rover (relayed from land station and its own commands)

Land Station

- Asynchronous connection to surface ship

  Receives data from surface ship

  Sends commands to surface ship

- Terminal output

  Displays confirmation of rover control commands

3.4 User Interface

The rover will collect data autonomously and continuously until all internal buffers have been filled. On this event, the rover will request for permission to upload to the surface ship. This operation can be carried out automatically or at the request of the user from the control systems on land. The user will send a request granted signal to the rover through a terminal window on their computer, and when the ship receives 1KB block of data from the rover, it will request permission to transfer to the land station. Similarly, the land station can grant upload permission automatically or at the request of the user through a terminal window. The land station terminal also allows the user to issue commands to start or stop data collection on the rover.

3.5 System Functional Specification

The system will consist of three parts and two forms of serial communication. First, an undersea rover will collect the various forms of data. After completing the collection phase, the rover will send the data to a surface ship over a serial data bus. Once the surface ship has received the data, it will contact the National Earthquake Information Center on land and send the rover data to a nearby land station. The process will repeat indefinitely to continuously collect geological data from ground underwater.

3.6 Operating Specifications

Collecting Data

The system will operate in such a way to ensure no loss of data occurs while transmitting data from the underwater rover the surface ship. The rover system will contain three packages, each with a one kilobyte data buffer. After one data buffer is 80 percent full, the rover will contact the surface ship and request permission to transmit. At the same time, another sensor is brought to stand-by mode, ready to collect data. When the buffer is 90 percent full, the second package set to stand-by will start collecting data. When the buffer of the first package is 100 percent full, the instrument is placed in standby mode while it waits for permission to transfer the data. Once permission is granted, the data is uploaded. After all the data has been uploaded, the instrument is set to stand-by mode.

Sending Data to the Surface Ship

Data will be transmitted to the surface ship synchronously once the surface ship has given the rover permission to transmit. One kilobyte of data will be transferred at a time, consisting of all the data from one complete buffer.

Sending data to the NEIC Station

After the surface ship has received one kilobyte of data from the undersea rover, it will request permission to transfer the data to the NEIC station. Once permission has been granted by the NEIC station, this data will be transferred over an asynchronous channel.

3.7 Reliability and Safety Specification

The rover will be fail-operational and will have three redundant data collection systems. It will be able to collect and send data even if it experiences a failure in two of its data systems (although data collection may be hampered with only one data system as it can't collect and send data simultaneously).

4 DESIGN SPECIFICATION

4.1 System Description

This specification describes and defines the high-level requirements and constraints of an undersea earthquake data rover and a surface communication ship. The rover will collect data on earthquake hazards, ground shaking, geographic faults, and plate motion and stress. It will upload this information asynchronously to the surface ship, which will relay that information to one or more ground stations around the world. This specification covers all aspects of Phase 1 and Phase 2. Namely, this includes all aspects of data collection and communication on the rover, communication and data upload on the surface ship, and communication, data upload, and command generation on the land station. The major components of the system will consist of the following parts:
Rover: 3 x 1KB SRAM, PIC microcontroller
Surface ship: 1KB SRAM, PIC microcontroller
Land station: PC with RS-232 serial connection

4.2 Specification of External Environment

The rover will operate on the sea floor and will face significant pressures, low temperatures, and saltwater. The surface ship will operate on the ocean surface and will face varying weather conditions and saltwater. Additionally, the rover should not be too heavy so that it can be lowered from the ship and pulled back up to it.

4.3 System Input and Output Specification

4.3.1 System Inputs

Rover

- Data inputs from sensors–stored in three 1KB buffers

- Synchronous connection with surface ship–sent over the synchronous SPI link, with data prefaced by the sync sequence 1011001000

  Receives commands from surface ship and from land station

Surface Ship

- Asynchronous connection to land station–using RS-232 protocol with baud rate of 19200 char/sec

  Receives commands from land station to rover

- Synchronous connection with rover–receiving 1KB at a time over the synchronous SPI link

  Receives command confirmations and requests to upload

Land Station

- Asynchronous connection to surface ship–using RS-232 protocol with baud rate of 19200 char/sec

  Receives data and command confirmations from surface ship

- Terminal input–using a standard serial communications terminal application with baud rate of 19200 char/sec

  Receives user-provided rover control commands

4.3.2 System Outputs

Rover

- Synchronous connection to surface ship–sent over the synchronous SPI link, with data prefaced by the sync sequence 1011001000

  Sends 1KB of collected data at a time

  Sends acknowledgment of received commands

Surface Ship

- Asynchronous connection to land station–using RS-232 protocol with baud rate of 19200 char/sec, 1 stop bit, even parity, and high-idle state

  Sends 1KB of collected data at a time

- Synchronous connection to rover–receiving 1KB at a time over the synchronous SPI link

  Sends commands to rover (relayed from land station and its own commands)

Land Station

- Asynchronous connection to surface ship–using RS-232 protocol with baud rate of 19200 char/sec, 1 stop bit, even parity, and high-idle state

  Receives 1KB of data from surface ship at a time

  Sends commands to surface ship

- Terminal output

  Displays confirmation of rover control commands

4.4 User Interface

The rover will collect data autonomously and continuously until it fills up its internal buffers. When this happens, the rover will request permission to upload to the surface ship. This process can be automated or manual–in the manual case, the surface ship communications operator will send a request granted signal to the rover via a terminal window. Once the surface ship has received a 1KB block of data from the rover, it will request permission to transfer to the land station. The land station communications operator will grant or deny this request via a terminal window.

4.5 System Functional Specification

The system will consist of three parts and two forms of serial communication. First, an undersea rover will collect the various forms of data. After completing the collection phase, the rover will send the data to a surface ship over a serial SPI data bus. Once the surface ship has received the data, it will contact the National Earthquake Information Center on land and send the rover data to a nearby land station over an asynchronous RS-232 serial channel. The process will repeat indefinitely to continuously collect geological data from ground underwater. The rover system comprises three major blocks as described below and shown in the block diagram given in figure 4.1.

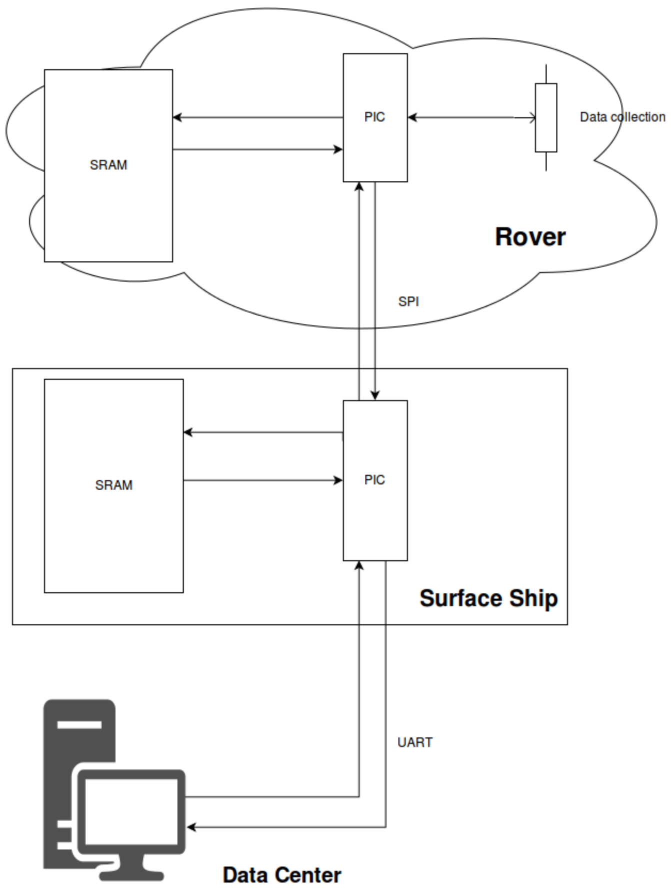

Figure 4.1: Initial block diagram of system.

