Conception, Design and Construction of a Remote Wifi Vehicle using Arduino

Author: Clayton Broman
Advisor: Dr. Tom Bensky

March 20, 2014
Contents

0.1 Acknowledgments .................................................. 1
0.2 Introduction ..................................................... 1

1 What is a Wifi Vehicle? ............................................. 3

2 The Components .................................................... 4
  2.1 The Chassis ..................................................... 6
  2.2 Microcontroller ............................................... 7
  2.3 Ethernet Shield ............................................... 8
  2.4 Wifi Bridge ................................................... 9
  2.5 The Router .................................................... 10
  2.6 Laptop Interface ............................................. 12
  2.7 The Joystick Controller .................................... 13

3 How it works ....................................................... 15
  3.1 Hacking the car ............................................... 15
  3.2 Programming the Ruggedduino ............................ 18
  3.3 Programming the Laptop ................................... 19
  3.4 Setting up the Router .......................................... 20
  3.5 Setting up the Wifi Camera ................................. 22
  3.6 Setting up the wifi bridge ................................. 24

4 Conclusion .......................................................... 25
  4.1 Appendix A .................................................... 25
  4.2 Appendix B .................................................... 32
    4.2.1 Vehicle wiring diagram ............................... 32
Abstract

The scope of this senior project was to make a wireless vehicle controlled via Internet Protocol. This vehicle operates remotely and without direct line of sight. Commands are sent from a program running on a laptop and transmitted using a wireless router. Visual data is retrieved from a network camera, mounted on the vehicle, in real-time, to see where you are going.
0.1 Acknowledgments

I could not have completed this project on my own and in fact I had a lot of help. First of all I would like to thank my senior project advisor Dr. Tom Bensky as the primary motivating force behind this project. Dr. Bensky, teaching the advanced electronics elective course, (Phys 357) was the gateway to my fascination with electronics and microcontrollers.

My lab instructor for Phys 357, and co-advisor to this project, was Dr. Matt who provided invaluable insight into how to go about tackling this project and how to properly troubleshoot my circuitry when things went wrong.

I would also like to thank lab technician Jim Hilsinger for teaching me how to soldier, giving me insight into which electronic components were available, and for listening and conversing at length about the technical details of this project.

Lab partners Spencer Herrick, Lisa Slaughter, Kate Miles, and Zechariah Thurman who over the years contributed their boundless enthusiasm, interesting conversations, and company at sleep deprived times while completing projects in the lab.

Physics majors Kara Zappitelli, Charlie Showley and Ivo Plamenac who became family, as well as the rest of the Cal Poly Physics department for their interest, encouragement and help to make this project a success.

Last and certainly not least my Mom and Dad for moral and monetary support and for never ceasing to be incredibly impressed by my projects.

0.2 Introduction

This senior project was inspired by the advanced electronics lab course (Phys 357) offered to physics majors to fulfill an upper division elective lab requirement. As part of this course curriculum students are required to design and build electronic circuitry that collected and store physical data as well as perform physical tasks. Since a great portion of the time spent building my senior project involved programming learned or least introduced too while taking this course, these are the aspects of Phys 357 I will elaborate on as my introduction to this project.

For our first project in Phys 357 we were given a kit of Legos Mindstorm NXT 2.0 and were required to build an electronic circuit that measured a physical quantity, analyzed the data in some fashion, and then used the Legos Mindstorm to respond to and alter the physical quantity being measured in someway. Aside from wheels and building blocks as are commonly associated with Legos- the heart of the Legos Mindstorm is the NXT brick, this “brick” is a programmable microcontroller that connects directly to multiple sensors which include ultrasonic, pressure and light sensing, etc. capabilities that come standard with the Legos Mindstorm kit. By connecting the NXT brick to a computer via USB it was possible to use the integrated development environment (IDE), LabView, to program the NXT brick to respond to physical stimuli and perform tasks. LabView made it easy to program the NXT brick because all tasks and functions within the LabView IDE are icons made to resemble the task or function to be carried out, i.e. a motor moving forward has a picture of a motor with small green arrows to indicate the forward direction (red for backwards). Once the user programed the NXT brick to perform the task desired the program could be uploaded to the NXT brick hard drive and the Legos would operate autonomously.
For our second assignment we were required to use the Arduino programmable microcontroller in conjunction with some of the many sensors available (GPS, CO₂, alcohol, light intensity, etc.) to record data from a physical phenomena or perhaps within a certain situation, i.e. how many people pass in and out of the electronics lab. Arduino, similar to the NXT brick is a programmable microcontroller, but instead of using LabView to program it, it comes with a free open source (not copyrighted) IDE called the Arduino playground. The programming language used in Arduino playground is Java. Java is based on the C language, but is more developed, and “further from the hardware” than C, is not however as highly developed as LabView, in that you still must type in commands rather than finding their respective icons.

Having little to no prior experience programming in Java, my partner and I set forth to programming the Arduino to take location information from GPS and light intensity from a small solar panel mounted on top of the GPS, and save these values to an SD card. In order to read the GPS data, we used a preexisting script function, called TinyGPS.h that could be downloaded from a website, this sort of file that has functions already written for you is called a user library. Using the library greatly reduced the amount of work we were required to do, that, we just figured out how to use the library and it gave us our GPS location. We ultimately succeeded in creating a light map of Cal Poly by combining our light intensity data with our GPS data.

