CSC 460: Real Time Systems

Project 3: Humans vs. Zombies

Jian Guan: V00732919, SEng Student

guan3d@gmail.com

Brandon Jacklyn: V00732611, SEng Student

brandonjacklyn@gmail.com

Date: April 7, 2015

Table of Contents

- 1.0 Introduction
  - 1.1 Game Overview
  - 1.2 Hardware
  - 1.3 Software
- 2.0 System Overview
- 3.0 Base Station
  - 3.1 Base Station Components
  - 3.2 Joystick Controllers
  - 3.3 Complete Base Station
- 4.0 Roomba Controller
  - 4.1 Roomba LEDs
    - 4.1.1 Resistors
  - 4.2 nRF24L01 Radio
    - 4.2.1 Game Packet
  - 4.3 Roomba Interface
    - 4.3.1 Roomba Commands
    - 4.3.2 Movement
    - 4.3.3 Sensors
    - 4.3.4 Ignoring Roomba Failsafes
  - 4.4 IR Emitter and Receiver
    - 4.4.1 IR Emitter
    - 4.4.2 IR Transmitter
  - 4.5 Roomba Music
  - 4.6 Roomba Autonomous Features
    - 4.6.1 Autonomous Testing
  - 4.7 RTOS Task Scheduling
    - 4.7.1 Radio Task Overview
  - 4.8 HC-06 Bluetooth Radio
    - 4.8.1 Configuring the Bluetooth Radio
    - 4.8.2 Communicating Over Bluetooth
    - 4.8.3 Reading Velocity and Rotation
    - 4.8.4 Testing the Bluetooth Received Values
  - 4.9 Base Station and Roomba Integration Testing
  - 4.10 Completed Roomba
- 5.0 Android App
  - 5.1 Tracking the User's Finger
  - 5.2 Mapping the Finger to Roomba Commands
  - 5.3 Using Bluetooth
  - 5.4 Android Bluetooth Driver
- 6.0 Conclusion
- 7.0 Future Work
- 8.0 Appendix

1.0 Introduction

In the final project of CSC 460 we teamed up with 3 other groups to create a game of Humans vs. Zombies (HvZ). Each of the groups controlled an IRobot's Create2 robot using a common base station which managed the overall game state, and sent joystick controller input. Any autonomous functionality was the responsibility of each group created independently.

The other groups we collaborated with are:

- Simon Diemert and Scott Low: https://github.com/sdiemert/csc460
- Paul Moon and Jordan Yu: https://github.com/paulmoon/csc460
- Justin Guze and Paul Hunter: https://github.com/paulhunter/csc460

Additional information can be found on each of their github wiki pages.

1.1 Game Overview

Humans vs. Zombies is a modified version of tag with 2 sides: humans and zombies. Humans win by surviving for a pre-determined time period, and zombies win when they have “infected” or turned every human into a zombie. Because of the small number of roomba’s we start with one zombie and three humans. The humans and zombie are chosen by the base station when the game is initialized.

In our game humans use the IR transmitters as “guns” which “stun” the zombies for a period of time. Zombies cannot be killed only stunned, however, each time a zombie is stunned the length of time they are stunned for increases. While stunned a zombie cannot move thus giving the human roomba time to move away.

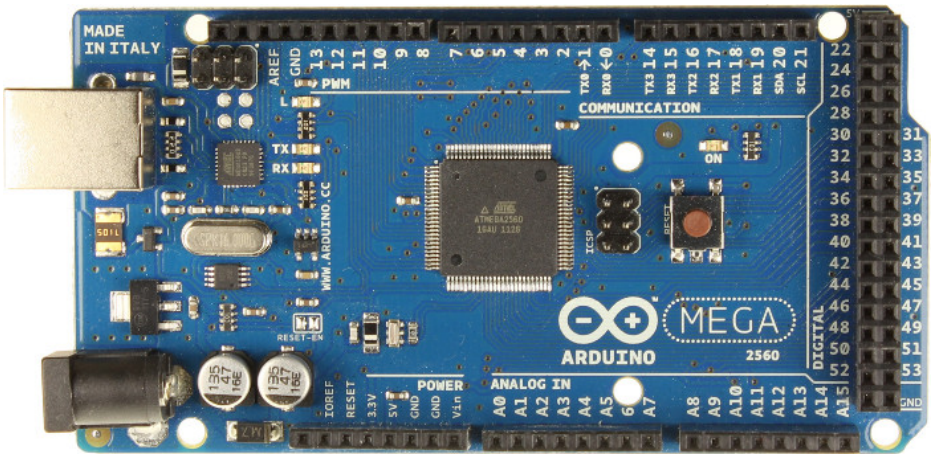
Human players start with a “shield” which allows them to take one hit from a zombie without becoming infected. After a period of time the shield will refresh, but if hit again the human will turn into a zombie.

A common base station provides the controllers which drive the each team’s roomba, and some teams have also implemented autonomous functionality such as wall detection. The following video shows how the game works.

https://www.youtube.com/watch?v=rHvumUfkcDA

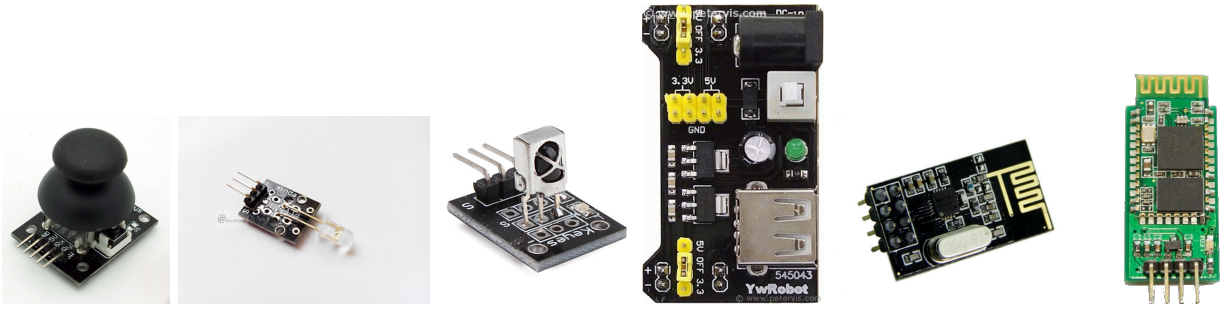
1.2 Hardware

Arduino 2560 Mega



The roomba was controlled by the ATMega2560 which has a microcontroller running at 16MHz. The board has 54 pins with 15 that can be used for PWM. One LED connected to pin 13 was primarily used to flash RTOS errors at the user. Other features include 4 UARTs (serial ports), a USB port, a power jack, and a mechanical button that resets the board.

