6-2011

# Design and Implementation of an RF Data Communication System

Aung K. Soe
*Union College - Schenectady, NY*

# Design and Implementation of an RF Data Communication System

**Aung K. Soe**

ECE 499: Senior Project

Advisor: Professor James Hedrick

03/18/2010

## Report Summary

Located just 8 miles northeast of Union College is Ballston Lake, a unique lake, which offers excellent research opportunities for faculty and students. The south basin of the lake is permanently stratified, and there has been no intermixing between water layers for thousands of years. The lower water layers contain no oxygen (anoxic). The Union College Geology Department is interested in a ten year study of Ballston Lake. Currently, there is no commercially available automatic system for collecting and transmitting data from a sensor package at the bottom of the lake to the lake shore and finally to Union College for long-term research purposes. For this project, I have designed and implemented a prototype for an RF data communication system between Ballston Lake and Union College.

This system will be used as a part of the long-term water monitoring system for Ballston Lake. The system will allow users to collect and transmit the water property data automatically without having most of the tedious human involvement. In addition, the system will not only offer a large amount of data storage space but also provide a convenient technique to manage and analyze data to help answer numerous questions concerning this fascinating lake.

The system design contain both hardware and software components. They both worked together to provide all essential characteristics to perform data transmission reliably between Ballston Lake and Union College. Several possibilities for each hardware component are explored carefully to meet system requirements. In order to communicate reliably between two sites, the Master/Slave protocol is designed and implement. The protocol has been verified working properly with error detection, receiver feedback and retransmission. Several different scenarios of data transmission protocol were tested in order to check the robustness of the protocol.

## Table of Contents

## List of Figures and Tables

# 1. Introduction

Located just 8 miles northeast of Union College is Ballston Lake, a unique lake, which offers excellent research opportunities for faculty and students. The lake is approximately 3.5 miles in length and 750 feet in width. The south basin of the lake is known to be meromictic. This part of the lake is permanently stratified, and there has been no intermixing between water layers for thousands of years. The water depth at this part of the lake is about 120 feet. The deep water in the lake has some unique characteristics. It contains no oxygen at the lower columns but has high levels of carbon dioxide and other gases. The Geology Department at Union College has been very interested in studying the water columns at Ballston Lake for a period of ten years. The department has purchased a multi-parameter water quality sensor package together with a data logging system to collect water property data such as temperature, pH, conductivity, salinity, redox, dissolved oxygen at various depths.

## 1.1. Problem Statement

Currently, there is no automatic method for collecting and transmitting the data from the sensors package for a long-term research purpose. At present, a person on-site takes periodic measurements by hand from the sensors package through the data-logging unit. It is found that the current manual water quality monitoring entails tedious process and is time consuming. For that reason, it is desirable to have a monitoring system with characteristics of autonomous, reliable and flexible. Figure 1 illustrates an overall design of a remote water monitoring system.

This system mainly consists of three different parts: a submersible sensor package control system, a data link between the bottom of the lake and a lake shore where a control computer is located, and a data link between Ballston Lake and Union College.

The sensor package control unit at the bottom of the lake takes periodic measurements at defined depth during a day using the sensor packet, and all the information is transmitted from the unit to the lake site control computer through the data communication link. The data communication link is extended up to about 200 feet and is primarily a wired data link using a RS-485 cable or a fiber optic cable. Finally, the collected data is transmitted from the lake site computer to a Union site computer using a radio frequency (RF) data communication link over a distance of approximately 8 miles.

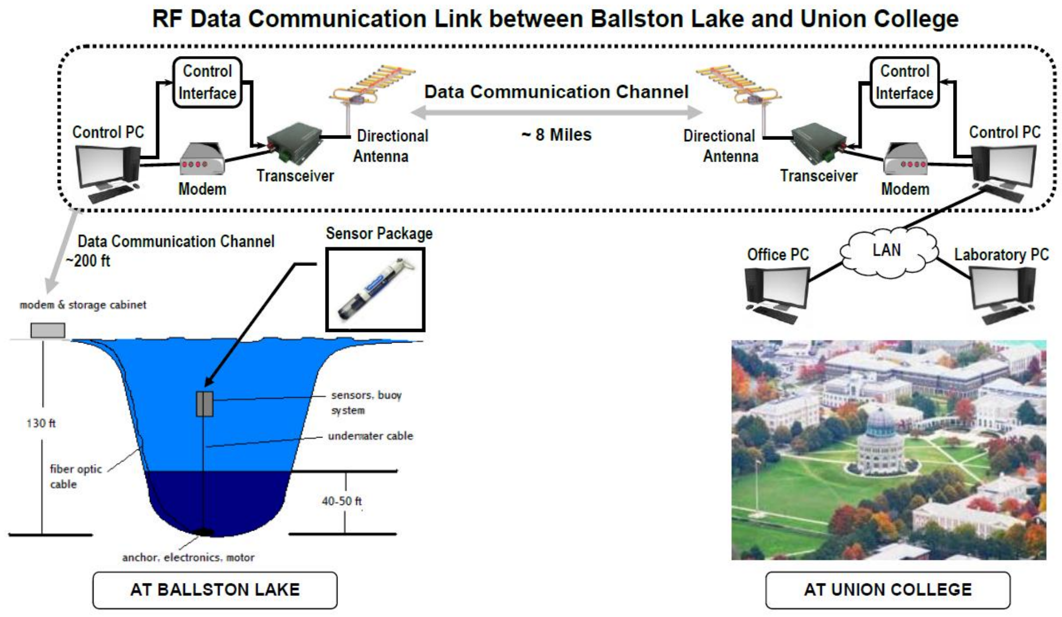


**Figure 1: A block diagram of a remote water monitoring system**

This system will be used for long-term water monitoring of Ballston Lake. The system will allow users to collect and transmit the water property data automatically without having most of the tedious human involvement. In addition, the system will not only offer a large amount of data storage space but also provide a convenient technique to manage and analyze data to help answer numerous questions concerning this fascinating lake.

In addition, the RF technology also supports more efficient way of operating system, as wireless systems can instantly transmit information from the remote water monitoring location. This allows more frequent and precise remote deep water monitoring at the lake, and results in smaller labor force requirements to operate equipments. Fewer human errors in collecting data, which might affect greatly on long term data analysis, will be expected when the system is in use.

## 1.2. Motivation

The purpose of my project is to design and implement a reliable RF data communication system between Ballston Lake and Union College for the long term deep water monitoring research project. Besides the needs of the system for the research purpose, my own personal interest also motivates myself to engage closely in the project. I always want to learn more about RF data communication system and continue my career path in this fascinating field of wireless communication. This project allows me to investigate in much more detail in the design of data communication. I understand that there are alternative designs to approach this problem: transmitting data through telephone network or over the internet. But I choose RF communication because I have a strong interest specifically in RF data communication system.

This paper presents the details of the design and implementation of a data communication system using ultra high frequency (specifically 433 MHz UHF) for remote water monitoring. This design report will begin with some background information on a RF data communication system. Design requirements will be discussed to address essential basic system functionalities for a reliable data communication channel. Possible design alternatives for major components of the design will be examined. Next the final design and implementation and results will be explained and discussed and will include pictures of the system. The design performance will be discussed in the context of conclusions and recommendations for future projects. A cost analysis of the prototype implementation will also be given. Finally, in order to support future use of the system, a user’s manual has been created and included in the report.

# 2. Background

## 2.1. Data communications

Data Communications deal with the transmission of data or information in a form of electric signals between a source and a receiver in a reliable and efficient manner. The source transmits the data and the receiver receives it. The actual generation of the information is not part of Data communications nor is the resulting action of the information at the receiver. Data communication is only interested in the transfer of data, the method of transfer and the preservation of the data during the transfer process.

The requirement for data communication is closely linked to the invention of the telegraph in the early 19th century. The first commercial electrical telegraph in the United States was invented by Samuel Morse in 1837 [1]. During the same year, William Cooke and Sir Charles Wheatstone built the telegraph independently in the United Kingdom [1]. These early establishments of data communication systems relied on wired connections to communicate between two stations. As wired data communication expanded, a separate form of data exchange that required no wires experienced a concurrent development. In 1873, James Maxwell mathematically predicted the existence of electromagnetic waves. In 1886, Heinrich Hertz demonstrated that rapid variations of electric current could be projected into space in the form of radio waves similar to those of light and heat and thus practically proved the existence of the electromagnetic waves. In 1894, Oliver Lodge, a British physicist and writer, demonstrated wireless communication over a distance of 150 yards [1]. In shortly thereafter, Guglielmo Marconi, an Italian inventor, established the first long distance wireless transmission by sending telegraph signals over two kilometer from ship to shore in 1896.[1] The fundamental work of scientists of the nineteenth century paved the way with invaluable theoretical and experimental discoveries related to both wired and wireless data communication systems.

The advancement in the digital computer during the late 1950s even induced more technological development in data communications. Data communications has become an extremely important aspect of the modern world. It has been a subject of great interest for many applications for many decades.

Two key aspects of data communication systems are the transmission media and the data communication protocols. The transmission medium is the physical path over which data travels from the source to the receiver. The transmission medium can be a twisted-pair of copper wires, coaxial cable, optical fiber or wireless media such as radio waves. On the other hand, the communication protocol is a set of rules and conventions essential for the source and the receiver to communicate reliably and efficiently. Two devices wishing to communicate with each other cannot just begin data transmission arbitrarily. The sender and the receiver must agree on a common set of rules, protocols before they can communicate with each other.

## 2.2. Related Research Work

Water systems need to grow as population increases, and wireless networks can make upgrading and expanding those systems easier and cost effective. Rather than excavating trenches or physically adding cables to existing installations, additional components and facilities can be added and monitored through signals sent via airwaves to control centers.

There are many different types of data transmission systems available in the market to meet most remote water monitoring application needs. Global Water offers several types including cellular data transmission systems, satellite data transmission systems, radio data transmission systems and telephone modem data transmission sets for the remote water monitoring system. In addition, Stevens remote telemetry systems offers effective wireless communication technologies data include cellular, radio and satellite telemetry. Although data telemetry systems from both of these two companies might provide viable solutions for this project, they are pre-configured with limited capabilities and difficult to expend for future integration of the system for this specific kind of research purpose. Moreover, these commercially available systems still have relatively high price.

# 3. Design Process

This section describes the step-by-step design process used to develop the RF communication system. The design process proceeds in the top-down manner not only to reduce the complexity of the system but also to enhance modularity of the design. Based on the problem statement that defined the need of the system, the first step of the design process is to define system requirements which are presented in detail in Section 4. These requirements essential elements in the design process as they provide all the information necessary to successfully produce a solution to the design problem. These requirements also serve as criteria that a design solution must meet or attributes that the system must possess to be considered successful.

The next step involves selecting accurate design parameters required for designing the RF communication system. The detailed description of design parameter is presented in Section 5. In this system design, design parameters such as the operating frequency and transmit power are required to set prior to choosing other system components. Specifically, these parameters are used to determine the specific type of radio equipment. Additionally, this section provides essential information of the licensing process in the United States, where licensing is governed by the Federal Communications Commission (FCC).

After choosing the design parameters, system modules are defined in accordance with system requirements. In order to analyze the complex system design in a simple way, it is broken down into a smaller unit call modules. There are three main system modules contained in this design. In Section 6, a high-level description of the system and functionalities of each system module are described together with a diagram. The advantages of breaking the design into parts are to clarify what needs to be done in order to accomplish the system design as well as to create less complicated design for easy modification and integration.

Once system modules are clearly specified, possible design alternatives of each module are investigated individually and then analyzed against each other in order to meet all system requirements. After analyzing alternative solutions, the decision for all system components is made to construct a final prototype design implementation. In

Section 7, general design alternatives for each system module are discussed, as well as the detailed analysis of design alternatives along with a specific recommendation for all functional components, both hardware and software is presented.

Section 8 contains the detailed documentation of the final design and prototype implementation of the RF communication system along with the description of all system components both hardware and software, and their corresponding functionalities. Finally, the performance of the final implemented system is discussed in Section 9 based on the preliminary design criteria. First, each system module is tested individually. Second, all the modules are integrated into the complete prototype system and tested again at the system level.

# 4. Design Requirements

In order for the RF data communication system to provide a useful service for the Geology Department, it must meet or exceed certain design requirements. This section provides detailed requirements which act as a guideline in determining whether a given system design is considered practical. The requirements that apply to this particular system design are based on the preliminary field server at Ballston Lake and the research conducted after a need for the system has been establish. These requirements also serve as the basis of the further design process.

## 4.1. Overall Design Requirements

The primary function of the RF data communication system is to transmit the water property data successfully and reliably from Ballston Lake to Union College and to store the data securely for future analysis. The system consists of both hardware equipment and software components; they cooperate with each other to provide an excellent data transmission service. The followings are design requirements:

- **Operate continuously for 10 years**

  The system must operate continuously at least 10 years of period for a long-term research purpose. According to Prof. Shaw from the Geology Department, the long-term research is necessary to observe possible unusual occurrences as well as unique water characteristics within the lake by analyzing the water property data. The durability and reliability of hardware equipments (especially antennas that will be exposed in the air for long periods) and the functionality of software components will greatly affect the overall performance of the system.

- **Operate over distance of 8 miles**

  The distance between two communicating station is another important criteria the system. Any RF transmission must account for how far apart are two antennas. The estimated direct-distance between Union College and Ballston Lake is about 8 miles. It is measured using Google Map as shown in Figure 2. Thus, the system must provide uninterrupted connection over the distance of 8 miles.

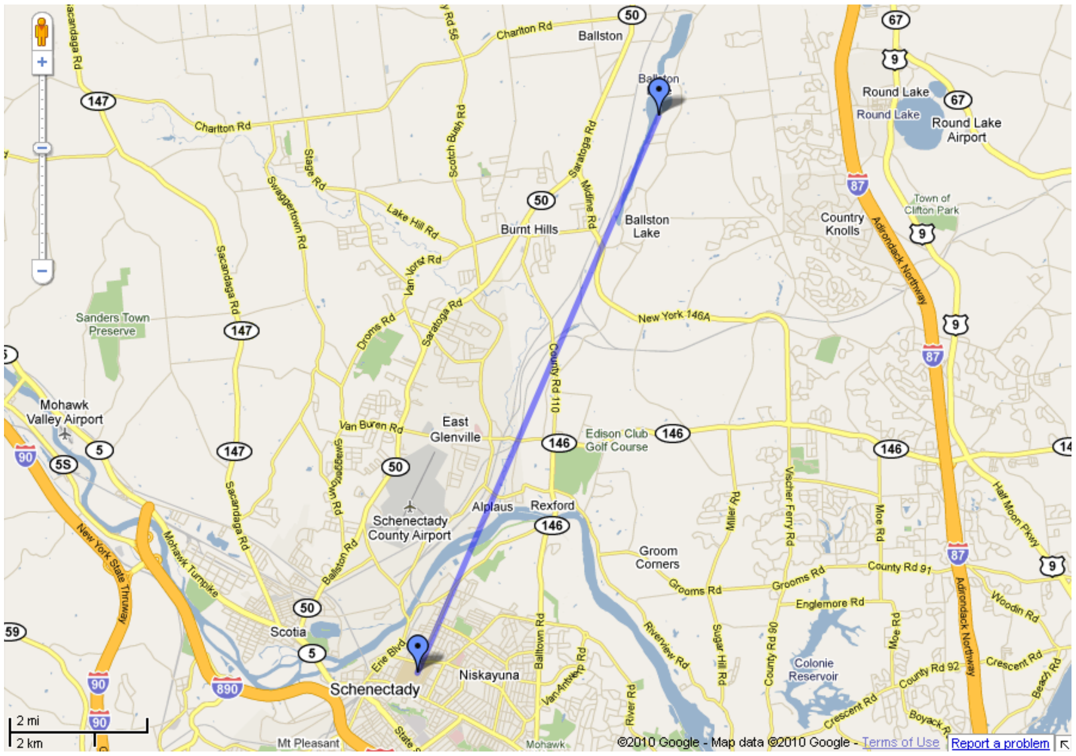


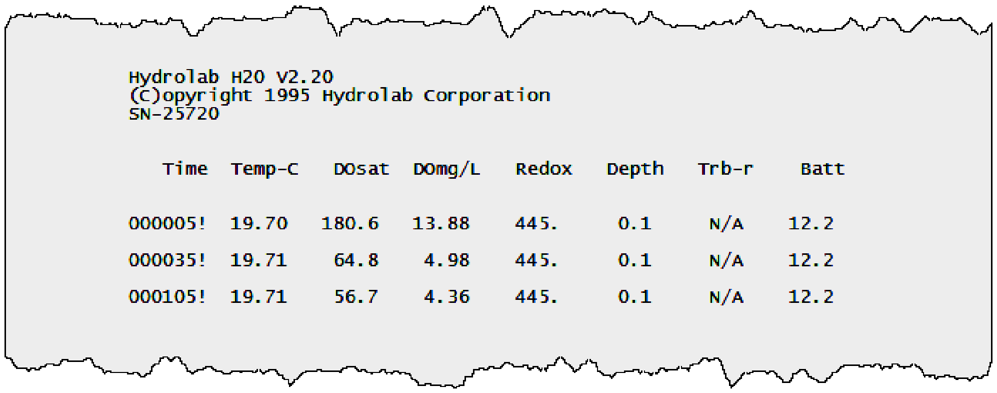
**Figure 2: The direct-distance between Union College and Ballston Lake on Google Map**

- **Operate properly in all weather conditions**

  Weather condition is an additional factor that affects the propagation of radio waves. As the wavelength becomes shorter with increases in frequency, precipitation has an increasingly important attenuation effect on radio waves. Therefore, the system needs to overcome attenuation of the radio signal caused by various forms of precipitation such as rain, snow, fog and hail. This important factor imposes a limitation on selecting the operating frequency for the system.

- **Provide large amount of data storage at Union College**

  Data storage and management are also critical aspect of the system. Although the amount of the data from single measurement of water properties from the sensor package (as shown in Figure 3) is relatively small, the cumulative data over 10 years of research will need a large amount of memory storage space on a computer at Union College site. Most computers nowadays usually provide a large amount of memory for data storage as well as an easy upgrade for an additional storage space.



```
Hydrolab H20 V2.20
(C)opyright 1995 Hydrolab Corporation
SN-25720

   Time   Temp-C    DOsat  DOmg/L    Redox    Depth    Trb-r     Batt

000005!   19.70    180.6   13.88     445.      0.1      N/A     12.2
000035!   19.71     64.8    4.98     445.      0.1      N/A     12.2
000105!   19.71     56.7    4.36     445.      0.1      N/A     12.2
```

**Figure 3: Sample water property data measurements**

- **Provide data storage at the lake when the link is down**

The system at Ballston Lake site must provide a temporary local storage space so that the data will not be lost during an emergency situation such as the loss of the data transmission channel for extended period of time due to natural disasters or human errors. After restoring the connection, all the data at the temporary storage space will be retransmitted to the intended destination. The system also needs to provide tight security for the stored data to avoid both intentional and unintentional deletion of data.

- **Minimum of 9600 bits/sec for the transmission rate**

According to Professor George Shaw from Union Geology Department, 3 measurements will be taken at every foot starting from the bottom of the lake. Since the lake is 120 feet deep, there will be total of 120 sets of measurements or ($120 \times 3 = 360$) measurements for every complete process of data collection. This process will be repeated every day for 10 years of period. Each measurement contains variable length of character which depends on the number of water property to be measured. Given the amount of data that will be collected in the lake through the sensor package, the system needs to have minimum of 9600 bps (bits per second) for the transmission rate in order to establish effective communication between two sites in timely manner.

- **Need error checking, reporting and retransmission**

The system requires additional capabilities such as error checking, error reporting and retransmission in order to achieve reliable data transmission over an unreliable radio communication channel. The error checking mechanism is needed to allow the receiver to detect when bit errors have occurred. Error reporting serves as the receiver's explicit feedback to notify the sender whether or not a packet is received correctly. If the receiver receives a packet with errors, the sender will retransmit by the same packet.

- **Design must be subdivided into functional modules**

The system design must be flexible. It is possible that developers implementing the system would like to add additional hardware or software components or even remove some of the included components. For instance, the transceiver hardware module breaks down during the 10 year operation, and it cannot be fixed or replaced with the same module for some reasons. In this situation, a new transceiver is needed to integrate easily into the system without varying other components. Therefore, all system components should be designed in a modular fashion, and detailed documentation should be provided.

## 4.2. Goals

The primary goal of this project is to design and implement a system that will meet all abovementioned design requirements. More specifically, the data communication system will operate reliably with least amount of human intervention over the course of research period. The system will provide a proper and secure data storage mechanism to help users manage large amount of data for future analysis. If the system can fulfill these system-level design requirements, it will provide a useful and educational service to both faculty and students in the Geology Department.

# 5. Design Parameters

In this system design, one of the most crucial steps is to specify design parameters such as operating frequency and transmit power that define the RF data communication system. This section discusses the characteristics of VHF and UHF frequency bands as they have great impact on the determination of the right frequency band. In addition, this section contains the detailed information about the licensing process for getting a dedicated frequency channel for the data communication. Finally, it discusses the transmit power limitation at certain frequency band.

## 5.1. Operating Frequency

Choosing the right type of radio frequency spectrum is extremely important when designing RF Links. Because the selecting operating frequency band will also drive many other design requirements for antenna and transceiver, it is a good place to begin the design process. Radio spectrum which is part of the electromagnetic spectrum has the range of frequencies from approximately 30 kHz up to more than 300 GHz for different radio communications. Each range of frequencies has unique characteristics and performs different functions. What is significant about frequency band for two-way radios system is that it affects transmission range under certain conditions. At higher frequencies, radio signals are more susceptible to attenuation, absorption, refraction, scattering and reflection due to atmosphere conditions, metallic objects, ground terrain, and large buildings or structures between two sites. All these undesirable conditions will cause signal fading in which radio energy is lost. The estimated path loss for line-of-sight range with the ideal isotropic antenna can be determined using the free space loss (FSL) equation as follow [2]:

$$L_{dB} = 20\log(f) + 20\log(d) - 147.56\ dB$$

where

f = operating frequency (MHz) and

d = propagation distance between antennas (m).