I was impressed with the immense data collecting abilities, the minimal programming knowledge required, and the seemingly endless combination of projects microcontrollers, in particular Arduino, was capable of performing. This technology which was previously unknown to me, became a great resource for data collecting and sharing. I believe that investigating the programs required to use these microcontrollers such as Java, would be a tremendous asset to any physicist or for that matter any scientist.

I will not elaborate here on the details of our third and final project for Phys 357, but it utilized the Internet to send and retrieve data via Dr. Bensky’s server, to a remote Arduino, which would respond by turning on and off a coffee pot, as well as tell us the amount of coffee in the pot. This fascinated me, because now the Arduino could be remote with only an Internet connection and we could send data too and receive data from it! Our final project was successful, but it was clunky and slow, so I wanted to take it one step further by using the Internet to connect to Arduino, but to do it wirelessly, and in real time, a sort of Wifi Vehicle.

This report will begin by defining what is meant by the concept of a Wifi Vehicle, it will then explain all constituent parts that were required for building this particular Wifi Vehicle, why they were chosen, and how they contributed to the flow of information. In the chapter entitled “How it works” I will explain the details of how each part was set up and integrated into the system. I will then conclude with what I learned and my plans for what to do with the vehicle in the future.

For more information on this project and others visit my youtube page at

http://www.youtube.com/channel/claybro2005
Chapter 1

What is a Wifi Vehicle?

The basis of this Wifi Vehicle utilizes a remote control (RC) car chassis, retrofitted with an Arduino microcontroller in conjunction with an Ethernet shield for control, a wireless bridge for communication, and an Ethernet camera for a visual feed. All the components will be explained in depth in future chapters, but in essence, the RC cars existing microchip is replaced with an Arduino programmable microcontroller which controls the cars existing circuitry. The Ethernet shield sits atop the Arduino, making it possible to adapt the Arduino platform to send and receive data via an Internet connection. The wireless bridge then connects directly into the Ethernet shield to make a wireless connection with a remote wireless router which is connected to a laptop (via wired or wireless connection).

The program chosen to run on the laptop is an IDE called Processing-2.0. The purpose of the Processing program is to create an easy and user friendly interface between the user and the Wifi Vehicle. Processing was chosen over other possible applications because it is open source (free to use, not copyrighted), has thousands of pre-existing user libraries, and similar to Arduino- is based on the Java programming language, meaning it is completely compatible with Arduino. In addition to these attributes it is possible in Processing to create graphical user interfaces (GUI), Fig.(2.7), which allow the user to visually see how commands being received are changing, as well as how you are interacting with the program using visual aids rather than seeing programming messages and errors, this is very useful to someone like me who is a visual learner.

The vehicle is controlled using an X-Box 360 controller, chosen because this controller has a USB plug that can be directly connected to a laptop, as well as Processing having a user friendly library that interprets commands sent from the X-Box controller. Mounted on top of the car is a wireless Ethernet camera which connects directly to the wifi router. This camera comes with its own software that was intuitive to set up and made it possible to operate the vehicle in real time without direct line of sight.
Chapter 2

The Components

This project required a number of parts, and each in turn was hand picked for its cross-platform compatibility - this was an important requirement because I am running Ubuntu Linux on my laptop and must be able to download drivers/programs/etc. that are compatible with the device. Following the data block diagram in Fig.(2.1) this section will give an outline of the flow of information beginning with Processing running on the laptop listening for commands sent to it from the X-Box controller, and upon receiving a command, packaging the signal into a UDP datapacket using the Ethernet.h library and send it via IP protocol to the router. The router is wirelessly linked to the Wifi bridge which reads the IP address and forwards the command sent from the laptop onto the Ruggeduino. The Ruggeduino then loaded with the EthernetUdp.h library is able to unpack the UDP packet, parse the code, and translate it into an integer value which it compares against a list of commands in order to control the vehicle. At the same time the router is linked with the Wifi camera mounted on top of the vehicle which is continuously returning .jpeg images to a browser running on the laptop for the user to see where they are going.

In addition to following the flow of information from user to Wifi vehicle I will briefly explain how each individual component makes a unique contribution to this system, and why I chose the part I did against the myriad of components available on the market.
Figure 2.1: Laptop running Processing-2.0.
2.1 The Chassis

The chassis for the vehicle is a Cyclone 7 remote control car seen in Fig.(2.2). The car runs on a 9.6 V battery pack (see Fig.(4.2) in Appendix B) stored in a compartment on the under side of the car. This car was chosen due to its chassis being large enough to carry the weight of the added components (wifi bridge, Arduino, Webcam, etc.). The Cyclone 7 came with a standard RX2 integrated chip (IC) which controlled the motor relays on the car with 5V outputs. It was easy to find out which outputs of the IC controlled which outputs by probing the IC with a volt meter while sending it messages using its original transmitter (Fig.(4.1) Appendix B).
2.2 Microcontroller

The original microcontroller chosen for this project was the Arduino Leonardo. Upon completing Phys 357 I knew I wanted to do some more projects with the Arduino so I bought myself an Arduino Leonardo simply because I had experience with the Arduino platform in Phys 357 and I didn’t yet feel comfortable branching off into other microcontrollers such as Raspberry Pi (Linux/Python based operating systems) or Beagleboard (Linux) just yet. Arduino boards are also very reasonably priced, the Arduino Leonardo was a few dollars cheaper at $17 U.S. than the Arduino Unos at U.S. $20. In addition the Arduino Leonards had the added functionality of connecting via usb micro as a virtual COM port. This means it can appear as a mouse/keyboard/joystick etc. to software running on the laptop. I thought I may in the future have some fun with this feature so I chose the Arduino Leonardo.

