Zigbee Mesh Network

By

Peter Fyon 100652096

Andrew Kusz 100685317

Bong Geon (John) Koh 100684909

Supervisor: Dr. Aitken, Prof. Eatherley

A report submitted in partial fulfillment of the requirements

of SYSC 4907 Engineering Project

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

April 7, 2010

Abstract

The purpose of the ZigBee Mesh Network project was to research, design and implement a system where a remote control (R/C) car could be controlled over a wireless mesh network. This system allows a human operator to input commands using the controller, which then sends the commands to the R/C car through the wireless mesh network. The R/C car performs the commands received by controller. The controller instructions and the car operations are done in real-time.

The ZigBee Mesh Network was designed as a proof of concept on the versatility of wireless mesh communications. This project provides the framework needed for future projects that require or rely on either wireless mesh communications or robust communication methods.

The project required learning different types of vehicle steering and control methods, research on various wireless technologies that support mesh networking, study of networks, protocols and packetization, and development of embedded software.

All of the code, documentation, images, and presentation slides should always be available at the repository hosted at http://zigbeemeshnetwork.googlecode.com/.

Acknowledgements

We would like to thank Professor Victor Aitken for providing us the opportunity for this project. He accepted our proposal for this project, even though he was on sabbatical. We greatly appreciate his generous opportunity and the help he has given us.

We would like to thank Professor Graham Eatherley for his inspiration on this project. He allowed us to gather our ideas and materialize them into a suitable project. He provided us with his multitude of knowledge and experience, along with his personal tools and equipment.

We would like to thank Jacob Hammer for his help on the construction of the controller.

We would also like to acknowledge the contributions of the Smart Rollator project in documenting portions of the serial code and the process to compile assembly code to be used in an Interactive C program.

Finally, we would like to acknowledge Daniel Lemay for his assistance in obtaining much of the hardware used in this project.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

This report is a detailed document describing our fourth-year-project accomplishments over the development lifetime of eight months. The ZigBee Mesh Network project was supervised by Professor V. Aitken and Professor G. Eatherley. The ZigBee Mesh Network project was active during the period from September 2009 to April 2010. This report describes the research, design, testing and implementation completed for the ZigBee Mesh Network project.

1.1 Problem Background (Andrew)

Wireless communications allow us to overcome many obstacles that would be difficult or unfeasible for wired communications. Whether it is harsh terrain, long distances, or temporary installations, wireless networks allow quicker and more inexpensive setup. Wireless communications do have some shortcomings though.

Unlike wired communications, where someone would have to have physical access to the transmission cable to “listen in” on the messages being sent, wireless is broadcast in the open air, allowing anyone potential access to your signal. This security issue becomes more apparent when the signal strength is increased to reach longer distances.

Line of sight is another issue which can lower the potential range of wireless communications. Physical barriers such as walls, mountains, or trees can lower the signal quality by quite a bit. This can be overcome by increasing the signal power, though this is a potentially inefficient solution.

1.2 Problem Motivation (Andrew)

Typical wireless communications consist of a point-to-point setup; a transmitter transmits a signal to the receiver. Cellphones, wireless routers, and remote control vehicles are some examples of real world applications of point-to-point communications. While point-to-point communication works in many cases, there are some major limitations that make it less than ideal in some situations.

A typical point-to-point wireless network is limited by line of sight. A receiver which is not in direct line of sight to the transmitter must rely on signal penetration of the obstacle and/or bouncing of the signal. Also, as with any wireless network, the coverage area may be larger than necessary. Figure 1 below demonstrates this fact. The wireless tower provides coverage to the cottages in the north, while simultaneously providing unnecessary coverage to the wilderness in the south.

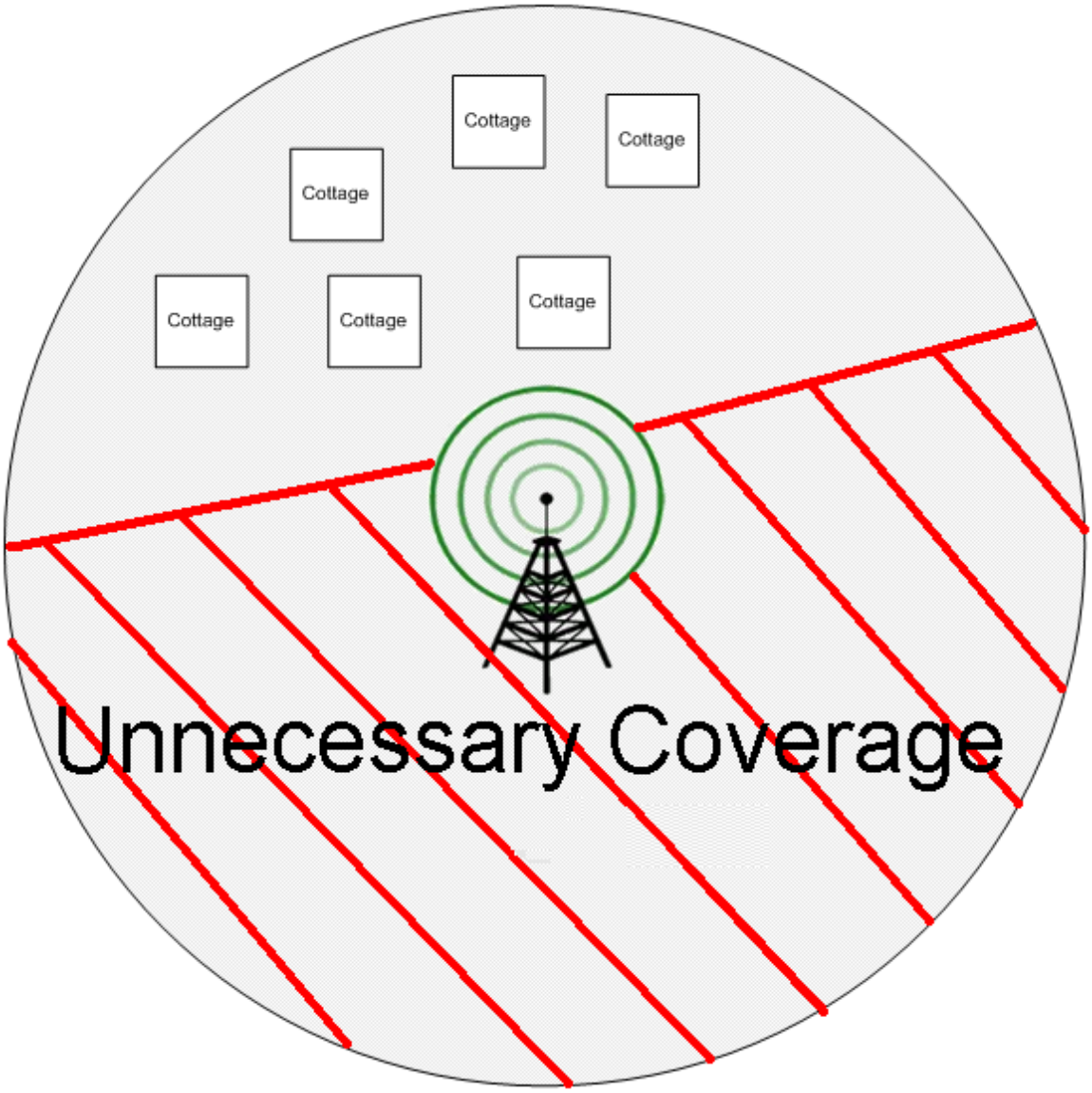

Figure 1: Unnecessary Coverage in a Wireless System[1]

One possible solution to this problem is to use a unidirectional antenna to focus the signal to the desired area, as can be seen in figure 2 below. One drawback to this option is if future expansion occurs in the area south of the transmitter, providing wireless coverage to them would require modifying the transmitter to provide coverage to the new area.

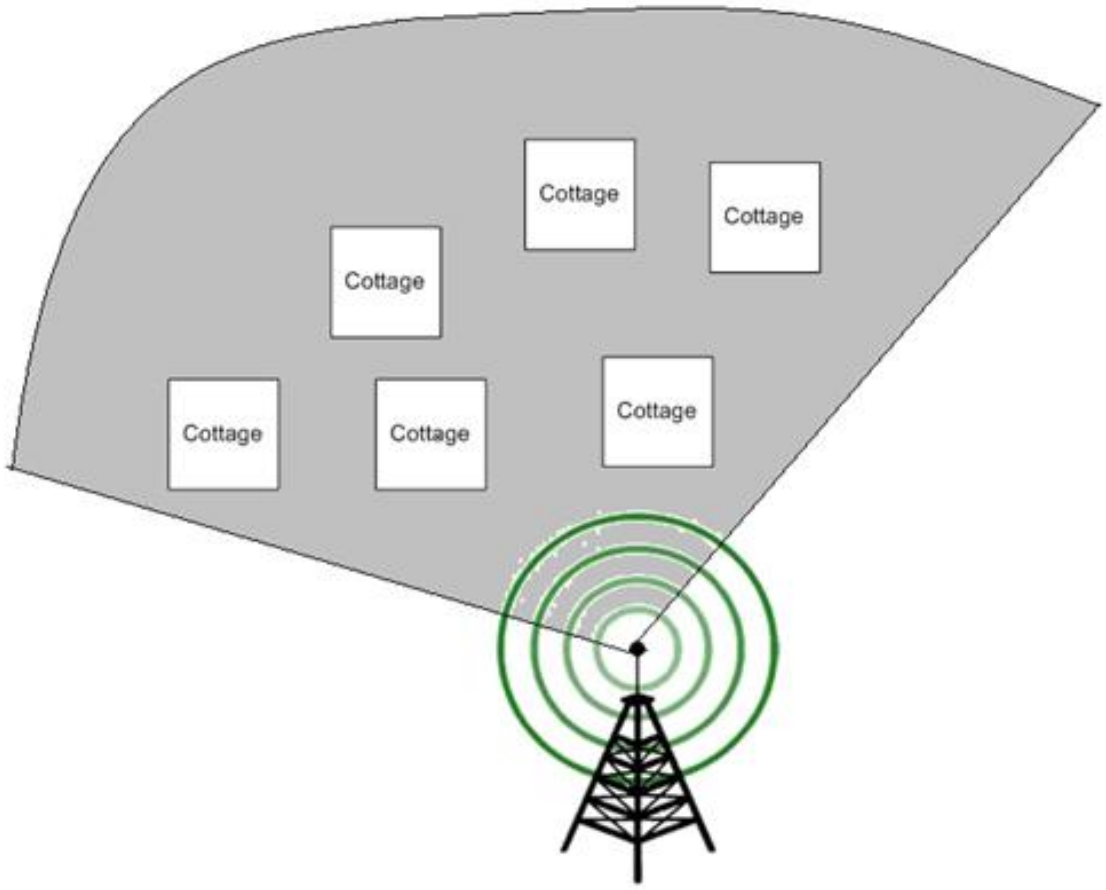

Figure 2: Unidirectional Antenna Focusing Signal to Desired Area[1]

1.3 Problem Statement (Peter)

Our project seeks to address and improve upon the attributes of range, expandability, reliability, and security of a standard long range transmitter/receiver design.

1.4 Proposed Solution (Andrew)

To overcome many of the drawbacks of wireless networks, we chose to implement an 802.15.4 based wireless mesh network using the ZigBee specification.

A ZigBee mesh network consists of three types of modules: a coordinator, several routers, and one or more end devices. The coordinator is the central part of the network, and has the responsibility of forming the network by assigning addresses to the other devices. Routers act as relay nodes that can pass the message along to the desired device. End devices are paired with the devices that you need to interact with.

The coordinator can send messages to any node on the network. If the node is not in range, the message will be sent to a neighbouring node which will then forward it onward to the destination. With optimal node placement, the mesh network can provide coverage to the same desired area as a point to point network while potentially using less power by greatly reducing the wasted coverage.

Because each node has a smaller transmission range, and relies on the relaying of messages between nodes to reach further destinations, the coverage area is more precise, which means it is less likely that somebody could listen in on the transmissions.

Line of sight can also be overcome by adding router nodes over or around the obstacles that would normally cause signal degradation in typical point to point communications.

The ZigBee mesh network is capable of growing or shrinking depending on one’s needs by adding or removing nodes. Should a node malfunction, the network will attempt to route packets around the failed node and give the impression of seamless operation.

2.0 The Engineering Project

2.1 Health and Safety (Peter)

The health and safety of the public was taken into account in the design of this system. The system was designed to use low power radio frequency transmitters which meet Conformité Européenne (CE) standards.

In addition, the system was designed to use small, battery operated remote control vehicles. The vehicle control software also provides a safety function to stop driving in the event that it loses connection to the network.

2.2 Engineering Professionalism (Peter)

The professional responsibilities of the group members were met in the progress and completion of this project by keeping logs of daily work, research, and in the completion of documentation for every major research component.

2.3 Project Management (Peter)

In all phases of the project, we used a number of tools to facilitate the management of the project. Weekly meetings were attended to facilitate group work and collaboration, and to ensure that every group member's work was progressing as expected.

In addition, log books were kept by each individual and every major research component and design decision was documented. Finally, an off-site Subversion repository was used to store all versions of the code, images, presentation slides, and documentation in a publicly available, web-accessible location to ensure no documentation was lost and provides a dated history of changes made to the above

documents.