Arduino Equipment



- Joystick - used to control the roomba movement and activation of the IR transmitter
- IR Transmitter - transmits a 38 KHz modulating signal to IR receiver
- IR Receiver - receives the signals sent by the transmitter
- Breadboard Power Supply (YwRobot) - provided power to the breadboard
- nRF24L01+Module - a radio that sends and receives packets
- HC-06 Bluetooth - a bluetooth radio used to receive commands from the phone

iRobot Roomba Create-2

The iRobot Roomba Create-2 is the roomba we use in the lab for project 3. It has an Arduino microcontroller mounted on top of it so that we can use both radios to transmit and receive packets from our ATMega 2560 microcontroller/Samsung Galaxy Nexus to control the roomba using information from iRobot's API.

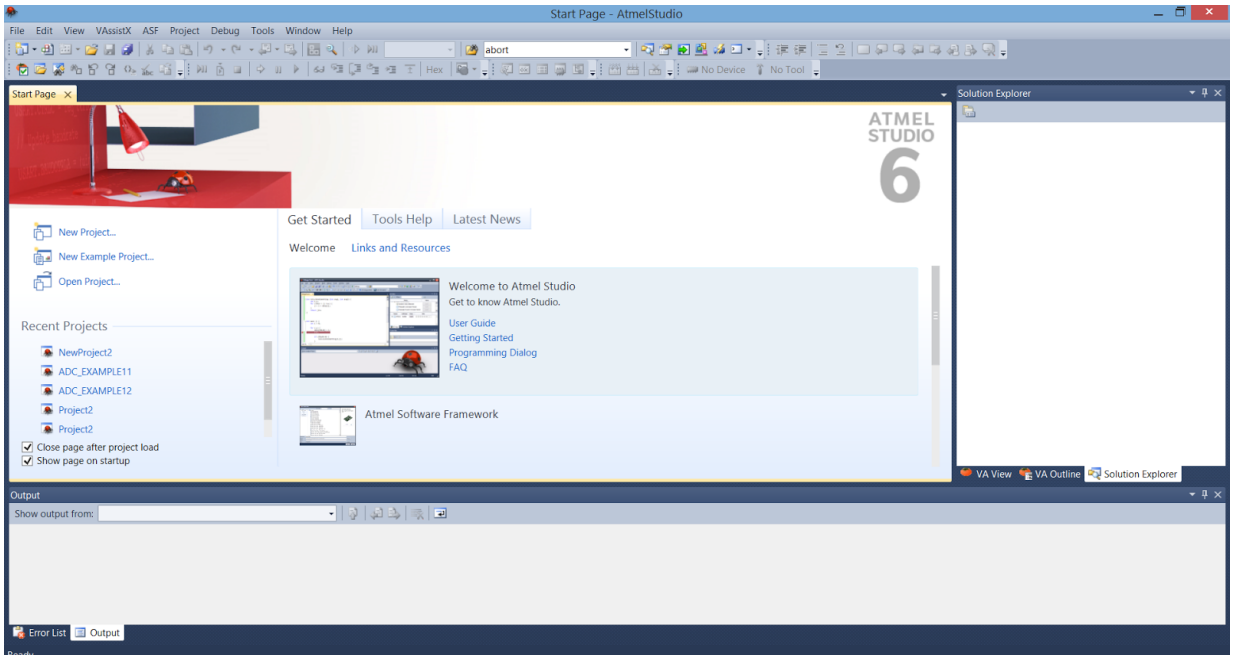
Samsung Galaxy Nexus

The Samsung Galaxy Nexus was used to connect to the bluetooth radio and send movement commands (velocity and rotation) to the roomba. An Android app was created which tracked the user's finger on the screen and mapped the movement to the desired roomba movement.

1.3 Software

Atmel Studio

Atmel Studio v6.2 is an IDE powered by Visual Studio used for developing programs for ARM and AVR microcontrollers. It provides a framework for compiling to ATMega2560 as the libraries have definitions for all the ports/pins, cpu speed, etc. We have added C++ compiler and linker settings so we can use some of the Arduino functionality in our project.

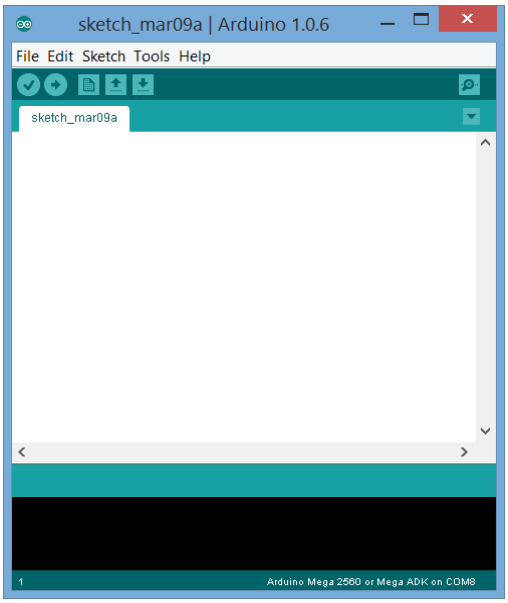


Arduino IDE

The Arduino IDE v1.6.3 is a cross-platform java application which makes it easy to write and upload code to the ATMega2560, as well as any other Arduino board.

Unfortunately, the IDE assumes your code wants to be called in a loop as fast as possible which is not possible in the RTOS we are using which requires control over the entire program, including main().

For these reasons, we switched to Atmel Studio instead.