The Leonardo performed well until, after some bad wiring on my part- it was accidentally burned out at the same time I burned out the steering circuit on the RC car. I then replaced the Arduino Leonardo with a Ruggedduino, which is a “rugged” 3rd party version of the Arduino Uno. The platform on the two microcontrollers is identical so moving to the Ruggedduino had no effect on the functionality of the vehicle. The Ruggedduino, even though it is twice as expensive as the Arduino Uno, ($40), has all the same features as the Arduino Uno, plus it has resettable fuses so it is harder to burn out in the future. The role that the Ruggedduino takes in the vehicle is to replace the original IC the car came with. Instead of receiving RF frequencies as the IC did, the Ruggedduino reads datapackets sent by the laptop with the use of the UDP library. Received datapackets are parsed by the Ruggedduino and include directions for controlling the steering and drivetrain relay modules.
2.3 Ethernet Shield

The Ethernet shield fits directly atop of the Ruggeduino see Fig.(2.4) and gives the Ruggeduino a MAC address and Port number for the router to send data too. This shield in an essential interface between the Internet and Ruggeduino. It receives an Ethernet signal from the Wifi bridge and imports all data to the serial port of the Ruggeduino.
2.4 Wifi Bridge

The wireless Wifi bridge is a Vonets VAP11G. Again this product was chosen because it is open source and in fact was very easy to install. The wireless bridge was not in the original plan (which was to attach the router directly to the top of the vehicle), but was added because it is less bulky and consumes less power from the batteries than the router. The bridge essentially mimics the setup of the router, relaying everything it receives from the router and passes it to the Ethernet shield atop the Ruggeduino.
2.5 The Router

The router used in this project was a Netgear N300 WNR2000v2 wireless router. The role of the router was to set up a local area network (LAN) and to forward all incoming messages destined for the Ruggeduino to the Ruggeduino. It achieved this via a technique called port forwarding (more about how to set up Port forwarding in Ch. 3). This router was chosen because it was conveniently lying around, and because it’s hardware is open source compatible. Routers typically come with a proprietary operating system called its “firmware.” Many router manufacturers don’t allow for anything but their stock firmware to be loaded on their routers, for example they will produce CPUs that are specific to only their particular software. There is a large demand however for routers that are in a sense, “universal,” for this reason some manufactures (including Netgear and LinkSYS) have capitalized on this demand by producing product lines that are cross platform compatible. Developers are then free to load Linux based router distributions onto these routers. The reasons for doing so varies from loading servers onto routers, overclocking, extending the range of the router by increasing power output, etc., all available with the use of a convenient web based GUI. The main reason I upgraded to a Linux based operating system, DD-WRT v24, is that this firmware upgrade has a number of added features the standard firmware didn’t come with including an easy to use port forwarding option (more on how to install this 3rd party software in Ch. 3).
Figure 2.6: Netgear N300 WNR2000v2 wireless router.
2.6 Laptop Interface

Processing 2.0 is a free, open source program that was easily installed on my laptop. The laptop used for this project is an Asus, Intel core i5 running 32-bit Ubuntu Linux v12.04. Processing is an object oriented language based on Java that has hundreds of user libraries that people have written for it. One package of particular interest, considering the scope of this project, was hypermedia.net. Hypermedia.net made it possible to send and receive user datapacket (UDP) (more on UDP in the Ch. 3) messages over the Internet by declaring an IP address and port number of the destined machine. Within Processing it is also easy to create graphical user interfaces (GUI), to control your program- see Fig.(2.7). By including the processing.dxf library in your code it was possible to create “triangle based graphics” such as boxes, lines, and circles, that you can manipulate quickly and efficiently with commands such as “mousepressed.” The processing.serial library in conjunction with the processing.procontroll library made it possible to then interact with your GUI and thus the vehicle with the use of an X-Box 360 controller. In the next chapter I will outline where to get user libraries, how to load them into the proper directory and how to use them.
2.7 The Joystick Controller

The vehicle is controlled via a standard X-Box 360 joystick controller Fig.(2.8) which connects via USB to the Laptop. This controller offered a way to comfortably, and professionally control the Wifi Vehicle. It was relatively inexpensive at $15 U.S. and was simple to integrate in to the design due to Processing having preexisting libraries, such as processing.proControll mentioned in the previous section, written for it.