2.4 Individual Contributions

2.4.1 Project Contributions

Peter Fyon:

- Project management (setup and management of Subversion repository, project website)
- Primary design and implementation of vehicle drive system
- Test code for the vehicle drive system and serial data transmission
- Primary design of both reliable data transfer alternating bit protocol and non-reliable protocol
- Research into assembly interrupt routine to handle packets received from the UART
- assisted in design, implementation, and testing of vehicle control system and ZigBee mesh network
- Initial car teardown analysis

Andrew Kusz:

- Research into XBee modules and ZigBee specification
- Initial car teardown analysis
- Design, setup, and implementation of the mesh network
- Testing of functionality of mesh network
- Assisted in design and implementation and debugging of vehicle control system and vehicle drive system

John Koh:

- Primary design and implementation of vehicle control system
- Primary design of the physical control mechanism
- Research into serial communications between Handy Boards and XBee nodes
- Assisted in vehicle drive system design and implementation
- Assisted in design and implementation and debugging of non-reliable protocol and vehicle drive system
- Initial car teardown analysis

2.4.2 Report Contributions

The primary contributor for each section and subsection is listed in brackets beside the section title. Unless otherwise noted, every section has been collaboratively edited by the other group members.

3.0 Research and Design

This section of the report covers the research and design work done for the project. Hardware and software specifications were obtained to find the most suitable and applicable solutions.

3.1 High Level Design (John)

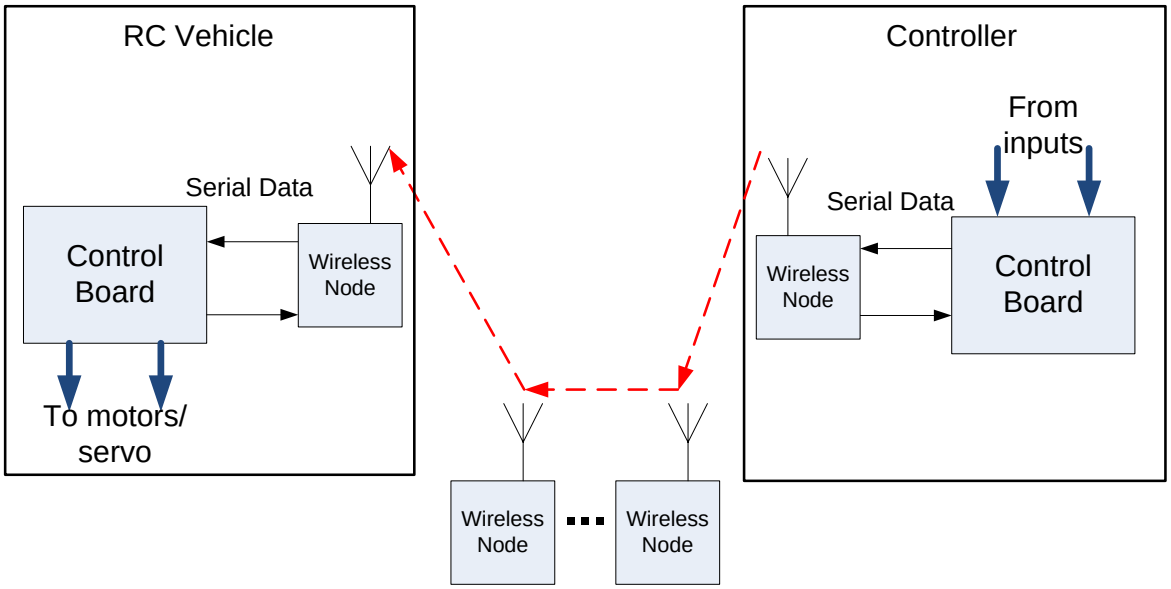

Figure 3: Block Diagram of the ZigBee Mesh Network System

The ZigBee Mesh Network system consists of three major subsystems: the RC vehicle, the wireless mesh network and the controller. The RC vehicle and the controller components include a control board and a wireless transmitter. The control boards are used for either encoding or decoding operational code (OpCode) packets. The wireless transmitters are used to send or receive data to or from devices on the network. The wireless mesh network is comprised entirely out of wireless transmitters, acting as routers. The wireless transmitters in the wireless mesh network are used to forward

instructions from the controller to the RC vehicle.

3.2 Hardware Platforms (Peter)

The embedded development boards were required to be able to communicate with the chosen wireless transmitter, receive inputs from a physical control mechanism, and drive the required motors and/or servo on the remote controlled vehicle.

There are a large number of embedded, however, given the past experience of group members, two boards were compared.

The Arduino line of boards are based off the Atmel AVR line of 8-bit microcontrollers. The standard Arduino, named the Duemilanove, uses the Atmel ATmega 168 or 328 at 16 MHz, and provides 14 digital pins: two can be used for serial communications at TTL levels, four for serial peripheral interface (SPI) bus, two for external interrupts, and 6 eight-bit pulse-width modulation (PWM) channels. [2]

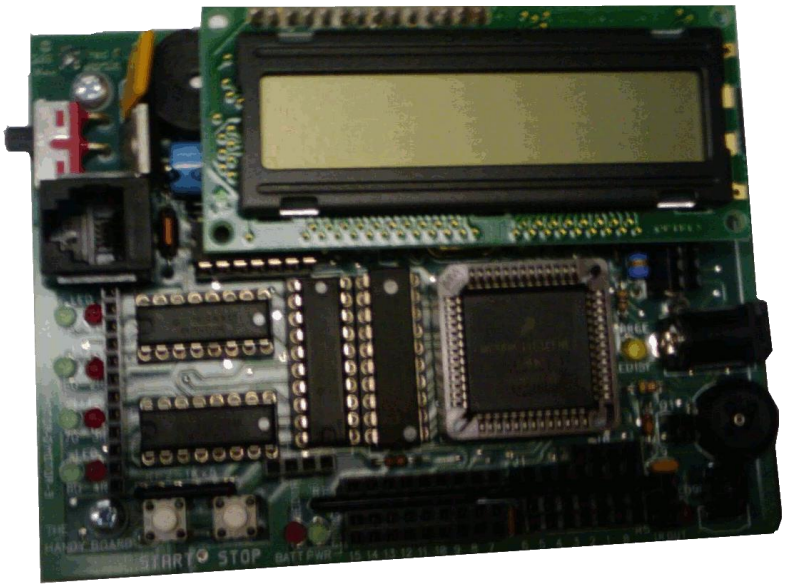

Figure 4: The Handy Board

The Handy Board is a development board based off the Motorola 68HC11 (now Freescale 68HC11) 8-bit microcontroller. The board includes a Motorola 68HC11 with system clock at 2 MHz, 32KB of static RAM, four motor ports capable of driving the motors at 1 amp each (using PWM at the internal battery voltage), a 16 x 2 character LCD screen, 9 digital pins, 7 analog pins, two user-programmable buttons, an analog knob, a piezo buzzer, and headers exposing the 9.6 V internal battery, 5 V from the regulator, the HC11 SPI bus, and the HC11 universal asynchronous receiver/transmitter (UART) at RS-232 levels [3]. The expansion board adds 10 analog sensor inputs, 9 digital outputs, and 6 servo control signals [4].

The Arduino and Handy Board are very similar with regards to the number of input/output ports available to the programmer, and both meet the requirements of the development boards.

The Handy Board was chosen to be used as the development platform for the project since it met all the requirements for the design, did not require any external hardware to drive the motors, were available immediately and at no cost to the project, and the group members all had previous experience with development for the 16-bit equivalent to the HC11, the Motorola 68HC12 microcontroller.

Three programming languages could be used with the Handy Board: Interactive C, C, and/or assembly language.

Assembly language provides the greatest control over the hardware, but is the most complex and requires custom subroutines to handle every aspect of control. C provides a large amount of control over the hardware and is less complex than assembly language, but still requires custom subroutines to handle most functions.

Interactive C is a C-like language that provides an operating system on the board. It uses time slicing to allow multiple processes to operate. Interactive C provides the least control over the hardware, but includes many libraries to handle complex functions for control of the hardware.

Interactive C was chosen to be used to program both the controller and RC vehicle boards as it was decided that the language provided sufficient control over the hardware, with the option to include assembly language interrupt routines if needed.

3.3 Wireless Transmitters (John)

The wireless transmitters needed to fulfill specific requirements, as wireless transmitter requirements were crucial to the system. The wireless transmitters must be able to run on very low power and must be capable of forming secure mesh networks.

The most suitable 802.15.4 wireless specification that met these requirements was the ZigBee specification. The ZigBee specification was capable of low power, secure, mesh networking. Other specifications such as WirelessHART were considered. These specifications had similar low power, secure, mesh networking capabilities [5]. However, the ZigBee specification was the only specification that had the ability to sleep when inactive and woke up on call. While inactive, average power consumption is drastically reduced. Since low power consumption was a key requirement of the system, the ZigBee specification was chosen.

There were many possible devices that were capable of running the ZigBee specification. The Atmel ZigBit and the XBee PRO devices were looked into. Both devices had similar specifications [6]. The XBee PRO node was suggested by Professor G. Eatherley as he advised us that the device was a popular hobbyist device and had a

good knowledge base, making it suitable for the system.

Figure 5: Key Features of the XBee PRO Node[7]

The XBee PRO was a very strong candidate, as it met all the requirements of the wireless transmitters and essentially, the system. The device supported low power, mesh networking along with secure data transmission, employing DSSS (Direct Sequence Spread Spectrum) and AES-128 encryption. A summary of the key features of the XBee PRO device is shown in Figure 5: Key Features of the XBee PRO Node Summarized in the XBee Pro Manual and shows how their characteristic features met the requirements for the system.

Table 1: Key Differences Between the Series 2.5 XBee Node and the XBee PRO Node

Specification	**Series 2.5 XBee**	**XBee PRO**
Indoor/Urban Range	133 ft. (40 m)	300 ft. (100 m)
Outdoor RF line-of-sight Range	400 ft. (120 m)	1 mile (1.6 km)
Transmit Power Output	2mW in boost mode, 1.25mW in normal mode	63mW
Receiver Sensitivity	-96 dBm in boost mode, -95 dBm in normal mode	-102 dBM
Supply Voltage	2.1 – 3.6 V	3.0 – 3.4 V
Operating Current (Transmit, max output power)	@ 3.3 V, 40mA in boost mode, 35mA in normal	@ 3.3 V, 295mA
Operating Current (Receive)	@ 3.3 V, 40mA in boost mode, 38mA in normal	@ 3.3 V, 45mA

Upon further research, the Series 2.5 XBee device was selected to be the wireless transmitters for the system. The Series 2.5 XBee device has the same functionality as the XBee PRO but featured a smaller range and lower sensitivity due to having a smaller operating current. The XBee PRO device had greater range and better receiver sensitivity but had a significantly higher operating current, resulting in larger power consumption. Because the range and sensitivity of the device was proportional to the operating current of the devices, the Series 2.5 XBee device was chosen in favour of lower power consumption and lower overall cost. It should be noted that the Series 2.5 XBee devices are pin compatible with XBee PRO devices.

3.4 Serial Data Transmission (John)

Serial data communication was chosen over parallel data communications for transmissions between the Handy Board to a connected XBee node, as serial communication is a popular method for device communication. The Handy Board provides a Serial Peripheral Interface (SPI) and a Universal Asynchronous Receiver/Transmitter (UART) for serial communications and the XBee nodes only support serial over UART. The UART was chosen over the SPI, as both the Handy Board and the XBee nodes supported it.

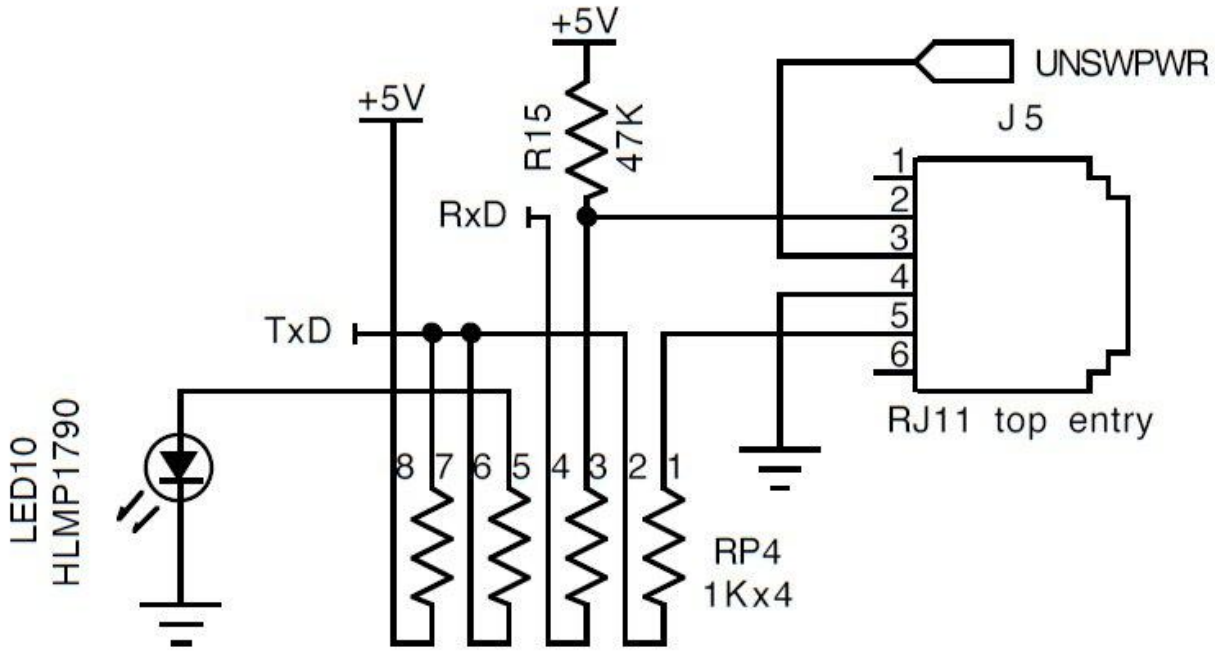

Figure 6: Serial Interface on the Handy Board Schematics[3]

The XBee nodes' UART pinout can be seen in Figure 11: XBee and XBee Pro Pinout Details.