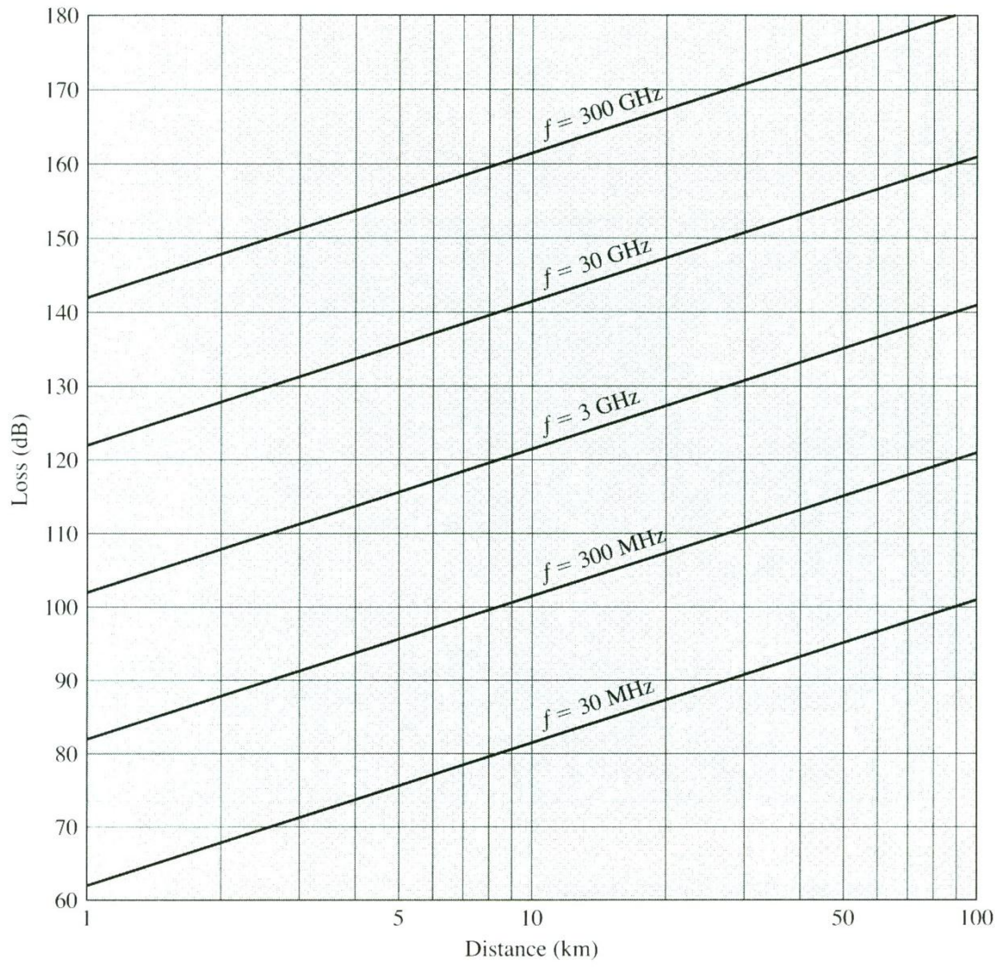


**Figure 4: Free space loss [2]**

Free Space Loss refers to the reduction of the signal strength as the signal radiates away from the source. Figure 4 illustrates the above free space loss equation using visual graph. In that figure, we can see that the loss can result at any frequency range, but in general is more severe at higher frequencies. In many circumstances, total losses can become so great that radio signals become too week for communication.

In order to avoid sever signal fading over the communication channel yet still able to achieve the data transmission rate of 9600 bps, I have considered two licensed radio frequency bands: very high frequency (VHF) and ultra high frequency (UHF). VHF operates in the range from 30 MHz to 300 MHz and UHF operates in the range from 300 MHz to 3 GHz. [2] Even though unlicensed frequency bands are available at higher frequency, I remain to choose a licensed frequency within VHF and UHF ranges as a viable solution. Besides a complicated FCC licensing process, the licensed frequency for RF data communication offers a dedicated link with far more security and system stability especially for this 10 year research project than unlicensed solutions. In addition, the licensed frequency is also less susceptible to radio signal interference.

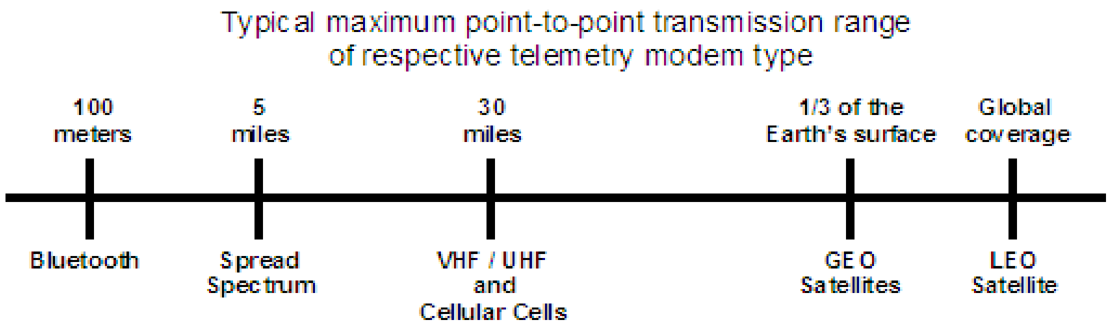

Typical maximum point-to-point transmission range of respective telemetry modem type
100 meters
5 miles
30 miles
1/3 of the Earth's surface
Global coverage
Bluetooth
Spread Spectrum
VHF / UHF and Cellular Cells
GEO Satellites
LEO Satellite

**Figure 5: Typical maximum point-to-point transmission range of respective frequency range**

VHF and UHF ranges also allow the longer distance communication under more obstructed circumstances. Based on the information presented in Figure 5 [3], the optimal range of communication offered by VHF and UHF is approximately 30 miles which is more than the distance between two sites in my projcet.

### 5.1.1. FCC Licensing Process

Based on the given information about the relationship between FSL and transmission range, the choice of frequency band within VHF and UHF range is practical for the RF data transmission system. Before purchasing two way radio transceivers and antennas to operate data transmission between Ballston Lake and Union College using a dedicated frequency band, the Federal Communications Commission (FCC) requires getting a license to operate that frequency band. Like all other government requirements, great amount of paperwork is involved in this licensing process, and it is confusing and time consuming.

Instead of going it alone through this complicated application process, I decide to use a licensing coordinator that handles all of the paperwork and processing. During this process, the coordinator acquires several questions about the sites, the radio transceivers and antennas that will be used in the system, and how those devices will be used. Then they fill out all the essential forms and get them submitted to the FCC for approval. In order to be able to provide precise answers to the coordinator, I have performed thorough RF link analysis which is presented with detailed information in the later section. The very first thing to do even before contacting the coordinator for the actual licensing, it is required getting an FCC Registration Number (FRN). This identifies who (or what entity) is applying for licenses. An FRN can be obtained online at http://wireless.fcc.gov/uls/ on the FCC's site.

Currently, I am working with WTC, a spectrum coordinator, for the FCC licensing process. Also, WTC will assign a frequency on which to operate, at a specified maximum power, within a specified geographic area. They will choose these frequencies carefully to avoid interference between your usage and other existing users. Although, the application process becomes easy because WTC helps with paper work, the FCC approval takes several months. Since this process takes longer than what I expected, it affects the final design and implementation of the system. In addition, the choice of the radio system hardware components for the final design is tentative.

Once the license is approved by the FCC, the actual radio equipment will replace the tentative ones in accordance with the license. Soon after the replacement of radio equipment, it is required to notify the FCC that the operation of the data transmission has begun. If not, the FCC will revoke the license. This is to prevent frequencies from being tied up by a licensee who is not actively using them.

## 5.2. Transmit Power

The determination of transmit power is usually driven by the FCC regulatory and power-consumption considerations. The FCC sets the specific limitation on transmit power not only at a certain frequency band but also for different transmission purposes. For example, FCC allows up to 1 W of transmit power in the United States in the 2.4

GHz band. [4] In this system design, this parameter is specified according to the operating frequency assigned by the spectrum coordinator.

With both operating frequency and transmit power for this particular RF data communication system determined, other hardware components such as transceiver and antennas that comprise of the radio system can be specified.

## 6. System Modules

This system design is developed using the top down approach aiming at determining the basic system modules. To reduce the complexity as well as to have the better understanding of the overall system design, the system is broken down into smaller modules. The advantages of the modular design are easy modification and integration of each distinct unit during the design process and easy testing and debugging not only at the low-level design but also at the system level implementation. In this section, each of the system modules is defined individually, and essential functional behavior of each module is presented.

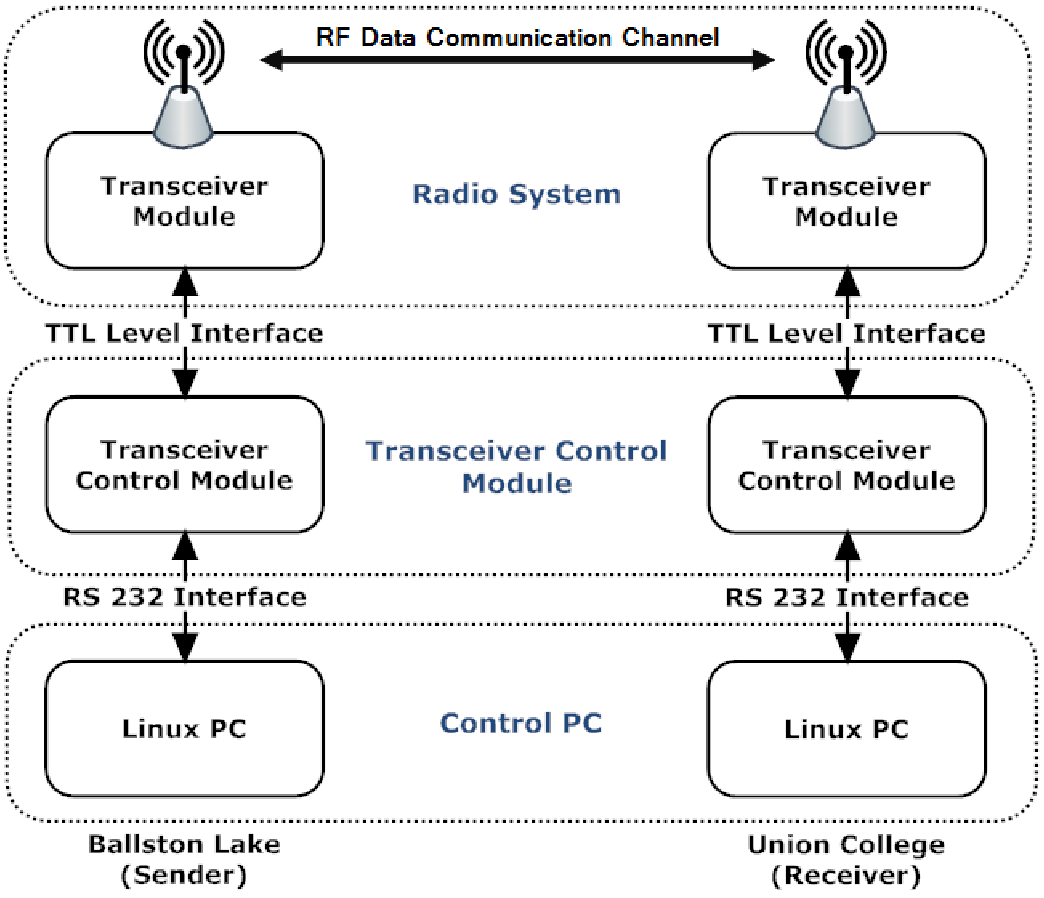


**Figure 6: A block diagram of system modules and interfacing between each module**

As I mentioned before, this design project only deals with the RF data transmission between Ballston Lake and Union College. Figure 6 depicts the overall design of the RF data communication system between the lake site and the Union site together with the block representation of essential system modules. In this system design, there are three major functional modules: radio system, control interface and control computer. Each set of functional modules is incorporated into an entire RF data communication system between Ballston Lake and Union College.

### 6.1. Radio System

Antenna, transceiver and modem function together as a radio system to establish the two-way radio communication system. Sometimes the transceiver comes together with modems in one piece. This system module is responsible for transmitting data from the control computer through the control interface or vice versa, and then generating RF signals for data transmission and receiving between the lake site and the Union site by converting RS232 signal through the RF modulation.

### 6.2. Transceiver Control Module

A control interface requires a hardware component as well as a software program that operate together not only to the control transceiver hardware unit but also to handle the data transmission between the control computer and the transceiver. In addition, this interface enhances the flexibility in replacement or integration of the radio system with another different radio system in the future without affecting any hardware or software components on control computers.

### 6.3. Control Computer

A control computer performs all the necessary control functionalities for the data transmission process with a specific set of rules and requirements called a protocol between the lake site and the Union site. It also serves as a data storage unit on both sites.

# 7. Design Alternatives

In following section, the detailed design process is further presented with design alternatives considered for each of the functional component, both hardware and software, in the overall system.

## 7.1. Hardware Alternatives

### 7.1.1. Antenna Alternatives

The selection of a suitable antenna design depends on the communication distance requirement, the operating frequency and the radio signal propagation mode. The physical distance, as I mentioned in the design requirements section, is approximately 8 miles. The RF data communication link between two sites is considered to be a bidirectional point-to-point link, and the data communication operates in line-of-sight propagation mode. The same antenna can be used with the same characteristics as transmit and receive antenna. Moreover, the antenna is characterized by its center frequency, polarization and gain. The center frequency is referenced to the operation frequency which I mentioned in the previous subsection. For best results in line-of-sight communication, antennas at both end system should also have the same polarization. Polarization is determined by the position of the radiating element or wire of the antenna with respect to the earth. [5] Since the RF link consists of two fixed stations communicating only with each other, the use of directional or gain antennas can offer an advantage. For the purpose of obtaining gain and directivity it is suitable to use a Yagi, a Quad or a Quagi (a hybrid Quad and Yagi) antenna. For this design project, I have chosen a Quagi Antenna design as shown in Figure 7. The Quagi antenna design consists of Yagi-style directors and Quad-style quadloops for reflectors. According to the article [6], Yagi-style directors deliver better gain than quad loops when the antenna is extended beyond four or five elements, and also a quad-style driven element and reflector provide good gain, good immunity to noise resulting from static buildup, and extreme ease of construction and impedance matching. A proper matching of the antenna to the feed point implies that maximum transmit power will be transmitted from one site to the other.

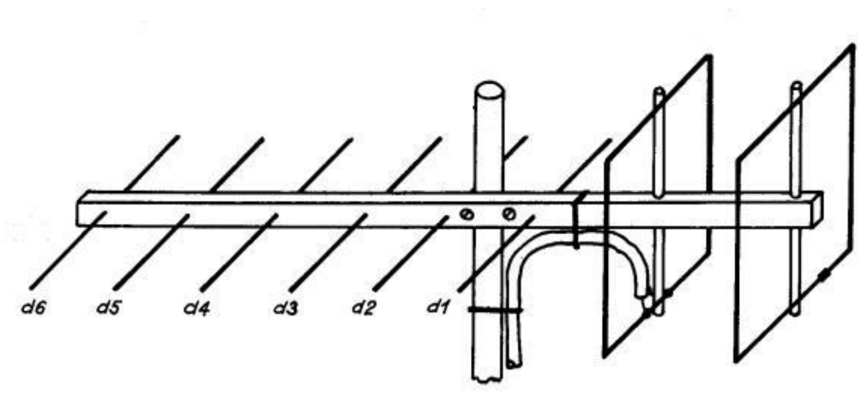


**Figure 7: An eight element Quagi antenna design**

### 7.1.2. Transceiver Alternatives

Transceivers are integral parts of this system design. Four major factors that lead to the proper choice of transceivers are the operating frequency, the transmit power, the receiver sensitivity, the data transmission rate requirement and the price. Based on the decision of the operating frequency range, VHF/UHF radio transceivers are considered using in the project. The determination of transmit power usually depends on the FCC regulatory and power-consumption considerations. Moreover, receive sensitivity indicates how faint an RF signal can be successfully received by the receiver. The lower the power level that the receiver can successfully process, the better the receive sensitivity. However it is also important to make sure that other transceiver specifications such as operating frequency range and transmit power from different manufacturers are equivalent when comparing the sensitivity. In most VHF/UHF transceivers, 6.25 kHz, 12.5 kHz and 25 kHz channel bandwidths are available in each frequency band which is usually pre-configured by manufacturers. RF modems, which are sometimes integrated into transceivers, also supports a variety of modulation schemes such as FSK, GMSK, four-level FSK and AM as well as different output power levels. Based on all abovementioned considerations, I have narrowed my choice of the transceiver to Ritron – DTXM and Satelline – M3TTL1. Satelline utilizes the 4FSK modulation to transmit about 9600 bps of raw data in a 12.5 kHz channel. On the other hand, Ritron offers higher power output than Satelline – M3TTL1. Both transceivers support sleep mode as power-saving technique to turn off the devices when there is no data to transmit. Table 1 provides more detailed descriptions of two transceiver models.

| **Manufacturer (Product)** | **Frequency Range** | **Transmit Power** | **Receiver Sensitivity** | **Estimated Price** |
|---|---|---|---|---|
| Ritron (DTXM) | 136 – 162 MHz VHF<br>148 – 174 MHz VHF<br>217 – 245 MHz<br>400 – 420 MHz UHF<br>450 – 470 MHz UHF | 1 – 6 watts VHF<br>1 – 3/6/9 watts UHF | < 0.28 uV | ~ $300 |
| Satel (Satelline – M3TTL1) | 135 – 174 MHz VHF<br>218 – 238 MHz<br>360 – 470 MHz UHF<br>869 MHz UHF | 100 – 500 mW VHF<br>10 mW – 1 W UHF | < 0.71 uV | N/A |

**Table 2: Descriptions of transceivers from Ritron and Satel**

### 7.1.3. Transceiver Control Interface Alternatives

The main function of the transceiver control interface is to isolate the characteristic of transceivers and control computers. This component is needed to achieve one of the important design requirements, modularity of system components. Some transceivers require pauses in between each byte of data; some utilize a carrier detect signal to indicate that data transmission or receiving is ready; and some even need to turn on and off to limit the power consumption. In order to provide these control functionalities, I consider using a microcontroller unit (MCU) as a control interface. Moreover, this control interface allows easy integration or replacement of the transceiver in the future. There is a wide variety of MCU development boards available from different manufacturers. Instead of exploring different development boards to determine the suitability for my project, two MCU development boards, which I most familiar with, from Silicon Lab and Parallax are chosen. I investigate Silicon Lab's C8051F020-TB microcontroller development board and Parallax's BASIC Stamp II microcontroller development board.

Silicon Lab's C8051F020-TB microcontroller development board utilizes CIP-51 core chip. Figure 8 shows a pin diagram of the Silicon Lab's C8051F020 microcontroller development board. There are two separate memory spaces in the on-board chip: program memory and data memory. The chip consists of 64K bytes of in-system programmable FLASH memory, 4352 bytes of data memory on-chip RAM and 64K bytes external data memory interface with address space. The chip also provides the following standard features: on-chip watchdog timer, 22 interrupt sources with 2 priority levels, 5 general purpose 16-bit timers run as system and a programmable counter array (PCA) with five capture/compare modules. It also contains additional serial communication interfaces such as UARTs implemented in hardware. This MCU can be programmed using assembly, C programming language and the combination of two. With the CIP-51's maximum system clock at 25 MHz, it has a peak throughput of 25 million of instructions per second (MIPS). [7] Most importantly, both Silicon Labs and Keil have web sites with extensive application notes about the 8051MCU.

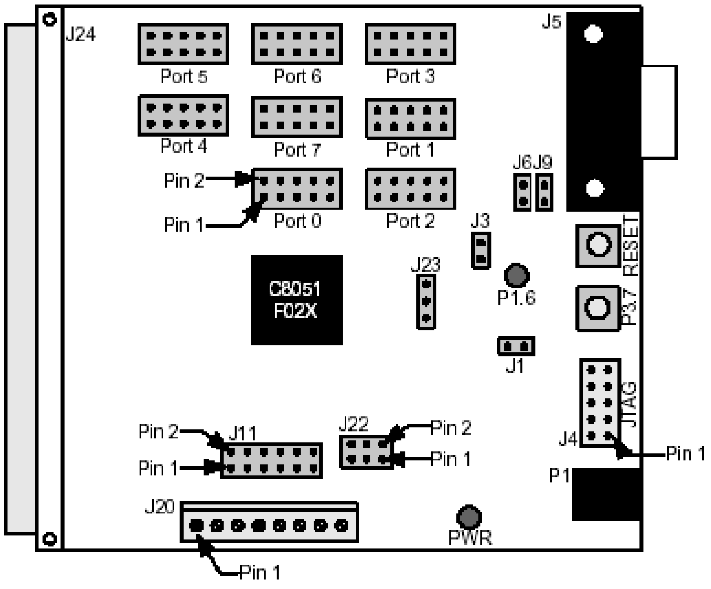


**Figure 8: A pin diagram of Silicon Lab's C8051F020 microcontroller development board [8]**

Parallax's BASIC Stamp II microcontroller development board uses a PIC16F57 core chip from Microchip Technology Inc. Like the C8051F020 module, the PIC16F57 chip's memory is organized into program memory and data memory. It has internal 2048 x 12 bytes of program memory and 32 bytes of RAM for data memory. Figure 9 shows a pin diagram of the Parallax's BASIC Stamp II microcontroller development board. The chip also supports 16 I/O ports and two special serial ports (1 input, 1 output). However, the chip not support interrupts routine. The features supported on this development board are far less compared to the Silicon Lab's board. The BASIC Stamp II module uses a hybrid form of the BASIC programming language called PBASIC, but it does not except assembly language routines. Based on the given operating speed of PIC16F57, the chip can achieve a peak throughput of 5 MIPS. The Basic Stamp II development board is a well designed experiment platform with excellent documentation and a plethora of support from Parallax website.

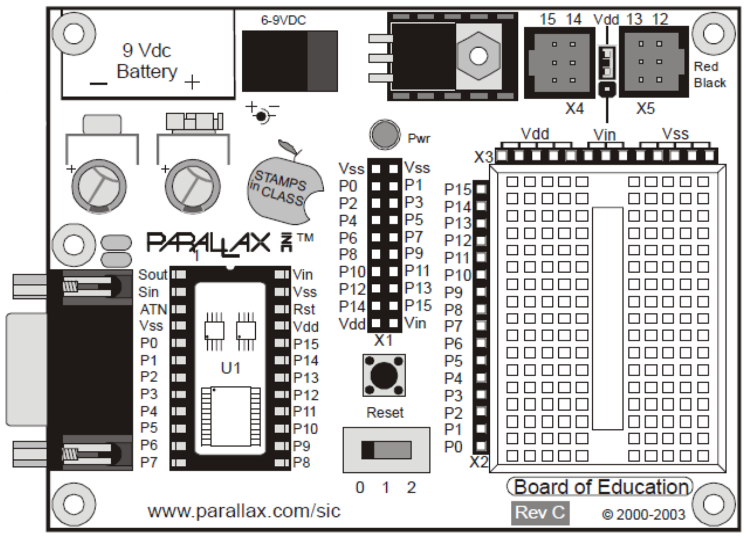

9 Vdc Battery
6-9VDC
15 14 Vdd 13 12
Red Black
X4 X5
Pwr
Vdd Vin Vss
X3
STAMPS in CLASS
PARALLAX
Sout Sin ATN Vss P0 P1 P2 P3 P4 P5 P6 P7
Vin Vss Rst Vdd P15 P14 P13 P12 P11 P10 P9 P8
U1
Vss P0 P2 P4 P6 P8 P10 P12 P14 Vdd
Vss P1 P3 P5 P7 P9 P11 P13 P15 Vin
X1
Reset
0 1 2
P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
X2
Board of Education
Rev C
© 2000-2003
www.parallax.com/sic

**Figure 9: A pin diagram of Parallax's BASIC Stamp II microcontroller development board [9]**

Although the Basic Stamp II is a great platform for learning and small simple projects, its processor is slower with less memory than the C8051F020-TB. Figure 10 shows a comparison of peak throughputs of two 8-bit microcontroller cores with their maximum system clocks. In addition, BASIC compilers are usually in the form of interpreters and the code produced is usually slow. Unlike C compilers, most BASIC compilers are not structured and this makes the programs maintenance a difficult task. On the other hand, the MUC with C programming language compatibility is more desirable in this project as the system design requires flexibility and modularity. If C programming language is used for both control interface and control computer, all the programs developed on each individual component will be easily interchangeable. Therefore, Silicon Lab's C8051F020-TB microcontroller development board is more suitable for this project. Also, I have taken ECE 352 Embedded Microcontroller Systems with focus on the exact development board. I am already familiar with the board, and development tools as well as support from faculty are readily available.