To recap Fig.(2.1) illustrates the flow of information from the the Laptop to the Wifi Vehicle. Each part makes a unique contribution to the system and is summarized as follows. The user watching the visual feed from the wireless Ethernet camera pushes a button on the X-Box controller. The program running on the Laptop listens to commands from the X-Box controller, and upon receiving a command, Processing packages a signal into an UDP datapacket and sends it via IP to the router. The router is then linked to the Wifi bridge which reads the IP address and forwards the command sent from the Laptop onto the Ruggeduino. The Ruggeduino then loaded with the EthernetUdp.h library is able to unpack the UDP packet, parse the code and translate it into an integer value which it compares against a list of commands in order to control the vehicle. At the same time the router is also linked with the Wifi camera which is continuously returning a .jpeg image to the browser on the laptop for the user to see where they are going.
Figure 2.8: Xbox 360 controller.
Chapter 3

How it works

3.1 Hacking the car

The initial plan was to use the original steering and power relays that came with the RC car. To do this it was necessary to determine which outputs of the RX2 IC on the car powered which control modules. It turned out that the RX2 IC that came with the car is used almost universally so the data sheet was easily found online and copied. see Fig.(3.1)

By probing the RX2 IC with a digital volt meter (DVM) while controlling it with the the original radio-frequency (RF) transmitter (TX IC- see Fig.(4.1 in Appendix A), it was possible to verify which outputs of the RX2 controlled which power modules by measuring a 5V spike at the output IC. It was then possible to remove the RX receiver entirely and solder leads directly to these modules see Fig.(3.2). These leads then run directly to digital output pins on the Ruggeduino to control the vehicle.

As mentioned in Chapter 2, the original plan for this project was to use the steering circuit that came with the RC car, but it was accidentally burned out from a shorted wire. After tracing out, troubleshooting and burning out many transistors trying to salvage the original steering circuit, it was scraped entirely and a new steering mechanism was designed. The steering circuit and motor were replaced with a Parallax continuous rotation servo motor discovered in the electronics lab by Dr. Bensky. This servo motor after loading the “servo” library onto the Ruggeduino had plug and go capability. With minimal effort it was easily installed into the vacated spot the original steering motor occupied and even utilized the original linkage system, see Fig.(3.4).
Figure 3.1: Wiring diagram of RX receiver.

Figure 3.2: The reverse side of the original microchip with added soldered leads.
Figure 3.3: The original microchip overlaid with what each section is responsible for doing what.

Figure 3.4: The continuous rotation servo taking the place of the original steering motor.
3.2 Programming the Ruggedduino

Once the leads to the motor and servo were properly inserted into Ruggedduino, it was time to program the Ruggedduino. Ruggedduino runs a program based in Java, an object based language that is somewhat intuitive to use. The full program code I named UD-Pread.ino and can be found in Appendix A of this paper. There are two noteworthy libraries that were included in the Ruggedduino code— the “EthernetUdp.h” which makes it possible to send and receive UDP datapackets, and the “Servo.h” which was included to operate the new steering servo motor. Both of these libraries were downloaded from the Internet free of charge. The EthernetUdp.h library was produced by MIT and can be downloaded from [http://roboticsclub.org](http://roboticsclub.org), servo.h produced by an independent developer, Michael Margolis, was downloaded from [https://github.com/arduino/Arduino/blob/master/libraries/Servo/Servo.h](https://github.com/arduino/Arduino/blob/master/libraries/Servo/Servo.h). Documentation on how to use both of these libraries and more can be found on the Arduino website at [arduino.cc/en/Reference/](https://arduino.cc/en/Reference/)

Once downloaded it is required to copy libraries to the correct directory so that your Arduino playground can find and load them, for my (Linux) machine the library directory is `/home/clayton/sketchbook/libraries`. One is able to find the proper library directory by going to File→Preferences in the Arduino IDE, and looking for the sketchbook location. Use a file manager to then navigate to this location on your computer and look for a sub directory called “libraries,” it will be here that you should store your downloaded library files. The Ruggedduino needs three items to use the EthernetUdp.h library properly. The first is a mac address to identify the Ethernet shield in the code the line; `byte mac[] = ( 0X90, 0XA2, 0xDA, 0X0D, 0X33, 0X59 )`, the mac address is specific to the Ethernet shield and is printed on the underside of the unit. Second, an IP address for the Laptop to send information too is necessary, the code; `IPAddress ip(192, 168, 1, 177)`, creates this variable. For those of you with some background in IP the default subnet mask is not declared is 255, 255, 255, 0, the host number was chosen at random between 0 and 999. Finally a port number for the Ruggedduino to listen on is required, this number is randomly chosen between 0 and 10,000, to identify the port number the code; `unsigned int localPort = 8888;` was inserted.

The main loop unwraps the datapackets, and parses the content using the parsePacket command. It saves this to the variable “packetSize” displayed in the line; `int packetSize = Udp.parsePacket();`. This command counts the number of characters in the message sent from the laptop. The size of packetSize determine what tasks the Ruggedduino is to perform. For instance a packetSize of 1 will send a 5V pulse to the power control module to send the vehicle forward. The logic for this task to occur is as follows.

```
if (packetSize == 1)
    digitalWrite(2, HIGH);
```

This logic for the forward and backward motion is linear, but the code got a little more tricky for the steering. Retrofitting the vehicle with the continuous rotation servo dramatically changed how the car steered such that it requires constant adjustments from the operator to control the car. The short turning radius of the car and the quick response of the servo motor made steering the car a bit tricky. Instead of returning to a neutral position, the car continues to turn until the operator turns the wheels back to a neutral position.