Table 2: Handy Board RJ14 Pinout

RJ14 Pin Out Standard	Handy Board "RJ11" Pin Out	Handy Board Pinout
	1	N/A
1	2	RxD – Receive
2	3	UNSWPWR – Unswitched Power
3	4	GND – Ground
4	5	TxD – Transmit
	6	N/A

Interactive C does not support serial communication over the UART by user programs and does not provide library functions to set up or access the serial port since it is normally reserved for communication between the console and the board. In order to enable communication over the UART, research was performed to determine what must be modified.

To enable sending data over the UART serial port from a user program, it was determined that a bit must be changed at a certain address. In addition, while the serial port could be polled to receive data, an assembly interrupt service routine could be installed [8][9] to automatically buffer characters as they are received in the serial port buffer.

3.5 Vehicles (Peter)

The requirements for the vehicles to be used in this project were fairly simple. The frame of the vehicle must be large enough to accommodate the hardware platform and the wireless transmitter, and the motors must be powerful enough to drive the loaded vehicle.

It was determined that the simplest solution would be a ground-based vehicle. A flying vehicle would add numerous physical challenges that are outside the scope of this project, and a water vehicle would be impractical.

In choosing the steering type for a ground based vehicle, one turn-steering vehicle and one skid-steering vehicle were compared. It was noted that the rate of turning for skid-steering is dependent on the friction between each wheel and the ground, and that different amounts of friction would change the rate of rotation of each side of the vehicle causing undesired variation in turning. Due to the issues related to skid-steering, a turn steering vehicle was deemed to be more reliable.

To control the degree of turn for the vehicle, a servo was chosen rather than a simple DC motor. A DC motor can only provide full left or full right turn, whereas a servo can control the degree of rotation to a fairly precise amount.

3.6 User Interface (John)

The user interface required in the system was the controller used to send commands to the vehicle. The main requirement for the controller was for the controls to be as intuitive as possible. The controls must be simple to understand without any previous use while allowing full and complete control over the vehicle.

Multiple controller designs were looked into. One of the simplest designs was using a single joystick to determine input commands. This controller design was very simple and very easy to understand. However, having all possible inputs on a single joystick resulted in human errors. For example, forward and left would be input at the same time when only forward was intended by the user.

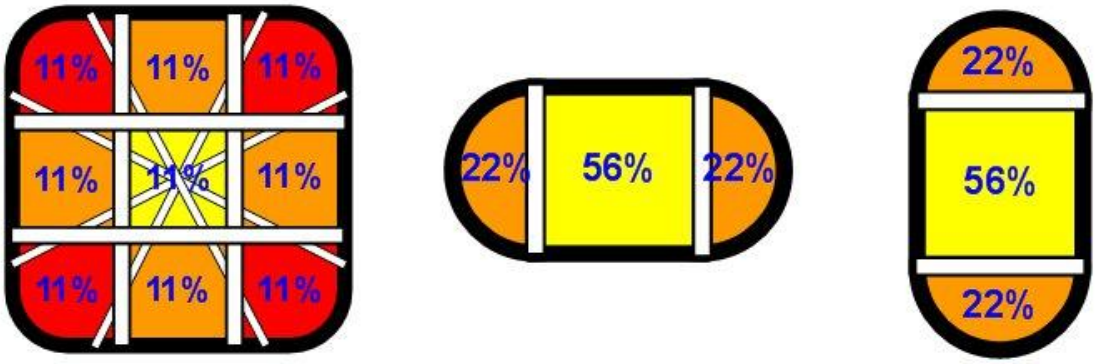

Figure 7: Joystick Engagement Possibilities of a One Joystick and a Two Joystick Controller Design [10]

In order to avoid this confusion while retaining simplicity, it was determined a controller design with two joystick inputs would be the best solution. One joystick was dedicated to sending either the forward or reverse command and the other joystick was dedicated to turn the car either left or right. This design was just as simple and intuitive as the one previous design while avoiding human input errors.

Since the vehicle's steering would be determined by a servo, a button was deemed necessary on the controller. This button would set the position of the servo so that the vehicle would drive straight, essentially "resetting" the steering of the vehicle.

3.7 Communication Protocol (Peter)

A communication protocol was designed for use between the controller and the remote controlled vehicle to provide a mechanism for reliable transmission and some small error detection.

The communication protocol was designed as a reliable transfer protocol, using one byte for all the data and control bits. The protocol is implemented as a negative-

acknowledgement (NAK)-less alternating bit protocol.

An alternating bit protocol uses one bit for sequence number, which alternates between 0 and 1 for each packet. This protocol has the disadvantage of only supporting one data packet in transit at a time, but reduces packet size due the single sequence bit.

A single byte was chosen since it is large enough to support the data that needs to be sent, reduces the processing time for the control board, and simplifies the interrupt routines and decoding process.

One bit for even parity was considered sufficient for the scope of the projects, as there was an assumption that the wireless modules forming the network would have some method for error checking and correction themselves. The parity bit was added to detect an odd number of bit errors between the receiving wireless module and the serial port on the Handy Board.

The choice was made to use a reliable alternating bit protocol as it was reasoned that the vehicle should not receive command packets out of order, nor should it lose a packet entirely. If the system were being used to control a full sized vehicle, a lost or out of order packet could have severe consequences. The protocol meets the above requirements by using ACK packets and a sequence number to notify the controller of successful, corrupted, and out of order packets so the correct packet can be retransmitted.

3.7.1 Packet Structure (Peter)

The structure of the packet is arranged as follows. One bit is used as a parity bit, one bit for the sequence number, one bit to denote an acknowledgement, and five bits for data, for a total of 8 bits, or one byte.

Table 3: Packet Bitfield Description

Parity	ACK	Seq	Data4	Data3	Data2	Data1	Data0

Table 4: Data Packet Bitfield

0/1	0	0/1	0/1	0/1	0/1	0/1	0/1

Table 5: Acknowledgement Packet Bitfield

0/1	1	0/1	0	0	0	0	0

3.7.2 Protocol Operation (Peter)

The protocol functions as follows (See Figure 8: Protocol State Diagram, Sender, and Figure 9: Protocol State Diagram, Receiver). The controller sends a packet containing the desired commands. When the receiver receives the packet, it first checks the parity bit and sequence number bit. If the parity bit indicates an error, or the sequence number is not the next expected sequence number, the receiver responds with an ACK addressed to the controller, with the sequence bit set to the last successfully received packet, or 0 if the vehicle has not yet received a packet. If the sequence bit is expected and the parity bit does not indicate an error, then the receiver decodes the data bits and performs the operations specified by the commands.

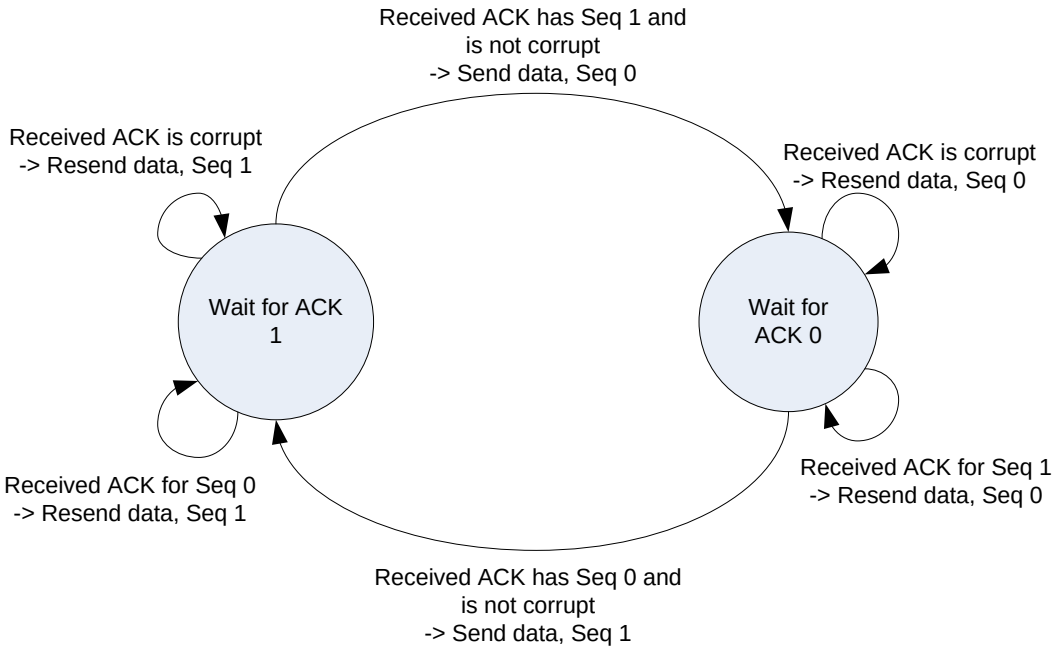

Figure 8: Protocol State Diagram, Sender

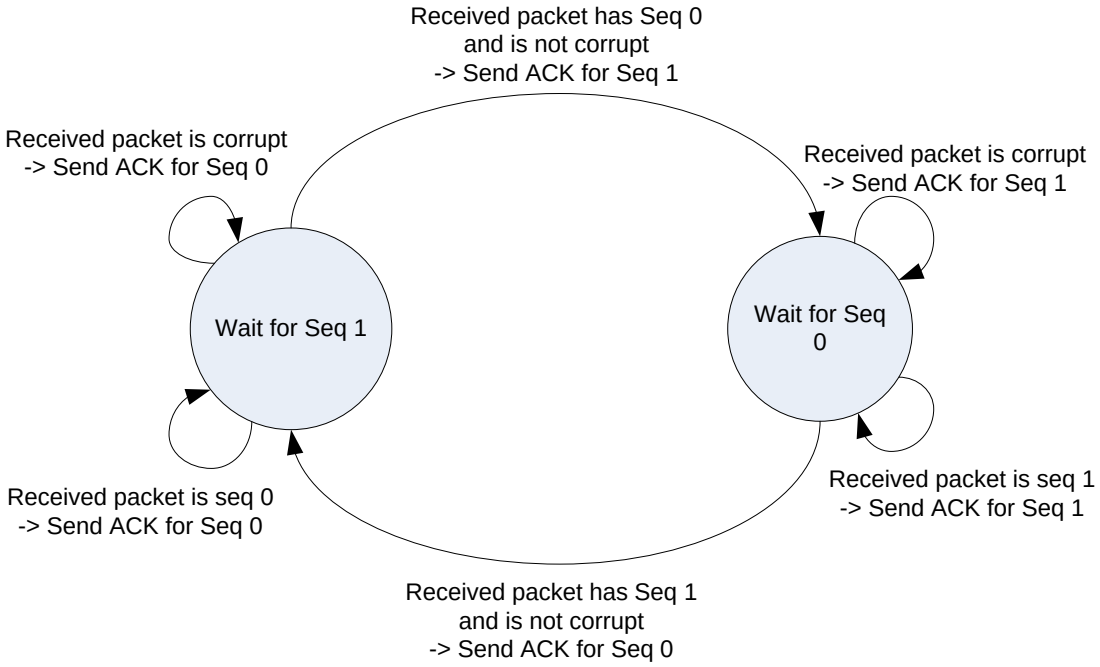

Figure 9: Protocol State Diagram, Receiver

When the controller receives a packet, it first checks the parity and ACK bit. If the parity bit indicates an error, or the ACK bit is not set (ie. a data packet), then the controller ignores the packet. If the parity bit does not indicate an error and the ACK bit is set, the controller compares the packet's sequence bit to the last sent packet. If the two sequence bits differ, the controller re-sends its last sent packet. If the two sequence bits are equal, then the controller sends the next packet.

3.8 XBee Interfacing and Software Overview (Andrew)

3.8.1 XBee Interface

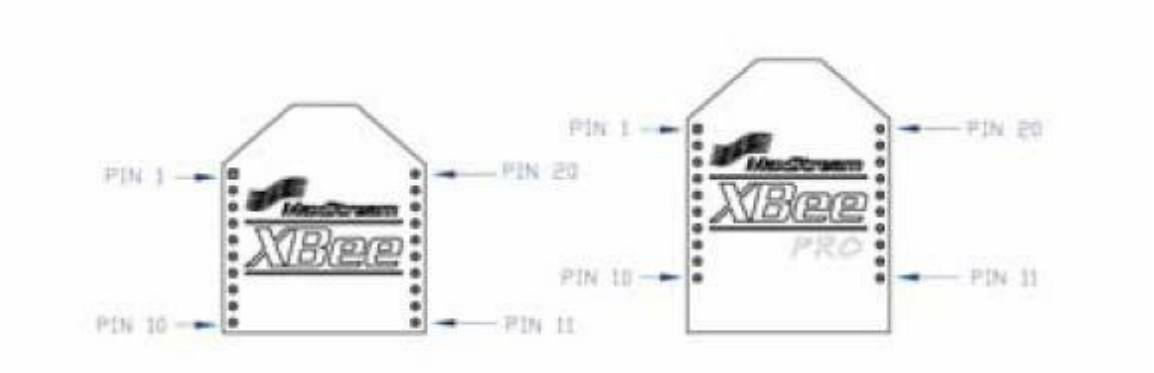

Figure 10: XBee and XBee Pro Pinout [7]

Pin #	Name
1	VCC
2	DOUT
3	DIN / **CONFIG**
4	DIO12
5	**RESET**
6	RSSI PWM / DIO10
7	DIO11
8	[reserved]
9	DTR / SLEEP_RQ/ DIO8
10	GND

Pin #	Name
11	DIO4
12	**CTS** / DIO7
13	ON / **SLEEP**
14	VREF
15	Associate / DIO5
16	**RTS** / DIO6
17	AD3 / DIO3
18	AD2 / DIO2
19	AD1 / DIO1
20	AD0 / DIO0 / Commissioning Button

Figure 11: XBee and XBee Pro Pinout Details [7]

The XBee and XBee Pro are pin compatible and their pinout details can be seen in the figures above. The XBee communicates with The Handy Board over UART through pins 2 and 3, DOUT and DIN. The XBees require a voltage between 2.1V to 3.6V to be supplied to pin 1 to operate. The Handy Board supplies only 5V or 9V as outputs, so a voltage divider circuit or a voltage regulator is required to bring down the voltage to the required levels for operation.