AVRDude

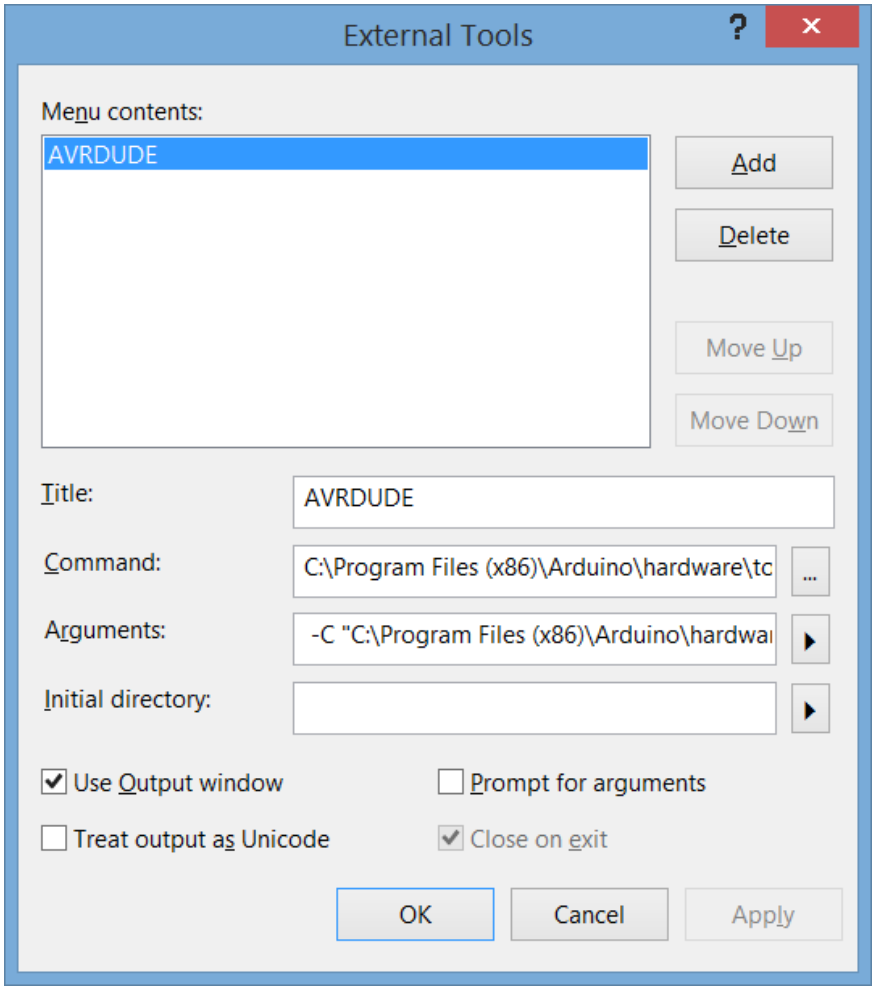
AVRDude is a command line tool which flashes the compiled .hex file to the ATMega2560's program memory.It is bundled with the Arduino IDE so we added it as an external tool in Atmel Studio.

The command points to the AVRDude executable:

```
C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avrdude.exe
```

The arguments specify which board architecture is being flashed, the USB port, the baud rate, and the location of the hex file:

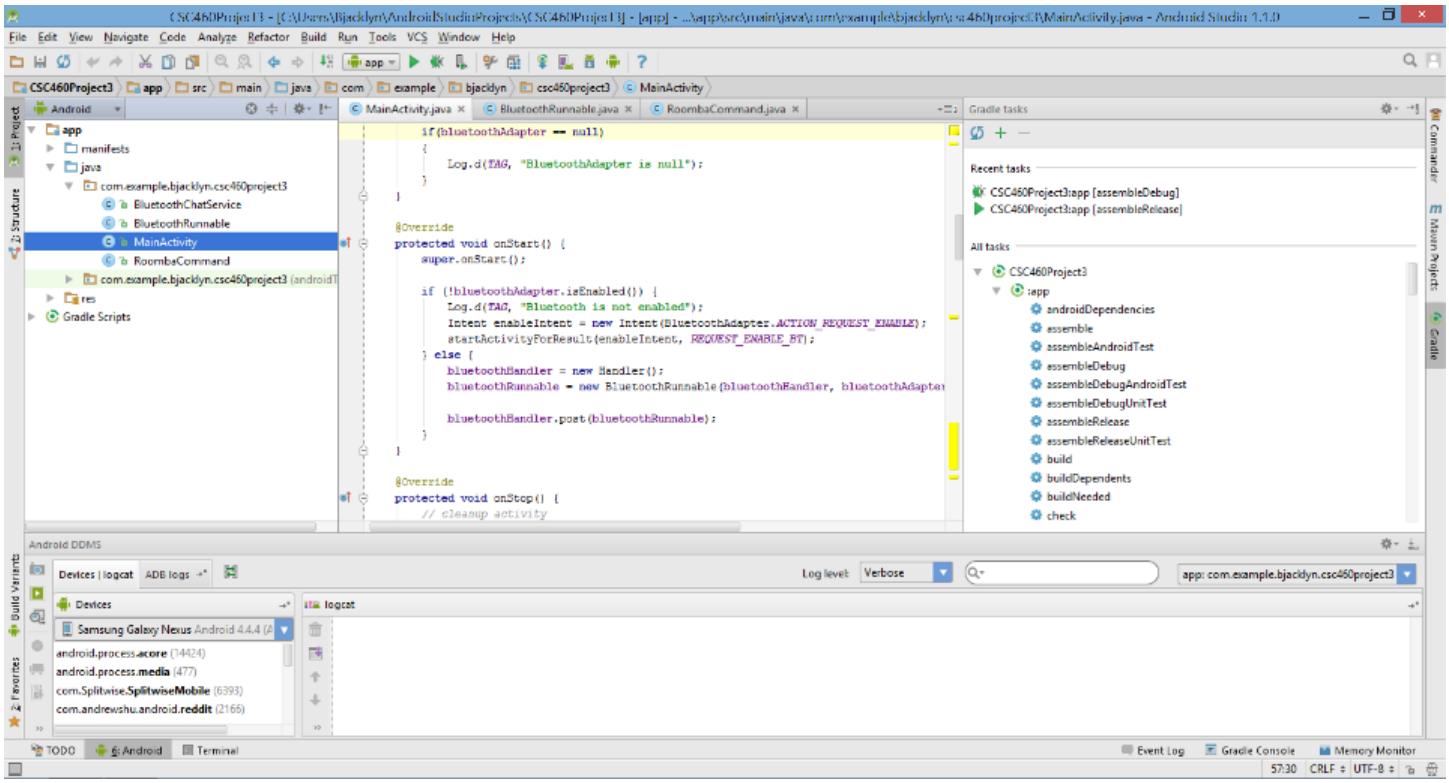
```
-C "C:\Program Files (x86)\Arduino\hardware\tools\avr\etc\avrdude.conf"
        -v -v -p m2560 -c wiring -P \\.\COM8 -b115200 -D -U
        flash:w:"$(ProjectDir)\Debug\YourProjectName.hex":i
```