To turn the car right the servo motor must receive a value less than 90 from the command “myservo.write” The smaller this number is (0 being the lowest) the faster the
motor turns to the right. If the number is greater than 90 the car turns to the left, the greater this number is (to a max of 180) the faster the car turns to the left. By setting this value too close to 90 the car took a long time to make a turn, but making it too big or too small the car would turn way too quickly. To combat this the code now makes the vehicle turn very fast (180 or 0) but for a very short duration of time $\approx 2$ ms, then stops turning. For example the command to turn the wheels to the right

```c
else if (packetSize == 4){
    myservo.write(0);
    delay(2);
}
```

If packetSize is equal to 7 the result is the vehicle does nothing, this is a safety feature included to prevent the car from running away if goes beyond the radius of the router signal.

The memory of the parse is wiped clean with each iteration of the loop by the code

```c
for(int i=0;i<UDP_TX_PACKET_MAX_SIZE;i++){
    packetBuffer[i] = 0;
}
```

this function walks through each message received by the Ruggedduino one character at a time and replaces it with zeros. This feature was added because if the variable data was not cleared each iteration, data would sometimes carry information over to the next iteration resulting in the vehicle receiving a signal different from the one sent by the operator.

3.3 Programming the Laptop

As previously noted the program running on the laptop is Processing-2.0. This program allows the user to easily create graphical user interfaces (GUI) to control your project. Processing, can be downloaded for free (or donation if you feel inclined) from the program website at https://processing.org/download. The main library that assists in the making of GUIs is “processing.dxf”. This library comes standard with Processing so no downloading or copying is needed. This feature allows you to set the size of figure, import a background image, and even draw figures that can be manipulated in real time. For this project the background image was set as a Mario Brothers mushroom (Fig.(2.7)), because for some reason in the back of my mind I envisioned the finalized vehicle being a real life Mario Kart drag racing down the halls of the physics department... A here we GO! On top of the the background image, rectangular sliders will relay to the operator what the current values of the controller joysticks are. The background was loaded using the command “background(bg)” where the variable bg is set to the image of the Mario mushroom.

Using the “procontroll;” library in combination with “net.java.games.input” library made it possible to include the use of the X-Box controller. The “net.java.games.input” library comes standard with Processing, but the procontroll library was downloaded from the website “creativecomputing.cc/p5libs/procontroll/,” and since Processing is so similar to the Arduino playground you can follow the EXACT same procedure as above in the Arduino section to find the libraries directory in Processing. The “println(Serial.list());” command prints a list of recognized USB devises on your computer. It also prints a list of which buttons, sliders and joysticks are available for these devices. It was then possible
to label the buttons using the “getButton” command, i.e. \( Y = \text{gamepad.getButton}(3); \). The value of these buttons can be set arbitrarily by the user. These values were then used to change the location or color of the buttons in the GUI. This allowed for interactive feedback to the user. For instance when the user moves the joystick on the X-Box controller left or right, the value of \( x \) changes from -128 to be full left, 0 is no motion, and 128 is full right. The value of \( x \) is converted to pixel number and results in a change in the position of the slider on the GUI.

```javascript
fill(0);
rect(leftLine, height/2 - leftY-10, 40, 20);
```

this code creates a rectangle \( 20 \times 40 \) pixels in size, situated on the left track of the GUI (leftLine), beginning at a height of the GUI divided by 2, where “leftY” is the value of \( x \) converted to pixel number. This directly ties \( x \) (and thus the joystick) to the position of this rectangle.

Using the “hypermedia.net.*;” package made it possible to send and receive UDP datapackets over IP. The reason we want to send data via UDP as opposed to TCP is because we want all commands to be in real time. UDP discards data that shows up late or out of order. TCP by contrast will request missed data be resent by the laptop, this does not apply to our situation because the vehicle is running in real time. For example if a command requesting the vehicle turn right shows up late then too bad, the user may not still want the vehicle to turn right (it may be on the edge of a cliff by then). UDP protocol is used for all real time applications where the quality of data isn’t as important as the data be current- such as voice over IP [1].

By defining an address to send the information to “String ip= ”192.168.1.177”;” and a port destination “int port = 8888;” makes it possible for the correct data to be sent to the correct location. Note that these are identical to the user defined IP address and port number of the Ruggeduino. The message to send to the Ruggeduino is created through a logic structure, first boolean variables are created and set to “false” for example, “boolean keydown = false;” next we’ll make a true false case that determines if the car will send a message to tell the vehicle to go in reverse. The point of the true false case is because there are eight directions the car travels; forward, backward, right and left, and combinations of the of two or more such as forward and right. There are only four controls however; forward, backward, right and left, so the logic structure is such that a combination of two or more sends a different message than if only one direction is pushed for example pushing one button

```javascript
if (keyleft) {message = "le"};
```

while a combination of two or more buttons sends a different message.

```javascript
if (keyup && keyleft) {message = "ule"};
```

Sending the message is accomplished using the function “udp.send(message, ip, port );;” This function is short and to the point, it requires three arguments; the message to send the IP address and the port number. Easy.

### 3.4 Setting up the Router.

Port forwarding is a method by which traffic arriving at the router with a particular IP address can be forwarded to a program listening on a specific port number. In the scope
of this project it is quite easy to set up because the vehicle and laptop are both within the same Local Area Network (LAN), and there is not a firewall setup on the router. Any IP packets arriving for 192.168.1.177 with port number = 8888, are directed by the router to the Ruggeduino. I chose to download some open source hardware for my router to make this portion of the project easier. Warning: loading 3rd party software onto a router that does not support it will “brick” or render your router inoperable. This is due to the fact that routers do not have bootloaders, small bits of native firmware from which you can load your operating system onto, therefore if your firmware upload gets interrupted or you download an incompatible firmware- your router is in a sort of worthless state of limbo (a brick).