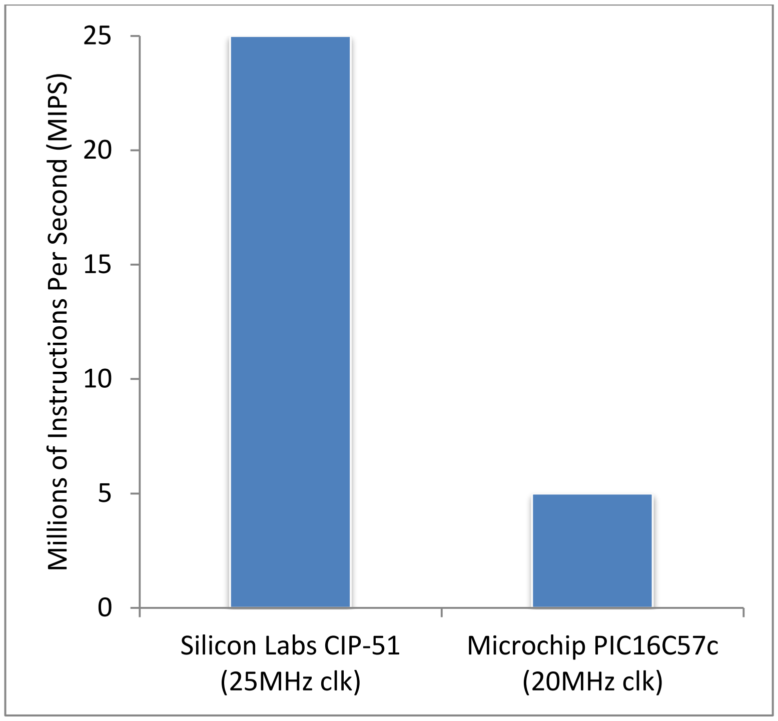


**Figure 10: Comparison of Peak MCU Execution Speed**

### 7.1.4. Control Computer Alternatives

The data transmission process in this design project operates on two separate computers, one at the lake site and another one at the Union site. Choosing an appropriate computer requires to meet system security as well as reliability requirements. There are numerous reasons why I have chosen Fedora as an operating system (Linux OS) over Window and Mac. The most obvious advantage of using Linux is the fact that all applications are free of charge, while Microsoft and Apple products are available for an expensive and sometimes recurring fee. Unlike Window and Mac, Linux even allows programmers to run all common UNIX software packages and create software programs in an extensive free development environment. Moreover, Linux is a highly secured operating system yet provides flexible file access permission systems to prevent access by unwanted visitors or viruses. In addition, Linux is a more stable operating system. It does not need to be rebooted periodically to maintain performance level. It rarely freezes up or slows down over extended period of time due to memory leaks and such. It also allows programmers to write programs that can easily access hardware components such as RS 232 port. Finally, Linux is also greatly suitable for this type of design project from the perspective of software programming.

## 7.2. Software Alternatives

There are a lot of programming languages available right now - everything from the extremely high level to the low level power of assembly, and a good variety of specialized options in between such as Python. In order to meet design requirements of system components modularity and system reliability, it is required to choose

There are several good reasons to use C programming language for both transceiver control interface and data transmission process in this project. First, there is a plenty of source codes and documentations available on the Internet. The website that I find particularly useful is http://www.java2s.com/Tutorial/C/CatalogC.htm. The website provides excellent C examples to be able to understand the basic structure of the language.

Moreover, C is a high level programming language that is portable across many hardware architectures which I have already described in Section 4.3.3. This means that with a few insertions of libraries and include files, architecture specific features such as register definitions, initialization and start up code, which must be made available to a program, can be specified.

Finally, C provides easy implementation as well as a better picture of advanced topics like exactly how serial data communication works. This higher level language makes the design process a little bit simpler, and still offers easy understanding of what is exactly going on at the low level, and so when things stop working, it is much easier to know what is going on and how to fix problems. Furthermore, a useful data communication management tool called Kermit is programmed in C. This program enables me to test my own program at various stages of design process.

# 8. Final Design and Prototype Implementation

This section contains the detailed documentation about the final design and the prototype implementation of the data communication system along with the description of each component and its corresponding function. First, the layout of hardware design section is presented with a depiction of overall system design at functional level. Second, software design section consists of the full description of data transmission protocol with complete diagrams and control interface program for transceivers on both Lake site and Union site.

## 8.1. Hardware Design

The system components of the hardware design consist of three parts: radio system, transceiver control modules and control computers running on Linux. Both the lake site and the Union site system contain the same set of hardware components. However, each computer on both sites operates separately with minor different capability for data transmission process while the rest of components have identical functionalities. Moreover, it is important to note that there will be a slight difference between the current prototype implementation and the final implementation. Since I could not manage to obtain FCC licensing approval in time, I could not get a dedicated frequency link to operate data transmission legally. Consequently, I could not get transceivers which require to be pre-configured at a specific approved frequency during the order. The current implemented radio system does not meet design requirements and is needed to be replaced with a better system in accordance with FCC regulatory and design requirements. The following hardware design phase focuses on interfacing between each component as shown in Figure 6. In the following section, system components are separately presented because the lake site system and the Union site system have slightly different function.

### 8.1.1. Lake Site Equipment

#### Radio System

The radio system consists of an antenna, a transceiver and a modem. The latter two components sometimes come together in one piece. An ultra low power data transceiver module with an omnidirectional antenna (HAC-UM96) from Sparkfun, which is shown in Figure 11, is chosen in this prototype design.

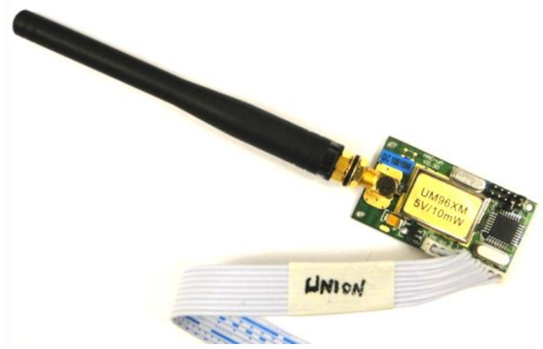


**Figure 11: HAC-UM96 ultra low power data transceiver module**

This module is suitable for a two-way communication; in other words it allows both data transmission and reception in half-duplex mode for point-to-point or point-to-multipoint. The carrier frequency operates at 433 MHz which is within the UHF range. It provides 8 frequency channels with easy pin configuration. Also, it offers the transmission distance of approximately 300 m and 500 m at the height of 2 m and 5 m respectively within the range of visibility. The data transmission rate is preconfigured by the manufacturer at 9600 bps. This module also comes with 3 interface modes including TTL level, standard RS-232 and RS485 for a flexible design. All modulation, demodulation, buffering and error detection is handled internally. There is no program implemented specifically for this module. Moreover, it supports the sleep function that can further reduce its, power consumption. This module can closely represent the component in the final design with reasonably low price and minor differences at the hardware level. However, it is important to note that the current transceiver module is used for a temporary testing device; it has to be replaced with a much higher power transceiver for long distance data communication.

#### Transceiver Control Module

The transceiver control module is implemented using an 8051 microcontroller development board which is a single chip microprocessor system. This board contains a C8051F020 system-on-a-Chip manufactured by Silicon Laboratories. This module handles data transmission between the transceiver module and the computer. The program that controls the data transmission is implemented on the board using C programming language. The detailed description of the program will be provided in the later section. Another important function of this module is to allow easy modification or replacing of the transceiver in the future without changing any hardware or software components in the system.

#### Control Computer

The control computer runs on Fedora, a Linux-based operating system. The lake site computer performs two functions: collect data directly from the sensor packet and store the data in the memory location; and transmit the collected data from the lake site computer to the Union site computer for analysis. In order to provide these two functions, two separate independent programs are implemented using C programming language.

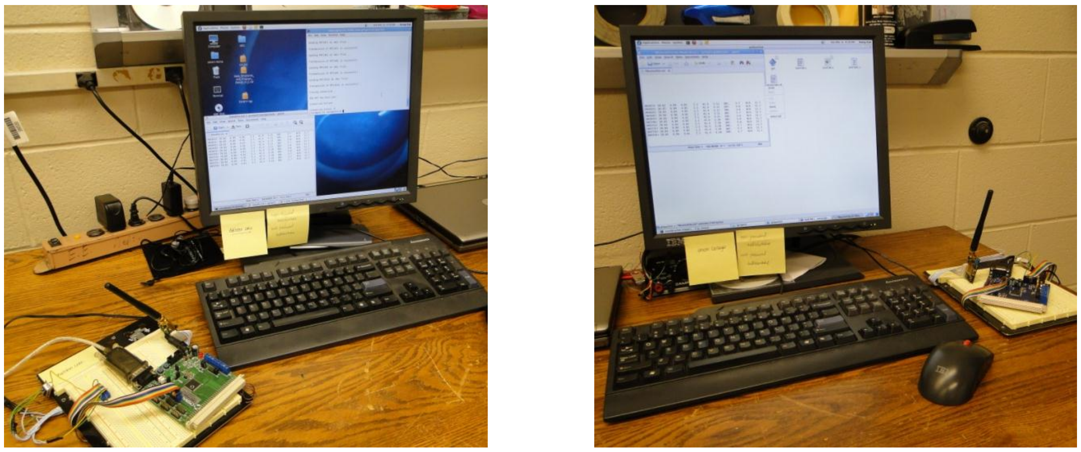
### 8.1.2. Union Site Equipment

#### Radio System and Transceiver Control Module

As you can see in Figure 12, components used in the radio system and the transceiver control module for the Union site is identical to those for the Lake site. Similarly the same functionalities are also provided by these units.

**Control Computer**

There is a difference in the function performed by the Union site control computer. It waits for incoming data from the lake site control computer and stores it in memory location for future analysis. To perform this task, a single program is developed using C programming language.



(a) Ballston site equipment (b) Union site equipment

**Figure 12: Hardware component setup for an RF data communication system**

Due to the multiple devices used in this system, interface is a vital part of overall system capability. Poorly matched interface system will limit functionality and impede performance. The interfacing between devices can be referred back to Figure 6. In order to allow proper data transmission to and from between the radio system and the transceiver control module, the TTL level serial interface is used by both functional units. On the other hand, the data transmission between the control computer and the transceiver control module is accomplished using an RS-232 serial cable. There are slight differences between TTL level and RS-232 serial interfaces mainly at hardware level. Serial communication at a TTL level will always remain between the limits of 0 V and Vcc, which is often 5V or 3.3V. A logic high is represented by Vcc while a logic low is 0 V. On the other hand, the RS-232 standard uses a negative voltage (from -3 to -25V) to represent a logic high, while a positive voltage (from +3 to +25V) is used for a logic low. Very short distance serial data exchange can use TTL level. Moreover, the high voltages of an RS-232 signal help to make it less susceptible to noise, interference, and degradation. This means that an RS-232 signal can generally travel longer physical distances than their TTL counterparts, while still providing a reliable data transmission.

## 8.2. Software Design

One of the most important parts of this project is the interaction between software and hardware components, they work together to provide functionality for the data communication system. Interfacing all essential hardware components correctly is not sufficient to accomplish this project. Software programs on both transceiver control interface and control computer is necessary for data transmission and control. In the following section, a protocol design process is described step by step, and a transceiver control program is explained in-depth.

### 8.2.1. Protocol Design

In order to establish reliable data transmission between two computers separated by the distance of 8 miles, there needs to have a proper way to exchange messages which can be understandable to both computers and to perform specific actions when these messages are sent and received. These are the key defining elements of a protocol. “A protocol defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.” [10] A data transmission protocol used in this project is called the master and slave model. In this model, a communicating computer (known as the master) initiates and controls the data transmission to and from another computer (known as slaves) at the different end separated by the distance of 8 miles. Once the master and slave relationship is established, the direction of control is always from the master to the slave. This model is also referred to as the primary and secondary model. The advantage of the master and slave model is simplicity in terms of implementation. Since this model operated at a simple level, there is no need to implement using protocol stacks or layers. In the following section, please note that I will use the sender and the receiver notation for the master and the slave respectively.

### 8.2.2. Packet Format

The initial step in developing data transfer protocol is to define packet types. In this protocol design, I defined two types of packets: control packet and data packet. The packet block structure is shown in Figure 13. The length of control packet is fixed, but the length of data packet is varied depending on the amount of data value.

| | | | | | |
|---|---|---|---|---|---|
| SOP | Opcode | Control Message | CRC Hi Byte | CRC Low Byte | EOP |

**(a)**

| | | | | | |
|---|---|---|---|---|---|
| SOP | Opcode | … Data Value … | CRC Hi Byte | CRC Low Byte | EOP |

**(b)**

**Figure 13: (a) Control packet format and (b) Data packet format**

All packets in this protocol design contain specific information. Each block except the data value block holds 1 byte of information. The contents include:

- **Start of Packet (SOP)**

  This is used as a unique way to recognize the beginning of a packet as well as to filter the incoming packet against other packets which contain useless data that might cause undesirable effects on the communication system. It also allows a receive end to reset if the packet is cut off during the transmission.

- **Opcode**

  This is used to identify the packet type. There are two types of packets: control packet and data packet. Since it holds 1 byte (8 bits), the total number of different packet types can be defined are 256. The available packet definition is more than adequate for this protocol design.

- **Control Message**

There are five different control messages defined in this protocol design. 3 control messages, CONNECT, READY and BYE are used for initializing the connection between two communicating computers. 2 control messages, ACK and NAK are used to indicate receive status of a data packet. The ACK message allows the receiver to let the sender know what has been received correctly. The NAK message is used to indicate what has been received in error and thus require retransmission.

- **Data Value**

This is a group of bytes, and the amount of the data depends on the number of the parameters such as temperature, conductivity, salinity, redox, depth and dissolved oxygen that are required to collect at the lake. Each data packet contains a single measurement (or a single line of information shown in Figure 3 from section 4) from the sensor package. All data values are represented as a string of ASCII bytes or as raw 8-bit bytes of data.

- **Cyclic Redundancy Check – CRC (high byte and low byte)**

This is used to detect if a packet goes wrong and some of the data is altered during the transmission using only a small number of redundant bits. One of the most common techniques for error detection is a technique known as the cyclic redundancy check (CRC). It is also a very powerful technique to check data reliably. The 16-bit CRC is used to compute over SOP, Opcode and Control Message or Data Value, not including EOP. The 16-bit CRC seems adequate for both control and data packets in this project. Although the CRC algorithm is relatively more complex than other error detection techniques such as parity and checksum, it offers stronger error detection algorithm.

The CRC calculation is performed using the source code available in electronic form at http://www.netrino.com/code/crc.zip. This code supports several formats for the implantation of CRC such as CRC-CCITT, CRC-16 and CRC-32. It also offers two types of implementations: bit-by-bit CRC calculation and byte-by-byte CRC calculation. The bit-by-bit CRC calculation is generally slow because it executes multiple C statements on each bit in the data. On the other hand, the byte-by-bye CRC calculation which is also known as the table driven CRC calculation uses a different technique than the bit-by-bit CRC routine. The idea behind a table driven technique is that instead of calculating the CRC bit by bit, pre-computed bytes of each of the possible byte-wide input character is stored in the memory and then the input data byte by byte is matched with the value in the table to determine the remainder. The advantage of the table driven technique is that it is faster than the bit-by-bit method. The drawback is that it consumes more program memory because of the size of the look-up table. In this protocol design, since the speed of the CRC calculation is more important than the required memory space, the table driven CRC calculation method is incorporated.

- **End of Packet (EOP)**

This simply indicates the end of each packet. This block does not contain essential information about the packet, if it is corrupted but the rest of the packet is still received without error, it will be ignored.

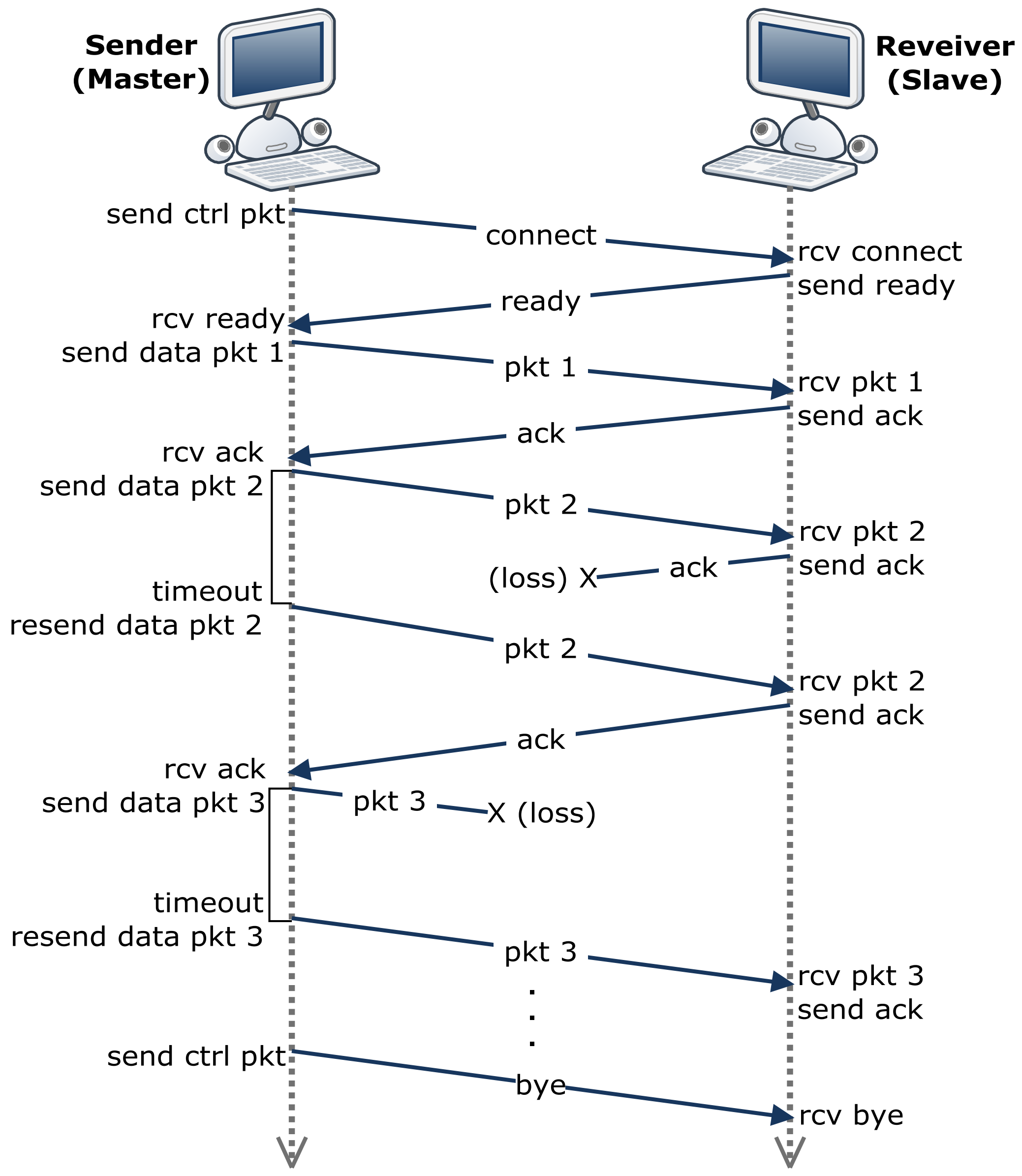


**Figure 14: A time sequence diagram of a protocol design**

### 8.2.3. Fundamental Behaviors of a Protocol

After defining packet types, a protocol's behavior should be identified in a proper manner. When designing a protocol, it is important to prepare the unexpected situations; a system crashes, a message get lost or something that is expected to happen quickly turns out to take a long time. A time sequence diagram shown in Figure 14 can often help understand what might go wrong in such cases and thus help me be prepared for every possibility of errors in the design.