Android Studio

Android Studio v1.1.0 was used to create an Android application which communicates with the bluetooth radio on the roomba. Android Studio is based on the IntelliJ IDEA platform created by JetBrains. Major features are as follows:

- Integrated with Android Debug Bridge (ADB) driver to load apps and receive log data from a live phone/tablet
- Provides an emulator which displays the app and allows the developer to interact with it on almost every Android platform
- Grade build support automatically compiles Java code into Dalvik bytecode, and assembles apps into .APK files for installation on devices



RTOS

We used Paul Hunter and Justin Guze's RTOS from project 2 on our roomba as well as on the base station. It is located at the following link.

https://github.com/paulhunter/csc460/tree/master/Project2/rtos

2.0 System Overview

The system consists of five ATMega2560 boards: four roombas and one base station. The base station manages the state of the gameplay as well as the input from the four controllers. All communication is through the radio sending a game packet to each roomba.

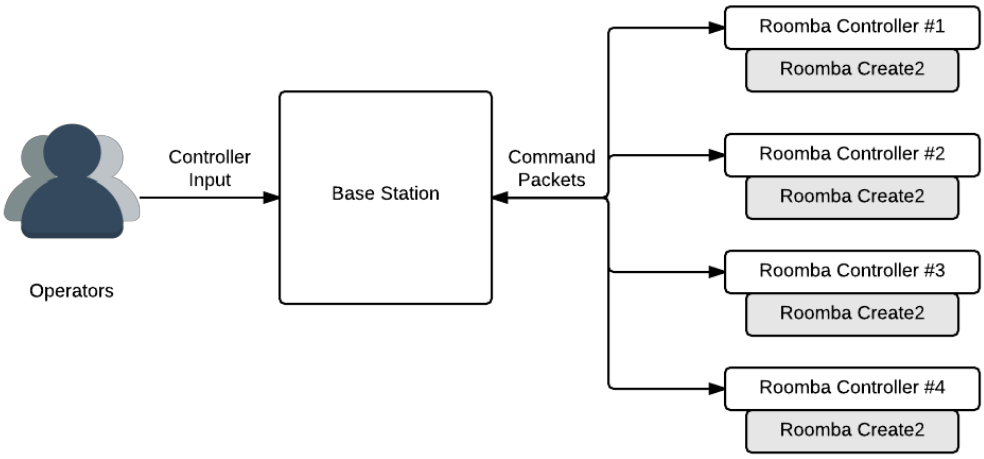


Figure 1: Block Diagram of the Overall System

3.0 Base Station

The base station interfaces with four custom built controllers each consisting of two joysticks and a push button. Each time can determine how to use the various inputs in their roomba, as the base station merely reads the value of each input and includes it in the game packet. A picture of one of the controllers is shown below.

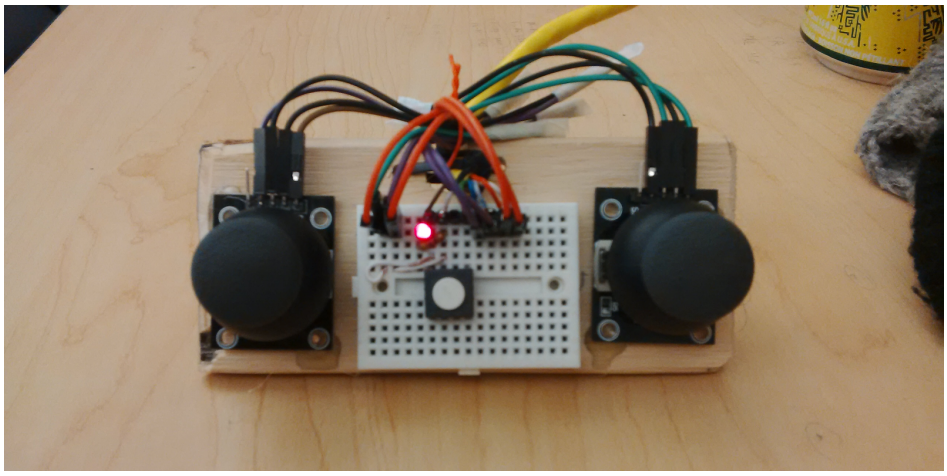


Figure 2: Custom Joystick Controller

3.1 Base Station Components

A total of 8 joysticks and 4 push buttons were connected to the ATMega2560 base station. Furthermore, a nRF24L01 radio was needed to communicate with the roombas, and several LEDs were included for debugging and game state indication.