After much research I chose to go with DD-WRT as my 3rd party Linux distribution of choice simply because it was the ONLY available 3rd party software available for my particular router distribution. It does however offer a large amount of options not available on the original firmware distribution. By going to the DD-WRT website at www.dd-wrt.com/site/index I was able to find my router under the “support” menu and download the proper distribution for my router. I then went to my router homepage at 192.168.1.1 and navigated to the “update firmware” tab, and after carefully following the advise listed on the DD-WRT website, including performing a hard reset on my router, and selectively ignoring warnings that I could potentially brick my router performing this procedure, I proceeded (successfully) to upload, “flash,” my router with the new firmware (operating system). The new homepage for the router was still 192.168.1.1, but looked entirely different. By navigating to the tab marked NAT/QoS → Port Forwarding it was a straightforward matter to set up the port forwarding option, see figure(3.5) for an example of how easy it is to set up port forwarding in the DD-WRT browser. Under the line “Port from,” 6000 is the application port number given to arriving data at the router sent by Processing. I chose this number at random, and initialized in the line of code in Processing

```java
    udp = new UDP( this, 6000 );
```

Also note in the figure that the field for arriving IP address “Source Net” does not need to be filled out, this is again because the vehicle and laptop are on the same LAN. The destination IP address (the address of the Ruggeduino) is inserted on the line marked “IP Address,” and finally the destination port number (the Port number for Ruggeduino) is inserted on the line labeled “Port to.”
3.5 Setting up the Wifi Camera

Setting up the Ethernet wifi camera was quite easy. The camera came installed with an Internet interface that could be accessed directly using the cameras IP address. Using this software it was possible to make the camera connect to the router wirelessly upon startup by entering; the name of router (SSID) and the password to connect. The image could then be retrieved by opening a browser window and entering the IP address of the wireless camera followed by the port number, Fig(3.6 and 3.7).
Figure 3.6: Driving the vehicle remotely with wifi camera.

Figure 3.7: Options available on the wifi camera.
3.6 Setting up the wifi bridge

Setting up the wifi bridge was also quite easy. Using the included software it was just a matter of entering the routers SSID, and the password to connect. Now any data the router received was automatically repeated to the the wifi-bridge. The wifi bridge has an Ethernet cable that could be plugged directly into the ethernet shield on top of the Ruggeduino.
Chapter 4

Conclusion

This was an ambitious project due to my minimal exposure to programming, but I’m happy with how it turned out, and feel that I learned a lot over the course of the last two quarters. I feel I have a better grasp on programming especially Java, which is a very powerful programming language that I have not even begun to scratch the surface of. I learned that small things such as putting a user library in an incorrect directory could result in hours of frustrating searches through Internet forums looking for an answer to your problem. I learned a lot of valuable information about how to troubleshoot programs and how the Internet works, including port numbers, IP addresses, and the difference between TCP and UDP protocols and under what circumstances each protocol is favorable.

As it stands the project does what it is intended to do, it will drive remotely and relays visual images back to the operator in real time, however it is not very rugged. If the camera is jolted too hard it disconnects from the router and wont reconnect without someone manually unplugging it and plugging it back in. The vehicle also requires a lot of battery power, these two problems led me to decide I will take apart the camera and disconnect the pan/tilt functionality to conserve energy. In addition I plan to get the Wifi camera working within the framework of Processing, i.e. perhaps get Processing to import a picture from the Internet using a library such as “processing.video” which would consolidate things and keep the project as a whole more compact, it may also reduce the problem of the camera disconnecting, if I can find a way to get Processing to restart the camera automatically when it does not receive a message from it.

The steering also needs an overhaul, I think that instead of a continuous rotation servo-an angular servo would be better suited for this application, i.e. when the user stops steering the car to left the car stops steering to the left and returns back to driving straight. After these fixes the options are limitless for this platform. Some other possible additions include solar power and controlling it via an Android application over a 4G network. I feel I have barely scraped the surface of the possibilities for the Arduino/Ruggeduino.

4.1 Appendix A

Programming the Ruggeduino: The following is the code written to Ruggeduino microcontroller.
Clayton Broman 03/18/14 UDPread.ino Arduino code

#include <Servo.h>
#include <SPI.h>  // needed for Arduino version s later than 0018
#include <Ethernet.h>
#include <EthernetUdp.h>  // UDP library from: bjoern@cs.stanford.edu 12/30/2008

// Enter a MAC address and IP address for your cont roller below.
// The IP address will be dependent on your local net work:
byte mac[] = { 0X90, 0XA2, 0XDA, 0X0D, 0X33, 0 X59 }; //Not important
IPAddress ip(192, 168, 1, 177);

unsigned int localPort = 8888;  // local port to listen on
unsigned int incoming = 0;

// buffers for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE];  // buffer to hold incoming packet,
//char ReplyBuffer[] = "Howdy Duty";  // a string to send back

// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

Servo myservo;  // create servo object to control a servo

void setup() {
  // start the Ethernet and UDP:
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);