Control Subsystem

The control subsystem manages which data collection module is storing data and which is transmitting data, as well as the transmitting of the data to the surface ship. It monitors how full each buffer is and requests permission to send when one hits 80 percent capacity and begins transmitting data once it's at 100 percent, assuming permission has been granted. The control subsystem sends the appropriate control signals to each SRAM in order to write or read data. This subsystem also has a counter for each buffer to allow it to keep track of the buffer usage. This subsystem is also responsible for handling and carrying out commands received from the surface ship and land station. The land station can automatically or manually give permission to the surface ship to upload the data it received from the rover. Further, the land station can send commands to the rover via the surface ship to stop or start data collection. These commands are user-generated via a terminal window and sent over the RS-232 connection to the surface ship, which then relays them to the rover. Once the rover has received a command, it sends an acknowledgment back through the surface ship to the land station. The control subsystem coordinates all of this on the rover, surface ship, and land station.

Data Collection Subsystem

The data collection subsystem includes the sensors on the rover collecting measurements and the buffers used to store the data before transmitting to the surface ship. This system will be implemented using simulated data inputs from a power supply and/or signal generator in Phase 1, with the three buffers implemented as a 2 kilobyte SRAM module each. This will provide easy design and control of the buffers and more data storage in case communication with the surface ship is flaky or permission to transmit is delayed. The data collection subsystem will be controlled directly by the control subsystem.

Rover-to-Ship Communication Subsystem

The rover-to-ship communication subsystem manages the transmission of data from the rover to the surface ship and commands from the surface ship to the rover. This module will communicate over a Serial Peripheral Interface (SPI) bus, a synchronous interface, with the rover configured as the SPI master and the ship as the slave. A binary synchronization sequence of 1011001000 will be transmitted by the rover at the start of each data stream, with the one kilobyte of collected data being transmitted immediately after. In Phase 1, one command this subsystem will process is a "permission granted" signal from the surface ship to begin sending data. Other commands could include "start" and "stop" recording, "power on/off", and fine control over which sensors are actively collecting data.

Ship-to-Land Station Communication Subsystem

The ship-to-land station communication subsystem manages the transmission of data from the surface ship to the land station and commands from the land station to the surface ship. This module will communicate over an RS-232 link, an asynchronous protocol that uses the PIC USART hardware module and a serial to USB adapter on a computer. Further, this subsystem will use a baud rate of 19200 characters/second, even parity, a single stop bit, and high-idle. This subsystem will handle collected data uploaded from the surface ship to the land station and will carry commands from the station to the ship and command acknowledgments from the rover back to the land station.

4.6 Operating Specifications

Collecting Data

The system will operate in such a way to ensure no loss of data occurs while transmitting data from the underwater rover the surface ship. The rover system will contain three packages, each with a one kilobyte data buffer implemented as a two kilobyte SRAM (with extra buffer room). After one data buffer is 80 percent full, the rover will contact the surface ship and request permission to transmit. At the same time, another sensor is brought to stand-by mode, ready to collect data. When the buffer is 90 percent full, the second package set to stand-by will start collecting data. When the buffer of the first package is 100 percent full, the instrument is placed in standby mode while it waits for permission to transfer the data. Once permission is granted, the data is uploaded. After all the data has been uploaded, the instrument is set to stand-by mode.

Sending Data to the Surface Ship

Data will be transmitted to the surface ship synchronously over an SPI bus, with the rover as the master and the surface ship as the slave. The binary synchronizing sequence that proceeds data will be 1011001000. The data will follow immediately after this sync sequence.

Sending Data to the Land Station

Data will be transmitted to the land station asynchronously over a RS-232 connection using a baud rate of 19200 char/sec, even parity, 1 stop bit, and high-idle. The data will be uploaded from the surface ship to the land station as soon as permission is granted for the ship to do so.

Land Station to Rover Commands

Commands for the rover to start or stop data collection will be sent from the land station over its RS-232 connection to the surface ship and then relayed to the rover over the ship-rover SPI connection. When the rover receives one of these commands, it will send an acknowledgment back through the surface ship to the land station. Essentially, the surface ship will function as a relay between the rover and land station.

4.7 Reliability and Safety Specification

The rover will be fail-operational and will have three redundant data collection systems. It will be able to collect and send data even if it experiences a failure in two of its data systems (although data collection may be hampered with only one data system as it can't collect and send data simultaneously).

5 DISCUSSION

5.1 System Description

This specification describes the high-level requirements and constraints of a data rover placed under the sea along with its surface communication ship. It also identifies the requirements and constraints of the ground station that retrieves the information from the communication ship.

Inputs

The system as a whole takes one input: data measured from rocks. This input is measured by the underwater rover portion of the system. In the actual implementation, this input is represented as the middle voltage in a voltage divider, implemented and varied with a potentiometer. This input interpreted by the ADC as a 10-bit number.

Outputs

The system has different outputs, depending on which section is being considered.

- Rover - The rover sends an 8-bit data via SPI to the surface ship
- Surface ship - This sends the 8-bit data via RS-232 to the land station
- land station - This sends data collection commands via RS-232 to the surface ship

The system overall does not have external outputs.

Basic Procedures

Timing Constraints

The system must follow the timing constrains of the integrated circuits. These constraints are outlined in the respective data sheets and vary. To avoid any timing issues or race conditions, delays were implemented uniformly across the system at about 10 milliseconds. This exceeded each of the timing constraints specified by every IC. The fallback to this method was that the data transfer rate was very slow, about 15 bytes a second. To implement faster data transfer, the delays could be fine-tuned to the actual specified delays required of each IC.

Error Handling

A failure mode was implemented on the undersea rover for when the two buffers on the SRAM fail. The two buffers will fail if they are both full and either waiting to send data or in the middle of sending data. Data cannot be stored on either buffer during this state. If the program every enters this state, a red LED will light and alert the user that the collected data is not being stored. The LED will turn off as soon as either buffer enters an idle state and can begin collecting again.

Ideally this signal would activate a string of buffers that allows data to be written to a second, backup SRAM. However, with limited breadboard space, all the necessary ICs could not be installed. Instead, a visual representation of failure shows the user when data is not being stored.

5.2 Hardware Implementation

The system is divided into three sections: underwater rover, surface ship and the data collection center. Each section has a unique hardware implementation, specified below.

Underwater Rover

The underwater rover incorporates an SRAM and 18F25k22 PIC microcontroller. There are are also shift registers that consolidate the number of pins needed on the PIC.