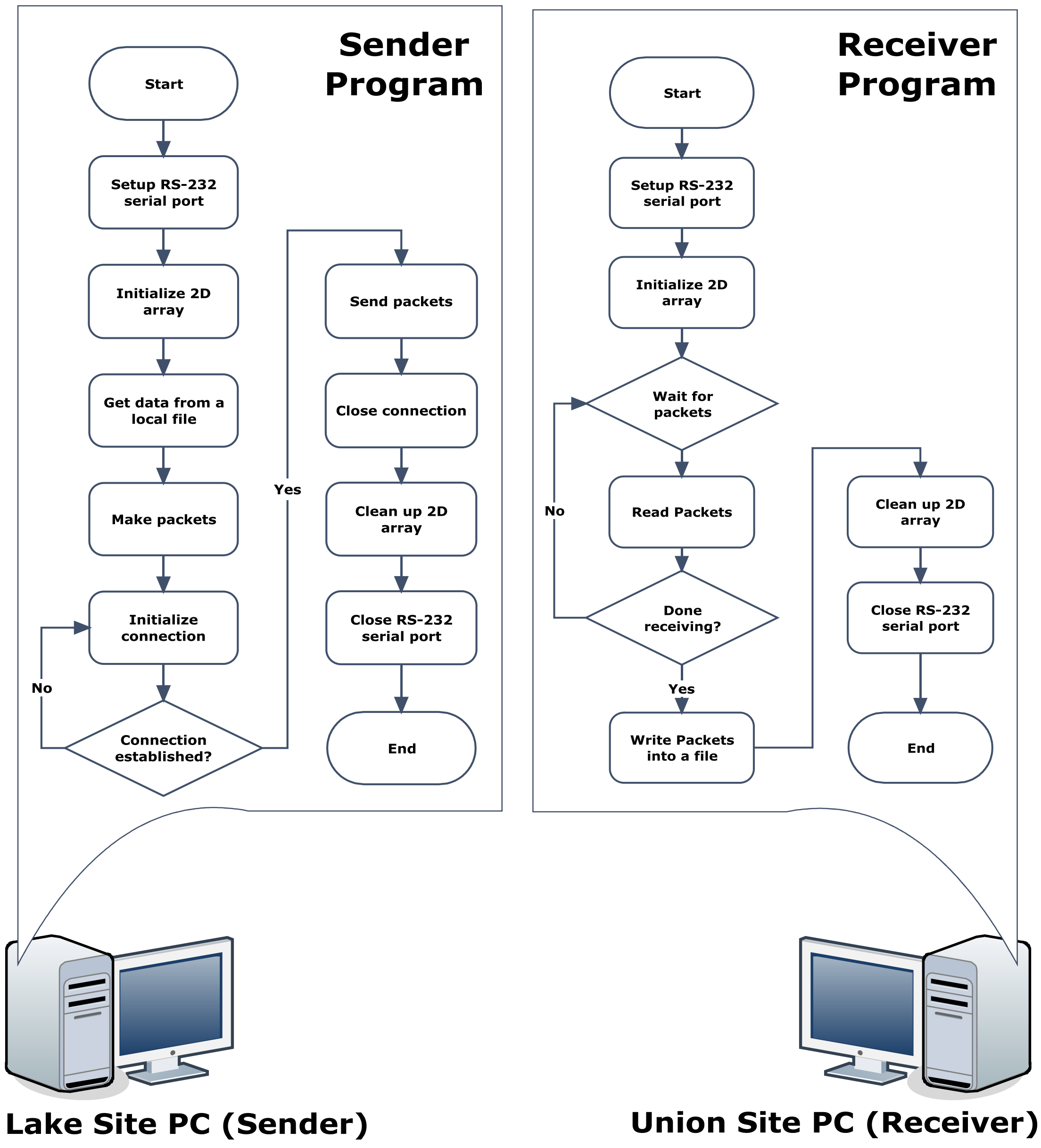


**Figure 15: A flowchart diagram for a sender program and a receiver program**

The algorithm used to implement the master and slave model is known as the stop-and-wait algorithm. The idea of stop-and-wait is straightforward; after the sender (the master) transmits one packet, it waits for a control packet from the receiver (the slave) before transmitting the next packet. If the control packet does not arrive after a certain period of time, the sender calls time out and retransmits the same packet. The strategy of using the retransmission technique to implement reliable delivery is called automatic repeat request (ARQ). The sending side is represented on the left, the receiving side is depicted on the right and time flows from top to bottom. One of the advantages of the stop-and-wait algorithm is that it can handle the data transmission process properly in half-duplex mode meaning there is only a single communication channel and the computer cannot send or receive data at the same time. Figure 14 illustrates three phases of the communication process: initializing connection, data transmission and closing connection involved in this protocol design.

When initializing connection, the sender sends out a CONNECT packet and waits for a READY packet from the receiver. The first two arrows in Figure 14 depict the initializing connection phase. If the READY packet is received before the timer expires, the connection established and it is ready to begin transmitting data packets.

In the data transmission phase, data packets are transmitted sequentially as the data is stored in the sender's computer. Figure 14 describes three different scenarios for the data transmission that might result from the protocol implementation. After establishing connection, the first packet is sent and the ACK packet is received before the timeout occurs. In the next situation, the second packet is sent successfully, but the ACK packet for the second packet is lost, and the retransmission of the same packet is occurred. During the transmission of the third packet, the original data packet is lost, and again the same packet is retransmitted.

After all data packets are successfully transmitted, the sender closes the connection by sending a BYE packet and simply ends the transmission process without waiting any response back from the receiver. The ideal of the initializing and closing connection in this protocol design is somewhat similar to the algorithm for a three-way handshake used by TCP to establish and terminate a connection.

There is one important detail in this protocol design. During the transmission of the second packet, the sender sends the packet and the receiver sends back the ACK packet, but the ACK packet is either lost or delayed in arriving. In this case, the sender calls time out and retransmits the same data packet, but the receiver thinks that the retransmitted packet is the next packet since it correctly received and acknowledged the previous packet. This potentially introduces duplicate copies of a data packet.

In order to see a complete picture of how this protocol design is incorporated separately into a sender program and a receiver program on the Lake site computer and the Union site computer respectively, I used a flowchart diagram as shown in Figure 15. Complete code listings of the sender program and the receiver program are provided in Appendix A and B respectively. The main() function of each corresponding program directly parallels to the flowchart diagram. There is one important point to note in the development of two programs; each block (a rectangular box) of a process step in the flowchart diagram is actually represented using a function. This offers not only better coherency in understanding those programs but also modularity in the program design. As you can see even in the flowchart diagram, there are some similarities in the use of function within sender program and receiver program. All functions are designed in such a way that they are reusable in different parts of programs. Both sender program and receiver program are described in much more detail in subsequent sections.

### 8.2.4. Sender Program

- **main () function**

The main() function is called when the sender program is stated. This function is composed of small functions, and each of the function operates independent of each other. Figure 16 contains the complete contents of the function. This function performs all essential operations of the sender program.

```
main () {

  int i, j, nrows, ncolumns, port;
  int connected = 0;
  int disconnected = 0;
  int total_packet, total_char;
  char data[NUM_ROW * NUM_COLUMN];
  char **packet2D;

  struct termios old_flags; /* will be used to save old port settings */
```

```
    struct termios new_flags; /* will be used for new port settings
*/

    /* open the RS-232 port */
    port = open_RS232port();
    if (port == -1) {
      printf("\n Error opening RS232 port\n");
      exit(-1);
    }

    // Set up raw/non-canonical mode
    // VTIME is set to 10 and VMIN is set to 0

    tcgetattr(port, &old_flags); /* save current port settings */
    new_flags = old_flags;
    new_flags.c_lflag &= ~(ECHO | ICANON | ISIG);
    new_flags.c_iflag &= ~(BRKINT | ICRNL);
    new_flags.c_oflag &= ~OPOST;
    new_flags.c_cc[VTIME] = 10;  /* inter-character timer (10 x
0.1)sec */
    new_flags.c_cc[VMIN] = 0;    /* non-blocking read*/
    new_flags.c_cflag |= CS8;    /* Set 8 bits per character. */

    if (tcsetattr(port, TCSAFLUSH, &new_flags) < 0) {
      printf("\n Error seting raw mode!! \n");
      exit(-1);
    }

    /* set baud rate */
    cfsetispeed(&new_flags, B9600);

    /* Allocate memory location for packet2D an 2D array */
    /* Size of packet2D: 360 x 100 (NUM_ROW x NUM_COLUMN)*/
    packet2D = create_2D_char_array(NUM_ROW, NUM_COLUMN);

    printf("\n Reading data from %s ... \n", FILE_NAME);
    get_Data(data, FILE_NAME);

    printf("\n Cteating packets ... \n");
    total_packet = makePacket(packet2D, data, &total_char);

    for(i = 0; i < total_packet; i++) {
      printf("\n Packet #%d: %s \n", (i+1), packet2D[i]);
    }

    printf("\n Total packet being created: %d\n ", total_packet);

    printf("\n Initializing connection... \n");
    connected = init_Connection(port, 1);
```

```
58     printf("\n Connection Status: %d \n", connected);
59
60     if (connected) {
61       printf("\n Initializing send process... \n");
62       send_packet (packet2D, port, total_packet, total_char);
63
64       printf("\n Closing connection... \n");
65       disconnected = init_Connection(port, 0);
66       printf("\n Connection Status: %d \n", disconnected);
67     }
68
69     cleanUp_2D_array(packet2D, NUM_ROW);
70
71     /* restore the old port settings before quitting */
72     tcsetattr(port, TCSANOW, &old_flags);
73     close(port);   /* close the serial port */
74   } // end of main() function
```

**Figure 16: The main() function of the sender program**

The RS-232 serial port is opened by calling the open_RS232port() function in Line 14 so that the serial communication can be executed. Since a serial port is a file, the built-in open() function is used to access it within the open_RS232port() function. This function returns the file descriptor on success or -1 on error. Next, Line 20-38 is executed to setups the port in the raw or non-canonical mode and the baud rate at 9600 baud. In the raw or non-canonical mode, the input characters at the serial port are unprocessed, and they are passed through exactly as they are received, when they are received. Two parameters control the behavior of this mode: c_cc[VTIME] in Line 28 sets the character timer in tenths of seconds, and c_cc[VMIN] in Line 29 sets the minimum number of characters to receive before satisfying the read. In this case, VMIN is set to zero and therefore VTIME servers as a timeout value. The reading at the serial port will be satisfied if a single character is read, or VTIME is exceeded. If TIME is exceeded, no character will be returned.

```
char** create_2D_char_array(int num_rows, int num_cols) {
  int i;
  char **2D_array;

  /*Allocate pointer memory for the first dimension of a matrix[][];*/
  2D_array = (char **) malloc(num_rows * sizeof(char *));
  if(2D_array == NULL){
    free(2D_array);
    printf("Memory allocation failed.\n");
    exit(-1);
  }

  /*Set all pointers to NULL*/
  memset(2D_array, 0, num_rows * sizeof(char*));

  /*Allocate integer memory for the second dimension of a matrix[][];*/
  for(i = 0; i < num_rows; i++) {
    2D_array[i] = (char *) malloc(num_cols * sizeof(char));
    if(NULL == 2D_array[i]){
      free(2D_array[i]);
      printf("Memory allocation failed.\n");
      exit(-1);
    }
  }
  /* Return an allocated memory address of 2D array */
  return *&2D_array;
}
```

**Figure 17: The create_2D_char_array() function to dynamically allocate a 2D array**

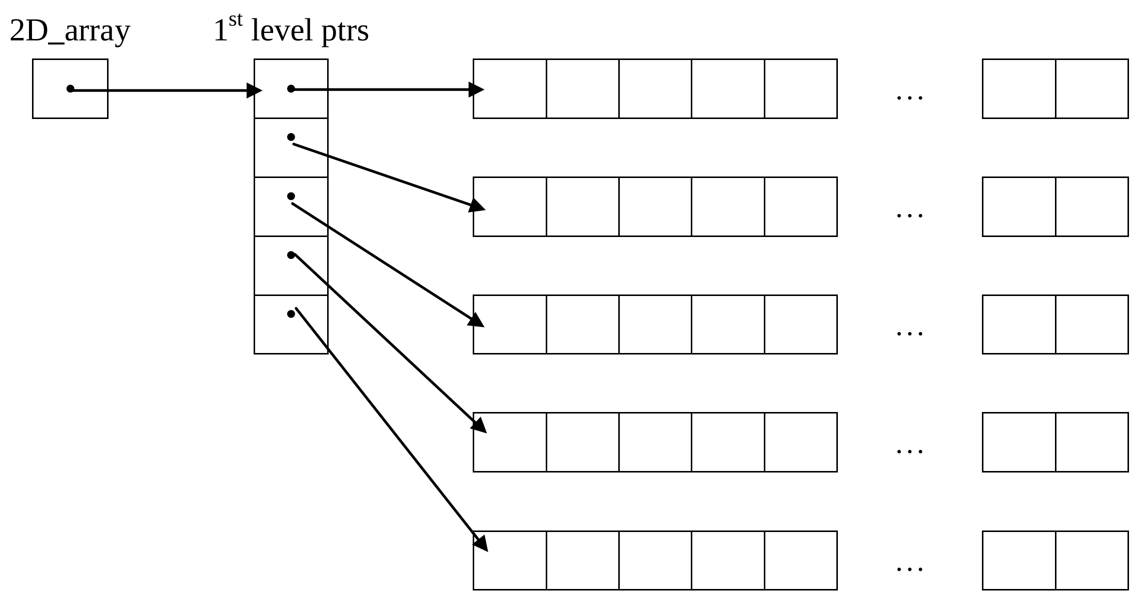


**Figure 18: Dynamic memory allocation of a 2D array**

- **Allocation of memory for a two dimensional array**

The sender program allocates a block of memory by calling the create_2D_char_array function to store several data packets in sequence before transmitting to the receiver site. In this program, a two-dimensional array is considered to store data packets, and each row of the array contains a single packet. For this reason, the malloc() function from the standard C library is used to allocate a continuous portion of memory. In Line 3, 2D_array is defined as a pointer-to-pointer-to-char. Memory allocation for the two-dimensional array involves two stages as shown in Figure 18. At first, 2D_array points to a block of the first-level pointers, one for each individual row. Line 6 calls the malloc () function to allocate num_rows elements of memory space for first level pointers. Next, after successfully allocating, each element is filled with a pointer to num_columns number of chars, the storage for a single row of the array in Line 18. Finally, this function returns memory addresses of the allocated 2D array.

```c
void cleanUp_2D_array(char **array, int x) {
  int i;
  for(i = 0; i < x; i++) {
    free(array[i]);     // free the first-level pointer
  }
  free(array);     // free the second-level pointer
}
```

**Figure 19: The cleanUp_2D_array() function to free a dynamically allocated 2D array**

- **Freeing an allocated memory space**

Once all data packets are transmitted successfully, it is required to deallocate or free the dynamically allocated "2D_array" using the cleanUp_2D_array() function shown in Figure 19. This function first frees the first level pointers and then all the elements in each row of the array. Freeing the allocated memory location is a good idea because it gives the memory back on the computer for other uses.

```c
void read_data (char *raw_data, char *fn) {
  char temp_data;
  int num_char;
  FILE *datafile;              /* need a pointer to FILE */
  datafile = fopen(fn, "r");  /* Open "file_name" for reading */
  if (datafile == NULL) {
    printf("\n Cannot open file (%s)! \n", fn);
  }

  num_char = 0;     /* Initialize character counter */

  /* Read until end-of-file */
  while ((temp_data = fgetc(datafile)) != EOF) {
    raw_data[num_char++] = temp_data;
  }
  raw_data[num_char++] = temp_data;     /* End data array with EOF */
  fclose(datafile);     /* close a file */
  }
```

**Figure 20: The read_data() function to read data from a specific file name**

- **Reading data from a text file**

Reading the data from a local file in C is simple. The read_data() function is shown in Figure 20. The standard C library includes the fopen() function that allows to open a file from a specific location in the indicated mode such as r for reading and w for writing. Since, an EOF (end of file) character generally indicates the end of the text files, it is used to specify where to stop reading. At the end of reading, the file is closed using the fclose() function. This function returns all the data from the text file in a single array.

```
  int make_packet (char **pkt, char *data, int *num_char) {
    char temp_pkt[NUM_COLUMN];    /* temporary storage of actual data */
    char temp_char;
    int total_char;               /* total number of actual data */
    int num_pkt;                  /* total number being created */
    int opCode;                   /* Opcode for data packet (Opcode = 1) */
    int i, row, column;

    crc crc_value;                /* CRC value for each packet */
    crcInit();                    /* initialize the CRC look up table */

    opCode = DATA_PKT;            /* opcode for data packet */
    i = 0;
    row = 0;
    column = 0;


    while (data[i] != EOF) {   /* read data until EOF (end-of-file) */
      if (data[i] != LF) {     /* read data until LF (line feed) */
        if (column == 0) {
          // begin each pkt with SOT
          pkt[row][column] = SOT;
          temp_pkt[column] = SOT;
          column++;
        }
        else if (column == 1) {
          // place an opCode for each packet
          pkt[row][column] = opCode;
          temp_pkt[column] = opCode;
          column++;
        }
        else {
          // read the actual data
          temp_char = data[i++];
          pkt[row][column] = temp_char;
          temp_pkt[column] = temp_char;
          column++;
        }
      }
      else {
        // keep a LF char at the end of the actual data
        temp_char = data[i++];
        pkt[row][column] = temp_char;
        temp_pkt[column] = temp_char;

        // number of char including SOT, Opcode and data in each pkt
        total_char = column + 1;
        // generate CRC for each pkt
        crc_value = crcFast(temp_pkt, total_char);
        // store crc high byte
        pkt[row][++column] = crc_value >> 8;
        // store crc low byte
        pkt[row][++column] = crc_value & 0x00FF;
        // end a packet with EOT
        pkt[row][++column] = EOT;

        *num_char = ++column;   /* number of char in the complete pkt */
        row++;                  /* increment row index of pkt[][] */
        column = 0;             /* set column index of pkt[][] */
      }
    }
    num_pkt = row;
    return (num_pkt);           /* Return number of pkt */
  }
```

**Figure 21: The make_packet() function to generate data packets in sequence**

- **Creating data packets**

The make_packet() function shown in Figure 21 is used to break the data from the local text file into a several small packets. The format of the data packet follows the one illustrated in Figure 13(b). Each packet begins with a SOT character and follows by an Opcode. The data value section of each packet contains exactly one measurement of water property data from the sensor packet. The data packet also includes the 16-bit CRC value which is calculated over SOT, Opcode and the data value using the table driven CRC calculation technique. Each complete data packet is stored in each row of the allocated "2D_array". The number of packets generated depends on the number of measurements contain in the text file.

- **Creating control packets**

The make_ctrl_packet() function is used to create a control packet with a specified control message. The code listing of this function is provided in Figure 22. The format of the control packet is illustrated in Figure 13(a). Similar to the data packet, each control packet starts with a SOT character, follows by an Opcode and then the control message. The control packet also includes the 16-bit CRC value which is calculated over SOT, Opcode and the control message. This function returns a complete control packet.

```
int make_ctrl_packet (char *ctrl_pkt, char ctrl_char) {
  crc crc_value;              /* CRC value of control packet */
  char temp_ctrl_char[3];
  int total_char;             /* total character in control packet */
  char opCode = CTRL_PKT;     /* Opcode for control packet (Opcode = 2) */
  int i = 0;
  crcInit();                  /* initialize the CRC look up table */

  temp_ctrl_char[0] = SOT;
  temp_ctrl_char[1] = opCode;
  temp_ctrl_char[2] = ctrl_char;

  ctrl_pkt[i] = SOT;             /* Begin a control pkt with SOT */
  ctrl_pkt[++i] = opCode;        /* Opcode for control packet */
  ctrl_pkt[++i] = ctrl_char;

```

```
    /* Generate CRC for ctrl packet */
    crc_value = crcFast(temp_ctrl_char, 3);

    ctrl_pkt[++i] = crc_value >> 8;       /* Store crc high byte */
    ctrl_pkt[++i] = crc_value & 0x00FF;   /* Store crc low byte */
    ctrl_pkt[++i] = EOT;                  /* End a control pkt with
EOT */

    /* Return number of character in control pkt */
    return (total_char = ++i);
  }
```

**Figure 22: The make_ctrl_packet function to create control packets**

The following two functions: connection_status() function and send_packet() function within the main() function of the sender program perform the most important functions in this protocol design in order to deliver packets in the order they are sent as well as guarantee packet delivery. Although the two functions perform a very similar task, they are separated into two in order to reduce the complexity of code development. The behaviors of both connection_status() function and send_packet() function are presented using state-transition diagrams in Figure 23 and 24 respectively. These state-transition diagrams are fairly easy to understand. Each circle denotes a state, and the transition from state to state is determined according to the direction of the arrow. The actions executed in each state are described using pseudo-codes in a rectangular box.

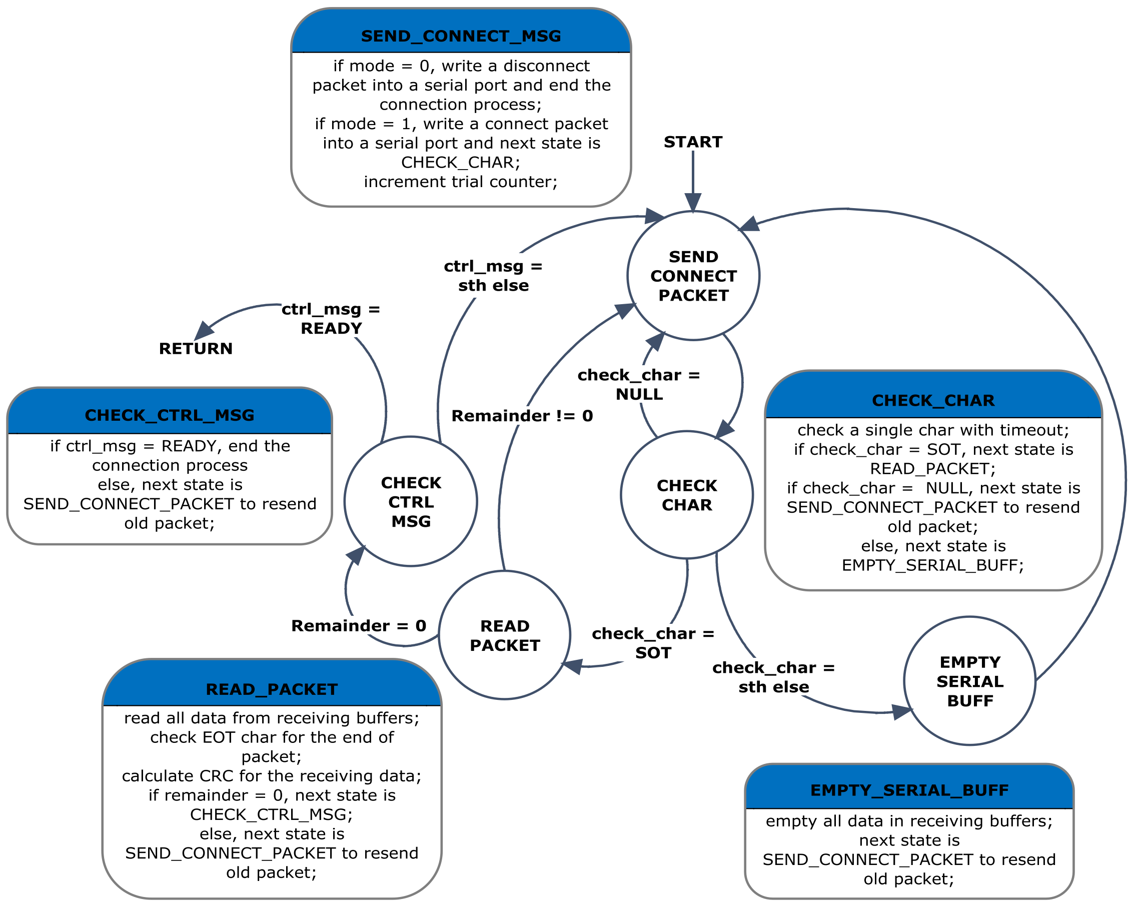

SEND_CONNECT_MSG
if mode = 0, write a disconnect packet into a serial port and end the connection process;
if mode = 1, write a connect packet into a serial port and next state is CHECK_CHAR;
increment trial counter;
START
SEND CONNECT PACKET
ctrl_msg = sth else
ctrl_msg = READY
RETURN
CHECK_CTRL_MSG
if ctrl_msg = READY, end the connection process
else, next state is SEND_CONNECT_PACKET to resend old packet;
check_char = NULL
Remainder != 0
CHECK_CHAR
check a single char with timeout;
if check_char = SOT, next state is READ_PACKET;
if check_char = NULL, next state is SEND_CONNECT_PACKET to resend old packet;
else, next state is EMPTY_SERIAL_BUFF;
CHECK CTRL MSG
CHECK CHAR
Remainder = 0
READ PACKET
check_char = SOT
check_char = sth else
EMPTY SERIAL BUFF
READ_PACKET
read all data from receiving buffers;
check EOT char for the end of packet;
calculate CRC for the receiving data;
if remainder = 0, next state is CHECK_CTRL_MSG;
else, next state is SEND_CONNECT_PACKET to resend old packet;
EMPTY_SERIAL_BUFF
empty all data in receiving buffers;
next state is SEND_CONNECT_PACKET to resend old packet;

**Figure 23: A state diagram for initializing and closing connection in the sender program**

- **Connection establishment and termination**

Now let's trace the typical transitions taken through the diagram in Figure 23. When the connetion_status function is called, the initial state usually starts in the SEND_CONNECT_PACKET state to send out a CONNECT packet into the UART serial port. In this state, a trial counter is implemented to keep track of the number of trial required to establish the connection. If the connection does not establish successfully until the third time, the connection process is indicated as unsuccessful.