  Serial.begin(9600);
  pinMode(2, OUTPUT);  // sets the digital pin as output
  pinMode(3, OUTPUT);
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop() {
  // Go grab a packet
  int packetSize = Udp.parsePacket();
  // if there’s data available, read a packet
  if(packetSize)
    {
      if (packetSize == 1)
        {
          digitalWrite(2, HIGH);  // move forward;
        }
      else if (packetSize == 2)
        {
          myservo.write(180);  // turn left;
          delay(2);
        }
      else if (packetSize == 3)  // left & forward
        {
          myservo.write(180);  // left
digitalWrite(1,HIGH); //forward
        }
      else if (packetSize == 4)  //
        {
          myservo.write(0);  //  turn right
          delay(2);
        }
      else if (packetSize == 5)
        {
          digitalWrite(2, HIGH);  // forward (up)
          myservo.write(0);  //right
        }
      else if (packetSize == 6)  // reverse
        {
          digitalWrite(3, HIGH);
        }
      else if (packetSize == 8)
        {
          digitalWrite(3, HIGH);  //reverse
          myservo.write(180);  //left
        }
      else if (packetSize == 10)
        {
          digitalWrite(3, HIGH);  // reverse (down)
          myservo.write(0);  // right
        }
      else if (packetSize == 7)  // if you pushed anything other than down
        {
          digitalWrite(3,LOW);  // coasts
digitalWrite(2,LOW);
          myservo.write(93);  //stops steering
        }
      // for(int i=0;i<UDP_TX_PACKET_MAX_SIZE;i++) { packetBuffer[i] = 0;  //This clears the buffer so it
doesnt get gunked up with previous codes
      //}

      // All that was if there was a packsize, so if there was
      // then do the following
else if (packetSize == false) // if you pushed anything other than down
  {
    digitalWrite(3,LOW);
    digitalWrite(2,LOW);
    myservo.write(94);  // stops steering
    //Serial.println("no packet");
  }

delay(10);
}
Clayton Broman 03/18/14 \ UDPrunner.pde Processing code

/*
Clayton Broman
Senior Project 2014
Processing
*/
import processing.serial.*;
import procontroll.*;
import net.java.games.input.*;
import processing.dxf.*;
import hypermedia.net.*;

//Booleans for controll
boolean keyup = false;
boolean keyright = false;
boolean keyleft = false;
boolean keydown = false;
String ip= "192.168.1.177";  // the remote IP address
int port = 8888;// the destination port
String message = "";
PImage bg; // set the background wallpaper
UDP udp; // define the UDP object
Serial myPort;
PFont f, b;
//char HEADER =#'"

// WEB Cam trials
//Capture cam; //uncomment in the future

// The define controller
ControllIO controll;
ControllDevice gamepad;
ControllStick leftStick;
ControllStick rightStick;
ControllStick brake;
ControllStick gas;
ControllCoolieHat DPad;
ControllSlider XBOXTrig;

float leftTriggerMultiplier, leftTriggerTolerance, leftTriggerTotalValue;
float rightTriggerMultiplier, rightTriggerTolerance, rightTriggerTotalValue;

float leftX = 0;
float leftY = 0;
float rightY = 0;
float rev = 0; // Reverse
float forward =0; //Drive
float centerY =0;
byte lOutput = 0;
byte rOutput = 0;
byte lDir = 0;
byte rDir = 0;
float angle;
float magnitude;
float output;
float tolerance = 0.17; // set tolerance value for joysticks
int mode = 0;
boolean selectSet = false; // for reading the state of the select button
boolean startSet = false; // for reading the state of the start button
boolean ySet = false;
boolean DEBUG = false;
boolean trexMode = false;
ControllButton A;
ControllButton B;
ControllButton Y;
ControllButton X;
ControllButton L1;
ControllButton L2;
ControllButton L3;
ControllButton R1;
ControllButton R2;
ControllButton R3;
ControllButton Select;
ControllButton Start;
ControllButton Up;
ControllButton Down;
ControllButton Left;
ControllButton Right;

void setup(){
size(700,700);
// cam = new Capture(this); //has future applications
// cam.start();

f = loadFont("ArialMT-48.vlw");
b = loadFont("Arial-BoldMT-48.vlw");
bg = loadImage("shroom.jpeg");
// The microcontroller will connect on the first port listed
// with a baud rate of 9600
// Don’t forget to uncomment myPort once you’re ready to start
// using the microcontroller
println(Serial.list()); //Discover what your computer is calling the controller you have plugged in.
//myPort = new Serial(this, Serial.list()[0], 9600);
controll = ControllIO.getInstance(this);

gamepad = controll.getDevice("Microsoft X-Box 360 pad");
gamepad.printSliders();
gamepad.printButtons();

// This is the section of the wrapper class I used
leftStick = new ControllStick(gamepad.getSlider(1), gamepad.getSlider(0));
gas   = new ControllStick(gamepad.getSlider(3), gamepad.getSlider(5));
brake = new ControllStick(gamepad.getSlider(3), gamepad.getSlider(2));

XBOXTrig = gamepad.getSlider(4);
leftTriggerTolerance = rightTriggerTolerance = XBOXTrig.getTolerance();
leftTriggerMultiplier = rightTriggerMultiplier = XBOXTrig.getMultiplier();