During research, it was discovered that, Adafruit Industries provides a pre etched breakout board as a kit specifically designed for the XBee. The boards come equipped with a level shifter to convert RS-232 level signals to TTL levels, a 3.3V regulator, allowing us to use the 5V supply voltage from The Handy Board to power the XBee, and other support electronics such as signal strength and data received LEDs. The breakout board also exposes the most useful pins with standard pin spacing; the XBee uses much smaller pin spacing, preventing it from being placed directly into the breadboards found in the lab. The pin arrangement also allows an FTDI USB to serial cable to be connected directly, allowing us to modify the XBee's firmware via USB on the lab computers. A picture of the XBee inserted into an Adafruit breakout board can be seen below.

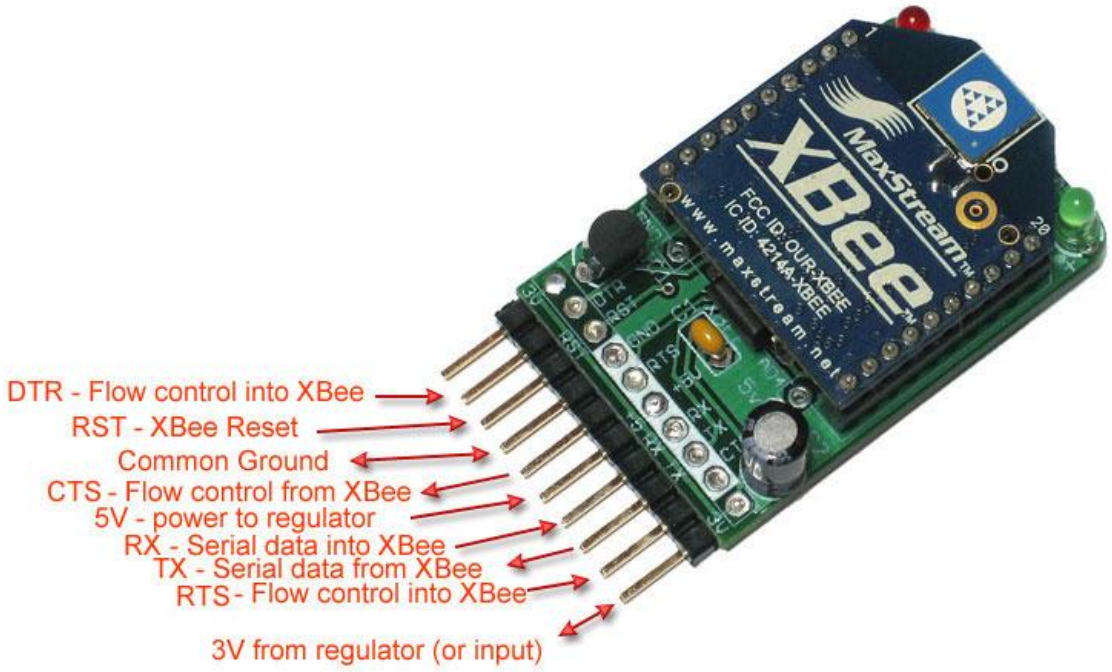

Figure 12: Adafruit Breakout Board Pinout with XBee [11]

3.8.2 X-CTU Software

X-CTU is a piece of software that Digi created for use with the XBee. It allows one to update the XBee's firmware as well as communicate with the XBee connected to the computer via the terminal.

The terminal allows the user to interact with the XBee node using AT commands. By typing "+++" in quick succession, the XBee enters AT mode and will remain in this mode until it detects no activity for 10 seconds. In this mode, the user can send the XBee commands. All commands must include the prefix "AT" before the command. For example, to see a list of all nodes on the network, one would type "ATND" and hit enter. The table below lists of some commands that can be sent to the XBee.

Table 6: XBee AT Commands Sample

ND	Usage: ATND Displays all the nodes on the network. Shows some parameters such as the nodes network address, serial number, and node identifier
DN	Usage: ATDN <address/node identifier> Sets the destination node to send packets to.
RO	Usage: ATRO <int value> Sets the packetization timeout. Setting to 0 causes the XBee to send out data as it arrives and not buffer it into a single packet.
NI	Usage: ATNI <name> Sets node identifier for this node. The node identifier is a label that can be used to address this node directly instead of using its network address or serial number.
WR	Usage: ATWR Writes changed settings to firmware.

3.9 ZigBee Mesh Network (Andrew)

The ZigBee mesh network consists of three types of nodes, the Coordinator, the Router, and the End Device.

The Coordinator is in charge of creating the network and addressing all the other nodes. All nodes are given a 16 bit network address by the Coordinator when they join the network. The Coordinator's address is always 0. Nodes may also be addressed by their 64-bit serial number which may be useful in addressing a specific node whose network address is not known at runtime.

Routers expand the range of the mesh network. Messages sent to nodes across the

network will be forwarded by the Routers using the least cost path as computed by the nodes. Routers are also capable of adding new nodes to the network that are out of range of the Coordinator.

End Devices have the ability to send or receive messages on the network. They may not forward messages, nor can they accept new nodes into the network. Of the three types of nodes, End Devices are the only nodes which may go into sleep cycles to conserve power.

To send messages across the network, the destination node variable must be set to the address of the desired recipient. This can be the node's 16 bit network address, or the node's 64 bit serial number. By setting the address to FFFF, the message will be broadcast to the entire network. A simple way to set the network address of the destination node is by using the node identifier. The node identifier is a label that is stored with the routing table and can be used in place of an address for the destination node. When the node identifier is used in place of an address, the Coordinator looks the label up in its routing table and sets the destination node as the 64 bit serial number corresponding to that label.

For this particular application, the packetization timeout, RO, is set to 0. This variable determines how long it should wait and buffer characters before sending them out. By setting it to 0, it acts in a transparent mode in which is forwards every packet as it arrives without buffering. This setting is actually hidden in X-CTU and can only be changed by sending AT commands through the terminal to the XBee module.

4.0 Implementation

4.1 Car Drive System (Peter)

The vehicle drive system takes the packets obtained through the network from the controller, decodes them, and drives the car forward, reverse, left, or right, in any logical combination.

The vehicle drive system uses an interrupt service routine [7] to handle incoming packets from the serial port. When a full byte is received in the serial port data buffer, the interrupt code executes and transfers the byte into a 4 byte buffer in memory.

The vehicle control software is written in Interactive C. The control program consists of two motor processes which drive the forward/reverse and the left/right motors, and a main process which polls the buffer for packets, does error checking on the packets, and sends the appropriate commands to each motor process.

The main process polls the buffer in an infinite loop. If there is no data in the buffer, the main process defers its processing time to other processes. When a packet arrives in the buffer, the program checks to ensure that no conflicting commands (See Table 8: Vehicle Control System Test Results) have been sent. Conflicting commands, such as forwards and reverse at the same time, are ignored and neither direction is activated. If a command is received to drive a motor, the drive software generates a pulse-width modulated signal to drive the motor for 50 ms.

An interrupt-driven approach was taken in the design of the vehicle drive software to ensure that no packets would be lost. The Handy Board does not provide any timing guarantees for each process, so polling the serial buffer directly was deemed too unreliable.

4.1.1 Deviation from Original Design (Peter)

The original design required the vehicle drive system and the vehicle control system to use a NAK-less alternating bit protocol to ensure reliable transmission of packets. The implementation would have had the main loop checking the sequence number and the parity bit to ensure that the received packet was expected and not corrupted during transmission.

The deviation occurred because it was decided that the reliable transport provided by the mesh network was sufficient for the purposes of the project. It was also observed during experimentation that the intended protocol did not solve any of the issues discovered with using the XBees to form a mesh network.

The designed and implemented protocols are very similar, with the non-reliable protocol simply ignoring the three bits that the alternating bit protocol used for parity, ACK/NAK, and sequence number.

In addition, the vehicle's turning was implemented using a simple dc motor rather than the servo in the design. This change was made as it was noted that the vehicle's poor quality tires and gearing would not be precise enough for the servo's precision to be fully utilized. The final implementation can be easily modified to use a servo, as all the required OpCode bits remain in the implemented protocol.

4.2 Vehicle Control System (John)

The vehicle control system takes the input commands given by the user through the joysticks and encodes the instructions into a packet. The possible instructions for the car are forward, reverse, left, right or any logical combination of commands.

Table 7: List of Possible Vehicle Instructions and Corresponding OpCodes

Instruction	**OpCode**
Forward	xxx0 1000
Reverse	xxx0 0100
Left	xxx0 0010
Right	xxx0 0001
Centre	xxx1 0000
Do Nothing/Idle	xxx0 0000

The joysticks are connected to switches, each dedicated to its own digital port on the Handy Board. The Handy Board polls the digital ports to gather which microswitches have been hit. The control system gathers the input, determines if the input is logical and encodes the instructions into a packet. Once the packet is verified for non-conflicting instructions, the packet is sent into the network for the vehicle drive system to receive and decode. If there is no input, the controller will not send any packets until a new input occurs.

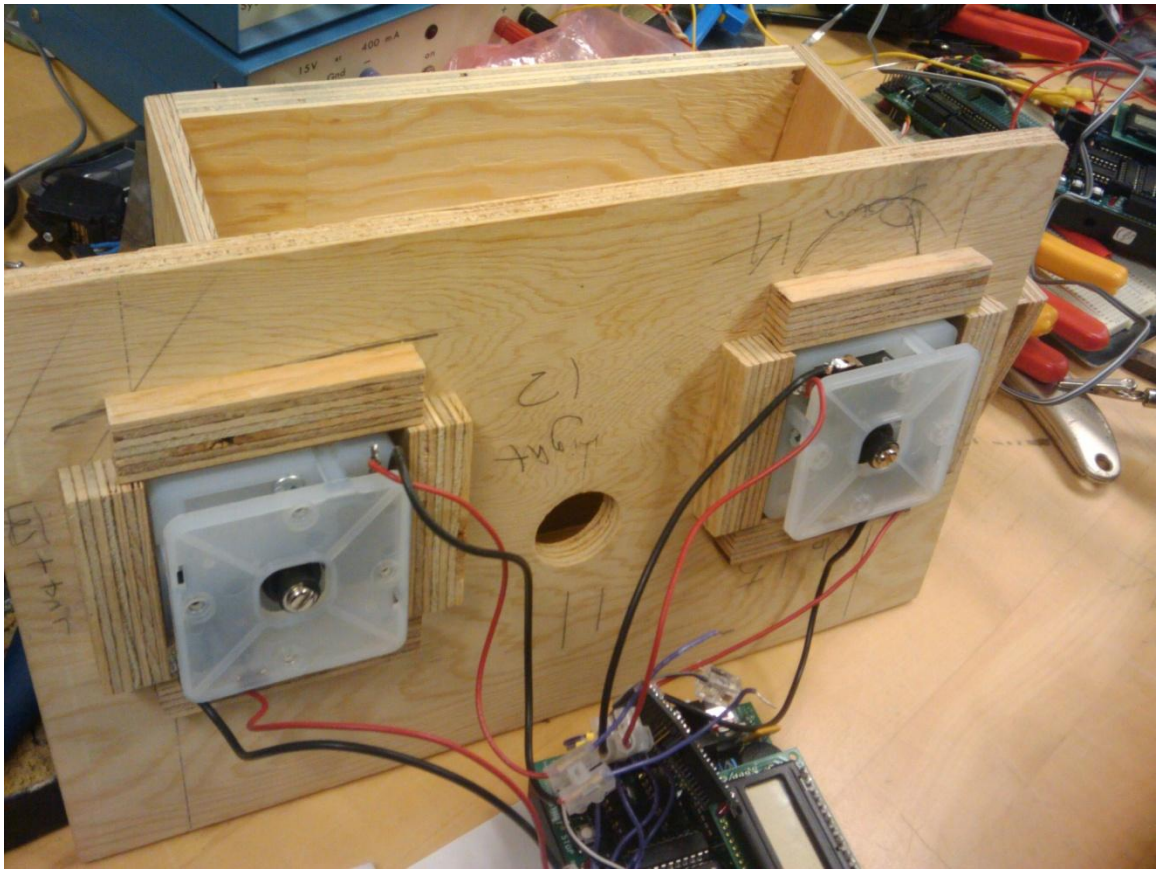

Figure 13: Joystick Microswitches Connected to the Handy Board

The vehicle control system was designed with intuitive controls in mind. By replacing the steering servo with a motor, the button to set the car in a straight driving position was removed. However, the centre OpCode has been left in the code if the RC car reverts back to a servo steering method. The controller is identical to most turn steering RC car controllers.

Using an interrupt service routine for the car was deemed unnecessary. Since the vehicle control code was done in Interactive C, polling for inputs was a simple and effective solution within the available resources of the Handy Board. The Handy Board would have not been able to run an interrupt based control scheme while running

Interactive C code.

4.3 Wireless Mesh Communication System (Andrew)

The XBee nodes were set up with Digi's X-CTU software. The node connected to the controller was assigned the Coordinator Role in the firmware. This node creates the mesh network and is in charge of assigning addresses to all the other nodes which attempt to join the network. It receives commands from the Handy Board over the serial port and attempts to forward them to the car. The node connected to the car was given the End Device role. It receives commands from the Coordinator and forwards them over the serial port to the Handy Board to control the car. The remainder of the nodes were assigned the Router role. These nodes serve as a means of expanding the network, and forward packets sent from the Coordinator to the End Device when it is not in range. They may also accept nodes onto the network when they are not in range of the Coordinator.