Next, the transition is moved to the CHECK_CHAR state in order to wait for an incoming control packet with the timeout. Currently, the timeout value is 5 seconds. If the timeout exceeds and there is no packet from the other side, the transition returns to the SEND_CONNECT_PACKET state. However, if there a packet before the timeout occurs, the first character of the packet is checked. If the received character is not a SOT character, the transition is made to the EMPTY_SERIAL_BUFF state to remove all "garbage" characters at the UART serial buffer. Finally, if the received character is a SOT character, the transition is made to the READ_PACKET state to continue reading the remaining character in the packet.

After reading the packet, the error checking mechanism is performed using the 16-bit CRC calculation on that packet. If the remainder of the CRC calculation is not equal to zero, the transition is made to the SEND_CONNECT_PACKET state. But if the remainder is zero, the next state is the CHECK_CTRL_MSG state to check the control message inside the received packet.

If the control message is READY, the process of establishing connection is indicated as successful. If not the next state simply return to the SEND_CONNECT_PACKET state until the maximum number of allowed connection trial is reached.

Turing our attention now to the process of terminating a connection, the important thing to keep in mind is that the terminating process is quite different from the establishing process. The sender program just sends out a BYE packet without waiting any response back from the other end.

- **Sending packets**

The state-transition diagram shown in Figure 24 describes the behavior of the process of sending data packets. When the send_packet() function is called after the connection is successfully established, the initial state usually starts in the SEND_PACKET state to send out a packet one at a time into the UART serial port. In this state, a trial counter is implemented to keep track of the number of trial required to send each packet. There is no limit to the number of time required to send each data packet. After sending the data packet, the next state becomes the CHECK_CHAR state.

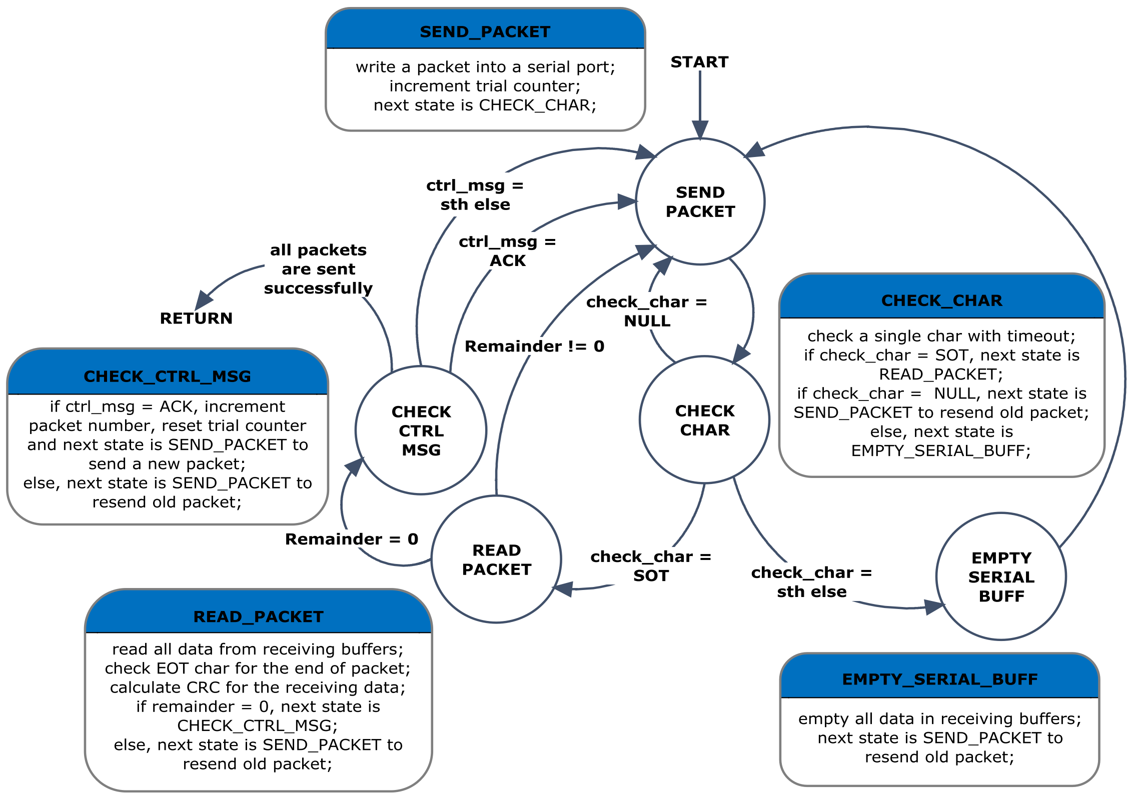

SEND_PACKET
write a packet into a serial port;
increment trial counter;
next state is CHECK_CHAR;
START
SEND PACKET
ctrl_msg = sth else
ctrl_msg = ACK
all packets are sent successfully
RETURN
check_char = NULL
Remainder != 0
CHECK_CHAR
check a single char with timeout;
if check_char = SOT, next state is READ_PACKET;
if check_char = NULL, next state is SEND_PACKET to resend old packet;
else, next state is EMPTY_SERIAL_BUFF;
CHECK_CTRL_MSG
if ctrl_msg = ACK, increment packet number, reset trial counter and next state is SEND_PACKET to send a new packet;
else, next state is SEND_PACKET to resend old packet;
CHECK CTRL MSG
CHECK CHAR
Remainder = 0
READ PACKET
check_char = SOT
check_char = sth else
EMPTY SERIAL BUFF
READ_PACKET
read all data from receiving buffers;
check EOT char for the end of packet;
calculate CRC for the receiving data;
if remainder = 0, next state is CHECK_CTRL_MSG;
else, next state is SEND_PACKET to resend old packet;
EMPTY_SERIAL_BUFF
empty all data in receiving buffers;
next state is SEND_PACKET to resend old packet;

**Figure 24: A state diagram for sending data in the sender program**

In the CHECK_CHAR state, an incoming control packet is read with the timeout. The actions executed in this state are exactly the same as those in the state from the connection_status() function. If the received control packet starts with a SOT character, the next state is the READ_PACKET state.

Similar to the READ_PACKET state in the connection_status() function, the reading process is followed by the error checking mechanism. If the CRC remainder is not equal to zero, the transition is preceded back to the SEND_PACKET state to resend the same data packet. But if the remainder is zero, the next state is the CHECK_CTRL_MSG state to check the control message inside the received packet.

If the control message is ACK, the current process of sending the data packet is indicated as successful, and the next state is the SEND_PACKET state to send a new data packet. If the control message is something else, the next state also simply returns to the SEND_PACKET state but to resend the same data packet until it is successfully sent.

The process of data transmission is completed only after the entire available data packets are sent successfully.

#### 8.2.5. Receiver Program

- **main() function**

This main() function is called when the receiver program is stated. Figure 25 contains the complete contents of the function. This function performs all essential operations of the sender program. The main() function of the receiver program looks similar to that of the sender program, but it does not contain make_packet() function, connection_status() function and obviously the send_packet() function. Instead, it contains two new functions: read_packet() function and write_ data() function. Several functions from the sender program are reused in the receiver program.

```
main() {
  int port, i;
  int total_pkt, total_char;
  int nrows, ncolumns;
  char **packet2D;
  struct termios old_flags; /* will be used to save old port settings */
  struct termios new_flags; /* will be used for new port settings */

  /* open the RS-232 port */
  port = open_RS232port();
  if (port == -1) {
      printf("\n Error opening RS232 port \n");
      exit(-1);
  }

  // Set up raw/non-canonical mode
  // VTIME is set to 10 and VMIN is set to 0
  tcgetattr(port, &old_flags); /* save current port settings */
  new_flags = old_flags;
  new_flags.c_lflag &= ~(ECHO | ICANON | ISIG);
  new_flags.c_iflag &= ~(BRKINT | ICRNL);
  new_flags.c_oflag &= ~OPOST;
  new_flags.c_cc[VTIME] = 10;  /* inter-character timer (10 x 0.1)sec */
  new_flags.c_cc[VMIN] = 0;    /* non-blocking read*/
  new_flags.c_cflag |= CS8;    /* Set 8 bits per character. */
```

```
    if (tcsetattr(port, TCSAFLUSH, &new_flags) < 0) {
      printf("\n Error seting raw mode!! \n");
      exit(-1);
    }

    /* set baud rate */
    cfsetispeed(&new_flags, B9600);

    /* Allocate memory location for packet2D an 2D array */
    /* Size of packet2D: 360 x 100 (NUM_ROW x NUM_COLUMN)*/
    packet2D = create_2D_char_array(NUM_ROW, NUM_COLUMN);

    printf("\n Initializing receiving process... \n");
    read_packet(packet2D, port, &total_pkt, &total_char);

    printf("\n Writing data into %s ... \n", FILE_NAME);
    write_data(packet2D, FILE_NAME, total_pkt, total_char);
    printf("\n Writing complete! \n");

    cleanUp_2D_array(packet2D, NUM_ROW);

    tcsetattr(port, TCSANOW, &old_flags);
    close(port);
  }
```

**Figure 25: The main() function of the receiver program**

- **Receiving packets**

Figure 26 shows the state-transition diagram which describes the behavior of the read_data() function. Notice that unlike those state-transition diagram for the sender program, there is an extra state called the CHECK_OPCODE state contained in the following state-transition diagram. This state is used to easily distinguish between control packet and data packet.

When the read_packet() function is called, the initial state usually starts in the CHECK_CHAR state to read a packet available at the UART serial port with the timeout. The timeout value is 5 seconds. If the timeout occurs and there is still no incoming packet, it remains in this state to continue waiting for the packet. However, if there a packet before the timeout occurs, the first character of the packet is checked. If the received character is not a SOT character, the next state is the EMPTY_SERIAL_BUFF state to remove all “garbage” characters at the UART serial buffer and finally returns to the CHECK_CHAR state. If the received character is a SOT character, the transition is made to the READ_PACKET state to continue reading the remaining character in the packet.

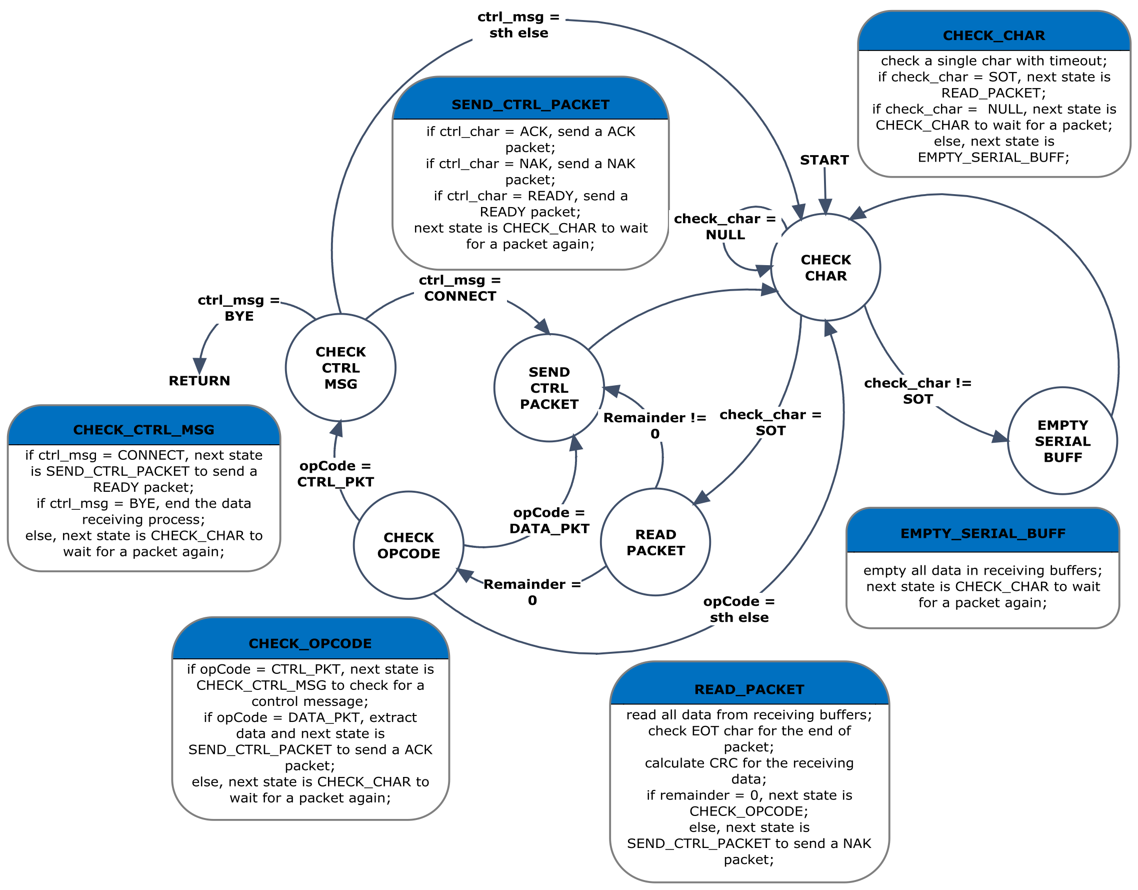

ctrl_msg = sth else
CHECK_CHAR
check a single char with timeout; if check_char = SOT, next state is READ_PACKET; if check_char = NULL, next state is CHECK_CHAR to wait for a packet; else, next state is EMPTY_SERIAL_BUFF;
SEND_CTRL_PACKET
if ctrl_char = ACK, send a ACK packet; if ctrl_char = NAK, send a NAK packet; if ctrl_char = READY, send a READY packet; next state is CHECK_CHAR to wait for a packet again;
START
check_char = NULL
CHECK CHAR
ctrl_msg = CONNECT
ctrl_msg = BYE
CHECK CTRL MSG
RETURN
SEND CTRL PACKET
check_char != SOT
EMPTY SERIAL BUFF
CHECK_CTRL_MSG
if ctrl_msg = CONNECT, next state is SEND_CTRL_PACKET to send a READY packet; if ctrl_msg = BYE, end the data receiving process; else, next state is CHECK_CHAR to wait for a packet again;
opCode = CTRL_PKT
Remainder != 0
check_char = SOT
opCode = DATA_PKT
CHECK OPCODE
READ PACKET
EMPTY_SERIAL_BUFF
empty all data in receiving buffers; next state is CHECK_CHAR to wait for a packet again;
Remainder = 0
opCode = sth else
CHECK_OPCODE
if opCode = CTRL_PKT, next state is CHECK_CTRL_MSG to check for a control message; if opCode = DATA_PKT, extract data and next state is SEND_CTRL_PACKET to send a ACK packet; else, next state is CHECK_CHAR to wait for a packet again;
READ_PACKET
read all data from receiving buffers; check EOT char for the end of packet; calculate CRC for the receiving data; if remainder = 0, next state is CHECK_OPCODE; else, next state is SEND_CTRL_PACKET to send a NAK packet;

**Figure 26: A state diagram for receiving data in the receive program**

After finish reading the packet, the error checking mechanism is performed using the 16-bit CRC calculation on that packet. If the remainder of the CRC calculation is not equal to zero, the transition is made to the CHECK_OPCODE state to check the opcode inside the received packet. But if the remainder is zero, the next state is the SEND_CTRL_PACKET state to indicate that there is an error in the received packet by sending a NAK packet.

The main action of the CHECK_OPCODE state is to determine the type of the packet received from the other end. If the opcode is DATA_PKT, the next state is the SEND CRTL_PKT state to indicate receiving data packet as successful by sending an

ACK packet. However, if the opcode is CTRL_PKT, the next state is the CHECK_CTRL_MSG state to check what type of control message is inside the packet.

The CHECK_CTRL_MSG state expects two types of control messages: CONNECT and BYE. These messages are used by the sender to establish and terminate the connection respectively. If the control message is CONNECT, the next state is the SEND CRTL_PKT state to send back the READY packet to the other end. If the control message is something else, the next state also simply returns to the CHECK_CHAR state to wait for another packet. However, if the control message is BYE, the read_data() function is ended, and the receiving process is indicated as successful.

### 8.2.6. Transmission Control Program

The transmission control program is developed in C and implemented on the Silicon Lab's C8051F020-TB microcontroller development board. The behavior of the program is described using a flowchart diagram shown in Figure 27. Notice that the flowchart diagram is broken into three pieces in order to fit in one pages and explain it clearly. The purpose of this program is to handle the data transmission between control computers and transceivers. This program can be used to isolate the characteristic of the control computers and that of transceivers. Replacing the transceivers for any reason will not affect any hardware or software components of the computers.

When the development board is powered up, this program is running continuously within the loop. This program fully utilizes all UARTs' interrupt flags available on the board to receive data from the control computer and transmit it to the transceiver or vice versa. Specifically, the data is received from the computer via the RX0 port of UART0 and transmit to the transceiver via the TX1 port of UART1 and the data from the transceiver is received via the RX1 port of UART1 and transmitted to the computer via the TX0 port of UART0.

RI0 and RI1 (Receive Interrupt) flags are used to detect the incoming data from the computer and the transceiver respectively. The detection mechanism requires a polling algorithm which is implemented using while loop. The top-left flowchart diagram in Figure 27 illustrates the polling algorithm and the RI flag checking.

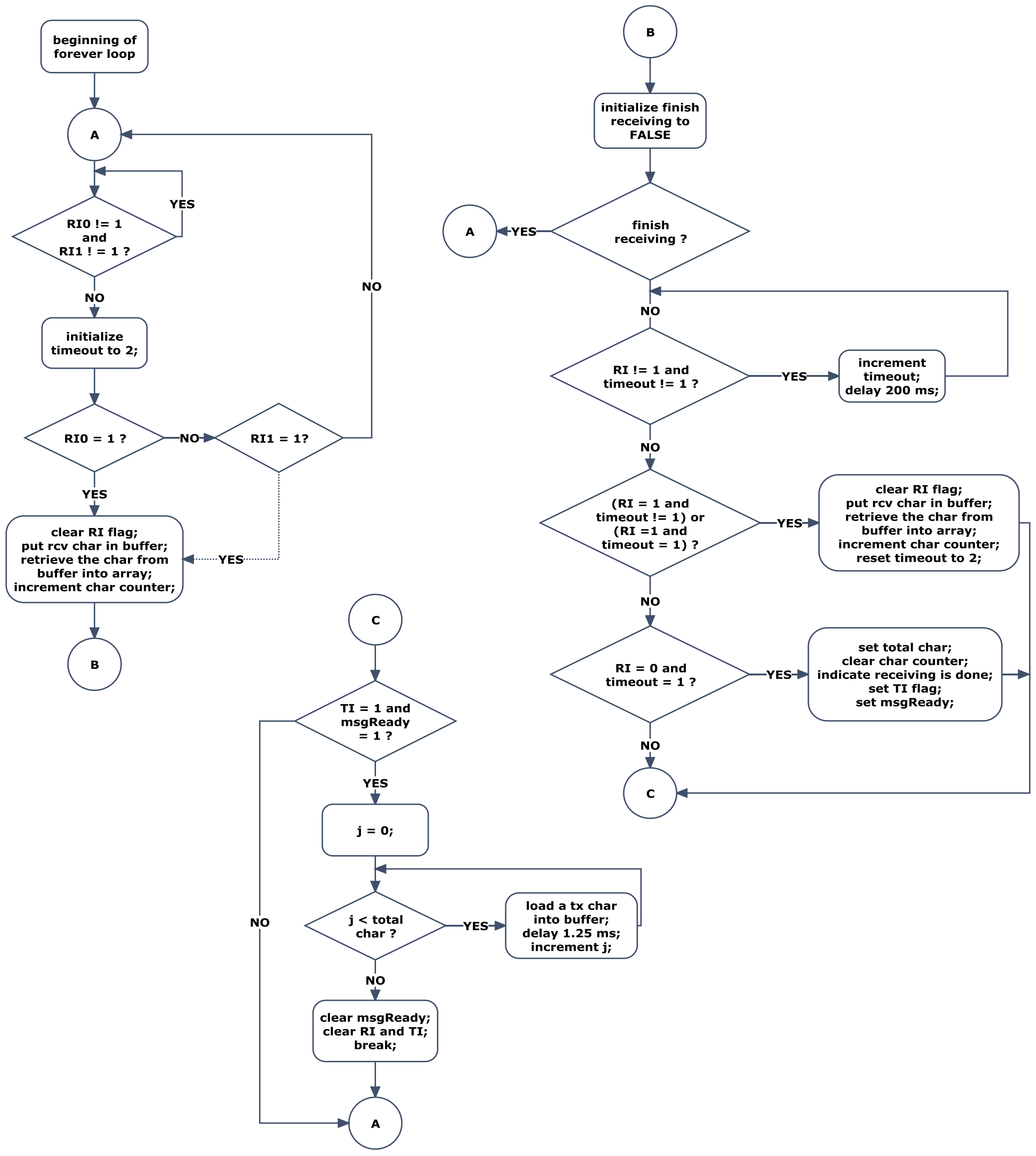

beginning of forever loop
A
RI0 != 1 and RI1 ! = 1 ?
YES
NO
initialize timeout to 2;
RI0 = 1 ?
RI1 = 1?
clear RI flag; put rcv char in buffer; retrieve the char from buffer into array; increment char counter;
B
initialize finish receiving to FALSE
finish receiving ?
RI != 1 and timeout != 1 ?
increment timeout; delay 200 ms;
(RI = 1 and timeout != 1) or (RI =1 and timeout = 1) ?
clear RI flag; put rcv char in buffer; retrieve the char from buffer into array; increment char counter; reset timeout to 2;
RI = 0 and timeout = 1 ?
set total char; clear char counter; indicate receiving is done; set TI flag; set msgReady;
C
TI = 1 and msgReady = 1 ?
j = 0;
j < total char ?
load a tx char into buffer; delay 1.25 ms; increment j;
clear msgReady; clear RI and TI; break;

**Figure 27: A flowchart diagram a transceiver control program**

When one of RIs is set, the inter-character timer begins, and the program keeps checking the RI for the next incoming character. The timeout value between subsequent characters is approximately 200 ms. If the timeout occurs and there is no more character from the RX port, all the received characters are placed in an array and prepared to transmit to the corresponding port. Notice that there is a delay of about 2 ms is between each character transmitted via TX ports. This delay is one of the requirements for the transceiver to be able to process the data transmission reliably. The top-right flowchart diagram in Figure 27 describes the checking of the subsequent character at the RX port along with the timeout. The bottom flowchart diagram in Figure 27 explains the data transmission via the corresponding TX port to the final destination.