Y = gamepad.getButton(3);
B = gamepad.getButton(1);
A = gamepad.getButton(0);
X = gamepad.getButton(2);
R1 = gamepad.getButton(5);
R3 = gamepad.getButton(9);
L1 = gamepad.getButton(4);
L3 = gamepad.getButton(8);
DPad = gamepad.getCoolieHat(11);
Select = gamepad.getButton(6);
Start = gamepad.getButton(7);

// create User Datagram Packet Protocol connection
on port 6000
// <-- print out the connection activity
// and wait for incoming message
udp = new UDP( this, 6000 );
//udp.log( true );
udp.listen( true );
}

void draw()
{

//image(cam,0,0); //Future
background(bg);

// set the coordinates for the left and right stick objects
int leftLine = width*2/16, centerLine = width*8/16, rightLine =width*14/16;
    mode = 0;


// begin computing and map values for joysticks
// .get() and map() both need float values

// if you want to use X-axis declare them here.
leftY = leftStick.getY();
rev = brake.getY();
forward = gas.getY();

// trig functions use RADIANS by default
angle = atan2(leftY, leftX);
magnitude = sqrt(pow(leftX,2) + pow(leftY,2));
output = cos(2 * angle);

// remap the magnitude eliminating any slack in the joystick
magnitude = map(magnitude, sqrt(2 * pow(tolerance,2)), 1, 0, 1);

// joystick is not a perfect circle, therefore ignore anything
// greater than 1.0 or less than zero
if(magnitude > 1.0)
magnitude = 1.0;
if(magnitude < 0)
magnitude = 0;

// set to zero if triggers are pressed
if(L1.pressed())
rev = 0;
keyup=false;
keydown = false;
if(R1.pressed())
forward = 0;
keyup=false;

// This maps things on the visual

//Put a right and left scale in here
if(leftY < -tolerance)
{
    leftY = map(leftY, -1, -tolerance, 127, 0);
    keyleft = true;
} else if(leftY > tolerance)
{
    leftY = map(leftY, 1, tolerance, -128, 0);
    keyright = true;
} else
{
    keyright =false; //car defaults to not doing anything
    keyleft =false;
}

// Reverse ***
if(rev < -tolerance)
{
Clayton Broman 03/18/14 \ UDPrunner.pde Processing code

rev = map(rev, -tolerance, -1, 0, -120);
keydown = false;
}
else if(rev > tolerance)
{
    rev = map(rev, tolerance, 1, 0, 128);
    keydown = true;
}
else
{
    keydown = false; //car defaults to nothing
}

// Drive
if(forward < -tolerance)
{
    forward = map(forward, -tolerance, -1, 0, -120);
    keyup = false;
}
else if(forward > tolerance)
{
    forward = map(forward, tolerance, 1, 0, 128);
    keyup = true; //car goes forward
}
else
{
    keyup = false; // car does not go forward
}

//---------------------------------------------------------
// Below this point the message to send is determined
if (keyup)
{
    message = "u";
}
if (keydown) 
{
    message = "downn";
}

// Add in the right rectangle: and make it turn red if R1 pressed
if(R1.pressed())
{
    fill(255,0,0); // red
    rect(centerLine, height/2 + 115, 40, 20);
}
else {
    fill(0);
    rect(centerLine, height/2 - forward- 5, 40, 20);
}

// *** Everything below this point is just labels ***
// textFont(f,16);
// fill(0);
textAlign(RIGHT, CENTER);
text(int(leftY), leftLine - 50, height/2);
// textAlign(LEFT, CENTER); // Hang on to these in case you want to add the specifics later
// text(int(rightY), rightLine + 50, height/2);

// Label the Sliders
textAlign(CENTER, TOP);
textFont(f,18);
text("Right", leftLine, height/2 + 130);
text("Rev.", centerLine, height/2 + 130);
text("Forward", rightLine, height/2 + 130);
text("Left", leftLine, height/2 - 140);

//---------------------------------------------------------
// Below this point the message to send is determined
if (keyup)
{
    message = "u";
}
if (keydown) 
{
    message = "downn";
}
if (keyleft)
{
    message = "le";
}

if (keyright)
{
    message = "rite"; //
}

if (keyup && keyleft) message = "ule"; //3
if (keyup && keyright) message = "urite"; //5
if (keydown && keyleft) message = "downleft"; // 8
if (keydown && keyright) message = "downright "; //10
    if (keyup & keydown) message = "nothing"; // 7
    if (keyright & keyleft) message = "nothing"; // 7
  no longer an issue with the xbox controller
    if (keyright == false & keyleft==false & keyup
==false & keydown == false) message = "nothing"
    ; //7

    // the message to send
    println(message); // see whats being sent
    udp.send(message, ip, port);
}

// void receive( byte[] data )
    // <-- default handler   COMMENTED ALL TH
    IS FOR SPEED!!!

    //Clear the data
    // for(int i=0; i < data.length; i++)
    // print(char(data[i]));
    // println();
    // }
    // This is for future applicaitons
    //void captureEvent(Capture c)
    // {
    // c.read();
    // }
4.2 Appendix B

4.2.1 Vehicle wiring diagram

Outline of what is happening on the vehicle itself, as well as some of the parts not mentioned above.
Figure 4.1: The original RF controller.

Figure 4.2: 9.6V Battery pack 2,000 mAh.
Bibliography