Figure 14: X-CTU Firmware Settings for Coordinator shows the X-CTU firmware settings for the Coordinator. Some notable variations from default settings are the Scan Channels, ZigBee Stack Profile, Destination Address, and Node Identifier variables. The Scan Channels setting is a bit field which controls which channel the network will operate on, ranging from 11-26. We have it set to F000 so the XBee only scans the upper 4 channels as we found there were fewer devices operating in these channels. The ZigBee Stack Profile is set to 2 in order to use the ZigBee-PRO spec over the older ZigBee2006 spec. The Destination Address, which is set to 0 in the image, is set to the 64-bit serial address of the car. For the End Device, it remains 0 as that is the address the Coordinator assigns to itself.

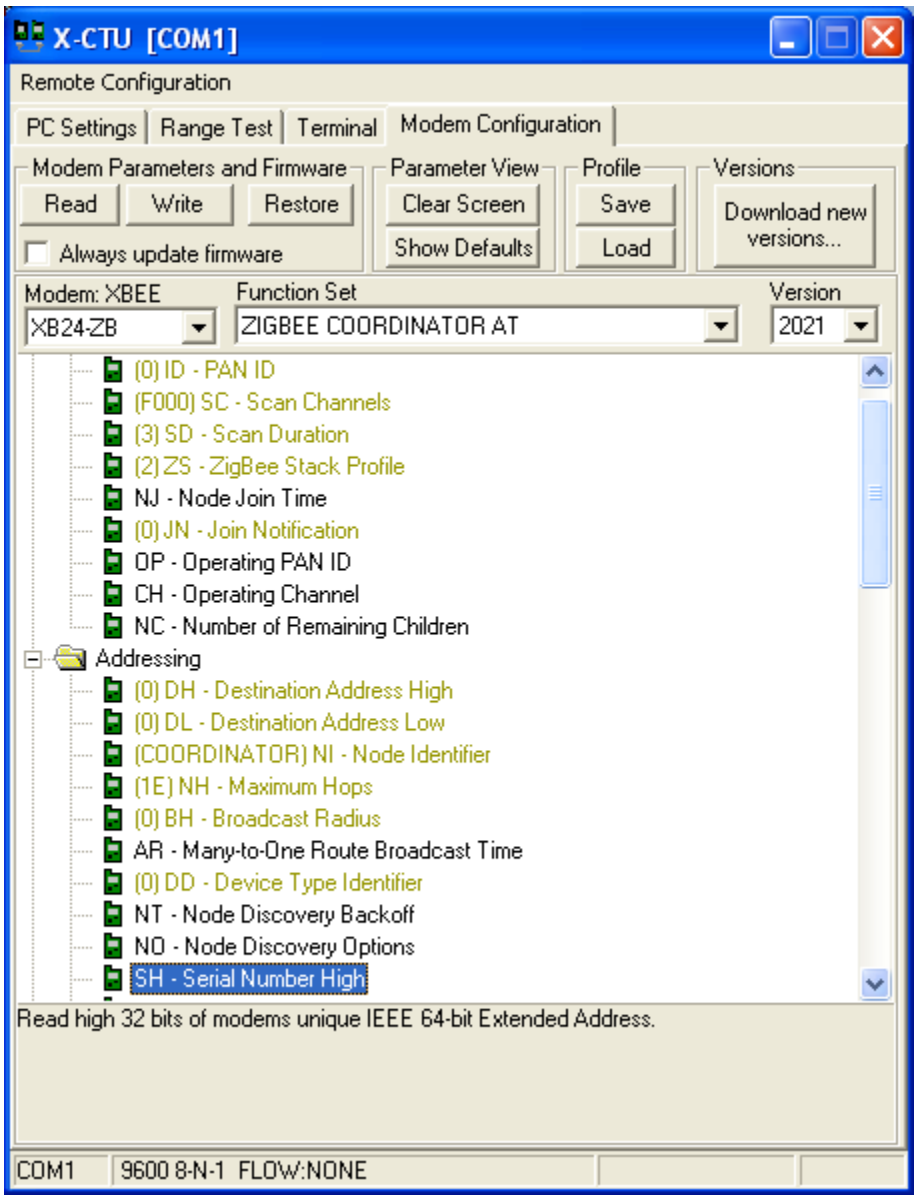

Figure 14: X-CTU Firmware Settings for Coordinator

5.0 Testing and Integration

5.1 Vehicle Drive System (Peter)

The vehicle drive system was tested separately from the full system using a separate test program residing on a separate Handy Board. The test program cycles through the possible operational codes (OpCodes), sending them one by one to the serial port.

The two Handy Boards are connected physically using a serial crossover cable to ensure there is no packet loss or delays due to packetization or wireless transmission.

The vehicle drive system displays the OpCode it receives on the LCD display, and the tester observes the actions the vehicle takes when it receives an OpCode to ensure that the command is received and interpreted properly.

5.2 Vehicle Control System (John)

The vehicle control system was initially simulated to ensure proper input recognition and packetization. The vehicle control system was tested with all possible and impossible combinations of instructions. Impossible combinations were tested during the simulation since testing the impossible combinations of instructions was only possible during the simulation since conflicting instructions were physically impossible to input. Case in point, it is physically impossible to input forward and reverse at the same time with one joystick.

Figure 15: Implemented Controller

Table 8: Vehicle Control System Test Results

Input Test Case	Interpreted OpCode	Resulted OpCode	Resulted Instruction
Possible Test Cases			
Forward	xxx0 1000	xxx0 1000	Forward
Reverse	xxx0 0100	xxx0 0100	Reverse
Left	xxx0 0010	xxx0 0010	Left
Right	xxx0 0001	xxx0 0001	Right
Centre	xxx1 0000	xxx1 0000	Centre
Forward + Left	xxx0 1010	xxx0 1010	Forward + Left
Forward + Right	xxx0 1001	xxx0 1001	Forward + Right
Reverse + Left	xxx0 0110	xxx0 0110	Reverse + Left
Reverse + Right	xxx0 0101	xxx0 0101	Reverse + Right
Impossible Test Cases			
Forward + Reverse	xxx0 1100	xxx0 0000	Do Nothing
Left + Right	xxx0 0011	xxx0 0000	Do Nothing

During a physical test, the vehicle control system displayed the OpCode the system was transmitting to the mesh network. The system was tested with all possible combinations of inputs on the physical test. No conflicting command packets were ever formed and the physical test results were identical to the results found in the simulated test.

This method of testing was aimed to eliminate the vehicle control system of possible errors during system integration. Even in the case of errors or conflicting commands, the controller interpreted the impossible test cases as “Do Nothing” instructions, suppressing any possible errors and eliminating the vehicle control system of any possible errors.

5.3 Wireless Mesh Communication System (Andrew)

The wireless mesh communication system was tested between multiple XBee nodes. A test program was written in Interactive C on the Handy Board for the end device to display all the codes that the connected XBee nodes were receiving from a transmitting node during a wireless mesh networking test.

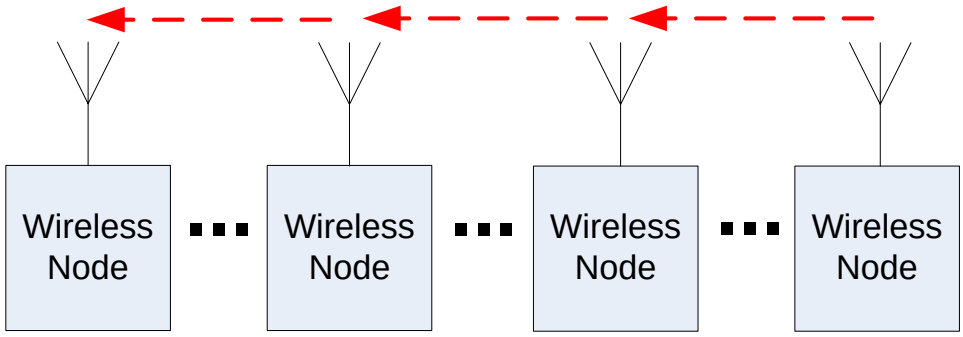

Figure 16: Wireless Mesh Communication Testing System; Before Node Removal

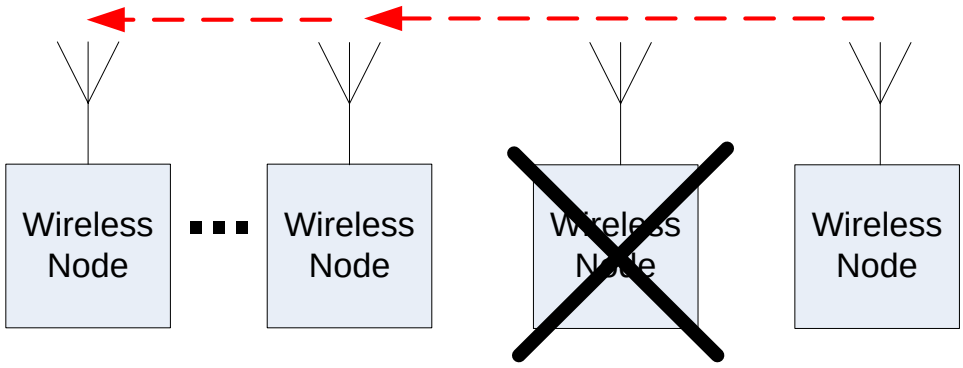

Figure 17: Wireless Mesh Communication Testing System; After Node Removal

Once the designated end device received the test packet sent out by the transmitting test node, one of the nodes used to transmit the data to the end device was turned off. The end device was observed to see if it would re-establish a connection using

other nodes in the wireless mesh network.

5.4 End to End Testing (Peter)

To ensure the system functions properly as a whole, end to end testing is required. The end to end test scheme is simple; the tester creates the wireless mesh network, connects the vehicle drive system and vehicle control system to the network, then attempts to drive the vehicle using the physical controller.

The tester then observes the behaviour the vehicle and determines whether it meets the tester’s requirements.

6.0 Conclusion (John)

6.1 Achievements

The ultimate objective of the ZigBee Mesh Network project was to drive an RC car by sending commands though the ZigBee wireless mesh network from a controller. In order to meet this objective, the goals laid out in our project proposal were met. In order, these following achievements have been met.

1. The Series 2.5 XBee node, an IEEE 802.15.4 capable device, was selected and designed in order to be used in a mesh network.
2. With the combination of the Handy Board and the XBee node, an RC car was designed to receive commands and drive over the mesh network.
3. With the combination of the Handy Board and the XBee node, a controller was designed to send commands across the mesh network and to the RC car.
4. Through the tests done on the mesh network, a greater range was demonstrated.
5. Through the tests done on the mesh network, the resilience and the self-healing ability of a mesh network was demonstrated.

With these accomplishments achieved, the ZigBee Mesh Network system was able to drive an RC car over a mesh network.

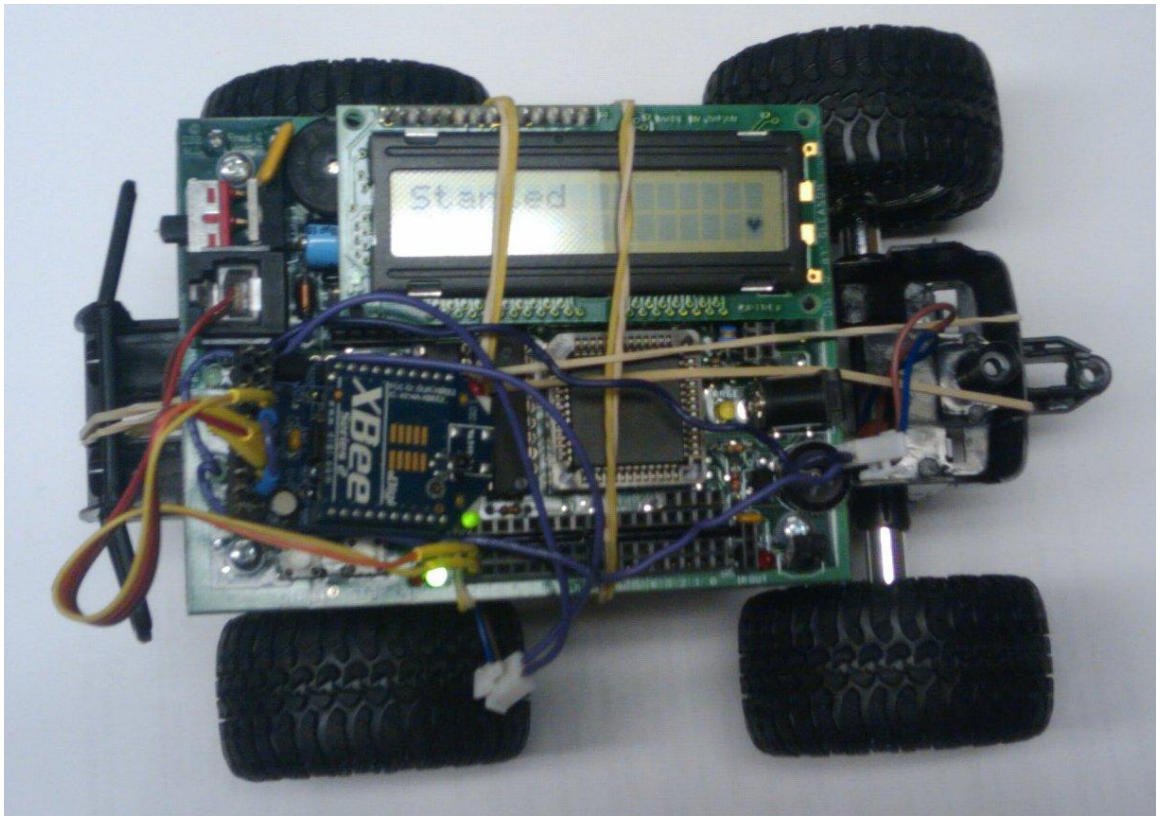

Figure 18: Implemented RC Car

6.2 Issues and Improvements

6.2.1 Hanging Instruction Packets

While the ZigBee Mesh Network project met the accomplishments set out in the initial proposal for the project, unexpected issues were discovered while driving the vehicle drive system. The vehicle would occasionally appear to ignore commands and continue executing the last command received for an extended period of time.