The advantage of this program is that prior to receiving the data, it does not need to know the number of incoming characters. The complete code listing of the transceiver control program is provided in Appendix C.

# 9. System Performance

In this section, the system performance of the final design and implementation is discussed based on the preliminary design requirements. First, the initial experiment results of the RF link analysis are presented. Second, the system level performance is discussed in detail.

### 9.1. RF Link Analysis

In order to determine the important design parameters such as operating frequency and transmit power, the RF link analysis is done at both Ballston Lake and Union College.

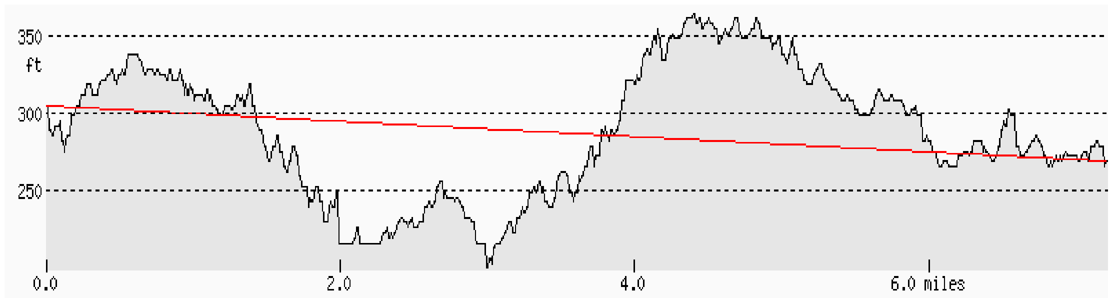


**Figure 28: Ground Elevation Map between Union College and Ballston Lake [11]**

In this system design, the line-of-sight communication was desirable to achieve a reliable data transmission over the distance of about 8 miles. Therefore, the communication link needed a careful survey through map-based or direct observation to avoid any metallic or wet objects which could cause severe attenuation and reflection in the transmission path. For the map-based approach, a ground elevation map shown in Figure 28 was used to verify to obstacles such as hills, tall buildings and other structures in between Union College and Ballston Lake. Also the direct observation of the communication path was done from the roof top of the Science and Engineering building. Unfortunately, the line-of-sight requirement could not confirm based on these two approaches.

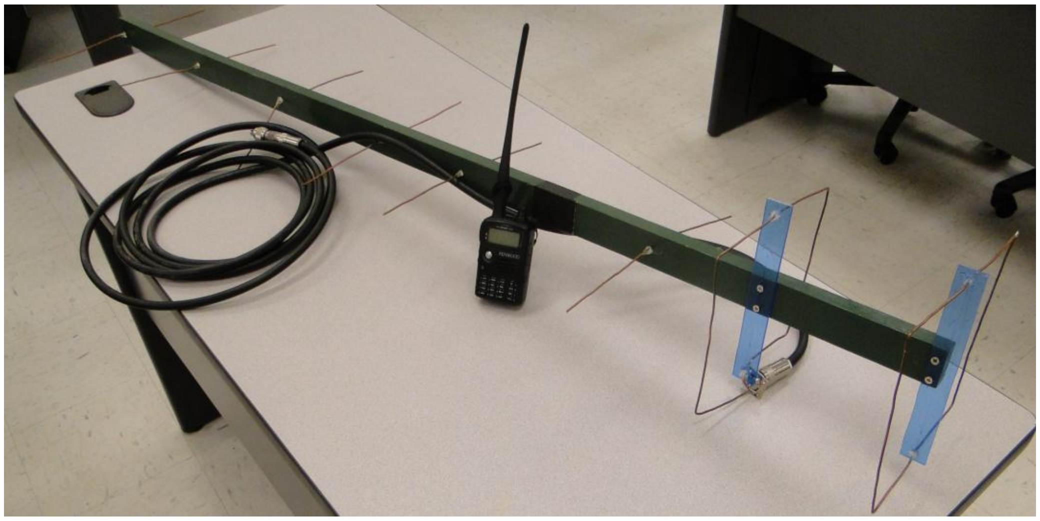


**Figure 29: A 446 MHz Quagi antenna and a transceiver**

Then, the radio field strength test is performed at Ballston Lake to further investigate design parameters. In this test, a home-made 446 MHz Quagi and a ham radio transceiver shown in Figure 29 are used. The transceiver was transmitting radio signals at 442.55 MHz and receiving at 447.55 MHz. Both the antenna and the transceiver were operating at the UHF range.

To obtain the best estimate from the field strength test, not only was the operating frequency important, the basic antenna characteristics such as gain and front-to-back ratio were essential. The antenna gain was estimated using a spectrum analyzer, and its gain was about 10dB and highly directional. The front-to-back ratio was determined using a SWR (standing wave ratio) meter, and this ratio is 1:4 which is significantly low. A low front-to-back ratio means that the amount of energy is radiated in the desired direction is greatly reduced. This resulted from the frequency mismatch between antenna and transceiver.

Despite the fact, we could still connect a 5-Watt repeater at Union College from Ballston Lake using these two hardware components. The radio field strength measurement confirmed that RF data communication was still possible using a 5-W transmitter and a high gain (with 10dB) directional antenna both operating in UHF frequency. Moreover, the fact that there was a power loss because of the mismatch between the antenna and the transceiver, it was predicted that the transmit power requirement would be less than 5 Watt.

### 9.2. Overall System Performance

As I mentioned before, the final implementation of the RF data communication system was hindered because of the FCC licensing process. Instead, the complete prototype implementation was accomplished. The main difference between the prototype implementation and the final implementation would be in the radio system. The rest of the components could remain the same for the final implementation.

In order to make sure that all the hardware components worked together as a unit in the prototype implementation, they are tested together with the software programs.

The operation of Master/Slave protocol has been verified working properly with error detection, receiver feedback and retransmission. Several different scenarios of data transmission protocol were tested in order to check the robustness of the protocol. The protocol could handle all three situations illustrated in Figure 14, but it could not handle duplicate packets. Programs on sender and receiver control computers were implemented in modular fashion for design flexibility, easy debugging and maintainability. Moreover, programs on control computers and microcontrollers operate independently. By changing the program on the microcontroller I do not need to change any part of the program on the control PC. Since all the programs functioned properly to handle the RF data transmission,

# 10. Cost Analysis

As with any design project, there is going to be some incurred costs for hardware, software, and other equipment. These costs are often limited by the available funding. For this project, the funding is limited to the $600 granted by the Internal Education Fund (IEF) Committee. Since the IEF Grant was submitted, the design has been changed to include 8051 microcontroller boards. In addition, due to the unexpected long processing time for the FCC license application, the actual radio system could not purchase and implement. Instead, the HAC-UM96 ultra low power transceivers were used for the prototype design. The cost of each individual component in this prototype implementation is provided in Table 2. The total cost of the prototype is only $287.90.

| Components | Quantity | Unit Price | Cost |
|---|---|---|---|
| 8051 Microcontroller Board | 2 | $99 | $198 |
| HAC-UM96 Ultra Low Power Data Transceiver | 2 | $44.95 | $89.90 |
| | | **Total** | $287.90 |

**Table 3: Cost of system components in the prototype implementation**

Note that there will be additional cost to include the FCC licensing fee, actual final radio equipment and high gain directional antennas for the final system implementation. Two similar brands of transceivers are recommended for this system design. In the future, there might be other transceivers that would be more suitable for this project.

## 11. Recommendation and Conclusion

The overall purpose of this project was to develop a reliable RF data communication system for the long-term deep water monitoring system. This project illustrates the prototype design of the RF data transmission system which meets most of the system requirements expect the long distance data transmission. The lack of long distance transmission is solely because of hindrance of the FCC licensing process. The current prototype implementation functions properly at the system level and it can be easily transformed into a final implantation because of careful consideration of the design modularity through the project development. The complex behavior of data transmission protocol is also successfully implemented to transmit the water property data reliably from one side of the computer to the other side.

Although the RF data communication system worked properly, there are still plenty of areas in which the system could be improved. To enhance the robustness of the data transfer protocol, the checking mechanism for duplicate packets is necessary. This can be done by implementing the sequence number in the data packet. An error log file is also desirable in order to keep track of errors occurred during the operation. Implementing the error log will enhance the system maintainability.

Overall, the project experience was worthwhile because it not only offers great deal of knowledge about the RF data communication system but also benefits students and faculty at Union Geology Department.

## 12. References

[2] Stallings, W., *Data and computer communication.* Pearson Education, Inc., New Jersey, 2004.

[3] Stevens. "Telemetry and Wireless Data Communications - Stevens Water Monitoring Systems" http://www.stevenswater.com/telemetry_com/index.aspx.

[4] Compliance Engineering. "Unlicensed Wireless Data Communications, Part II: Specifying RF Parameters" http://www.ce-mag.com/archive/02/Spring/cutler2.html.

[5] *The ARRL Handbook for Radio Communications*. 2011 88th Edition. The American Radio Relay League, Inc., 2010.

[6] The Quagi antenna. "The Quagi antenna" http://commfaculty.fullerton.edu/woverbeck/quagi.htm.

[7] Silicon Labs. "F8051F02x Data Sheet" http://www.silabs.com/Support%20Documents/TechnicalDocs/C8051F02x.pdf.

[8] Silicon Labs. "Support Document" http://www.silabs.com/pages/DownloadDoc.aspx?FILEURL=Support%20Documents/TechnicalDocs/C8051F02x-DK.pdf&src=SupportDocLibrary.

[9] RobotShop. http://www.robotshop.com/content/PDF/board-of-education-rev-c-manual-28150.pdf.

[10] Kurose, J. and Ross, K., *Computer Networking: A Top-Down Approach.* Fifth Edition. Pearson Education, Inc., Boston, MA, 2010.

[11] Profiler. "HeyWhatsThat Path Profiler" http://www.heywhatsthat.com/profiler.html.