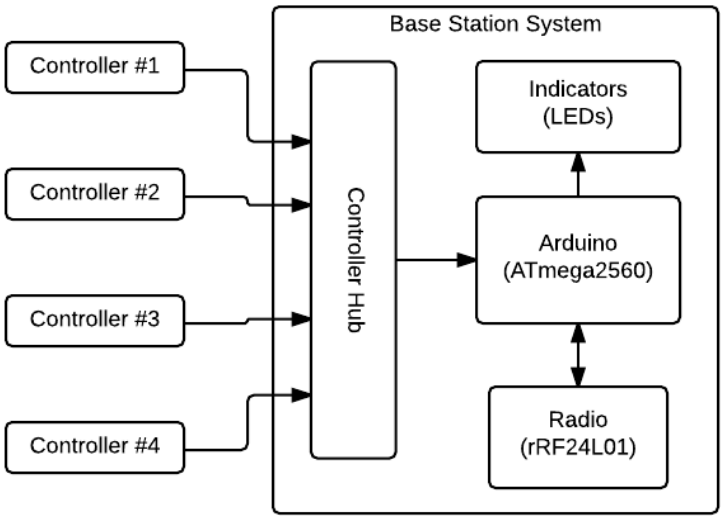


Figure 3: Block Diagram for Base Station

3.2 Joystick Controllers

Each controller consists of two joysticks, a push button, and an LED indicator as well. The wiring diagram is shown below in figure 4; note that the pull-up resistor is missing for the push button.

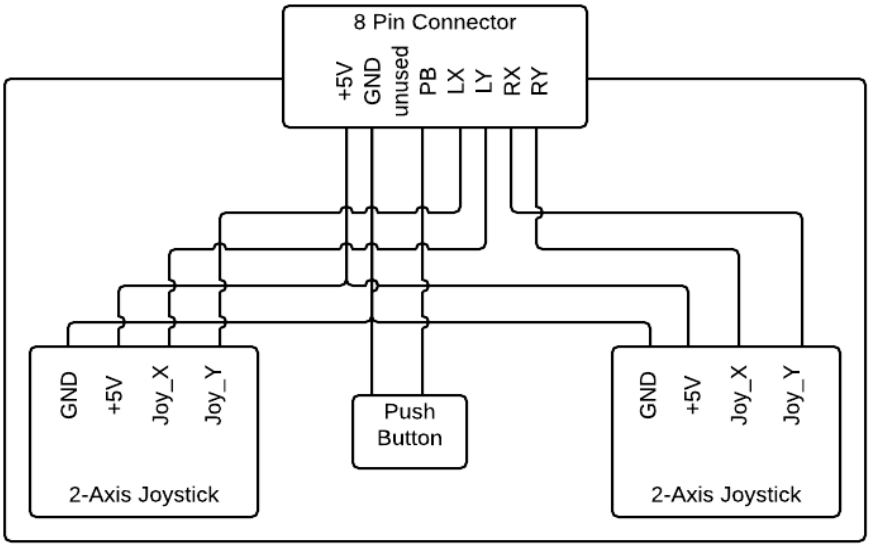


Figure 4: Controller Wiring Diagram

Since there were so many wires to the base station, a base station hub was used to standardize the connections of the controllers. An 8 pin connector was used where seven of the pins carried power to the controller. Pins 0 and 1 were GND and VCC, and pins 3 through 7 carried signals back to the base station. Note that pin 2 was not used, and that figure 5 also contains the pull-up resistors.

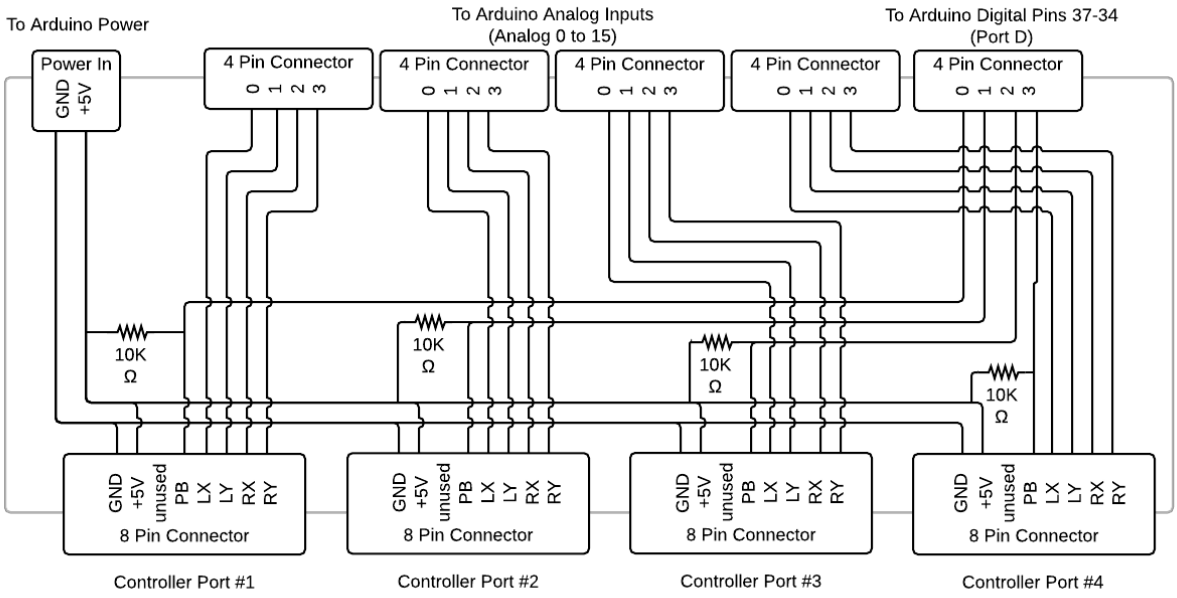


Figure 5: Controller Hub Schematic

3.3 Complete Base Station

The following figures 6 and 7 show the controller hub on the breadboard and the completed base station. The photos also show the ethernet cables which were used to keep the wires together.