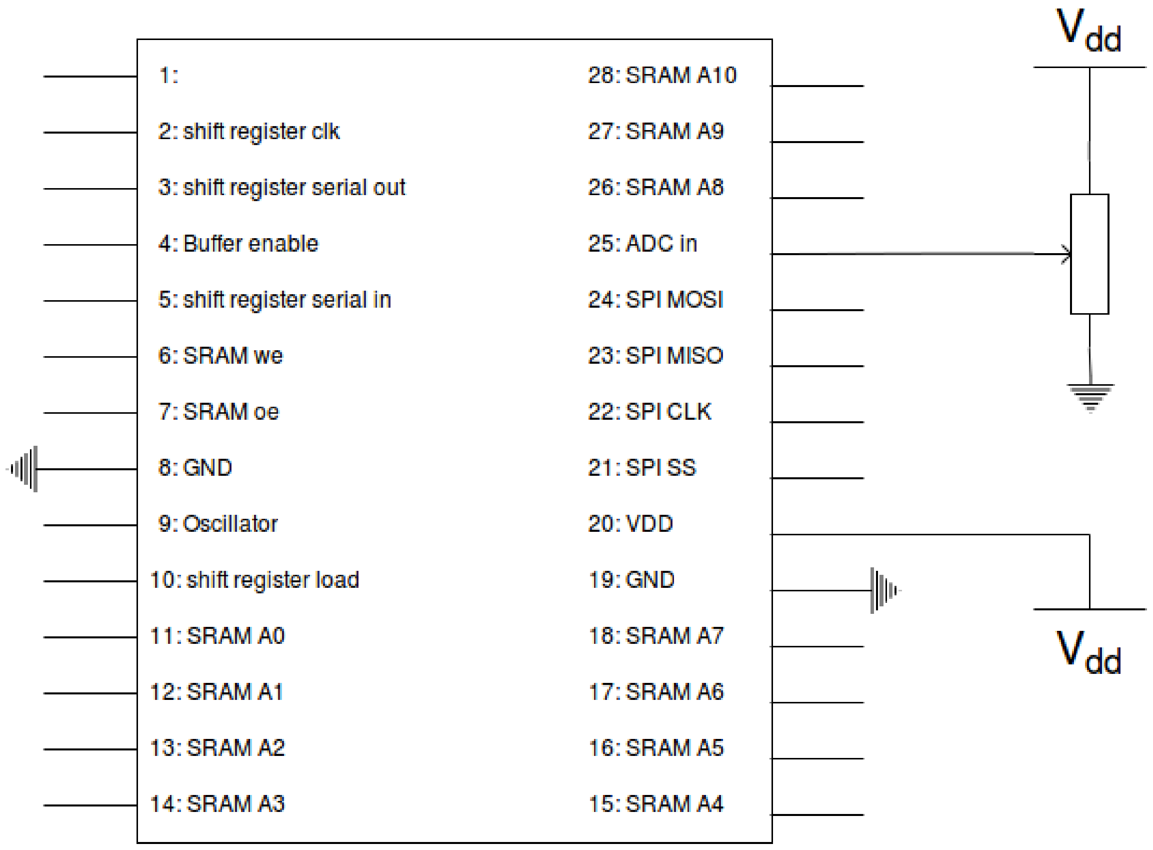

Figure 5.1: Diagram of the pins used in the underwater rover PIC with potentiometer.

The PIC microcontroller pin specification is found in Figure 5.1. Pin 25 in configured as an ADC input while taking voltage data from a potentiometer. The PIC, SRAM and shift registers are connected as described in Figure 5.2.

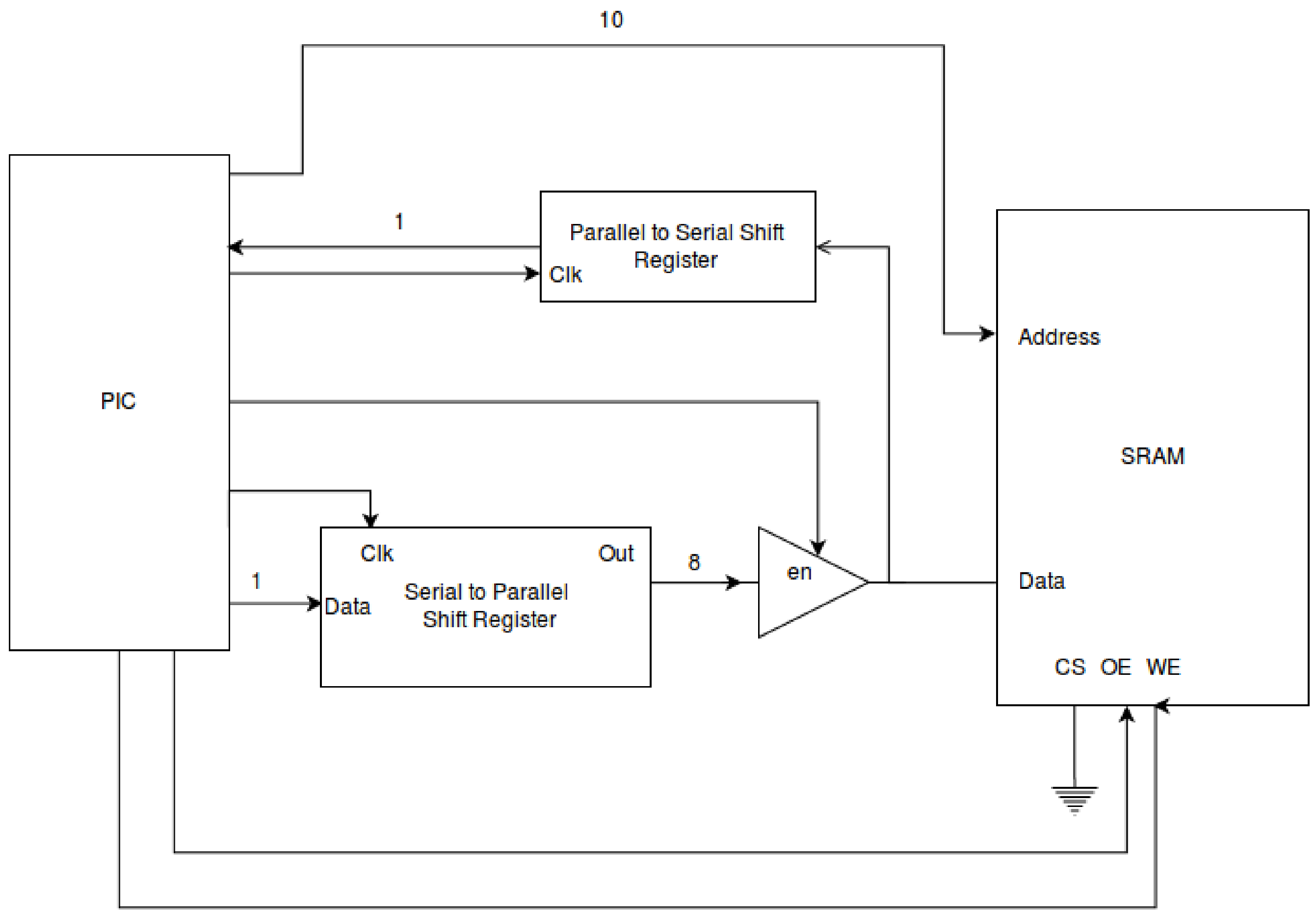
10
1
Parallel to Serial Shift Register
Clk
Address
PIC
SRAM
Clk
Out
8
en
1
Data
Serial to Parallel Shift Register
Data
CS OE WE

Figure 5.2: Schematic of PIC, SRAM and shift register connections.

The purpose of the shift registers was to consolidate the number of pins needed on the PIC for the 8-bit data inputs/outputs of the SRAM. The PIC only has 24 available in/out pins, thus shift registers were needed. An 8-bit serial-to-parallel ship register controls the data flow into the SRAM. The address is controlled by pins 11 - 18 and 26 - 28 on the PIC (see Figure 5.1). Next, an 8-bit parallel to serial shift register controls the data coming out of the SRAM, turning eight bits into one bit. The `oe` `we` pins are controlled by the PIC.

The PIC sends the collected data to the surface ship via SPI. Pins 21 - 24 are configured as SPI master inputs and outputs. They are connected directly to another PIC configured as the slave.

Surface Ship

The surface ship acts as the SPI slave to the underwater rover. The surface ship also has SRAM storage with similar architecture to the rover. The SRAM input data goes through an 8-bit serial to parallel register while the output data goes through an 8-bit parallel to serial register. The other SRAM inputs (`oe`, `we` and the 10-bit address select) are controlled by the PIC.

Pins 21-24 are configured to be SPI slave inputs and outputs. They receive data from the underwater rover to be stored in the SRAM. Pins 17 and 18 are configured at RS-232 inputs and outputs. These pins receive signals from the land station, which tell the system commands such as start data collection, stop data collection, and begin data upload. These pins also transfer data to the center.

land station

The land station is implemented on a separate computer and communicates with the system via RS-232. Data is transferred to and from the computer via a USB connection to the UART.

5.3 Software Implementation

All system logic and procedures are programmed in software on the PIC microcontrollers. Each PIC has its own program to control its unique procedures and settings.

Underwater Rover PIC Software

The program first crates the appropriate settings to configure an SPI. This involved setting certain registers to certain values. The procedure is outlined in the PIC 18F25K22 manual.

Next, the program configures the ADC. Again, this involves setting certain registers, a procedure outlined in the manual. When the ADC is sampled, the register `GO_nDONE` must be set high. This register will automatically be pulled low when the ADC is done sampling. The data is found on registers `ADRESH` and `ADRESL`.