## 13. Appendices

## 13.1. Appendix A: Complete code listing of a sender program

```
/*******************************************************************************
*****
    Program:    SendFSM.c
    Author:     Aung Soe
    Date:       02/18/2011

This program reads data from a local file, "DataFile.txt" and divides data into small
packets
together with CRC value. Pre-arranged data packets are stored in a 2D array. Next,
connection
is initialized. After establishing the connection, each packet is sent in sequence from
Ballston Lake to Union College. Then, the connection is closed and the data transmission
process is ended.

This program contains following functions:
connection_status()      : Initialize or close the connection based on the chosen mode
read_data()              : Read data from a local file and store it in an array
make_packet()            : Break data into several packets. Each packet starts with SOT,
                           follows by a piece of data and CRC value of that data and ends
with
                           EOT.
make_ctrl_packet()       : Create a control packet based on a chosen parameter. Parameters
                           include CONNECT, READY, BYE, ACK and NAK.
send_packet()            : Send all data packets in sequence through the serial port.
read_timeout()           : Read a single character with timeout.
create_2D_char_array()   : Allocate memory addresses for an 2D array to store data packets
cleanUp_2D_array()       : Free memory space of the 2D array after the data transmission
                           process is finished.
open_RS232port()         : Open a RS-232 serial port.

*******************************************************************************
****/

#include <stdio.h>       /* Standard buffered input/output */
#include <termios.h>     /* POSIX terminal control definitions */
#include <unistd.h>      /* UNIX standard function definitions */
#include <stdlib.h>      /* Standard library definitions */
#include <fcntl.h>       /* File control definitions */
#include <string.h>      /* String operations */

#include "crc.h"


/*******************************************************************************
*****
    Function prototypes
*******************************************************************************
****/
char read_timeout(int, int);
int init_Connection (int, int);
void send_packet (char **, int, int, int);
void read_data(char *, char *);
int makePacket (char **, char *, int *);
int makeCtrlPacket (char *, char);
void cleanUp_2D_array(char **, int);
int open_RS232port(void);
char** create_2D_char_array(int, int);

/*******************************************************************************
*****
    Global CONSTANTS
*******************************************************************************
****/
#define SOT 0x02        /* Start of Text */
#define EOT 0x03        /* End of Text */
#define ACK 0x06        /* Acknowledge */
#define NAK 0x15        /* Neg. Acknowledge */
```

```
#define CONNECT 0x43    /* C: Connect */
#define READY 0x52      /* R: Ready */
#define BYE 0x42        /* B: Disconnect */
#define CR 0x0D
#define LF 0x0A

#define CTRL_PKT_SIZE 10            /* Control packet size */
#define PKT_SIZE 100                /* Data packet size */
#define MAX_PKT 360                 /* Maximum number of packet */
#define PORT_NAME "/dev/ttyS0"
#define FILE_NAME "DataFile.txt"

/* Size of 2 Dimension Array */
#define NUM_ROW 360         /* number of row */
#define NUM_COLUMN 100      /* number of column */

/* define states for state machine */
#define SEND_CONNECT_PACKET 0
#define SEND_PACKET 1
#define CHECK_CHAR 2
#define READ_PACKET 3
#define CHECK_CTRL_MSG 4
#define EMPTY_SERIAL_BUFF 5
#define SEND_CTRL_MSG 6

/* define Opcode */
#define DATA_PKT 1
#define CTRL_PKT 2


/*****************************************************************************************
*****
    Main Routine
*****************************************************************************************
****/
main () {

    int i, j, nrows, ncolumns, port;
    int connected = 0;
    int disconnected = 0;
    int total_packet, total_char;
    char data[NUM_ROW * NUM_COLUMN];
    char **packet2D;

    struct termios old_flags;
    struct termios new_flags;

    /* open the RS-232 port */
    port = open_RS232port();
    if (port == -1) {
        printf("\n Error opening RS232 port\n");
        exit(-1);
    }

    /*
        Set up raw/non-canonical mode
        VTIME is set to 10 and VMIN is set to 0
    */
    tcgetattr(port, &old_flags);
    new_flags = old_flags;
    new_flags.c_lflag &= ~(ECHO | ICANON | ISIG);
    new_flags.c_iflag &= ~(BRKINT | ICRNL);
    new_flags.c_oflag &= ~OPOST;
    new_flags.c_cc[VTIME] = 10;
    new_flags.c_cc[VMIN] = 0;
    new_flags.c_cflag |= CS8;

    if (tcsetattr(port, TCSAFLUSH, &new_flags) < 0) {
        printf("\n Error setting raw mode!! \n");
        exit(-1);
    }
    /* set baud rate */
    cfsetispeed(&new_flags, B9600);

    /* Allocate memory location for packet2D an 2D array */
    /* Size of packet2D: 360 x 100 (NUM_ROW x NUM_COLUMN)*/
    packet2D = create_2D_char_array(NUM_ROW, NUM_COLUMN);

    printf("\n Reading data from %s ... \n", FILE_NAME);
    read_data(data, FILE_NAME);

    printf("\n Cteating packets ... \n");
    total_packet = makePacket(packet2D, data, &total_char);

    for(i = 0; i < total_packet; i++) {
        printf("\n Packet #%d: %s \n", (i+1), packet2D[i]);
    }

    printf("\n Total packet being created: %d\n ", total_packet);

    printf("\n Initializing connection... \n");
    connected = init_Connection(port, 1);
    printf("\n Connection Status: %d \n", connected);

    if (connected) {
        printf("\n Initializing send process... \n");
        send_packet (packet2D, port, total_packet, total_char);

        printf("\n Closing connection... \n");
        disconnected = init_Connection(port, 0);
        printf("\n Connection Status: %d \n", disconnected);
    }

    cleanUp_2D_array(packet2D, NUM_ROW);

    tcsetattr(port, TCSANOW, &old_flags);
    close(port);

}   // end of main


/*****************************************************************************************
*****
    Read character with timeout

    char read_timeout(int port_ID, int timeout_sec);

    int port_ID     : (input) file descriptor for the RS232 serial port
    int timeout_sec : (input) timeout (in sec)

    return a single character found at Serial port; if no character is found, return
NULL.

*****************************************************************************************
****/
char read_timeout(int port_ID, int timeout_sec) {

    char found_char;        /* Character found at Serial port */
    int counter, i;         /* Time counter for each sec */
    int found;              /* indicator for found character */
    int time_out;           /* indicator for timeout */

    /* read a character with timeout */
    counter = 0;
    found = FALSE;
    time_out = FALSE;

    while ((! found) & (! time_out)) {
        /* read must be non-blocking */
        if ((i = read(port_ID, &found_char, 1)) > 0) {
            found = TRUE;   /*character found*/
        }
        else {
            counter++;
            if (counter >= timeout_sec) {
                time_out = TRUE;
            }
            else {
                /* sleep for 1 ms - 1 character time at 9600 BAUD */
                usleep(1000);
            }
        }
    }

    if (found) {
        return (found_char);    // return a single found character
    }

    if (time_out) {
        return (found_char);    // return NULL char if timeout
    }

}


/************************************************************************************
*****
    Create control packet with CRC

    int make_ctrl_packet (char *ctrl_pkt, char ctrl_char);

    char ctrl_char  : (input) a chosen control message
    char *ctrl_pkt  : (output) a control packet with the control message

    Return value from this function is the number of character in the control packet.

    Packet data format:
        [0]        [1]         [2]          [3]        [4]         [5]
    |---------||----------||----------||----------||----------||---------|
    |   SOT   ||  opCode  || CTRL MSG ||  CRC HI  || CRC  LOW ||   EOT   |
    |---------||----------||----------||----------||----------||---------|
    | 1  Byte || 1   Byte || 1   Byte || 1   Byte || 1  Byte  || 1  Byte |

    Byte 0 consists of the data header (SOT). Byte 1 is an opcode for control packet
    (opCode=2). Byte 2 is a control character. Byte 3 and Byte 4 consists of a 16-bit (2-
byte)
    Cyclic Redundancy Check (CRC) Checksum value calculated using the Byte 1. The
calculation
    is done by calling the function, crcFast() from "crc.h". Byte 5 is the end of packet.

*************************************************************************************
****/
int make_ctrl_packet (char *ctrl_pkt, char ctrl_char) {

    crc crc_value;                          /* CRC value of control packet */
    char temp_ctrl_char[3];
    int total_char;                         /* total character in control packet */
    char opCode = CTRL_PKT;                 /* Opcode for control packet (Opcode = 2) */
    int i = 0;

    crcInit();                              /* initialize the CRC look up table */

    temp_ctrl_char[0] = SOT;
    temp_ctrl_char[1] = opCode;
    temp_ctrl_char[2] = ctrl_char;

    ctrl_pkt[i] = SOT;                      /* Begin a control pkt with SOT */
    ctrl_pkt[++i] = opCode;                 /* Opcode for control packet */
    ctrl_pkt[++i] = ctrl_char;

    crc_value = crcFast(temp_ctrl_char, 3); /* Generate CRC for ctrl packet */
    ctrl_pkt[++i] = crc_value >> 8;          /* Store crc high byte */
    ctrl_pkt[++i] = crc_value & 0x00FF;      /* Store crc low byte */
    ctrl_pkt[++i] = EOT;                     /* End a control pkt with EOT */

    return (total_char = ++i);               /* Return number of character in control pkt
*/
}


/*****************************************************************************************
*****
    Initialize/Close connection

    int connection_status (int port_id, int mode);

    int port_id : (input) file descriptor for RS232 serial port
    int mode    : (input) mode 0 is used in closing connection (disconnect)
                          mode 1 is used in initializing connection (connect)

    return connection status (0/1); 0 indicate connection failed and 1 indicated
connection
    established.

    There is a significant difference in the establishing connection and the closing
    connection.

    The protocol for establishing connection is as follows (shown is a simplified form):
    1)  If mode 1 is selected, a CONNECT packet is created and sent into a serial port.
Also,
        the number of trial to establish the connection is specified.
    2)  Wait for a return packet with the timeout, if the timeout occurs before receiving
a
        control packet, the same CONNECT packet is resent.
    3)  If there is a incoming control packet, a first element of the packet (SOT) is
checked.
        If the packet begins with SOT, the remaining packet is read. Otherwise, the
remaining
        packet is discarded, and then the same CONNECT packet is resent.
    4)  After successfully reading the control packet, the CRC value is calculated on
that
        packet for an error checking purpose. If the remainder of CRC is zero, the packet
is
        further checked for a control message inside the packet. If the remainder is
something
        else, the same CONNECT packet is resent.
    5)  If the control message is READY, the connection process is indicated as success
by
        returning true; But if the control message is something else, the same CONNECT
packet
        is resent.

    The protocol for closing connection is as follows (shown is a simplified form):
    1)  If mode 0 is selected, a BYE packet is created and sent into a serial port. After
that,
        the process is simply ended, and return false. There is no additional process.

*****************************************************************************************
****/
int connection_status (int port_id, int mode) {

    char ctrlPkt[CTRL_PKT_SIZE];    /* control packet */
    char rcv_pkt[CTRL_PKT_SIZE];    /* receive packet */
    char opCode;
    char ctrl_msg;
    char check_char, temp_char;
    int pkt_len;                    /* control packet length */
    int rcv_pkt_len;                /* receive packet length */
    int char_cnt;                   /* character counter for receive packet */
    int trial_cnt;                  /* connter/disconnect trial counter */
    int done;                       /* indicator for connection process */
    int connected;                  /* indicator for connect/disconnect status */
    int state;                      /* state indicator for the state machine */
    int max_trial;                  /* maximum allow trial for connect/disconnect */
    int i;

    crc crc_remainder;              /* remainder for CRC calculation of receive packet */
    crcInit();                      /* initialize the CRC look up table */

    trial_cnt = 0;
    done = FALSE;                   /* initialize connection process */
    connected = FALSE;              /* initialize connection status */
    state = SEND_CONNECT_PACKET;    /* initialize state with SEND_CONNECT_PACKET */

    while ((! done) && (! connected)) {

        if (mode == 1) {
            pkt_len = make_ctrl_packet (ctrlPkt, CONNECT);
            max_trial = 3;          /* number of trial to establish connection */
        }
        else if (mode == 0) {
            pkt_len = make_ctrl_packet (ctrlPkt, BYE);
        }
        else {
            printf("\n Incorrect input parameter in make_ctrl_packet() function! \n");
            done = TRUE;
        }

        switch (state) {

            case SEND_CONNECT_PACKET:

                write(port_id, ctrlPkt, pkt_len);

                trial_cnt++;

                if (mode == 1) {
                    printf("\n CONNECT PKT has been sent. \n");
                    state = CHECK_CHAR;
                }
                else if (mode == 0) {
                    printf("\n BYE PKT has been sent. \n");
                    done = TRUE;
                }

                break;  // end of case SEND_CONNECT_PACKET


            case CHECK_CHAR:

                printf("\n Connecting (Trial #%d) ... \n", trial_cnt);

                /* read a single char with 5 sec timeout */
                check_char = read_timeout(port_id, 5);

                if (check_char == SOT) {
                    state = READ_PACKET;
                }
                else if (check_char == 0x00) {
                    if (trial_cnt < max_trial) {
                        state = SEND_CONNECT_PACKET;
                        printf("\n Connection Failed! \n");
                    }
                    else {
                      done = TRUE;
                    }
                }
                else {
                    state = EMPTY_SERIAL_BUFF;
                }

                break;  // end of case CHECK_CHAR
            case EMPTY_SERIAL_BUFF:

                i = 0;

                // check a character at the buffer and read it to clear the receiving
buffer
                while ((i = read(port_id, &temp_char, 1)) > 0);

                printf("\n Connection Failed! \n");

                if (trial_cnt < max_trial) {
                    state = SEND_CONNECT_PACKET;
                }
                else {
                  done = TRUE;
                }
                break;  // end of case EMPTY_SERIAL_BUFF


            case READ_PACKET:

                char_cnt = 0;
                i = 0;

                rcv_pkt[char_cnt++] = SOT;
                //printf("\n rcv_pkt[%d]: %02X \n", char_cnt, rcv_pkt[char_cnt]);
                //char_cnt++;

                // check a character at the buffer
                while ((i = read(port_id, &temp_char, 1)) > 0) {
                    if (temp_char != EOT) { // check whether the packet ends with EOT or
not
                        rcv_pkt[char_cnt++] = temp_char;
                        //printf("\n rcv_pkt[%d]: %02X \n", char_cnt, rcv_pkt[char_cnt]);
                        //char_cnt++;
                    }
                }

                rcv_pkt_len = char_cnt;
                //printf("\n Receive PKT Lenght: %d \n", rcv_pkt_len);

                crc_remainder = crcFast(rcv_pkt, rcv_pkt_len);
                //printf("\n Remainder: %02X\n", crc_remainder);

                if (crc_remainder == 0) {

                    opCode = rcv_pkt[1];

                    if (opCode == CTRL_PKT) {
                        state = CHECK_CTRL_MSG;
                    }
                    else {
                        state = SEND_CONNECT_PACKET;
                    }
                }
                else {
                    printf("\n Connection Failed!\n");

                    if (trial_cnt < max_trial) {
                        state = SEND_CONNECT_PACKET;
                    }
                    else {
                        done = TRUE;
                    }
                }

                break;  // end of case READ_PACKET


            case CHECK_CTRL_MSG:
                ctrl_msg = rcv_pkt[2];

                if (ctrl_msg == READY) {
                    printf("\nConnection established.\n");
                    done = TRUE;
                    connected = TRUE;
                }
                else {
                    printf(" Connection Failed!\n");

                    if (trial_cnt < max_trial) {
                        state = SEND_CONNECT_PACKET;
                    }
                    else {
                        done = TRUE;
                    }
                }

                break;  // end of case CHECK_CTRL_MSG

        }   // end of switch

    }   // end of while

    if (connected) {
        return(TRUE);   /* Return 1 when connection is established */
    }
    else {
        return(FALSE);  /* Return 0 when connection is not established */
    }

}   // end of function


/*****************************************************************************************
*****
    Send packets

    void send_packet (char **pkt2D, int port_id, int num_pkt, int num_char);

    char **pkt2D    : (input) packets are organized into 2D array
                      <size of pkt2D: num_pkt x num_char>
    int port_id     : (input) file descriptor for the RS232 serial port
    int num_pkt     : (input) total number of packet available to transmit
    int num_char    : (input) number of character in each packet

    The protocol for transmitting each data packet is as follows (shown is a simplified
form):
    1)  Send a data packet into a serial port.
    2)  Wait for a return packet with the timeout, if the timeout occurs before receiving
a
        control packet, the same data packet is resent.
    3)  If there is a incoming control packet, a first element of the packet (SOT) is
checked.
        If the packet begins with SOT, the remaining packet is read. Otherwise, the
remaining
        packet is discarded, and then the same data packet is resent.
    4)  After successfully reading the control packet, the CRC value is calculated on
that
        packet for an error checking purpose. If the remainder of CRC is zero, the packet
is
        further checked for a control message inside the packet. If the remainder is
something
        else, the same data packet is resent.
    5)  If the control message is ACK, the process repeat from step 1 to send a new data
        packet. But if the control message is NAK or something else, the same data packet
is
        resent.
```

```
*******************************************************************************************
****/
void send_packet (char **pkt2D, int port_id, int num_pkt, int num_char) {

    char send_pkt[MAX_PKT];         /* point for each individual packet */
    char rcv_pkt[CTRL_PKT_SIZE];    /* receive packet */
    char temp_char;
    char check_char;
    char ctrl_msg;
    char opCode;
    int packet_num;                 /* need to keep track of packet of number */
    int rcv_pkt_len;                /* receive packet length */
    int char_cnt;
    int state;                      /* state indicator for the state machine */
    int done;                       /* indicator for sending process */
    int trial_cnt;                  /* number of trial to send each pkt */
    int i;

    crc crc_remainder;              /* remainder for CRC calculation of receive packet */
    crcInit();                      /* initialize the CRC look up table */

    packet_num = 0;                 /* intitialize the packet number */
    trial_cnt = 0;
    done = FALSE;                   /* initialize sending process */
    state = SEND_PACKET;            /* initialize state with SEND_PACKET */

    while (! done) {

        switch (state) {

            case SEND_PACKET:

                for(i=0; i<num_char; i++) {
                  send_pkt[i] = pkt2D[packet_num][i];
                  //printf("\n send_pkt[%d]: %02X \n", i, send_pkt[i]);
                }

                write(port_id, send_pkt, num_char);

                trial_cnt++;
                printf("\n Sending PKT[#%d] on <#%d> Trial... \n", (packet_num + 1),
trial_cnt);

                state = CHECK_CHAR;

                break;  // end of case SEND_PACKET


            case CHECK_CHAR:

                /* read a single char with 5 sec timeout */
                check_char = read_timeout(port_id, 5);

                if (check_char == SOT) {
                    state = READ_PACKET;
                }
                else if (check_char == 0x00) {
                  printf("\n Sending PKT[#%d] Failed! \n", (packet_num + 1));
                  state = SEND_PACKET;
                }
                else {
                    state = EMPTY_SERIAL_BUFF;
                }

                break;  // end of case CHECK_CHAR


            case EMPTY_SERIAL_BUFF:

                i = 0;
                // check a character at the buffer and read it to clear the receiving
buffer
                while ((i = read(port_id, &temp_char, 1)) > 0);

                printf("\n Sending PKT[#%d] Failed! \n", (packet_num + 1));

                state = SEND_PACKET;

                break;  // end of case EMPTY_SERIAL_BUFF


            case READ_PACKET:

                char_cnt = 0;
                i = 0;

                rcv_pkt[char_cnt++] = SOT;
                //printf("\n rcv_pkt[%d]: %02X \n", char_cnt, rcv_pkt[char_cnt]);
                //char_cnt++;

                // check a character at the buffer
                while ((i = read(port_id, &temp_char, 1)) > 0) {
                    if (temp_char != EOT) { // check whether the packet ends with EOT or
not
                        rcv_pkt[char_cnt++] = temp_char;
                        //printf("\n rcv_pkt[%d]: %02X \n", char_cnt, rcv_pkt[char_cnt]);
                        //char_cnt++;
                    }
                }

                rcv_pkt_len = char_cnt;
                //printf("\n Receove PKT Lenght: %d \n", rcv_pkt_len);

                crc_remainder = crcFast(rcv_pkt, rcv_pkt_len);
                //printf("\n Remainder: %02X \n", crc_remainder);

                if (crc_remainder == 0) {

                    opCode = rcv_pkt[1];

                    if (opCode == CTRL_PKT) {
                        state = CHECK_CTRL_MSG;
                    }
                    else {
                        state = SEND_PACKET;
                    }
                }
                else {
                    printf("\n Sending PKT[#%d] Failed! \n", (packet_num + 1));

                    state = SEND_PACKET;

                }

                break;  // end of case READ_PACKET


            case CHECK_CTRL_MSG:

                ctrl_msg = rcv_pkt[2];

                if (ctrl_msg == ACK) {

                    printf("\n Transmission of PKT[#%d] is successful! \n", (packet_num +
1));

                    packet_num++;               /* increment packet number */

                    /* if packet numner is less than total packet avaliable */
                    if (packet_num < num_pkt) {
                        state = SEND_PACKET;
                        trial_cnt = 0;          /* reset trial counter */
                    }
                    else {
                        done = TRUE;            /* indicate sending all packets is
complete */
                    }
                }
                else {
                    printf("\n Sending PKT[#%d] Failed! \n", (packet_num + 1));

                    state = SEND_PACKET;
                }

                break;  // end of case CHECK_CTRL_MSG

        }   // end of switch

    }   // end of while

}   // end of function


/*****************************************************************************************
*****
    Read characters from a file into an array

    void read_data (char *raw_data, char *fn);

    char *fn        : (input) file name of the actual data file in the memory
    char *raw_data  : (output) return the actual data in an array

*****************************************************************************************
****/
void read_data (char *raw_data, char *fn) {

    char temp_data;
    int num_char;

    /* open a file */
    FILE *datafile;             /* need a pointer to FILE */

    datafile = fopen(fn, "r");  /* Open "file_name" for reading */

    if (datafile == NULL) {
        printf("\n Cannot open file (%s)! \n", fn);
    }

    num_char = 0;               /* Initialize character counter */

    while ((temp_data = fgetc(datafile)) != EOF) {    /* Read until end-of-file */
        raw_data[num_char++] = temp_data;
    }

    raw_data[num_char++] = temp_data;     /* End data array with EOF */

    /* close a file */
    fclose(datafile);

}


/*****************************************************************************************
*****
    Create an 2D array with each row contains a singal data packet including CRC value

    int make_packet (char **pkt, char *data, int *num_char);

    char **pkt      : (output) an 2D array of data packets
    char *data      : (input) raw data read from a local file
    int *num_char   : (output) total number of character in each packet including SOT,
actual
```

```
                    data, CRC value (Hi byte and Low byte) and EOT

    Break data into several packets. Each packet starts with SOT, follows by a piece of
data,
    CRC value of that data, and ends with EOT. Return total number of packets, number of
    char in each packet and a 2D array of data packets.

    Packet data format:
        [0]         [1]         [2]            [n +1]     [n + 2]     [n + 3]     [n + 4]
    |---------||----------||----------|    |----------||----------||----------||---------
|
    |   SOT   ||  opCode  ||    Data   |  ~ |   Data    ||  CRC HI  || CRC  LOW ||   EOT
|
    |---------||----------||----------|    |----------||----------||----------||---------
|
    | 1  Byte || 1   Byte || 1   Byte        1   Byte || 1   Byte    1   Byte || 1  Byte
|

    Byte 0 consists of the data header (SOT). Byte 1 is an opcode for data packet
(opCode=1)
    Byte 2 through n+1 are n number of byte of the actual data value. Byte n+2 and Byte
n+3
    consists of a 16-bit (2-byte) Cyclic Redundancy Check (CRC) Checksum value calculated
    using the Byte 1 through n  together. The calculation is done by calling the
function,
    crcFast() from "crc.h" file. Byte n+4 is the end of packet.
*****************************************************************************************
****/
int make_packet (char **pkt, char *data, int *num_char) {

    char temp_pkt[NUM_COLUMN];              /* temporary storage of actral data */
    char temp_char;
    int total_char;                         /* total number of actual data */
    int num_pkt;                            /* total number being created */
    int opCode;                             /* Opcode for data packet (Opcode = 1) */
    int i;
    int row, column;

    crc crc_value;

    crcInit();                              /* initialize the CRC look up table */

    opCode = DATA_PKT;                      /* Opcode for data packet (Opcode = 1) */
    i = 0;
    row = 0;
    column = 0;

    /*

    */
    while (data[i] != EOF) {                /* Read data until EOF (end-of-file) */
        if (data[i] != LF) {
            if (column == 0) {
                pkt[row][column] = SOT;     /* Begin a pkt with SOT */
                temp_pkt[column] = SOT;
                column++;
            }
            else if (column == 1) {
                pkt[row][column] = opCode;  /* place an opCode for each data packet */
                temp_pkt[column] = opCode;
                column++;
            }
            else {
                temp_char = data[i++];
                pkt[row][column] = temp_char;
                temp_pkt[column] = temp_char;
                column++;
            }
        }
        else {
            temp_char = data[i++];
            pkt[row][column] = temp_char;
            temp_pkt[column] = temp_char;

            total_char = column + 1;                    /* Number of char in each pkt */

            crc_value = crcFast(temp_pkt, total_char);  /* Generate CRC for each pkt */
            pkt[row][++column] = crc_value >> 8;        /* Store crc high byte */
            pkt[row][++column] = crc_value & 0x00FF;    /* Store crc low byte */
            pkt[row][++column] = EOT;                   /* End a packet with EOT */

            *num_char = ++column;   /* Number of char in the complete pkt */

            row++;                  /* Increment row index of pkt[][] */
            column = 0;             /* Set column index of pkt[][] */
        }
    }

    num_pkt = row;
    return (num_pkt);       /* Return number of pkt */

}


/*****************************************************************************
*****
    Clean up the allocated 2-D array
*****************************************************************************
****/
void cleanUp_2D_array(char **array, int x) {

    int i;

    for(i = 0; i < x; i++)
        free(array[i]);

    free(array);
}


/*****************************************************************************
*****
    Open RS232 Serial port 1
*****************************************************************************
****/
int open_RS232port(void) {
    int port_id;
    port_id =  open(PORT_NAME, O_RDWR);
    return(port_id);
}


/*****************************************************************************
*****
    Allocate memory address for an 2D array

    char** create_2D_char_array(int num_rows, int num_cols);

    int num_rows    : (input) number of rows in the 2D array
    int num_cols    : (input) number of column in the 2D array

    This function return memory addresses allocated for a 2D array.
*****************************************************************************
****/
char** create_2D_char_array(int num_rows, int num_cols) {

    char **2D_array;

    /* Allocate pointer memory for the first dimension of a matrix[][]; */
    2D_array = (char **) malloc(num_rows * sizeof(char *));
    if(2D_array == NULL){
        free(2D_array);
        printf("Memory allocation failed while allocating for dim[].\n");
        exit(-1);
    }

    /* Set all pointers to NULL */
    /* This will make it possible to call free() if out of memory during data allocation
*/
    memset(2D_array, 0, num_rows * sizeof(char*));

    /* Allocate integer memory for the second dimension of a dim[][]; */
    register int i;
    for(i = 0; i < num_rows; i++) {
        2D_array[i] = (char *) malloc(num_cols * sizeof(char));
        if(NULL == 2D_array[i]){
            free(2D_array[i]);
            printf("Memory allocation failed while allocating for dim[x][].\n");
            exit(-1);
        }
    }
    return *&2D_array;  /* Return an allocated 2D array */
}
```