While the exact cause of the issue is currently unknown, the source of was narrowed down to the XBee node. This was determined by connecting the vehicle control system and the vehicle drive system directly using a serial crossover cable.

The RC car would "hang" on approximately every three or four instruction packets. Since we ran into the issue of hanging instructions, an obvious future goal would be to shorten or eliminate the duration of these "hangs".

6.2.2 Remote Control Vehicle

The RC car used in the project was generally of poor quality. The RC car was sufficient for the purposes of the project, as it was able to move forward and reverse and steer left and right, however, the cheap plastic tires and the overall low weight of the RC car did not allow for precise control over the steering of the RC car. Moving to a well-constructed physical platform would create opportunities for more precise controls and additional sensors and modules.

6.2.3 Handy Board to Arduino

Another goal would be to use a wireless mesh network on a device besides the Handy Board. The Handy Board allowed us to prove that wireless communication through a mesh network between embedded devices is possible, as long as the devices are capable of serial communications. The Handy Board is an older control board, moving to the Arduino platform could be a good future goal, as the Arduino is a modern development board with a much more active development community. There is also an add-on module, the XBee shield, allowing simple integration and control over an XBee node without additional hardware or programming.

6.2.4 Peripherals

Adding peripherals to the RC car that communicate over the ZigBee Mesh Network system would be another good goal for future work. Having additional

peripherals would enhance the project overall greatly. For example, having a camera attached to the RC car would allow the communication and control over the car as long as the RC car is within the mesh network. If a video stream is too much for the mesh network to handle or in an area with poor visibility, sonar could be attached instead. Sonar imaging would allow the RC car to report back an approximate map of a specific area.

6.3 Conclusion

While there were many obstacles and difficulties, the overall project was a success, as we met our goals in terms of our proposal. We were able to communicate and control the car over a wireless mesh network. However, there were some technical difficulties within our project.

The components behind our project worked flawlessly. The vehicle drive system, the vehicle control system and the ZigBee mesh network were all functioning as planned before integration. Once we integrated the components together, we realized there was a slight problem communicating between the controller and the car. While the car was able to receive the encoded instructions from the controller, there would be a considerable hang every couple of instructions. We could not measure the exact occurrence of the hang since we were unable to reproduce the problem consistently. We suspect that XBee nodes are the source behind these hangs.

While testing mesh networking simply with the nodes, we found a similar hang. We initially attributed this problem to latency between nodes. However, after further testing after integration of components, we suspect that the XBee nodes are clearing a cache or buffer before receiving new instructions, hanging on an instruction until it is

confirmed. The car would receive instructions input before, during and after the hang, suggesting the behaviour mentioned. Another possibility is the reliable transmission of the XBee specification in action. The vehicle drive system was designed so that the RC car would stop after consuming a packet. During these delays, the car would be frozen with the latest input it received before the freeze. The XBee node may actually be constantly transmitting the same data in order to ensure reliable transmission.

During tests, we connected the controller and car Handy Boards through a wire to test the protocol. We noticed that the physically touching the wire would sometimes produce garbage packets for the car to read. The car would not decode the instruction since the drive system saw that it was an invalid instruction packet. However, the response between the controller and the RC car was consistent, providing no interference. The Handy Board could not be a cause for the delay between packets.

In conclusion, through our project, we discovered that while a wireless mesh network is a great method of low-power wireless communication over long distances, it may not be suitable for a constant stream of data over the network. The hang between the controller and car over the network was restricting control of the car, making seamless car control impossible.

7.0 References

[1] antenna.thumbnail.jpg, [Online]. http://blackberrycool.com/wp-content/uploads/cell-antenna.thumbnail.jpg. [Accessed Apr. 6, 2010].

[2] "Arduino – Arduino Board Duemilanove," Feb. 5, 2010. [Online]. Available: http://arduino.cc/en/Main/ArduinoBoardDuemilanove. [Accessed Apr. 6, 2010].

[3] F. G. Martin, *The Handy Board Technical Reference,* Fred G. Martin, 2000.

[4] F.G. Martin, "Expansion Board – The Handy Board," June 2009. [Online]. Available: http://handyboard.com/hb/hardware/expansion-board/. [Accessed Apr. 6, 2010].

[5] "WirelessHART Technical Data Sheet," May 15, 2007. [Online]. Available: http://www.cds.caltech.edu/~shiling/wirelesshart_datasheet.pdf. [Accessed Apr. 6, 2010].

[6] "ZigBit 2.4 GHz Wireless Modules," Jun. 2009. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc8226.pdf. [Accessed Apr. 6, 2010].

[7] "XBee ZNet 2.5/XBee-PRO ZNet 2.5 OEM RB Modules," Feb. 11, 2008. [Online]. Available: http://ftp1.digi.com/support/documentation/90000866_C.pdf. [Accessed Apr. 6, 2010]

[8] "Creating .ICB Files From .ASM (Assembly) Files," Mar. 26. 2008. [Online]. Available: http://xperts-wiki.sce.carleton.ca/uploads/SmartRollator/Compiling%20ASM%20to%20ICB%20Procedure.pdf. [Accessed Apr. 6, 2010].

[9] "Interrupt Handler for Serial Communication," Jul. 7, 1998. [Online]. Available: http://cpansearch.perl.org/src/DCOPPIT/grepmail-5.3034/t/results/not_body_handy. [Accessed Apr. 6, 2010].

[10] "Joystick Controller – Joystick Engagements and Restrictors," Feb. 25, 2009. [Online]. Available: http://www.slagcoin.com/joystick/restrictors.html. [Accessed Apr. 6, 2010].

[11] "XBee Adapter – Simple Wireless Communication," Mar. 15, 2010. [Online]. Available: http://www.ladyada.net/make/xbee/. [Accessed Apr. 6, 2010].

Appendix A: Reports

- Report: Remote Control Car Teardown
- Report: Handyboard Serial Port

Report: Remote Control Car Teardown

Wireless Mesh Network
Peter Fyon 100652096
peter.fyon@gmail.com

1.0 Introduction

The purpose of this paper is to report on our findings from tearing down one skid-steer (SS) and one turn-steer (TS) remote control car. The report will cover the method used in taking apart the vehicles, the hardware at a general level, and the important lessons learned from the process.

2.0 Method

The deconstruction of each vehicle was simple. The plastic casing was removed with a Phillips head screwdriver to expose the control board inside. Tests were performed on the motors by connecting one to a motor port on the handyboard and using a program, written in Interactive C, to vary the rotational speed (using pulse width modulation) of the motor in both directions. It was noted that the LEDs on the board did not dim visibly with motors from either car, and concluded that they were not drawing more current than the board could provide.

3.0 Hardware

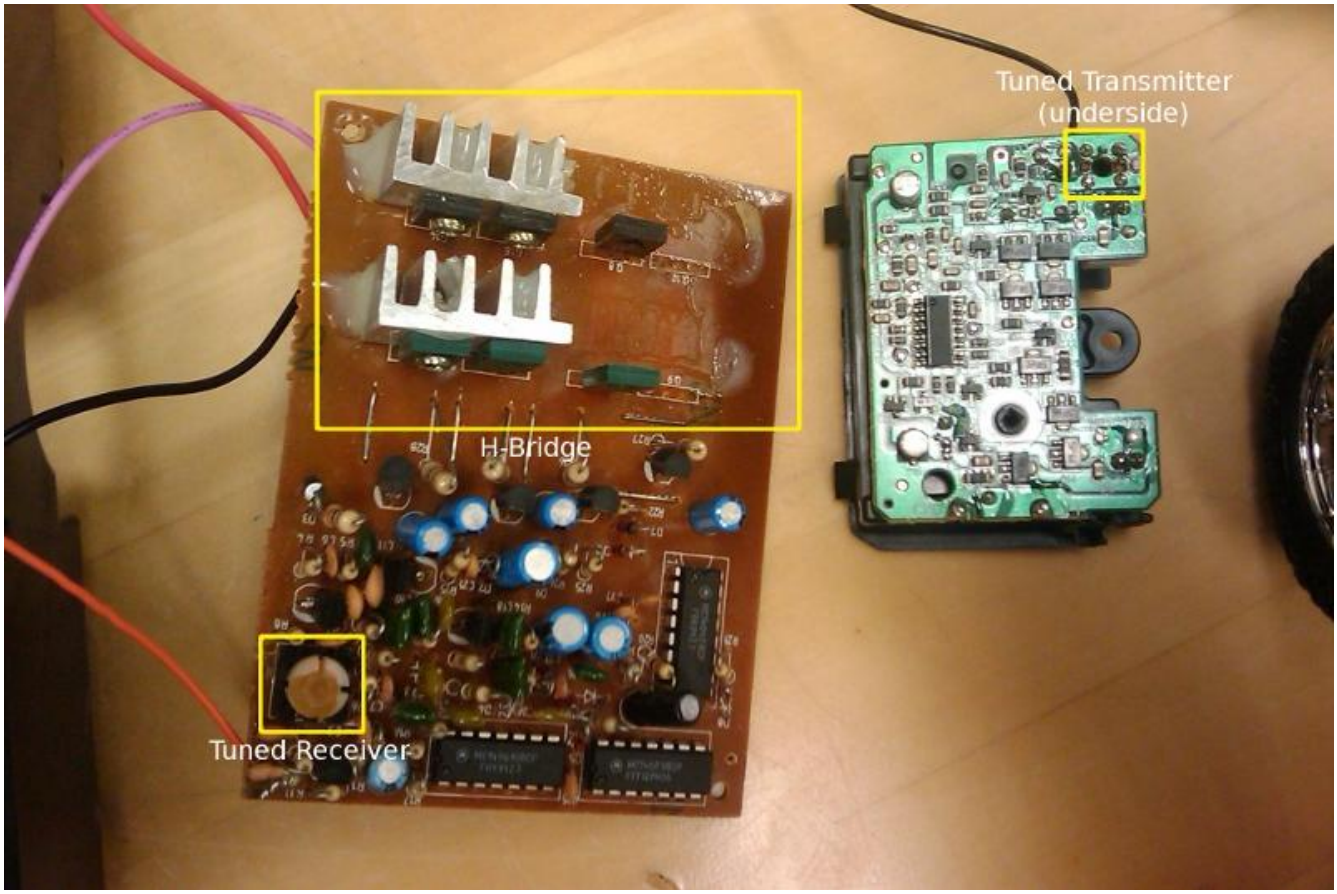

Figure 1: Car Driver Boards. Skid steering car on left, turn steering car on right. Note that two power transistors and two heatsinks were removed from the skid steering car board.

3.1 Skid Steer Car

- Uses a variety of analog components to filter the incoming signal before sending it to the control circuitry
- Uses two independently controllable motors on each side of the vehicle to control steering and forward/reverse at the same time
- Motors are supplied approximately 12 volts from eight AA cell batteries.
- Each motor is driven forwards or backwards with their own H bridge made up of four heat-sunk power transistors.
- Turning while moving requires fine control of the rotation speed of the motors
- The car is capable of turning 360˚ while remaining in place

3.2 Turn Steer Car

- Uses much fewer analog components, and more integrated circuits than the SS car.
- Uses one motor to drive the rear wheels and one motor to turn the front wheels left or right.
- Motors are supplied approximately 3 volts from two AA cell batteries.
- The default position for the front wheels is straight.
- Turning the front wheels is limited to all left or all right.

Both of the remote control cars use a radio tuned to the same frequency as the transmitter in the controller. Also, both vehicles have a small capacitor shorting the motor's terminals, presumably to filter out high-frequency noise caused by the motors.

4.0 Lessons

We learned the following things during the deconstruction of this vehicle:

1. SS vehicles are less precise in their steering than TS vehicles as the rate of turn is related to the friction between each drive wheel and the ground. The friction may not be the same for each wheel due to different ground materials, differing wear on each wheel, etc.

2. The motors are simple to control with pulse width modulation (PWM), and can be driven directly from the Handyboard without additional circuitry.

3. The turn angle for our TS vehicle can be made more precise by replacing the motor responsible for turning with a servo and some modifications with the gearing ratio between the servo and the axle.

4. Both controllers used simple microswitches to detect forward/reverse and left/right directives from the controller.

These lessons may influence the design for our controller and steering.

Report: Handyboard Serial Port

Wireless Mesh Network
John Koh 100684909
johnnykoh.is@gmail.com

1.0 Introduction

The purpose of this paper is to report on our findings on the serial port and communication method used on the Handyboards. This report will cover the hardware functionality of the serial port on the Handyboard.

2.0 Analysis

In order to communicate with the Handyboard, the compiler must connect through the serial interface. By analyzing the hardware in between the serial interface and the Handyboard, the Handyboard can be configured to work with the XBee node.

3.0 Technical Specifications

See the attached diagram on the CPU and Memory Circuit Schematic.(page 37 in manual)

The Handyboard only has one pair of receive (RxD) and transmit (TxD) but uses an RJ14 plug, which normally houses two pairs. The following table is how the RJ14 plug is laid out on the Handyboard.