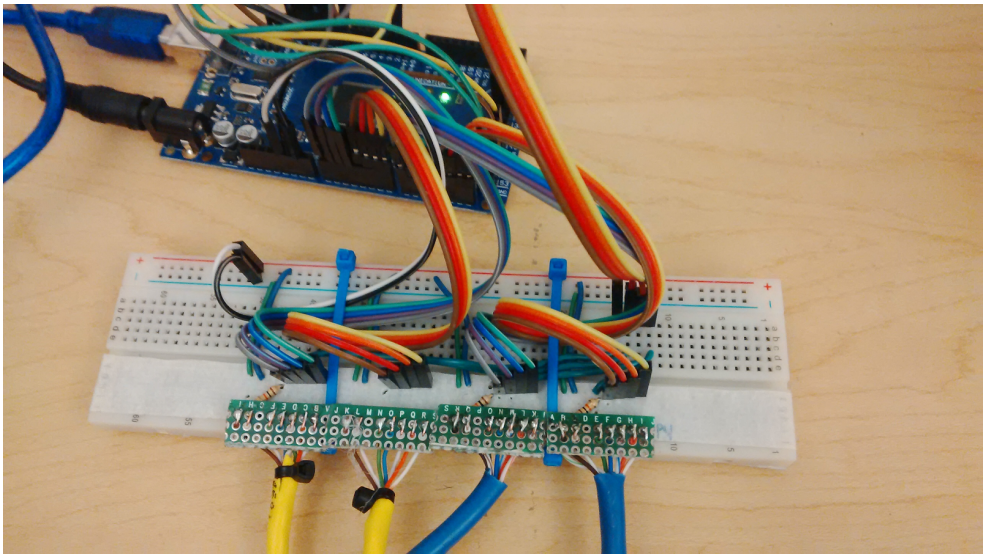


FIgure 6: Complete Controller Hub on Breadboard

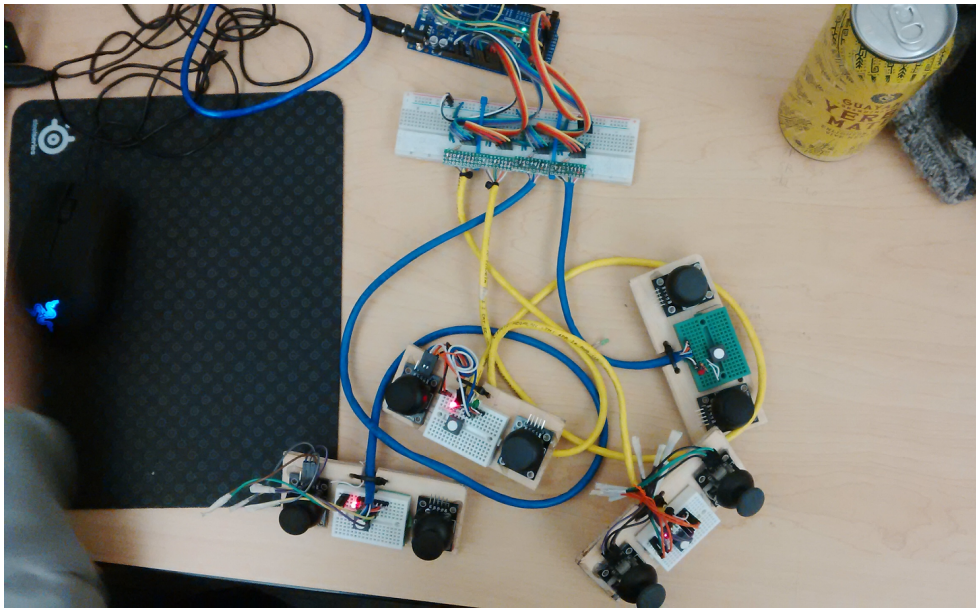


Figure 7: Photo of Complete Base Station

4.0 Roomba Controller

Each Roomba has an ATMega2560 attached to it and communicates with the Roomba using the serial connection with the Open Interface protocol. Attached to the Roomba is a nRF24L01 radio and a Bluetooth Radio which are used to communicate with the Base station control the movement of the Roomba.

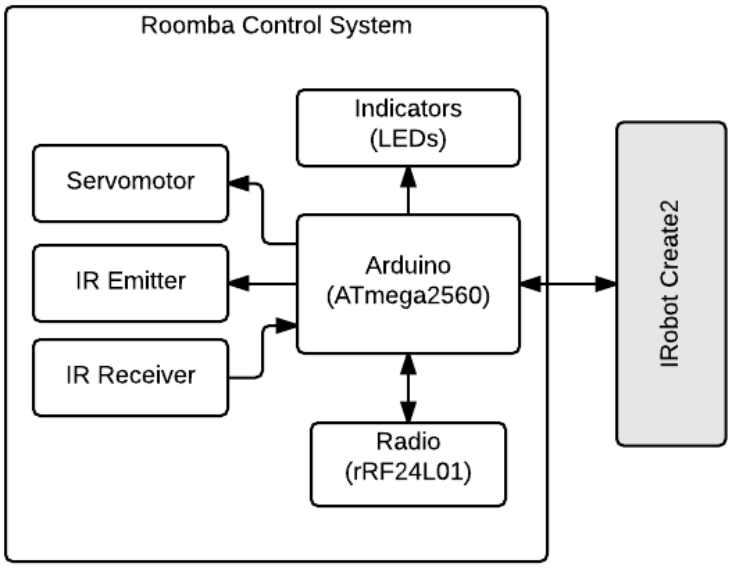


Figure 8: Block Diagram of Roomba Controller

The overall communication between the roombas and base station is shown in figure 9. Every 100ms the base station cycles through to the next joystick and sends a game packet to the roomba with the current input values. The roomba handles the new input parameters and sends back its current state to the base station. The base station then processes the new information (changing roomba status from human to zombie, tracking who has been hit by whom, etc) and waits for the next cycle to begin.