The general program flow follows the behavior specification outlined in Section 3.6. Figure 5.3 shows a state diagram of the program logic.

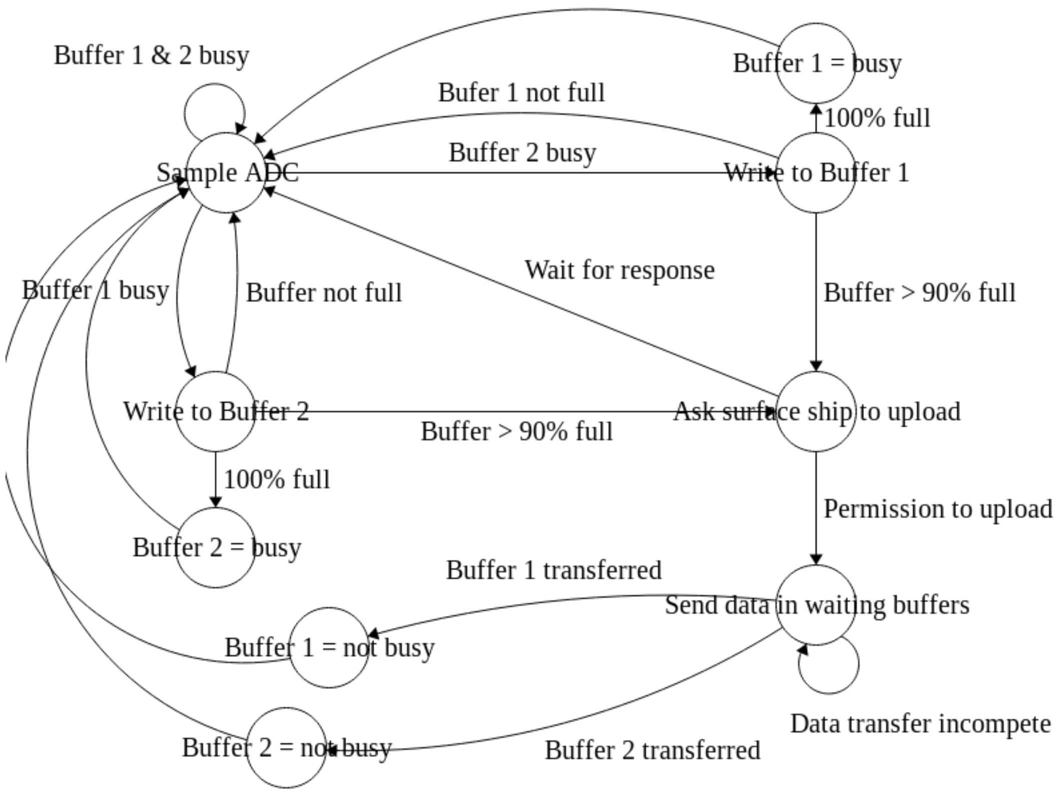

Figure 5.3: State diagram of underwater rover software

Surface Ship PIC Software

Similar to the underwater rover program, the surface ship program must configure the PIC to receive SPI signals, but this time as a slave. This involves setting certain registers according to values specified by the PIC18F25K22 manual.

This program must also configure the UART on the PIC in order to talk to the land station. This also involves setting certain registers according to specifications in the PIC manual. The UART will receive commands from the land station (computer) and will run software according to those commands.

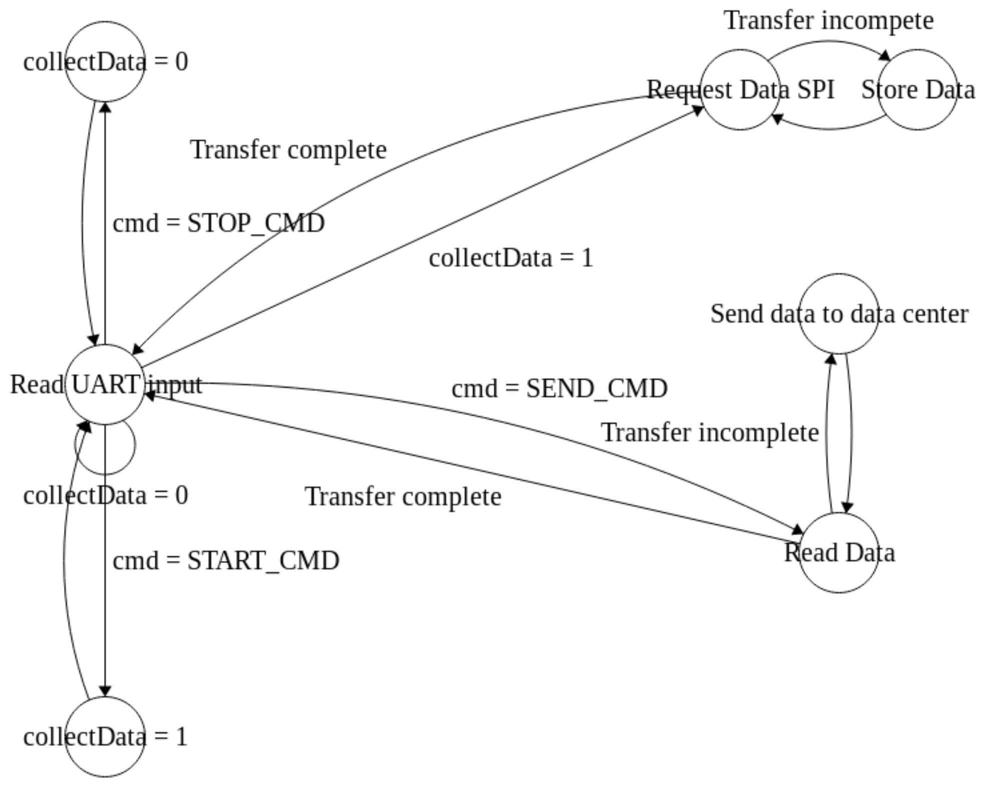

Figure 5.4: State diagram of procedure for surface ship

land station

land station commands on the computer are entered and sent through the USB port via a Putty serial terminal.

5.4 Design Procedure

We began designing the system by breaking the system into subsystem/modules and tackling each module one at a time. Specifically, we broke the system into data collection, SPI communications, UART communications, control, and remote command modules. We designed and tested each of these modules in isolation before bringing them together incrementally to form the full system. For each subsystem, we started with the most basic functionality, like being able to send over SPI, and got it working before moving onto more complicated functions, like uploading data over SPI. As we got each more complicated function working, we tested it to verify that it was working. Then, we integrated the module with the rest of the system and tested it more to ensure it was working as intended with the rest of the system. With this design paradigm, we got each module working in sequence and were able to get the whole system working. It proved to be an excellent strategy for us and this project.

6 TEST PLAN

Our plan for testing this system involved testing each subsystem individually and strategically connecting them together and testing again. Continuing from our work in Lab 1 where we learned to read and write an SRAM, we didn't spend much time testing the basic functionality of the SRAM modules used in this lab. Instead, once we verified that reading and writing the SRAM worked, we tested the output from reading the SRAM to be sure it matched what we expected to be stored from our sensors. Moving on, we tested the SPI communications in a similar way, starting with getting a single PIC transmitting data over SPI and verifying it using the SPI serial function on our logic analyzer. We then tested SPI receiving by using a second PIC to echo back whatever the master PIC was sending. After SPI, we worked on testing the RS-232 UART module, again beginning with simply transmitting to the logic analyzer and moving onto echoing data back from a computer serving as the land station. We then connected our station to the surface ship PIC and verified we could send and receive arbitrary data. Next, we tested the command function by sending commands from the land station to the surface ship and receiving acknowledgments. We moved onto relaying the commands through the surface ship to the rover and verifying that the land station received acknowledgments and the rover actually followed the given command. Finally, we tested everything together, making sure our data collection system was recording to the SRAM, the rover was uploading this data when it had permission to the surface ship, the surface ship relayed this data to the land station when it had permission, and the land station was able to control the rover with remote commands.