Position	**RJ14 Pin**	**Pair**	**Tip/Ring**	+/-	**Colours (wire, Handyboard-side)**
1					
2	1	2	Tip	+	Black
3	2	1	Ring	-	Red
4	3	1	Tip	+	Green
5	4	2	Ring	-	Yellow
6					

Appendix B: Initial Designs

- Handy Board Communication Protocol Version 1

Report: Handyboard Communication Protocol

Wireless Mesh Network

Peter Fyon 100652096

peter.fyon@gmail.com

John Koh 100684909

johnnykoh.is@gmail.com

1.0 Introduction

The purpose of this paper is to define a standard protocol for communication between the controller handyboard and the car drive handyboard. The document will cover the functional requirements and technical implementation details.

2.0 Requirements

The following requirements were defined during the requirements elicitation process:

1. The controller will not send conflicting signals to the vehicle (Left and right, or forward and backwards).
2. The controller will be able to send both forward or backwards, and directional commands at the same time.
3. The controller will be able to centre the directional wheels.
4. If there is no command to send, the controller will send an opcode representing "do nothing".
5. There will not be a significantly noticeable delay between when an input is supplied to the controller and when the corresponding action is taken by the vehicle.
6. The controller will send a packet approximately every 50 milliseconds.[1]
7. The vehicle will consume a packet approximately every 50 milliseconds.

3.0 Technical Specifications

The packet structure will be arranged as follows:

- Single byte packet containing opcode
    - Low five bits for forwards/backwards, left/right, and centre commands.
    - High 3 bits reserved for future use.

	Reserved	**Reserved**	**Reserved**	**Centre**	**Forward**	**Backward**	**Left**	**Right**	
Centre Wheels:									
	X	**X**	**X**	**1**	**X**	**X**	**0**	**0**	
Do-Nothing									
	X	**X**	**X**	**0**	**0**	**0**	**0**	**0**	
Forward:									
	X	**X**	**X**	**X**	**1**	**0**	**X**	**X**	
Reverse:									
	X	**X**	**X**	**X**	**0**	**1**	**X**	**X**	
Left:									
	X	**X**	**X**	**0**	**X**	**X**	**1**	**0**	
Right:									
	X	**X**	**X**	**0**	**X**	**X**	**0**	**1**	

Note that while the controller may send multiple commands in the same packet, it is the responsibility of the controller to ensure that each command does not conflict with any others.

3.1 Encoding

Encoding is done on the controller. Every directional switch will add their corresponding instruction to the instruction packet. After collecting inputs, the controller will send the instruction packet to the R/C car.

3.2 Decoding

Decoding is done on the R/C car. The R/C car will decode the instructions by performing a bit-wise AND with the instruction packet and each possible instruction, except Do-Nothing. When a bit-wise AND returns true, the corresponding instruction will be performed. Once all the possible instructions are checked, the car will sleep.

[1]Through experimental testing, it was determined that an action taken by the vehicle every 50 ms (such as moving forward or backwards) was indistinguishable from continuous motion.

Appendix C: Code

- Vehicle Drive Software, Unreliable Protocol
- Vehicle Control Software, Unreliable Protocol
- Common Header
- Serial Interrupt Routine
- Serial Helper Functions
- OpCode Test Software
- Serial Test Software, Receiving
- Serial Test Software, Sending
- Vehicle Drive Software, Reliable Protocol
- Vehicle Control Software, Reliable Protocol
- Protocol Function Definitions

car_drive-noack.ic

```
/*
 * car_drive is a program which receives commands over the UART
 * to drive motors and (possibly in a later version) a servo.
 * It is intended to control a remote control car over a mesh
 * network provided by ZigBee capable chips that transmit com-
mands
 * from a controller to the vehicle running this program.
 *
 * @author Peter Fyon
 *
*/
#include sci_isr.c
#include header.ic
// Motor number definitions
#define MOTORDRIVE 0 // fwd/reverse drive motor
#define MOTORTURN 1 // turn motor
// Motor speed definitions
#define FWDSPEED 50 // Speed to set forward motor
#define REVSPEED -50 // Speed to set reverse motor
#define STOPSPEED 0
#define LEFTSPEED 120
#define RIGHTSPEED -120
#define DEBUGGING 1

//Arbitrary number to send to kill processes
#define KILLPROCESS 99

//direction
// 0  - stop
// 1  - forward
// -1 - reverse
int direction = 0;
void goForward(){direction = 1;}
void goReverse(){direction = -1;}
void stop(){direction = 0;}

//leftRight
// 0  - centre wheels
// 1  - right
// -1 - left
int leftRight = 0;
void turnRight(){leftRight = 1;}
void turnLeft(){leftRight = -1;}
void centreWheels(){leftRight = 0;}
void main()
{
    //Initialize code variable
    int code = DONOTHING;
    int forward = 0;
    int reverse = 0;
    int left = 0;
```

```
    int right = 0;
    int centre = 0;
    int counter = 0;


    //Variables to store process IDs of the motor processes
    int driveProcess = start_process(driveProcess());
    int turnProcess = start_process(turnProcess());

    //Disable infra red decoding since we're not using it
    disableIRDecoding();

    //Disable control of the UART by interactive C and set it up
to use interrupts
    initSerial();

    //Loop
    while(1)
      {
        while(!dataAvailable())
          {
            defer();
            counter++;
            if (counter == 40){
                direction = 0;
                leftRight = 0;
                counter = 0;
            }

        }
        counter = 0;
        code = serialGetChar();
        forward = code & FORWARD;
        reverse = code & REVERSE;
        left = code & LEFT;
        right = code & RIGHT;
        centre = code & CENTRE;

        //if(DEBUGGING) printf("\ncode: %b",code);

        //If the opcode is DONOTHING, stop moving
        if(code != DONOTHING)
          {
            //BEGIN forward/reverse code
            if((forward > 0) && (reverse == 0))
              {
                //Go forward
                goForward();
            } else if((reverse > 0) && (forward == 0))
              {
                //Go reverse
                goReverse();
```

```
            }else{
                stop();
            }
            //END forward/reverse code


            //BEGIN left/right/centre code
            if((left > 0) && (right == 0))
              {
                //Turn left
                turnLeft();
            } else if((right > 0) && (left == 0))
              {
                //Turn right
                turnRight();
            }

            //Need to use an if instead of else if because inter-
active C can't handle
            // more than one else if in a row...apparently.

            //Centre opcode takes precedence over the other sig-
nals
            //if(centre > 0)

            //END left/right/centre code
        } else{
            //Do nothing
            stop();
            leftRight = 0;

        }

        //Sleep for 45ms
       // msleep(50L);
    }

    //Code currently cannot reach here
    direction = KILLPROCESS;
    leftRight = KILLPROCESS;


}

void driveProcess()
{
    while(direction != KILLPROCESS)
      {
        //off(MOTORDRIVE);
        if(direction == 1)
          {
            //Go forward
```

```
            motor(MOTORDRIVE,FWDSPEED);
        }
        if(direction == 0)
          {
            //Stop
            off(MOTORDRIVE);
        }
        if(direction == -1)
          {
            //Go reverse
            motor(MOTORDRIVE,REVSPEED);
        }

        //Reset instruction to 0
        //direction = 0;
//        msleep(50L);
    }
}
void turnProcess()
{
    while(leftRight != KILLPROCESS)
      {
        if(leftRight == 1){
            //Turn right
            motor(MOTORTURN,RIGHTSPEED);
        }
        if(leftRight == 0){
            //Centre wheels
            off(MOTORTURN);
        }
        if(leftRight == -1){
            //Turn left
            motor(MOTORTURN,LEFTSPEED);
        }

        //msleep(50L);
    }
}
```

controller-noack.ic

```
/*
 * controller is a program which sets the code upon receiving
 * instructions from the microswitch signals from the
 * Handyboard.
 *
 * @author John Koh
 * @date February 19, 2010
 *
*/

#include "header.ic"
#include sci_isr.c

//switch definitions, direction switch_number
#define go_forward 15
#define go_back 14
#define go_left 13
#define go_right 12
#define center 11

void main() {
    int code = DONOTHING;
    int dnflag = 0;
    initSerial();
    disableIRDecoding();

    while (1) {
        if (digital(go_forward)) {
            code = code + FORWARD;
        }
        if (digital(go_back)) {
            if ((digital(go_forward)) ||
                (code & REVERSE == REVERSE)  ||
                (code & FORWARD == FORWARD)) {
                code = code;
            } else {
                code = code + REVERSE;
            }
        }
        if (digital(go_left)) {
            code = code + LEFT;
        }
        else if (digital(go_right)) {
            code = code + RIGHT;
        }
        if (digital(center)) {
            code = CENTRE;
        }


        printf("%d \n",code);
```

```
        if ((code == DONOTHING) && (dnflag == 1)) {

        }else if (code == DONOTHING){
            dnflag = 1;
            serialPutChar(code);
        }else{
            serialPutChar(code);
            dnflag = 0;
        }

        msleep(100L);
        code = DONOTHING;
    }
}
```

header.ic

```
/*
 * This is a header file with various constants defined for use
by car_drive and car_control programs.
 *
 * @author Peter Fyon
 * @date February 5, 2010
 *
 */

//Define opcodes
//Included in the comments are the required bits and the 'don't
cares'
//Be sure you don't send conflicting opcodes

#define FORWARD 0x08 // xxxx 10xx
#define REVERSE 0x04 // xxxx 01xx
#define LEFT 0x02 // xxxx xx10
#define RIGHT 0x01 // xxxx xx01
#define CENTRE 0x10 // xxx1 xx00
#define DONOTHING 0x00 // xxx0 0000

//IR decoding takes up 11% of the processor when it's enabled
(which is default)
void disableIRDecoding()
{
    bit_clear(0x39, 0b00000010);
}

void enableIRDecoding()
{
    bit_set(0x39, 0b00000010);
}
```

```
sci_isr.asm
/* This sets up the serial interrupt service routine */
/* First Version:     Written by Anton L. Wirsch  20 Nov 1997 */
/* Second Version: Written by Tim Gold   27 May 1998
                              BYU Robotics Lab
                          goldt@et.byu.edu

     Really, the only thing left from the original code are a few
     lines in the .asm file.  Everything else I pretty much had
to
     rewrite from scratch to get it to work the way I wanted to.
     But the orignal code by Anton was a very helpful starting
point.

  Needed files:   serial_isr.c
                  serial_isr.icb
             serial_isr.asm (needed to change the buffer size)

  The buffer size here is 32 bytes (probably much larger than it
needs
  to be.)  To change the buffer size, do the following:
              1. Change the BUFFER_SIZE constant in serial_isr.c
to the
                 desired number of bytes.
             2. Edit the line in this fils which contains
                the word "EDIT" in the comment so that the value
              matches that of BUFFER_SIZE.
             3. Recreate the serial_isr.icb file by typing the
following:
                 > as11_ic serial_isr.asm
*/


/* change this line to match your library path... */
#include "6811regs.asm"

        ORG MAIN_START
variable_CURRENT:
      FDB    00        * ptr to next data to be read by user

variable_INCOMING:
        FDB    00        * number of bytes received (circular
count)

variable_BASE_ADDR:
      FDB    00        * base address of buffer (to be set by
init routine)

variable_DATA_FLAG:
        FDB    00        * flag set when data is available

variable_buffer_ptr:
```

```
        FDB    00        * pointer to CURRENT buffer

subroutine_initialize_module:
/* change this line to match your library path... */
#include "ldxibase.asm"

        ldd     SCIINT,X
        std     interrupt_code_exit+1
        ldd     #interrupt_code_start
        std     SCIINT,X
       rts

interrupt_code_start:
        ldad    variable_INCOMING       * store INCOMING into AB
        cmpb    #00                     * compare B with 0
        bhi     skip                    * goto "skip" if (B > 0)
        ldx     variable_BASE_ADDR      * STORE ADDRESS OF ARRY
IN X
        inx                             * SKIP THE FIRST (?)
        inx                             * TWO BYTES      (?)
        inx                             * OFFSET TO THE HIGHER
BYTE (?)
        stx     variable_buffer_ptr     * SAVE PTR VALUE
        bra     cont

skip:
        ldx     variable_buffer_ptr     * load buffer pointer
into x
cont:
        ldad    variable_INCOMING       * load INCOMING into AB
        incb                            * increment INCOMING
     cmpb    #4                      * compare B and 4   --EDIT
TO CHANGE BUFFER SIZE--
     beq     reset_count             * if a=32, goto reset_count
     bra     cont1
reset_count:
     ldad    #00                     * set count to zero
cont1:
        stad    variable_INCOMING       * store AB into INCOMING
        ldab    SCSR                    * load SCSR (SCI status
register) into B (why?)
        ldab    SCDR                    * load SCSR (SCI data
register) into B
        stab    ,X                      * store data in array
        inx                             * increment by two bytes
        inx
        stx     variable_buffer_ptr     * save the pointer value
     ldad    #01                     * load 1 into AB
     stad    variable_DATA_FLAG      * store AB into DATA_FLAG
(indicating data is available)
interrupt_code_exit:
        jmp     $0000
```

```
sci_isr.c
/* C program to read serial port with interrupt service routine
*/
/* First version:  Written by Anton Wirsch   20 Nov 1997 */

/*

   Second Version: Written by Tim Gold   27 May 1998
                                BYU Robotics Lab
         goldt@et.byu.edu

     Really, the only thing left from the original code are a few
     lines in the .asm file.  Everything else I pretty much had
to
     rewrite from scratch to get it to work the way I wanted to.
     But the orignal code by Anton was a very helpful starting
point.

  Needed files:   serial_isr.c
                  serial_isr.icb
    serial_isr.asm (needed to change the buffer size)

  The buffer size here is 32 bytes (probably much larger than it
needs
  to be.)  To change the buffer size, do the following:
              1. Change the BUFFER_SIZE constant below to the
          desired number of bytes.
       2. Edit the line(s) in the serial_isr.asm which contain
          the word "EDIT" in the comment so that the value
   matches that of BUFFER_SIZE.
       3. Recreate the serial_isr.icb file by typing the follow-
ing:
          > as11_ic serial_isr.asm

 */

//Peter Fyon, Feb 5, 2010: Changed filename from serial_isr.c to
sci_isr.c as some
// older programs have issues with filenames greater than 8 char-
acterss


#define BUFFER_SIZE 4  /* change buffer size here  -- see above
*/

/* various constants used by the program... */
#define BAUD 0x102b   /* baud rate set to 9600 */
#define SCCR2 0x102d
#define SCCR1 0x102c
#define SCSR 0x102e
#define SCDR 0x102f
#use "SCI_ISR.ICB"
```

```
int buffer[BUFFER_SIZE]; /* this is the actual buffer */
int return_char;

void initSerial()
{
    /* Call this routine to activate the serial interrupt han-
dler. */
    int i,temp;

    /* clear out buffer */
    for(i=0; i<BUFFER_SIZE; i++)
      {
        buffer[i] = 0;
    }

    /* clear vairous flags */
    DATA_FLAG = 0;
    INCOMING = 0;
    CURRENT = 0;
    return_char = 0;

    /* pass address of buffer to interrupt routine */
    buffer_ptr = (int) buffer;
    BASE_ADDR = (int) buffer;

    /* activate interrupt routine */
    temp = peek(SCCR2);
    temp |= 0x24;
    poke(SCCR2, temp);
    poke(0x3c, 1);
}

void closeSerial()
{
    int temp;

    /* deactivate the interrupt routine */
    temp = peek(SCCR2);
    temp &= 0xdf;
    poke(SCCR2, temp);
    //Peter Fyon: Not sure why this line is here, READ_SERIAL
doesn't even exist in the asm routine
    //READ_SERIAL = 0x0000;
    poke(0x3c, 0);

}

void serialPutChar(int c)
{
    /* call this function to write a character to the serial port
*/
```

```
    while (!(peek(0x102e) & 0x80));
    poke(0x102f, c);

}


int dataAvailable()
{
    /* This function can be used to check to see if any data is
available */
    return DATA_FLAG;
}


int serialGetChar()
{
    /* Create blocking getchar for serial port... */

    /* loop until data is available */
    while(!DATA_FLAG){};

    /* get the character to return */
    return_char = buffer[CURRENT];

    /* check for wrap around... */
    CURRENT++;
    if(CURRENT == BUFFER_SIZE)
      CURRENT = 0;
    if(CURRENT == INCOMING)
      DATA_FLAG = 0;
    return return_char;

}
```