## 13.2. Appendix B: Complete code listing of a receiver program

```
/*****************************************************************************************
*****
    Program:    GetFSM.c
    Author:     Aung Soe
    Date:       02/18/2011

This program receives data from Ballston Lake and save it into a local file,
"DataFile.txt"
on Union PC.

This program contains following functions:
read_packet()           : Receive data packets from a serial port and store them in an 2D
                          array.
write_data()            : Write the receive data packets into a local file for storage.
make_ctrl_packet()      : Create a control packet based on a chosen parameter. Parameters
                          include CONNECT, READY, BYE, ACK and NAK.
read_timeout()          : Read a single character with timeout.
create_2D_char_array()  : Allocate memory addresses for an 2D array to store data packets
cleanUp_2D_array()      : Free memory space of the 2D array after the data transmission
                          processs is finished.
open_RS232port()        : Open a RS-232 serial port.

*****************************************************************************************
****/

#include <stdio.h>      /* Standard buffered input/output */
#include <termios.h>    /* POSIX terminal control definitions */
#include <unistd.h>     /* UNIX standard function defintions */
#include <stdlib.h>     /* Standard library definitions */
#include <fcntl.h>      /* File control definitions */
#include <string.h>     /* String operations */

#include "crc.h"


/*****************************************************************************************
*****
    Function prototypes
*****************************************************************************************
****/

int makeCtrlPacket (char *, char);
void read_packet (char **, int, int *, int *);
void write_Data (char **, char *, int, int);
char read_timeout(int, int);
void cleanUp_2D_array(char **, int);
int open_RS232port(void);
char** create_2D_char_array(int, int);

/*****************************************************************************************
*****
    Global CONSTANTS
*****************************************************************************************
****/
#define SOT 0x02        /* Start of Text */
#define EOT 0x03        /* End of Text */
#define ACK 0x06        /* Acknowledge */
#define NAK 0x15        /* Neg. Acknowledge */
#define CONNECT 0x43    /* C: Connect */
#define READY 0x52      /* R: Ready */
#define BYE 0x42        /* B: Disconnect */
#define CR 0x0D
#define LF 0x0A

#define CTRL_PKT_SIZE 10            /* Control packet size */
#define PKT_SIZE 100                /* Data packet size */
#define MAX_PKT 360                 /* Maximun number of packet */
#define PORT_NAME "/dev/ttyS0"
#define FILE_NAME "DataFile.txt"

/* Size of 2 Dimension Array */
```

```
#define NUM_ROW 360                  /* number of row */
#define NUM_COLUMN 100               /* number of column */

/* define states for state machine */
#define SEND_CONNECT_PACKET 0
#define SEND_PACKET 1
#define CHECK_CHAR 2
#define READ_PACKET 3
#define CHECK_OPCODE 4
#define CHECK_CTRL_MSG 5
#define EMPTY_SERIAL_BUFF 6
#define SEND_CTRL_PACKET 7

/* define Opcode */
#define DATA_PKT 1
#define CTRL_PKT 2


/*****************************************************************************************
*****
    Main Routine
*****************************************************************************************
****/
main() {

    int port;               // file descriptor for the serial port
    int i;
    int total_pkt, total_char;
    int nrows, ncolumns;
    char **packet2D;

    struct termios old_flags, new_flags;

    /* open the RS-232 port */
    port = open_RS232port();
    if (port == -1) {
        printf("\n Error opening RS232 port \n");
        exit(-1);
    }

    /*
        Set up raw/non-canonical mode
        VTIME is set to 10
        VMIN is set to 0
    */
    tcgetattr(port, &old_flags);
    new_flags = old_flags;
    new_flags.c_lflag &= ~(ECHO | ICANON | ISIG);
    new_flags.c_iflag &= ~(BRKINT | ICRNL);
    new_flags.c_oflag &= ~OPOST;
    new_flags.c_cc[VTIME] = 1;
    new_flags.c_cc[VMIN] = 0;
    new_flags.c_cflag |= CS8;

    if (tcsetattr(port, TCSAFLUSH, &new_flags) < 0) {
        printf("\n Error seting raw mode!! \n");
        exit(-1);
    }

    /* set baud rate */
    cfsetispeed(&new_flags, B9600);


    /* Allocate memory location for packet2D an 2D array */
    /* Size of packet2D: 360 x 100 (NUM_ROW x NUM_COLUMN)*/
    packet2D = create_2D_char_array(NUM_ROW, NUM_COLUMN);

    printf("\n Initializing receiving process... \n");
    read_packet(packet2D, port, &total_pkt, &total_char);

    printf("\n Writing data into %s ... \n", FILE_NAME);
    write_Data(packet2D, FILE_NAME, total_pkt, total_char);
    printf("\n Writing complete! \n");

    cleanUp_2D_array(packet2D, NUM_ROW);

    tcsetattr(port, TCSANOW, &old_flags);
    close(port);
}

/*****************************************************************************************
*****
    Receive data packets from a serial port and store them in an 2D array.

    void read_packet (char **pkt2D, int port_id, int *num_pkt, int *num_char);

    char **pkt2D    : (output) packets are organized into 2D array
                      <size of pkt2D: num_pkt x num_char>
    int port_id     : (input) file descriptor for RS232 serial port
    int *num_pkt        : (output) total number of packet available to transmit
    int *num_char   : (output) number of character in each packet

    The protocol for receiving each data packet is as follows (shown is a simplified
form):
    1)  Wait for an incoming packet with the timeout, if the timeout occurs before
receiving
        any packet, keep checking for the packet.
    2)  If there is the incoming packet, a first element of the packet (SOT) is checked.
If
        the packet begins with SOT, the remaining packet is read. Otherwise, the
remaining
        packet is discarded, and continue checking for the packet.
    3)  After successfully reading the packet, the CRC value is calculated on that packet
for
        an error checking purpose. If the remainder of CRC is zero, the packet is further
        checked for an opcode inside the packet. If the remainder is something else, a
NAK
        packet is sent back to the sender to indicate that there is an error in the
packet,
        and then keep checking for another packet.
    4)  If the opcode is CTRL_PKT, the packet is further checked for a control message
inside
        the packet. If the opcode is DATA_PKT, a ACK packet is sent back to the sender to
        indicate that there is on error in the packet, and then keep checking for another
        packet. But if the opcode is something else, a NAK packet is sent back to the
sender
        to indicate that there is an error in the packet, and then keep checking for
another
        packet.
    5)  If the control message is CONNECT, a NAK packet is sent back to the sender to
indicate
        that it is ready to send data packets. If the control message is BYE, the
receiving
        process is ended. But if something else, keep checking for another packet.

*****************************************************************************************
****/
void read_packet (char **pkt2D, int port_id, int *num_pkt, int *num_char) {

    char rcv_pkt[PKT_SIZE];         /* point for each individual receive packet */
    char ctrl_pkt[CTRL_PKT_SIZE];   /* control packet */
    char ctrl_char;                 /* need to indicate which control char to send */
    char ctrl_msg;
    char temp_char;
    char check_char;
    char opCode;                    /* opcode */
    int ctrl_pkt_len;               /* control packet length */
    int packet_num;                 /* need to keep track of packet of number */
    int rcv_pkt_len;                /* receive packet length */
    int char_cnt;
    int state;                      /* state indicator for the state machine */
    int done;                       /* indicator for sending process */
    int i;

    crc crc_remainder;                /* remainder for CRC calculation of receive packet */
    crcInit();                        /* initialize the CRC look up table */

    packet_num = 0;                   /* intitialize the packet number */
    done = FALSE;                     /* initialize reading process */
    state = CHECK_CHAR;               /* initialize state with CHECK_CHAR */

    while (! done) {

        switch (state) {

            case CHECK_CHAR:

                /* read a single character with 5 seconds timeout */
                check_char = read_timeout(port_id, 50);
                //printf("The first character is: %02X\n", check_char);

                if (check_char == SOT) {
                    state = READ_PACKET;
                }
                else if (check_char == 0x00) {
                    state = CHECK_CHAR;
                    printf("\n Waiting... \n");
                }
                else {
                    printf("\n Waiting... \n");
                    state = EMPTY_SERIAL_BUFF;
                }

                break;  // end of case CHECK_CHAR


            case EMPTY_SERIAL_BUFF:

                i = 0;

                // check a character at the buffer and read it to clear the buffer
                while ((i = read(port_id, &temp_char, 1)) > 0);

                state = CHECK_CHAR;

                break;  // end of case EMPTY_SERIAL_BUFF


            case READ_PACKET:

                char_cnt = 0;
                i = 0;

                rcv_pkt[char_cnt++] = SOT;
                //printf("\n rcv_pkt[%d]: %02X \n", char_cnt, rcv_pkt[char_cnt]);
                //char_cnt++;

                // check a character at the buffer
                while ((i = read(port_id, &temp_char, 1)) > 0) {

                    if (temp_char != EOT) { // check whether the packet ends with EOT or
not
                        rcv_pkt[char_cnt++] = temp_char;
                        //printf("\n rcv_pkt[%d]: %02X \n", char_cnt, rcv_pkt[char_cnt]);
                        //char_cnt++;
                    }
                }

                rcv_pkt_len = char_cnt;
                //printf("\nReceove PKT Lenght: %d\n", rcv_pkt_len);

                crc_remainder = crcFast(rcv_pkt, rcv_pkt_len);
                //printf("\n Remainder: %02X \n", crc_remainder);
                if (crc_remainder == 0) {
                    state = CHECK_OPCODE;
                }
                else {
                    ctrl_char = NAK;
                    state = SEND_CTRL_PACKET;
                }

                break;  // end of case READ_PACKET


            case CHECK_OPCODE:

                opCode = rcv_pkt[1];

                if (opCode == CTRL_PKT) {
                    state = CHECK_CTRL_MSG;
                }
                else if (opCode == DATA_PKT) {
                    ctrl_char = ACK;
                    state = SEND_CTRL_PACKET;

                    /* return number of char in a complete pkt
                    not include SOT, opCode, CRC value and EOT */
                    *num_char = rcv_pkt_len - 4;

                    /*
                        Save the packet without opCode and CRC value into a 2D array
(pkt2D)
                    */
                    for (i = 2; i < (rcv_pkt_len - 2); i++) {
                        pkt2D[packet_num][i - 2] = rcv_pkt[i];
                    }

                    packet_num++;       /* increment packet_number */
                }
                else {
                    ctrl_char = NAK;
                    state = SEND_CTRL_PACKET;
                }

                break;


            case CHECK_CTRL_MSG:

                ctrl_msg = rcv_pkt[2];

                if (ctrl_msg == CONNECT) {
                    state = SEND_CTRL_PACKET;
                    ctrl_char = READY;
                }
                else if (ctrl_msg == BYE) {
                    printf("\n Close Connection!\n");
                    done = TRUE;    /* Receiving process is done */
                }
                else {
                    state = CHECK_CHAR;
                }

                break;  // end of case CHECK_CTRL_MSG


            case SEND_CTRL_PACKET:

                if (ctrl_char == ACK) {
                    printf("\n Receiving PKT[#%d] is successful! \n", (packet_num));
                    ctrl_pkt_len = make_ctrl_packet (ctrl_pkt, ctrl_char);

                    write(port_id, ctrl_pkt, ctrl_pkt_len);
                    printf("\n ACK PKT is sent back to the sender. \n");

                    state = CHECK_CHAR;
                }
                else if (ctrl_char == NAK) {
                    printf("\n ERROR receiving PKT[#%d] \n", (packet_num));
                    ctrl_pkt_len = make_ctrl_packet (ctrl_pkt, ctrl_char);

                    write(port_id, ctrl_pkt, ctrl_pkt_len);
                    printf("\n NAK PKT is sent back to the sender. \n");

                    state = CHECK_CHAR;
                }
                else if (ctrl_char == READY) {
                    printf("\n Connection Ready... \n");
                    ctrl_pkt_len = make_ctrl_packet (ctrl_pkt, ctrl_char);

                    write(port_id, ctrl_pkt, ctrl_pkt_len);
                    printf("\n READY PKT is sent back to the sender. \n");

                    state = CHECK_CHAR;
                }

                break;  // end of case SEND_CTRL_PACKET

        }  /* end of switch */

    }  /* end of while */

    *num_pkt = packet_num;      /* return number of received packets */

}  /* end of function */


/*****************************************************************************************
*****
    Create control packet with CRC

    int make_ctrl_packet (char *ctrl_pkt, char ctrl_char);

    char ctrl_char  : (input) a chosen control message
    char *ctrl_pkt  : (output) a control packet with the control message

    Return value from this function is the number of character in the control packet.

    Packet data format:
        [0]        [1]         [2]          [3]        [4]         [5]
    |---------||----------||----------||----------||----------||---------|
    |   SOT   ||  opCode  || CTRL MSG ||  CRC HI  || CRC  LOW ||   EOT   |
    |---------||----------||----------||----------||----------||---------|
    | 1  Byte || 1   Byte || 1   Byte || 1   Byte || 1  Byte  || 1  Byte |

    Byte 0 consists of the data header (SOT). Byte 1 is an opcode for control packet
    (opCode=2). Byte 2 is a control character. Byte 3 and Byte 4 consists of a 16-bit (2-
byte)
    Cyclic Redundancy Check (CRC) Checksum value calculated using the Byte 1. The
calculation
    is done by calling the function, crcFast() from "crc.h". Byte 5 is the end of packek.

*****************************************************************************************
****/
int make_ctrl_packet (char *ctrl_pkt, char ctrl_char) {

    crc crc_value;                          /* CRC value of control packet */
    char temp_ctrl_char[3];
    int total_char;                         /* total character in control packet */
    char opCode = CTRL_PKT;                 /* Opcode for control packet (Opcode = 2) */
    int i = 0;

    crcInit();                              /* initialize the CRC look up table */
    temp_ctrl_char[0] = SOT;
    temp_ctrl_char[1] = opCode;
    temp_ctrl_char[2] = ctrl_char;

    ctrl_pkt[i] = SOT;                          /* Begin a control pkt with SOT */
    ctrl_pkt[++i] = opCode;                     /* Opcode for control packet */
    ctrl_pkt[++i] = ctrl_char;

    crc_value = crcFast(temp_ctrl_char, 3); /* Generate CRC for ctrl packet */

    ctrl_pkt[++i] = crc_value >> 8;             /* Store crc high byte */
    ctrl_pkt[++i] = crc_value & 0x00FF;         /* Store crc low byte */
    ctrl_pkt[++i] = EOT;                        /* End a control pkt with EOT */

    return (total_char = ++i);                  /* Return number of character in control pkt
*/
}


/*****************************************************************************************
*****
    Allocate memory address for an 2D array

    char** create_2D_char_array(int num_rows, int num_cols);

    int num_rows    : (input) number of rows in the 2D array
    int num_cols    : (input) number of column in the 2D array

    This function return memory addresses allocated for a 2D array.

*****************************************************************************************
****/
char** create_2D_char_array(int num_rows, int num_cols) {

    char **2D_array;

    /* Allocate pointer memory for the first dimension of a matrix[][]; */
    2D_array = (char **) malloc(num_rows * sizeof(char *));
    if(2D_array == NULL){
        free(2D_array);
        printf("Memory allocation failed while allocating for dim[].\n");
        exit(-1);
    }

    /* Set all pointers to NULL */
    /* This will make it possible to call free() if out of memory during data allocation
*/
    memset(2D_array, 0, num_rows * sizeof(char*));

    /* Allocate integer memory for the second dimension of a dim[][]; */
    register int i;
    for(i = 0; i < num_rows; i++) {
        2D_array[i] = (char *) malloc(num_cols * sizeof(char));
        if(NULL == 2D_array[i]){
            free(2D_array[i]);
            printf("Memory allocation failed while allocating for dim[x][].\n");
            exit(-1);
        }
    }
    return *&2D_array;  /* Return an allocated 2D array */
}


/*****************************************************************************************
*****
    Write data into a local file for storage
*****************************************************************************************
****/
void write_data (char **pkt, char *fn, int num_pkt, int num_char) {

    char temp_char[PKT_SIZE];
    int i, j;

    /* open a file */
    FILE *datafile;              /* need a pointer to FILE */

    datafile = fopen(fn, "a");   /* Open "file_name" for appending */

    printf("\n Number of packet: %d \n", num_pkt);
    printf("\n Number of character in each packet: %d \n", num_char);

    for (i = 0; i < num_pkt; i++) {

        if (datafile == NULL) {
            printf("Cannot open file (%s)!\n", fn);
        }

        for (j = 0; j < num_char; j++) {
            temp_char[j] = pkt[i][j];
        }

        fwrite(temp_char, num_char, 1, datafile);

    }

    fclose(datafile);
}


/*****************************************************************************************
*****
    Read character with timeout

    char read_timeout(int port_ID, int timeout_sec);
    int port_ID     : (input) file descriptor for RS232 serial port
    int timeout_sec : (input) timeout (in sec)

    return a single character found at Serial port; if no character is found, return
NULL.

*****************************************************************************************
****/
char read_timeout(int port_ID, int timeout_sec) {

    char found_char;        /* Character found at Serial port */
    int counter, i;         /* Time counter for each sec */
    int found;              /* indicator for found character */
    int time_out;           /* indicator for timeout */

    /* read a character with timeout */
    counter = 0;
    found = FALSE;
    time_out = FALSE;

    while ((! found) & (! time_out)) {
        /* read must be non-blocking */
        if ((i = read(port_ID, &found_char, 1)) > 0) {
            found = TRUE;               /*character read*/
        }
        else {
            counter++;
            if (counter >= timeout_sec) {
                time_out = TRUE;
            }
            else {
                /* sleep for 1 ms - 1 character time at 9600 BAUD */
                usleep(1000);
            }
        }
    }

    if (found) {
        //printf("Message found: %02X\n", found_char);
        return (found_char);    // return a single found character
    }

    if (time_out) {
        //printf("Time out error!\n");
        //printf("Message found: %02X\n", found_char);
        return (found_char);    // return NULL char if timeout
    }
}


/*****************************************************************************************
*****
    Open RS232 Serial port 1
*****************************************************************************************
****/
int open_RS232port(void) {
    int port_id;
    port_id =  open(PORT_NAME, O_RDWR);
    return(port_id);
}


/*****************************************************************************************
*****
    Clean up the allocated 2-D array
*****************************************************************************************
****/
void cleanUp_2D_array(char **array, int x) {
    int i;

    for(i = 0; i < x; i++)
        free(array[i]);

    free(array);
}
```

### 13.3. Appendix C: Completer code listing of a transceiver control program

```
#pragma CODE    //generate assembly code in the LST file
```

```
//-----------------------------------------------------------------------------
------
// Program : SeniorProject_8051_Final_Code.c
// Author  : Aung Soe
// Date    : 01/16/2011
//
// This program is used to control data transmission between the PC and the
radio.
// The purpose of this program is to isolate the characteristic of the PC and
that of
// radio. Replacing the radio will not affect any hardware or software
components of
// the PC. This program uses two UARTs to transmit data from the PC to the
radio and
// vice versa. The data is received from the PC via RI0 of UART0 and transmit
to the
// radio via TX1 of UART1. The data from the radio is received via RI1 of UART1
and
// transmit the PC via TX0 of UART0. RI0 and RI1 (receive interrupt flags) are
used
// to detect the incoming data from the PC and the radio respectively. The
advantage
// of this program is that prior to receiving the data, it does not need to
know the
// number of characters. The program uses timeout to indicate the end of the
received
// data.
//
// I am not sure the way I implement the timeout is efficient but it works the
way I
// want. First, I initialize the timeout as an int variable (16 bits) starting
from
// 2. After receiving the first character, the timeout is start counting and
each
// remaining character is read at the same time. The timeout is incremented
from 2 up
// to 2^16. If the timeout is greater than 2^16, the timeout becomes overrun
and
// returns back to 0. It keeps incrementing again. The timeout period between
each
// character is approximately 200 ms which is counting from 2 until reching 1
again.
// If the subsequent character is not received within 200 ms, the receiving
process
// is ended and the transmission process is initialized.
//-----------------------------------------------------------------------------
------

#include <c8051f020.h>                     // SFR declarations
#include <stdio.h>

//-----------------------------------------------------------------------------
------
// 16-bit SFR Definitions for 'F02x
//-----------------------------------------------------------------------------
------

sfr16 DP       = 0x82;                     // data pointer
sfr16 TMR3RL   = 0x92;                     // Timer3 reload value
sfr16 TMR3     = 0x94;                     // Timer3 counter
sfr16 ADC0     = 0xbe;                     // ADC0 data
sfr16 ADC0GT   = 0xc4;                     // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                     // ADC0 less than window
```

```
sfr16 RCAP2    = 0xca;                    // Timer2 capture/reload
sfr16 T2       = 0xcc;                    // Timer2
sfr16 RCAP4    = 0xe4;                    // Timer4 capture/reload
sfr16 T4       = 0xf4;                    // Timer4
sfr16 DAC0     = 0xd2;                    // DAC0 data
sfr16 DAC1     = 0xd5;                    // DAC1 data


//----------------------------------------------------------------------------
------
// Global CONSTANTS
//----------------------------------------------------------------------------
------
#define SYSCLK 22118400                 // approximate SYSCLK frequency in Hz
#define BAUDRATE 9600                   // Baud rate of UART0 and UART1 in bps
#define TRUE 1
#define FALSE 0

unsigned char xdata PC_msg[1000];
unsigned char xdata Radio_msg[1000];
unsigned char *char_ptr0;
unsigned char *char_ptr1;
unsigned char buffer0, buffer1;
int Timeout;
int total_char;
int char_count0;
int char_count1;
int i, j;
bit PC_msgReady, Radio_msgReady;
bit finish_receiving;


//----------------------------------------------------------------------------
------
// Function PROTOTYPES
//----------------------------------------------------------------------------
------
//

void Port_IO_Init(void);
void UART0_Init(void);
void UART1_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);


//----------------------------------------------------------------------------
------
// MAIN Routine
//----------------------------------------------------------------------------
------
//

void main (void) {

    WDTCN = 0xde;            // Disable watchdog timer
    WDTCN = 0xad;

    Oscillator_Init();       // Enable external oscillator
    Port_IO_Init();          // Enable ports
    UART0_Init();            // Enable UART0
    UART1_Init();            // Enable UART1
    Interrupts_Init();       // Enable global interrupts
    PC_msgReady = 0;
    Radio_msgReady = 0;

    char_count0 = 0;
    char_count1 = 0;

    char_ptr0 = PC_msg;
    char_ptr1 = Radio_msg;

    while (1) {             // Loops forever

        // While RI0 is not set and RI1 is not set (polling) ...
        while ((RI0 != 1) && (!(SCON1 & (1 << 0))));

        Timeout = 2;                 // Initialize the timeout

        // If RX0 is set, incoming char from PC is read and transmitted to
Radio
        // Receive port     : RI0
        // Transmit port    : TX1
        if (RI0 == 1) {

            RI0 = 0;                       // Reset receive interrupt flag (RI0 =
0)
            buffer0 = SBUF0;               // Put a received char in a buffer
            *(char_ptr0++) = buffer0;      // Retrieve incoming char from PC and
                                           // store it into an array (PC_msg)
            char_count0++;                 // Increment character counter

            finish_receiving = FALSE;      // Initialize reading process indicator

            while (! finish_receiving) {

                // while RI0 is not set and there is no Timeout...
                // Timeout between each incoming char is about 200 ms
                while ((RI0 != 1) && (Timeout != 1)) {
                    for(i=0;i<5;i++);
                    ++Timeout;
                }

                // if incoming character from PC is ready without Timeout...
                // or if incoming character from PC is ready when Timeout
occurs...
                if (((RI0 == 1) && (Timeout != 1)) || ((RI0 == 1) && (Timeout
== 1))) {

                    RI0 = 0;                       // Reset receive interrupt flag
(RI0)
                    buffer0 = SBUF0;               // Put a received char in a
buffer
                    *(char_ptr0++) = buffer0;      // Retrieve incoming char from
PC and
                                                   // continue stroing it in the
array
                    Timeout = 2;                   // Reset Timeout

                    char_count0++;                 // Character counting
                }

                // if incoming character from PC is not ready and Timeout
occurs...
                else if ((RI0 == 0) && (Timeout == 1)) {
                    total_char = char_count0;    // total characters in received
msg
                    char_count0 = 0;             // Clear character counter
                    finish_receiving = TRUE;     // Indicate message reading
finish
                    SCON1 |= 0x02;               // Set TI1 (TX interrupt flag)
to 1
                    PC_msgReady = 1;             // Indicate message from PC is
ready
                }
            }

            // if the transmitter (TX1) avaliable and PC message is ready...
            if ((SCON1 & (1 << 1)) && PC_msgReady) {

                char_ptr0 = PC_msg;

                for (j = 0; j < total_char; j++) {
                    while (!(SCON1 & (1 << 1)));// While TI1 = 0
                    SBUF1 = *(char_ptr0++);      // Load char to send through
TX1
                    for(i=0;i<2500;i++);         // Delay between each byte
                }

                char_ptr0 = PC_msg;
                PC_msgReady = 0;                 // Mark msg buffer empty
                SCON1 &= 0xFC;                   // Clear UART1 interrupt flags
                break;
            }
        }   // End of if statement for checking whether RI0 or RI1 is set



        // If RX1 is set, incoming message from radio is read and transmitted
to PC
        // Receive port      : RI1
        // Transmit port     : TX0
        else if ((SCON1 & (1 << 0))){

            SCON1 &= ~(1 << 0);              // Reset receive interrupt flag
(RI1 = 0)
            buffer1 = SBUF1;
            *(char_ptr1++) = buffer1;        // Retrieve incoming char from PC
            char_count1++;                   // Start character counter

            finish_receiving = FALSE;

            while (! finish_receiving) {

                // while RI1 is not set and there is no Timeout...
                // Timeout between each incoming char is about 200 ms
                while ((!(SCON1 & (1 << 0))) && (Timeout != 1)) {
                    for(i=0;i<5;i++);
                    ++Timeout;
                }

                // if incoming character from radio is ready without Timeout
                // or if incoming character from radio is ready when Timeout
occurs
                if (((SCON1 & (1 << 0)) && (Timeout != 1)) || ((SCON1 & (1 <<
0)) && (Timeout == 1))) {
```

```
                    SCON1 &= ~(1 << 0);          // Reset receive interrupt flag
                    buffer1 = SBUF1;
                    *(char_ptr1++) = buffer1;    // Retrieve incoming char from
PC

                    Timeout = 2;                 // Reset Timeout

                    char_count1++;               // Character counting
                }

                // if incoming character from radio is not ready and Timeout
occurs
                else if ((!(SCON1 & (1 << 0))) && (Timeout == 1)) {

                    total_char = char_count1;    // Total char in received
packet
                    char_count1 = 0;             // Clear character counter
                    finish_receiving = TRUE;     // Indicate message reading
finish
                    TI0 = 1;                     // Set TI0 (TX interrupt flag)
to 1
                    Radio_msgReady = 1;          // Indicate Radio message ready
                }
            }

            // if the transmitter (TX0) avaliable and Radio message is ready...
            if ((TI0 == 1) && Radio_msgReady) {

                char_ptr1 = Radio_msg;

                for (j = 0; j < total_char; j++) {
                    while (TI0 == 0);            // While TI0 = 0
                    SBUF0 = *(char_ptr1++);      // Load char to send through
TX0
                    for(i=0;i<2500;i++);         // Delay between each byte
                }

                char_ptr1 = Radio_msg;
                Radio_msgReady = 0;              // Mark msg buffer empty
                SCON0 &= 0xFC;                   // Clear UART0 interrupt flags
                break;
            }
        }   // End of else if statement for checking whether RI0 or RI1 is set

    }   // End of while loop

}   // End of main


//-----------------------------------------------------------------------------
------
// UART0_Init
//-----------------------------------------------------------------------------
------
//

void UART0_Init()            // Configures UART to our specifications.
{
    PCON      |= 0x80;        // Disable Baud Rate / 2 (Set SMOD0 = 1)
    SCON0      = 0x50;        // SCON0: mode 1, 8-bit UART enable RX
    SCON0     &= 0xFC;        // Clear interrupt pending flags
    //SCON0     |= 0x02;      // Set TI0 (TX interrupt flag) to 1
    CKCON     |= 0x10;                    // Timer 1 uses SYSCLK as time base
    TCON      = 0x40;                     // Timer 1 enabled
    TMOD      = 0x20;                     // TMOD: timer 1, mode 2. 8-bit reload
    TH1       = -(SYSCLK/BAUDRATE/16);  // Set Timer 1 reload value for
baudrate (9600bps)
}


//-----------------------------------------------------------------------------
------
// UART1_Init
//-----------------------------------------------------------------------------
------
//

void UART1_Init()           // Configures UART to our specifications.
{
    PCON     |= 0x10;       // Disable Baud Rate / 2 (Set SMOD1 = 1)
    SCON1     = 0x50;       // SCON1: mode 1, 8-bit UART, enable RX
    SCON1    &= 0xFC;       // Clear interrupt pending flags
    //SCON1    |= 0x02;     // Set TI1 (TX interrupt flag) to 1

    CKCON    |= 0x40;                     // Timer 4 uses SYSCLK as time base
    T4CON     = 0x34;                     // Set baudrate generator for UART1
    RCAP4     = -(SYSCLK/BAUDRATE/32);  // Set Timer 4 reload value for
baudrate (9600bps)
    T4        = RCAP4;
}


//-----------------------------------------------------------------------------
------
// Port_IO_Init
//-----------------------------------------------------------------------------
------
//

void Port_IO_Init()       // Configures ports to our specifications.
{
    // P0.0  -  TX0 (UART0), Push-Pull,  Digital
    // P0.1  -  RX0 (UART0), Open-Drain, Digital
    // P0.2  -  TX1 (UART1), Push-Pull,  Digital
    // P0.3  -  RX1 (UART1), Open-Drain, Digital

    P0MDOUT   = 0x05;     // Set P0.0 & P0.2 to be Push-Pull Output
    XBR0      = 0x04;     // Enable crossbar
    XBR1      = 0x00;
    XBR2      = 0x44;
}


//-----------------------------------------------------------------------------
------
// Oscillator_Init
//-----------------------------------------------------------------------------
------
//

void Oscillator_Init()    // Configures the external oscillator to our
specifications.
{
    int i = 0;
    OSCXCN    = 0x67;               // Enable 22.1184 MHz as external
oscillator
    for (i = 0; i < 3000; i++);     // Wait 1ms for initialization
    while ((OSCXCN & 0x80) == 0);
    OSCICN    = 0x08;
}


//------------------------------------------------------------------------------
------
// Interrupts_Init
//------------------------------------------------------------------------------
------
// Initiate interrupt service routine

void Interrupts_Init(void)
{
    EA         = 1;        // Enable Global Interrupt (IE: Interrupt Enable)
}
```