7 TEST SPECIFICATION

7.1 Testing SPI

In order to test the SPI, we will configure one PIC to transmit data as an SPI master and monitor the signals using the logic analyzer. Once we are able to do this, we will configure a second PIC as an SPI slave and have it echo data back to the master device so we can test receiving. This will again be monitored using the logic analyzer's SPI serial connection tool. Once we are able to send and receive specific data over SPI, we will test sending arbitrary data and large sequences of data (up to 1KB in size).

7.2 Testing SRAM

Because we had already worked in depth with the SRAM in Lab 1, we will not thoroughly test the basic functionality of the SRAM by itself. Rather, we will write data from our data collection sensors to the SRAM and read it out on our logic analyzer, checking that the values match what was expected.

7.3 Testing UART

In order to test the UART/RS-232 connection, we will proceed as we did with testing the SPI. Namely, we will ensure transmitting data works by sending data from one PIC to the logic analyzer. Then, we will test receiving data by connecting the PIC to a computer serving as our land station and having the PIC echo back data it receives from the station. Finally, we will test sending arbitrary commands over the UART from the station to the PIC.

7.4 Testing Commands

Continuing on from the UART testing, we will first check that commands from the land station are received by the surface ship. Next, we will test that command acknowledgments are sent automatically from the surface ship to the land station. Then, we will relay the commands from the ship to the rover and ensure that the rover performs the actions requested by the command and acknowledges them. Finally, we will ensure that the land station is able to receive acknowledgments from the rover.

7.5 Testing Data Collection

In order to test the data collection subsystem, we will first configure one of the ADCs on the PIC to receive input from a potentiometer and write its converted data to the SRAM. We will monitor the data being sent to the SRAM on a logic analyzer to determine if the ADC is providing reasonable data for a given input signal. We will also test that the data collection system is always running by default, switches between the three data buffers as needed, and follows the start/stop commands issued by the land station.

7.6 Testing Failure Handling

Continuing from our testing of the data collection system, we will test the failure handling functionality by disabling one of the SRAM and making sure that the rover stores its collected data in a backup SRAM buffer. This transition should be seamless so we don't lose any data in the handover process.

8 TEST CASES

8.1 Testing SRAM

- Test writing to SRAM–sweep through each address from 0 to 1023, writing the address value to the address
- Test reading values from SRAM

Input Signals

- SRAM write enable, chip enable, and output enable control lines
- SRAM address
- SRAM serial write data

Output Signals

- SRAM parallel read data

Expected Results

- We expect the SRAM read data to match the data that was written to it at the same address

Pass/Fail Criterion

The SRAM subsystem will pass if we can successfully write arbitrary data to any address in the SRAM and read it back with no errors. This subsystem will fail if we cannot read or write the SRAM or if data gets corrupted within the SRAM.

8.2 Testing Data Collection

- Test values from ADC for: stability, sensible changing with respect to input signal voltage
- Test writing ADC values to SRAM
- Test reading ADC values from SRAM
- Test requesting permission to upload at 80% full
- Test uploading to surface ship when 100% full

Input Signals

- Analog input to ADC

Output Signals

- ADC output to SRAM

Expected Results

- We expect the converted output signal from the ADC to change in value proportionately to the ADC input as we change the input signal
- We expect the collected data to be stored in the SRAM buffer and able to be read from the buffer intact

Pass/Fail Criterion

This subsystem will pass if the output from the ADC appears stable and changes in proportion to changes made in the ADC input signal and if the data collected is successfully written to the SRAM buffer. This subsystem will fail if any of these criteria are not met.

8.3 Testing Failure Handling

- Test SRAM switch-over (should be seamless)

Input Signals

- SRAM disable signal

Output Signals

N/A

Expected Results

- We expect the transition to the backup buffer to be seamless and not allow any data to be dropped

Pass/Fail Criterion

The failure handling subsystem will pass if it is able to swap to the backup buffer without losing any data. It will fail if it does lose data or is unable to switch to the backup buffer.

8.4 Testing SPI

- Test clock signal generation
- Test sending data
- Test receiving data
- Connect with slave SPI device and test sending/receiving
- Test request/acknowledgment for uploading data to slave

Input Signals

- Receive line

Output Signals

- Transmit line
- Slave select
- Clock

Expected Results

- We expect the logic analyzer to show the same data on the transmit line as we are sending out in software
- We expect the slave PIC to echo back the same data the master PIC is sending out
- We expect the master PIC to preface data with the synchronizing sequence 1011001000

Pass/Fail Criterion

The SPI subsystem will pass all tests when it is able to transmit and receive arbitrary data between a master PIC and a slave PIC (i.e. the rover and surface ship). It will fail if it is unable to transmit or receive arbitrary data with the surface ship.

8.5 Testing UART

- Test transmitting data
- Test receiving data

Input Signals

- Receive line

Output Signals

- Transmit line

Expected Results

- We expect the UART to be able to send arbitrary data out on its transmit line with even parity, 1 stop bit, high idle-state, and at 19200 baud
- We also expect the UART to be able to receive arbitrary data on its receive line with the same configuration as above

Pass/Fail Criterion

The UART subsystem will pass all tests if we are able to transmit and receive arbitrary data at the specified parameters of even parity, 1 stop bit, high idle-state, and 19200 baud. This subsystem will fail if it is unable to send or receive arbitrary data, or if it cannot perform at the given parameters.

8.6 Testing Remote Rover Commands

- Test sending commands from land station to surface ship
- Test relaying commands from surface ship to rover
- Test sending command acknowledgments from rover to surface ship
- Test relaying command acknowledgments from surface ship to land station
- Test that the rover follows the commands given to start/stop data collection

Input Signals

- Commands from terminal window on land station computer

Output Signals

- Command acknowledgments on terminal window on land station computer

Expected Results

- We expect the rover to follow the commands issued by the land station and to send an acknowledgment, which will be displayed on the station terminal window
- We expect the surface ship to receive commands from the station over UART and relay these to the rover over SPI
- We expect the surface ship to receive acknowledgments from the rover over SPI and relay these to the station over UART

Pass/Fail Criterion

The command subsystem will pass all tests if we are able to input a command on the land station terminal window and receive an acknowledgment from the rover that it received the given command and performed the commanded action. This subsystem will fail if the command doesn't reach the rover, the rover fails to act properly on a command, or the rover fails to send an acknowledgment.

9 RESULTS

Overall, this lab was a success. At our demo, we were able to successfully show off all major components required of our rover system. Our rover continuously collected data from a sensor through its ADC, stored this data in rotating buffers implemented in SRAM, uploaded this data to a surface ship over SPI, and the ship uploaded it to the land station over RS-232. We were also able to send commands to the rover to start or stop data collection from the land station, although this functionality had some timing/usability issues. Yet, we received full points on the demo. In addition to getting the hardware and software working, we gained more experience using the oscilloscope/logic analyzer, learned how to program and use a PIC, and gained a lot of experience following a formal development cycle. We created requirements and design specification documents, a bill of materials, and a Gantt chart to track our progress. As we progressed through the design cycle, we updated these documents as needed. We also went through a couple design reviews to make sure that we were meeting the requirements specified by the customer and that our design was progressing appropriately. All in all, we met the requirements of this lab and learned about the design cycle as we went through it. These skills will help us greatly as we move out of college into the working world and begin to establish our careers as engineers.

10 ERROR ANALYSIS AND PROBLEMS ENCOUNTERED

Overall, we only faced a few major challenges in this lab. Firstly, configuring the ADC to work correctly took some time and a lot of looking at the PIC manual. Secondly, getting SPI to work on a single PIC and to communicate between two PICs using this interface took several hours.

Thirdly, the UART communication between a PIC and the computer took a couple hours to get fully working. Finally, we faced difficulties with the remote rover commands from the land station, and did not have the time to fully resolve them to our satisfaction. Despite these difficulties elaborated upon below, we did get all of the major requirements for this lab project completed by our final demonstration of our system.