opcode_test.ic

```
/*
 * Program to test the opcodes for car_drive over the serial
port.
 */
#include sci_isr.c
#include constants.ic

void main()
{
    int val;
    int opcode[] = {CENTRE, FORWARD, REVERSE, LEFT, RIGHT,
DONOTHING};
    int i = 0;

    initSerial();
    while(1)
      {
        if(start_button()){
            val = opcode[i];
            i++;
            if(i == 6){
                i = 0;
        }
      }

        if(stop_button())
          {
            serialPutChar(val);
            printf("\nSent: %d", val);
            msleep(50L);
          }
    }

    /*
        //for(i = 0; i < 6; i++)
        while(i < 6)
          {
            val = opcode[i];
            serialPutChar(val);
            printf("\nSent: %d", val);
            sleep(0.2);

            while(!start_button()){}
            i++;
        }
        i = 0;
*/
}
```

serial_test_receive.ic

```
/*
 * Program to test the assembly serial interrupt routine on the
handyboard.
 */

#include sci_isr.c

void main()
{

    int receivedChar = 0;


    initSerial();
    printf("\nStarting");

    while(!stop_button())
      {
        receivedChar = serialGetChar();
        printf("\nReceived: %d",receivedChar);
    }
    printf("\nStopping");
    closeSerial();
}
```

serial_test_send.ic

```
/*
 * Program to test the assembly serial interrupt routine on the
handyboard.
 */

#include sci_isr.c

void main()
{
    int val = 0;

    initSerial();
    printf("\nStarting");

    while(!stop_button())
      {
        if(start_button())
          {
            val = knob();
            serialPutChar(val);
            printf("\nSent: %d", val);
            sleep(0.2);
        }
    }
    printf("\nStopping");
    closeSerial();
}
```

car_drive.ic

```
/*
 * car_drive is a program which receives commands over the UART
 * to drive motors and (possibly in a later version) a servo.
 * It is intended to control a remote control car over a mesh
 * network provided by ZigBee capable chips that transmit com-
mands
 * from a controller to the vehicle running this program.
 *
 * @author Peter Fyon
 *
*/

#include sci_isr.c
#include header.ic
#include protocol.ic

// Motor number definitions
#define MOTORDRIVE 0 // fwd/reverse drive motor
#define MOTORTURN 1 // turn motor

// Motor speed definitions
#define FWDSPEED 75 // Speed to set forward motor
#define REVSPEED -75 // Speed to set reverse motor
#define STOPSPEED 0

#define LEFTSPEED 30
#define RIGHTSPEED -30

#define SLEEP_DURATION 50L

#define DEBUGGING 1

//Arbitrary number to send to kill processes
#define KILLPROCESS 99

//direction
// 0  - stop
// 1  - forward
// -1 - reverse
int direction = 0;
void goForward(){direction = 1;}
void goReverse(){direction = -1;}
void stop(){direction = 0;}

//leftRight
// 0  - centre wheels
// 1  - right
// -1 - left
int leftRight = 0;
void turnRight(){leftRight = 1;}
void turnLeft(){leftRight = -1;}
```

```
void centreWheels(){leftRight = 0;}

void main()
{
    //Initialize code variable
    int code = DONOTHING;
    int forward = 0;
    int reverse = 0;
    int left = 0;
    int right = 0;
    int centre = 0;
    int lastSeqNum = 0;

    //Variables to store process IDs of the motor processes
    int driveProcess = start_process(driveProcess());
    int turnProcess = start_process(turnProcess());

    //Disable infra red decoding since we're not using it
    disableIRDecoding();

    //Disable control of the UART by interactive C and set it up
to use interrupts
    initSerial();

    if(DEBUGGING)
      {
        printf("\nStarted");
    }

    //Loop
    while(1)
      {
        while(!dataAvailable())
          {
            defer();
        }

        code = serialGetChar();
        if(DEBUGGING) printf("\n%d : %d",code, getSeqNum(code));

        if(matchesParity(code) && getSeqNum(code) != lastSeqNum)
          {
            //Packet's parity is correct and seq num is the next
expected one
            forward = code & FORWARD;
            reverse = code & REVERSE;
            left = code & LEFT;
            right = code & RIGHT;
            centre = code & CENTRE;

            //Alternate sequence number
            lastSeqNum++;
```

```
            lastSeqNum = lastSeqNum % 2;

            //Send an ACK for the packet received
            sendAck(lastSeqNum);
            printf("Ack %d",lastSeqNum);

            if(DEBUGGING) printf("\ncode: %b",code);

            //If the opcode is DONOTHING, stop moving
            if(code != DONOTHING)
              {
                //BEGIN forward/reverse code
                if((forward > 0) && (reverse == 0))
                  {
                    //Go forward
                    goForward();
                } else if((reverse > 0) && (forward == 0))
                  {
                    //Go reverse
                    goReverse();
                }
                //END forward/reverse code


                //BEGIN left/right/centre code
                if((left > 0) && (right == 0))
                  {
                    //Turn left
                    turnLeft();
                } else if((right > 0) && (left == 0))
                  {
                    //Turn right
                    turnRight();
                }

                //Need to use an if instead of else if because
interactive C can't handle
                // more than one else if in a row...apparently.

                //Centre opcode takes precedence over the other
signals
                if(centre > 0)
                  {
                    //Centre wheels
                    centreWheels();
                }
                //END left/right/centre code
            } else
              {
                //Do nothing
                stop();
            }
```

```
            //Sleep for 45ms
            //msleep(SLEEP_DURATION);
        } else
          {
            //Sequence number is out of order or packet is some-
how mangled
            //Reply with an ACK for the last successfully re-
ceived packet
            sendAck(lastSeqNum);
            printf("Ack %d",lastSeqNum);
        }
    }
    //Code currently cannot reach here
    direction = KILLPROCESS;
    leftRight = KILLPROCESS;


}

void driveProcess()
{
    while(direction != KILLPROCESS)
      {
        if(direction == 1)
          {
            //Go forward
            motor(MOTORDRIVE,FWDSPEED);
        }
        if(direction == 0)
          {
            //Stop
            off(MOTORDRIVE);
        }
        if(direction == -1)
          {
            //Go reverse
            motor(MOTORDRIVE,REVSPEED);
        }

        //Reset instruction to 0
        //direction = 0;
        msleep(SLEEP_DURATION);
    }
}

void turnProcess()
{
    while(leftRight != KILLPROCESS)
      {
        if(leftRight == 1)
          {
```

```
            //Turn right
            motor(MOTORTURN,RIGHTSPEED);
        }
        if(leftRight == 0)
          {
            //Centre wheels
            off(MOTORTURN);
        }
        if(leftRight == -1)
          {
            //Turn left
            motor(MOTORTURN,LEFTSPEED);
        }

        msleep(SLEEP_DURATION);
    }
}

void sendAck(int seqNum)
{
    //Create an ACK packet
    int packet = createPacket(ACK, seqNum, 0);
    serialPutChar(packet); }
```

controller.ic

```
/*
 * controller is a program which sets the code upon receiving
 * instructions from the microswitch signals from the
 * Handyboard.
 *
 * @author John Koh
 * @date February 19, 2010
 *
*/

#include header.ic
#include protocol.ic
#include sci_isr.c

//switch definitions, direction switch_number
#define go_forward 15
#define go_back 14
#define go_left 13
#define go_right 12
#define center 11

void main() {
    int code = DONOTHING;
    int sequenceNo = 1;
    int receivedPacket = 0;

    initSerial();
    disableIRDecoding();

    while (1)
      {
        if (digital(go_forward)) {
            code = code + FORWARD;
        }
        if (digital(go_back)) {
            if ((digital(go_forward)) ||
                (code & REVERSE == REVERSE)  ||
                (code & FORWARD == FORWARD)) {
                code = code;
            } else {
                code = code + REVERSE;
            }
        }
        if (digital(go_left)) {
            code = code + LEFT;
        }
        else if (digital(go_right)) {
            code = code + RIGHT;
        }
        if (digital(center)) {
            code = CENTRE;
```

```
        }

        while(1)
          {
            printf("\n%d",code);
            serialPutChar(createPacket(DATA, sequenceNo, code));
            msleep(100L);
            if(dataAvailable())
              {
                receivedPacket = serialGetChar();
                if (getSeqNum(receivedPacket) == sequenceNo)
                  {
                    printf(",%d",receivedPacket);
                    //Alternate the sequence number
                    sequenceNo++;
                    sequenceNo = sequenceNo % 2;

                    code = DONOTHING;
                    break;
                }
            }
        }

    }
}
```

protocol.ic

```
/*
 * This file defines the protocol and provides methods to create
and access contents of packets
 *
 */
//Default ACK packet for receiver: X1X00000
//Default data packet:X0XXXXXX
//Parity bit is calculated by adding together all the other bits
in the packet
#define paritybit 0b10000000

//Used for the receiver to send ACKs back
//1 for an ACK, 0 for a NAK
//Set to ACK if it's a data packet
#define acknakbit 0b01000000

//Alternating sequence bit, initially 0
#define seqnumbit 0b00100000
#define databit0  0b00000001
#define databit1  0b00000010
#define databit2  0b00000100
#define databit3  0b00001000
#define databit4  0b00010000

#define NAK 0
#define ACK 1
#define DATA 1

//Returns true if the packet matches the parity bit, or false if
it doesn't
int matchesParity(int packet)
{
    int bitcount = 0;
    int parity = 0;

    //Get the parity bit
    if(packet & paritybit)
      {
        parity = 1;
    }

    if(packet & acknakbit)
      {
        bitcount++;
    }

    if(packet & seqnumbit)
      {
        bitcount++;
    }
```

```
    if(packet & databit0)
      {
        bitcount++;
    }

    if(packet & databit1)
      {
        bitcount++;
    }

    if(packet & databit2)
      {
        bitcount++;
    }

    if(packet & databit3)
      {
        bitcount++;
    }

    if(packet & databit4)
      {
        bitcount++;
    }

    if(bitcount % 2 == parity)
      {
        return 1;
    } else
      {
        return 0;
    }
}

int createPacket(int ack, int seqNum, int data)
{
    int packet = 0;
    int parity;

    //Set the data bits
    packet |= data;

    //Set the bit if it's an ACK/data packet
    if(ack)
      {
        packet |= acknakbit;
    }

    //Set the seqnum bit to whatever it should be
    if(seqNum)
      {
        packet |= seqnumbit;
```

```
    }

    parity = createParity(ack, seqNum, data);
    if(parity)
      {
        packet |= paritybit;
    }
}
/*
 * createParity takes the ack/nak, sequence number, and data bits
and
 * adds them together, returns the result mod 2 for parity bit
 */

int createParity(int ackNak, int seqNum, int data)
{
    int temp = 0;
    int i;
    int bitcount = 0;

    bitcount += ackNak;
    bitcount += seqNum;

    //Add up the data bits
    for(i = 0; i < 5; i++)
      {
        temp = data << 1;
        if(temp)
          {
            bitcount++;
        }
    }
    return (bitcount % 2);
}

int getSeqNum(int packet)
{
    if(packet & seqnumbit)
      {
        return 1;
    }
    return 0;
}
//The opposite is true, meaning !isAck() means it's a NAK
int isAck(int packet)
{
    if(packet & acknakbit)
      {
        return 1;
    }
    return 0;
}
```