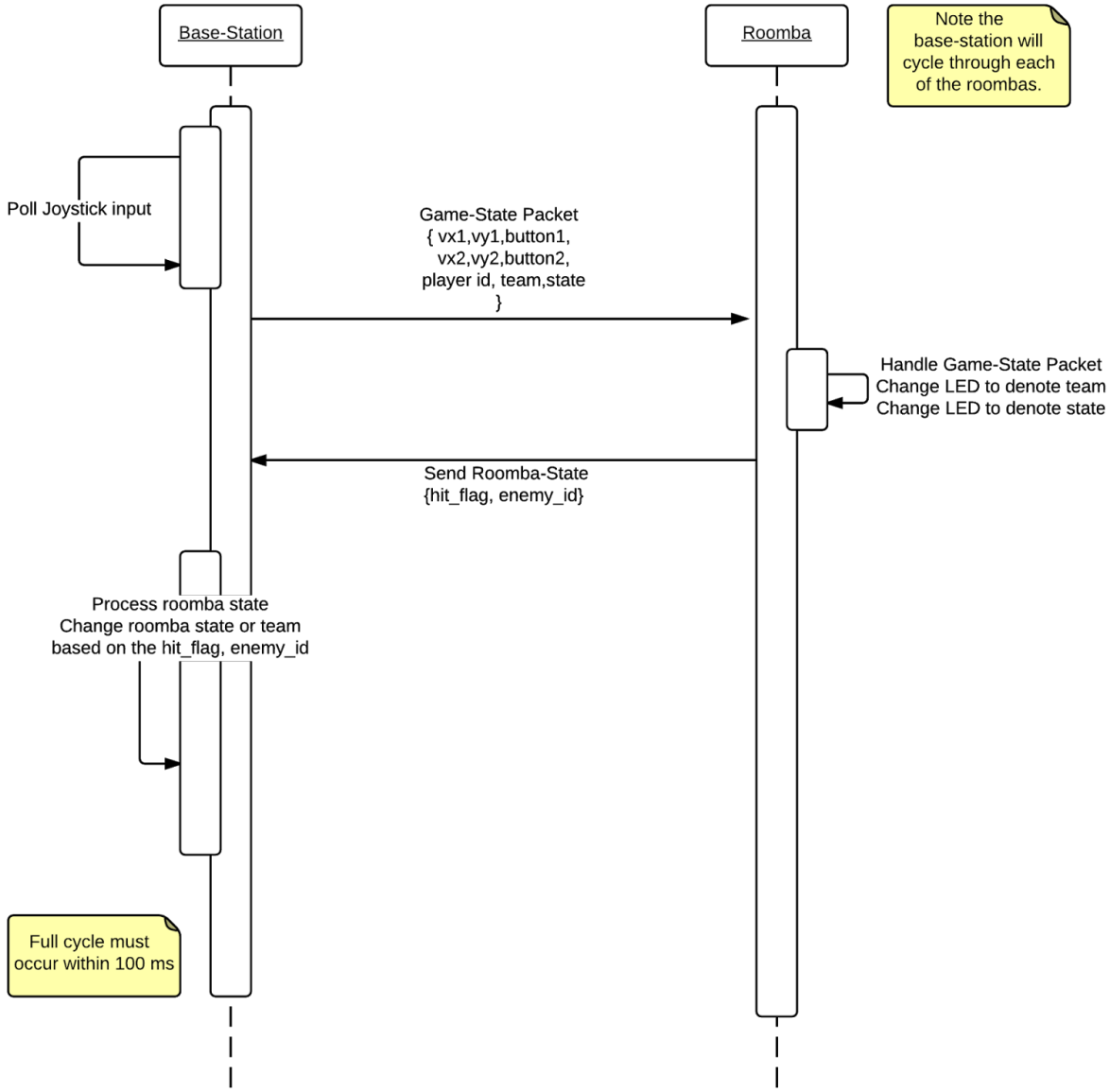


Figure 9: Sequence Diagram of the Overall System Communication

4.1 Roomba LEDs

Each roomba has six LEDs which describe its current state. Three LEDs are used for debugging purposes and they flash in certain situations: receiving a packet from the base station, transmitting a packet, and firing the IR transmitter. These LEDs are a great assistance in determining if each roomba is connected to the base station, and whether commands are

being interpreted correctly; for example, flashing the IR transmitter LED when the user pushes the push button to fire their gun.

An additional three LEDs are used to describe the game state of each roomba. Two LEDs of differing colour are used to show whether the roomba is a human (green LED) or zombie (red LED). The last LED shows any additional affects the player may have: humans can have a shield and zombies can be stunned.
Figure 10 shows the pin mapping layout with the ATMega2560, and table 1 summarizes each LED's details.