10.1 ADC

We only faced minor configuration issues with the ADC. We used our knowledge of how the SPI inputs/outputs were configured and applied it to the ADC input. After debugging a wrong bit number for the analog input, our ADC provided data to our system flawlessly.

10.2 SPI

We had several major challenges and bugs in our SPI implementation, but were able to figure them out and get data sending and receiving. The major problems we encountered were caused by mis-configuration of the SPI. The documentation in the PIC18F25K22 for the SPI pins and settings are adequate, but the number of register bits that need to be individually and specifically set is pretty high. It took a while to figure out the correct configuration for the SPI master. We did, and we were able to successfully send out data. The next major challenge was figuring out the configuration required to receive data on the master. This also took some time, as we didn't know we had to specify the SPI input pin as an analog input as well as setting its tri-state register as an input. Once we figured this out, we could read in data on the master by looping its output to its input. Next, we faced some trouble configuring the SPI slave. Documentation for the slave wasn't as easy to find or parse as for the master, but we were able to get our SPI slave configured properly.

10.3 UART Communication

We only had issues trying to send data at the right time using the UART. Having previous knowledge in this field was beneficial, as it helped set up tools to allow easier communication through effective user interfaces, and once the baud rate was set up and everything was wired correctly, the communication system worked flawlessly and we were able to echo back what we sent out.

10.4 Remote Commands

In our testing of the command subsystem, we were able to reliably control the data collection on the rover from the land station. This was made easier by the fact we were testing the subsystem with a reduced functionality data collection system where it was taking many fewer samples. When we added the command subsystem to the full rover software, we experienced issues where the commands from the land station would only be received when the rover is waiting for a communication from the surface ship. During this time, the ship would relay the command from the land station and the rover would respond and acknowledge the command as designed. However, when the rover is not waiting for a signal on its SPI bus, the command is not received by the rover as it is busy collecting data. This meant that we only had a few seconds per data collection cycle to issue commands to the rover. This could have been solved by using a basic real-time operating system to switch between the data collection, communications, and other tasks several times a second so that the rover would appear to always be listening while also always collecting data. But, we simply did not have enough time to implement this functionality.

11 BILL OF MATERIALS

Table 11.1: A table showing how the project was planned and carried out

Part No.	Part Name	Description	#	$/Pc	Cost ($)
18F25K22	PIC Microcontroller	Micro controller with basic I/O	2	2.50	5.00
CD4040B	12-bit Counter	12-bit ripple-up counter	3	0.50	1.50
CY7C128A	2K SRAM	2K x 8-bit bidirectional SRAM	4	4.00	16.00
SN74LS241	Octal Buffer	Octal tri-state buffer	4	0.82	3.28
N/A	Breadboard	Solderless breadboard	5	3.50	17.5
3362P-103LF-ND	Potentiometer	Turn Top 10K Potentiometer	3	0.48	1.44
3Wire-MW-26-1/4	Copper Wire	Copper wire	5m	1 / m	5.00
				Total:	49.72

12 FINAL UPDATED SCHEDULE

Table 12.1: A table showing how the project was carried out.

Task	Start Date	End Date	Assigned To
Initial Understanding of Lab	13/10	14/10	Everyone
Phase I Design Plan	14/10	18/10	Sean
Purchase Materials	15/10	15/10	Katie
Design Review Phase I Implementation	15/10	16/10	Katie/Sean
Program Initial PIC Settings	16/10	27/10	Sean
Test/Debug Phase I	21/10	27/10	Vishesh
Finalize Phase II Design Plan	25/10	27/10	Sean
Demonstrate Phase I	27/10	27/10	Everyone
Review Phase II Design	28/10	28/10	Vishesh
Implement Complete SPI	29/10	30/10	Katie
Implement UART Communications	1/11	3/11	Sean
Test/Debug Phase II	3/11	4/11	Vishesh
Demonstrate Complete Project	4/11	4/11	Everyone
Final Documentation	4/11	7/11	Everyone

13 SUMMARY AND CONCLUSION

In summary, this lab provided us with the necessary design cycle experience for implementing a system to measure, collect and transfer data using remote systems. We began our design using the specification provided from the customer, with their requirements and worked on ensuring we met their needs. These requirements involved collecting three different forms of data from under the sea, including waveform data, earthquake hazard data, ground shaking data and fault information. These also consist of plate motion, distortion and crustal data that was represented through the use of a potentiometer. Through the use of the potentiometer, we could change values as they would change using sensors, and see whether the data was collected, saved and transmitted when requested.

First, we wired the system so that we could collect data from one sensor in the form of a potentiometer. We connected the PIC to an SRAM and two shift registers, and successfully programmed the PIC to carry out the task. We then implemented two more similar systems for the other sensors and then implemented the UART communication system to communicate with the ground station (represented by the PIC) to successfully communicate commands. Once we had that programmed, we programmed the commands to carry out their required functions.

Finally, we continued to document our design and update our reports and system design to match our updates from debugging and vigorous tests. We went through several changes in design as we learned the power of the tools we had at hand. In conclusion, once we figured out how to program and use the PIC, the requirements of the customer were fairly easy to meet and were delivered with success.

14 APPENDICES

Table 14.1: Division of work among group members

Name	Role	Hours
Vishesh Sood	UART commands, debugging of SPI and wiring	35
Sean Happenny	SPI and UART communications, UART commands	35
Katie Neff	SRAM hardware and software, UART commands, wiring	35

Listing 1: C program for the rover

```
/******************************************************************************
    */
/* Files to Include
    */
/******************************************************************************
    */
#if defined(__XC)
    #include <xc.h>         /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>        /* HiTech General Include File */
#elif defined(__18CXX)
    #include <p18cxxx.h>    /* C18 General Include File */
#endif

#if defined(__XC) || defined(HI_TECH_C)
#include <stdint.h>         /* For uint8_t definition */
#include <stdbool.h>        /* For true/false definition */
#endif

#include <stdio.h>
#include "system.h"         /* System funct/params, like osc/peripheral config
    */
#include "user.h"           /* User funct/params, such as InitApp */


/******************************************************************************
    */
/* User Global Variable Declaration
    */
/******************************************************************************
    */

/* i.e. uint8_t <variable_name>; */
extern unsigned char canSend_buff1 = 0;
extern unsigned char canSend_buff2 = 0;
unsigned char collectData_buff1;
unsigned char collectData_buff2;
unsigned char isIdle_buff1;
unsigned char isIdle_buff2;
unsigned char activeBufferId = 1;
unsigned char collectData, nextByte;
/*****************************************************************************
    */
/* Main Program
    */
/*****************************************************************************
    */
void delay(int s);
void insertTask(TCB* node, TCB** head, TCB** tail);
void sram_write(unsigned int, unsigned int);
void reset_counter();
unsigned char sram_read(unsigned int);
void test_shift_register();
void test_spi();
void address_select(unsigned int);
void set_s2p_shift_register(unsigned int data);
void toggle_buffer_clock();
unsigned char get_data_p2s_register();
unsigned char spi_Send_Read(unsigned char byte);
void ADC_init();
unsigned char sample_adc(unsigned char channel);
unsigned char PORTA_shadow, PORTB_shadow, PORTC_shadow;
unsigned char ADCON0_SHAD, ADCON1_SHAD, ADCON2_SHAD;

void main(void)
{
    collectData_buff1 = 1;
    collectData_buff2 = 0;
    isIdle_buff1 = 0;
    isIdle_buff2 = 1;
    // RA1 = serial in from buffer, RA4 = serial in from SRAM, RA5 = SRAM I/O
       tri-state buffer
    TRISA = 0x10;
    // Need B2 as input for SPI IN, B4 as pot input
    TRISB = 0x14;
    TRISC = 0x00;

    ANSELB = 0x00;
    // Set ADC
    ANSELBbits.ANSB4 = 1;

    PORTA_shadow = 0x00;
    PORTB_shadow = 0x00;
    PORTC_shadow = 0x00;

    InitApp();

    SPI1_Init();
    SPI1_Enable();

    ADC_init();

    // buff1 count
    int i = 0;
    // buff2 count
```

```
int j = 100;
unsigned char a;
while(1)
{

    if (collectData_buff1) {
        a = sample_adc(11);
        sram_write(a, i);
        i++;
        }
    if (i >= 80 && !canSend_buff1)
    {
        // Ask permission from surface
        SPI_CSN = 0;
        spi_Send_Read(UPLOAD_REQ);
        //spi_Send_Read(UPLOAD_REQ1);
        SPI_CSN = 1;
        // Wait for response
        delay(10);
        SPI_CSN = 0;
        unsigned char ack = spi_Send_Read(UPLOAD_REQ);
        unsigned char ack1 = spi_Send_Read(UPLOAD_REQ0);
        //unsigned char ack1 = spi_Send_Read(UPLOAD_REQ0);
        //unsigned char ack0 = spi_Send_Read(UPLOAD_REQ1);
        SPI_CSN = 1;
        canSend_buff1 = (ack == UPLOAD_REQ);// && (ack1 == UPLOAD_ACK1);
    // Buffer 90% full; start collection on second buffer <-- u sure?
    }
    if (i >= 90 && isIdle_buff2)
    {
        // Switch buffer
        collectData_buff2 = 1;
        isIdle_buff2 = 0;
    }
    if (i >= 100) {
        collectData_buff1 = 0;
    }



    if (collectData_buff2) {
        a = sample_adc(11);
        sram_write(a, j);
        j++;
    }
    if (j >= 180 && !canSend_buff2)
    {
        // Ask permission from surface
        SPI_CSN = 0;
        spi_Send_Read(UPLOAD_REQ);
        //spi_Send_Read(UPLOAD_REQ1);
        SPI_CSN = 1;
        // Wait for response
        delay(10);
        SPI_CSN = 0;
        unsigned char ack = spi_Send_Read(UPLOAD_REQ);
        unsigned char ack1 = spi_Send_Read(UPLOAD_REQ0);
        //unsigned char ack1 = spi_Send_Read(UPLOAD_REQ0);
        //unsigned char ack0 = spi_Send_Read(UPLOAD_REQ1);
        SPI_CSN = 1;
        canSend_buff2 = (ack == UPLOAD_REQ);// && (ack1 == UPLOAD_ACK1);
    // Buffer 90% full; start collection on second buffer <-- u sure?
    }
    if (j >= 190 && isIdle_buff1)
    {
        // Switch buffer
        collectData_buff1 = 1;
        isIdle_buff1 = 0;
    }
    if (j >= 200) {
        collectData_buff2 = 0;
    }

    if (canSend_buff1)
    {
        // Send synchronization sequence
        SPI_CSN = 0;
        spi_Send_Read(SYNC_SEQ >> 8);
        delay(10);
        spi_Send_Read(SYNC_SEQ);
        SPI_CSN = 1;
        for (int k = 0; k < 100; k++) {
            int data = sram_read(k);
            delay(10);
            SPI_CSN = 0;
            spi_Send_Read(data);
            SPI_CSN = 1;
            delay(10);
        }
        canSend_buff1 = 0;
        isIdle_buff1 = 1;
        i = 0;
    }

    if (canSend_buff2)
    {
        // Send synchronization sequence
        SPI_CSN = 0;
        spi_Send_Read(SYNC_SEQ >> 8);
        delay(10);
        spi_Send_Read(SYNC_SEQ);
        SPI_CSN = 1;
        for (int k = 0; k < 100; k++) {
            int data = sram_read(k + 100);
            delay(10);
            SPI_CSN = 0;
            spi_Send_Read(data);
            SPI_CSN = 1;
            delay(10);
        }
        canSend_buff2 = 0;
        isIdle_buff2 = 1;
```

```
            j = 100;
        }

        // Failure
        if (!isIdle_buff1 & !collectData_buff1 & !isIdle_buff2 & !
           collectData_buff2) {
            PORTB_shadow = PORTB_shadow | (1 << 5);
            PORTB = PORTB_shadow;
        } else {
            PORTB_shadow = PORTB_shadow & ~(1 << 5);
            PORTB = PORTB_shadow;
        }


    }
}

void ADC_init()
{
    ADCON0_SHAD &= 0x81;
    ADCON0_SHAD |= (0x2D);
    ADCON0 = ADCON0_SHAD;
    ADCON1_SHAD &= 0x81;
    ADCON1_SHAD |= (0x00);
    ADCON1 = ADCON1_SHAD;
    ADCON2_SHAD &= 0x81;
    ADCON2_SHAD |= (0x00);
    ADCON2 = ADCON2_SHAD;
    ADFM = 1;
}

unsigned char sample_adc(unsigned char channel) {
   delay(20000);                    //Acquisition time to charge hold
      capacitor
    delay(20000);
    GO_nDONE = 1;                   //Initializes A/D conversion
    while(GO_nDONE);                //Waiting for conversion to complete
    return ((ADRESH<<8)+ADRESL);    //Return result
}

void address_select(unsigned int n) {
        PORTC = (n & 0xFF);
        PORTB_shadow = (PORTB_shadow & (0x1F)) | ((n >> 3) & 0xc0);
        PORTB = PORTB_shadow;
        delay(1000);
        return;
}

void delay(int s) {
    int a = 0;
    int i;
    for (i = 0; i < s; i++) {
        a++;
    }
}
```

```
void insertTask(TCB* node, TCB** head, TCB** tail)
{
    if(NULL == (*head)) // If the head pointer is pointing to nothing
    {
        *head = node; // set the head and tail pointers to point to this node
        *tail = node;
    }
    else // otherwise, head is not NULL, add the node to the end of the list
    {
        (*tail)->next = node;
        node->prev = *tail; // note that the tail pointer is still pointing
                            // to the prior last node at this point
        *tail = node;       // update the tail pointer
    }
    // Always set node next pointer to null for end of list
    node->next = NULL;
    return;
}

void reset_counter() {
    PORTA_shadow = PORTA_shadow | (1 << 1);
    PORTA = PORTA_shadow;
    delay(10000);
    PORTA_shadow = PORTA_shadow & ~(1 << 1);
    PORTA = PORTA_shadow;
    delay(10000);
}

void set_s2p_shift_register(unsigned int data) {
    int i;
    unsigned char serial_out;
    int shift = 0;
    for (i = 0; i < 8; i++) {
        serial_out = (data >> shift) & 0x1;
        PORTA_shadow = (PORTA_shadow & ~(1 << 1)) | (serial_out << 1);
        PORTA = PORTA_shadow;
        toggle_buffer_clock();
        shift = shift + 1;
    }
    return;
}

void toggle_buffer_clock() {
    PORTA_shadow = PORTA_shadow | 1;
    PORTA = PORTA_shadow;
    delay(1000);
    PORTA_shadow = PORTA_shadow & ~1;
    PORTA = PORTA_shadow;
    delay(1000);
    return;
}
void sram_write(unsigned int data, unsigned int address) {

    // oe = 1
    PORTA_shadow = PORTA_shadow | (1 << 3);
    PORTA = PORTA_shadow;
    // buffer en = 0
    PORTA_shadow = PORTA_shadow & ~(1 << 5);
    PORTA = PORTA_shadow;

    address_select(address);
    set_s2p_shift_register(data);
    // we = 0
    PORTA_shadow = PORTA_shadow & ~(1 << 2);
    PORTA = PORTA_shadow;
    delay(1000);
    // we = 1
    PORTA_shadow = PORTA_shadow | (1 << 2);
    PORTA = PORTA_shadow;
    delay(1000);
  return;
}

void test_shift_register() {
    return;
}

void test_spi() {
    unsigned char status = 0;
    unsigned char data[6];
    //write TX_ADDRESS register
    SPI_CSN = 0;            //CSN low
    data[0] = spi_Send_Read(0xAF);
    unsigned char portc_bit4 = PORTBbits.RB2; //spi_Send_Read(0xAF);
    delay(1000);
    data[1] = spi_Send_Read(0xAF);
    delay(1000);
    data[2] = spi_Send_Read(0xAF);
    delay(1000);
    data[3] = spi_Send_Read(0xAF);
    delay(1000);
    data[4] = spi_Send_Read(0xAF);
    delay(1000);
    data[5] = spi_Send_Read(0xAF);
    SPI_CSN = 1;            //CSN high

    //delay(1000);

    //read TX_ADDRESS register
    //Check that values are correct using the MPLAB debugger
    SPI_CSN = 0;                      //CSN low
    status = spi_Send_Read(0x10);
    delay(1000);
    data[0] = spi_Send_Read(0x00);    // 0x11
    delay(1000);
    data[1] = spi_Send_Read(0x00);    // 0x22
    delay(1000);
    data[2] = spi_Send_Read(0x00);    // 0x33
    delay(1000);
    data[3] = spi_Send_Read(0x00);    // 0x44
    delay(1000);
    data[4] = spi_Send_Read(0x00);     // 0x55
    SPI_CSN = 1;                       // CSN high
}

unsigned char get_data_p2s_register() {
    // oe = 0
    PORTA_shadow = PORTA_shadow & ~(1 << 3);
    PORTA = PORTA_shadow;
    delay(1000);
    // PL = 0
    PORTA_shadow = PORTA_shadow & ~(1 << 6);
    PORTA = PORTA_shadow;
    delay(1000);
    // PL = 1;
    PORTA_shadow = PORTA_shadow | (1 << 6);
    PORTA = PORTA_shadow;
    delay(1000);
     // oe = 1
//    PORTA_shadow = PORTA_shadow | (1 << 3);
//    PORTA = PORTA_shadow;
    int i;
    unsigned char data = 0x00;// = PORTAbits.RA4;
    for (int i = 0; i < 8; i++) {
        data = data | (PORTAbits.RA4 << i);
        toggle_buffer_clock();
    }
    return data;

}
unsigned char sram_read(unsigned int address) {
    // we = 1
    PORTA_shadow = PORTA_shadow | (1 << 2);
    PORTA = PORTA_shadow;
    address_select(address);

    // buffer en = 1
    PORTA_shadow = PORTA_shadow | (1 << 5);
    PORTA = PORTA_shadow;

    unsigned char data =  get_data_p2s_register();
    return data;

}
```

Listing 2: SPI Master

```
/******************************************************************************
    */
/* Files to Include
    */
/******************************************************************************
    */

#if defined(__XC)
```

```
    #include <xc.h>         /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>        /* HiTech General Include File */
#elif defined(__18CXX)
    #include <p18cxxx.h>    /* C18 General Include File */
#endif

#if defined(__XC) || defined(HI_TECH_C)

#include <stdint.h>         /* For uint8_t definition */
#include <stdbool.h>        /* For true/false definition */

#endif

#include "user.h"

/*****************************************************************************
   */
/* User Functions
   */
/*****************************************************************************
   */

/* <Initialize variables in user.h and insert code for user algorithms.> */
unsigned char nextByte;
unsigned char uploadReq;
extern unsigned char canSend_buff1;
extern unsigned char canSend_buff2;
extern unsigned char collectData;


void InitApp(void)
{
    /* Define global vars */
    nextByte = 0;
    uploadReq = 0;
    collectData = 1;

    /* TODO Initialize User Ports/Peripherals/Project here */

    /* Setup analog functionality and port direction */

    /* Initialize peripherals */

    /* Configure the IPEN bit (1=on) in RCON to turn on/off int priorities */

    /* Enable interrupts */
}

void CollectData(void *taskDataPtr)
{

}

void WriteSram(void *taskDataPtr)
{
}

void ReadSram(void *taskDataPtr)
{

}

void SpiComms(void *taskDataPtr)
{
  SpiCommsData *data = (SpiCommsData*) taskDataPtr;
    // Check if we have permission to send
    if (canSend_buff1 || canSend_buff2)
    {
        // Send synch sequence
        SpiWrite(SYNC_SEQ);
        // Send data in 1 byte packets
        for (uint16_t i = 0; i < data->writeSize; ++i)
        {
            SpiWrite((*((unsigned char *) data->writeData) >> (8*i)) & 0xFF);
        }
    }
}

unsigned char SpiRead(void)
{
  /* Read a byte, send dummy byte */
    return spi_Send_Read(0x00);

}

void SpiWrite(unsigned char byte)
{
  /* Send a byte, ignore read value */
  spi_Send_Read(byte);
}

unsigned char spi_Send_Read(unsigned char byte)
{
    SSP2BUF = byte;
    while (!SSP2STATbits.BF);
    char data = SSP2BUF;
    // Check for commands from the surface ship
    if (byte == START_RX)
    {
        // Send acknowledgement to ship and start data collection
        nextByte = START_ACK;
        collectData = 1;
    } else if (byte == STOP_RX)
    {
        // Send acknowledgement to ship and stop data collection
        nextByte = STOP_ACK;
        collectData = 0;
    }
    //SSP2BUF = nextByte;
    //while (!SSP2STATbits.BF);
    return SSP2BUF;
}
```

Listing 3: SPI Slave

```
/******************************************************************************
    */
/* Files to Include
    */
/******************************************************************************
    */

#if defined(__XC)
    #include <xc.h>        /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>       /* HiTech General Include File */
#elif defined(__18CXX)
    #include <p18cxxx.h>   /* C18 General Include File */
#endif

#if defined(__XC) || defined(HI_TECH_C)

#include <stdint.h>        /* For uint8_t definition */
#include <stdbool.h>       /* For true/false definition */

#endif

#include "system.h"        /* System funct/params, like osc/peripheral config
    */
#include "user.h"          /* User funct/params, such as InitApp */

/******************************************************************************
    */
/* User Global Variable Declaration
    */
/******************************************************************************
    */
//unsigned char nextByte;  // Holds next sending byte for acknowledgement
unsigned char roverUploadReq, canSendUART, startCollection, stopCollection;

/******************************************************************************
    */
/* Main Program
    */
/******************************************************************************
    */
void testUSART(void);

void main(void)
{
    // 19.2Kb/s: baudRate = ((18432000 / (19200 * 16)) - 1) / 2, BRGH = 0
    unsigned int baudRate = ((18432000 / (19200 * 16)) - 1) / 2;

    TRISA = 0x10;
    // Need B0, B1, B2 as input for SPI_SS, SPI_CLK, SPI_IN
```

```
TRISB = 0x07;
// For USART need R7 in, R6 out;
// Need RC3 for p2s register input (reading from sram))
//  1000 1000
TRISC = 0x88;

ANSELB = 0x00;
ANSELC = 0x00;

InitApp();
SPI1_Init();
SPI1_Enable();
OpenUSART1(baudRate);



unsigned char cmd, data;
while(1)
{


    //char flag = 1;
    // Receive from USART if data is available
    while (DataRdy1USART())
    {
        cmd = readcUSART();

        // Parse command
        if (cmd == START_RX)
        {
            startCollection = 1;
            //stopCollection = 0;
        } else if (cmd == STOP_RX)
        {
            //startCollection = 0;
            stopCollection = 1;
        }
    }
    // Send/read SPI
    spiSendRead(0x78);
    delay(10);
    // If rover requested SPI upload, store incoming data bytes in SRAM
    if (!roverUploadReq)
    {
        int i;
        for (i = 0; i < 1024; ++i)
        {
            data = spiSendRead(SPI_IDLE);
            sram_write(i, data);
            delay(10);
        }
        // Upload complete
        roverUploadReq = 0;
    }
    // Upload data to land station if allowed
    if (canSendUART)
```

```
        {
            int i;
            for (i = 0; i < 1024; ++i)
            {
                data = sram_read(i);
                putc1USART(data);
                delay(10);
            }
        }
    }
}

void testUSART()
{
    char c = 'A';  // A = 65 = 0x41
    while (1)
    {
        if (DataRdy1USART())
        {
            c = readcUSART();
            delay(10000);
            putc1USART(c);
        }

    }
}

void delay(unsigned long s)
{
    unsigned long a = 0;
    unsigned long i;
    for (i = 0; i < s; i++) {
        a++;
    }
}
```