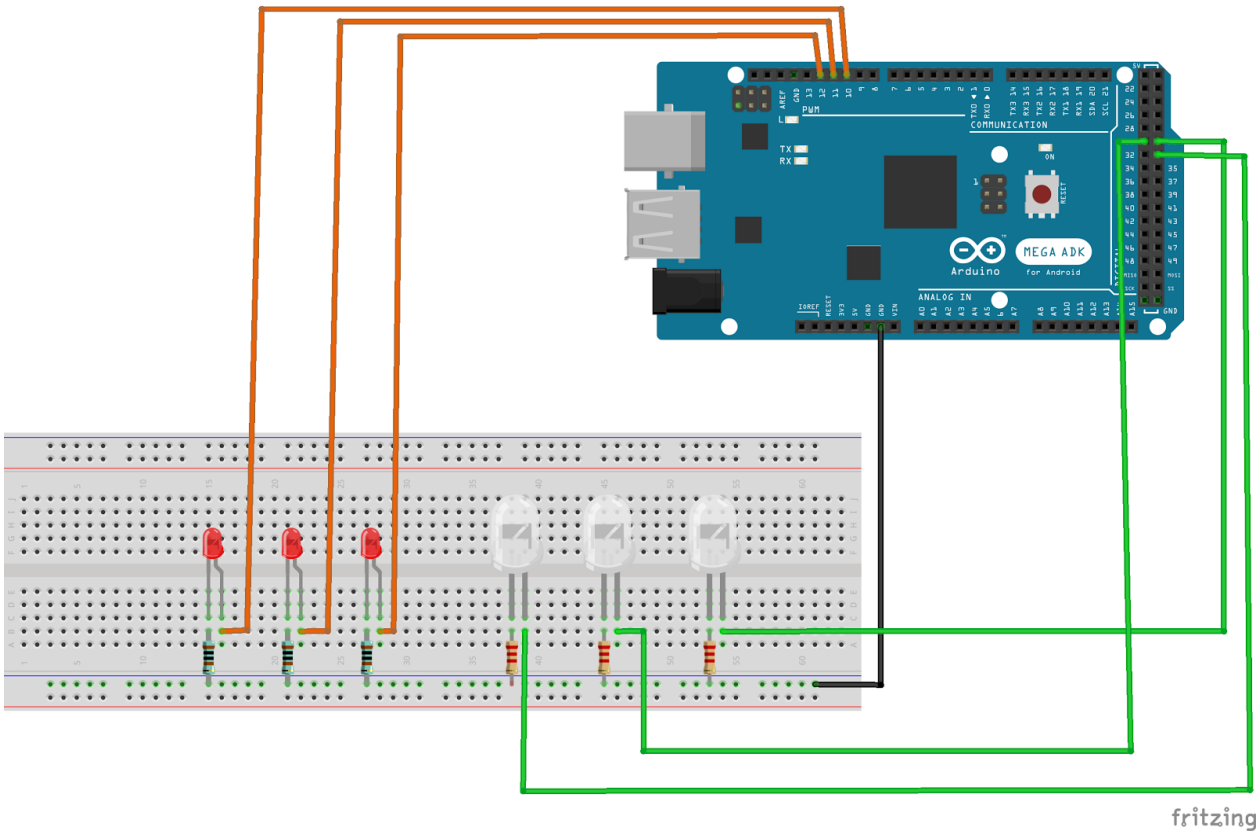


Figure 10: LED Layout of the Roomba Controller

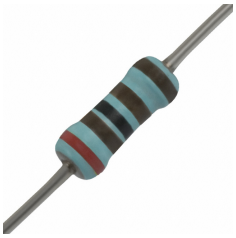
Arduino Pin	LED	Resistor	What it does
Digital 10	Red LED	2.1 KΩ ± 2%	Receives Packet
Digital 11	Red LED	2.1 KΩ ± 2%	Transmits packet
Digital 12	Red LED	2.1 KΩ ± 2%	Hit by IR

Digital 32	White LED	2.2 KΩ ± 5%	Human Pin
Digital 33	Green LED	2.2 KΩ ± 5%	Zombie Pin
Digital 34	Yellow LED	2.2 KΩ ± 5%	Effect Pin

Table 1: LED Pin Layout

4.1.1 Resistors

2.2 KΩ ± 5% tolerance



2.1 KΩ ± 1% tolerance

1.0 KΩ ± 5% tolerance

These resistors are used for reducing the currents that flow through it. The 2.2 and 2.1 KΩ resistors are used for the LEDs so there is not too much current going to them. As well as using 2.2KΩ in conjunction with 1.0 KΩ for the Bluetooth radio to reduce the amount of current going through it.

4.2 nRF24L01 Radio

Our current setup for the Radio communication foundation come from files prepared by Neil and Daniel. The files for the radio communication come from `radio.cpp`, `radio.h`, `packet.h`, `spi.cpp`, `spi.h` files. The radio is connected to the ATMega2560 in the same way as in project 1 phase 2, and it sends a modified packet structure we call the game packet.

When the ATMega2560 first turns on, our r_main task sets up the radio. To begin we power cycle the radio before calling Radio_Init(). We configure the radio with our own address, and we set the speed to 1 MBPS.

All four roomba's have a unique ID between 0 and 3. Using the IDs the base station remembers the radio address of each roomba with the corresponding ID. The base station polls each individual player's joysticks and button every 15 ticks, and sends a packet to their roomba.

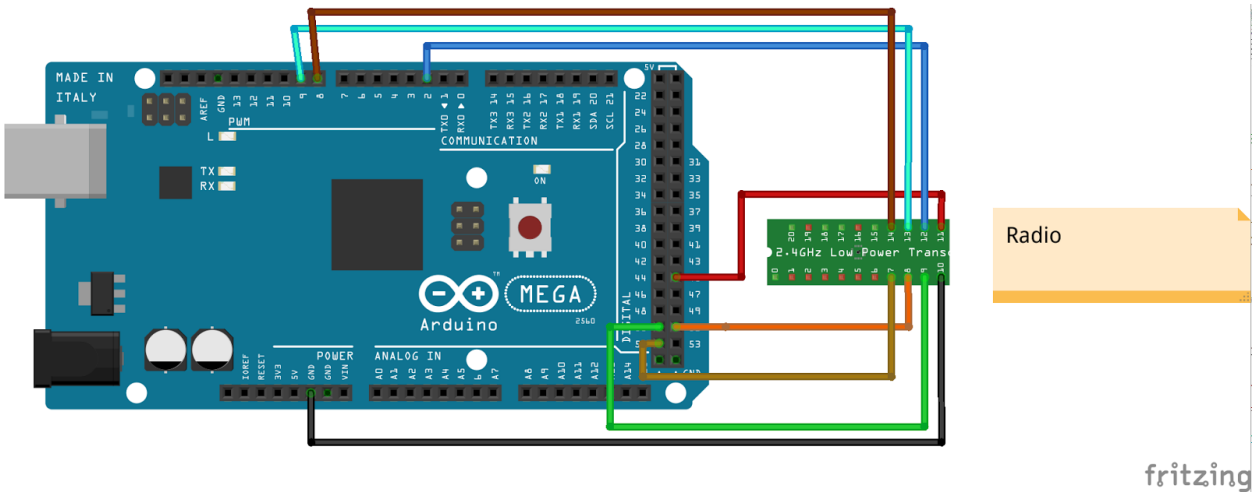


Figure 11: nRF24L01 Radio layout of the Roomba Controller

Pin Label	Pin Connected	Pin Type
CE	Digital 9	Output
CSN	Digital 8	Output
SCK	Digital 52	Output
MO	Digital 51	Output
MI	Digital 50	Input
IRQ	Digital 2	Output
GND	GND	Ground
VCC	Digital 47	Power

Table 2: nRF24L01 Pin Layout

4.2.1 Game Packet

```
typedef struct _game_t {
    uint8_t sender_address[5];
    uint8_t velocity_x;
    uint8_t velocity_y;
    uint8_t servo_velocity_x;
    uint8_t servo_velocity_y;
    uint8_t button;
} pf_game_t;
```

The roomba receives game packets from the base station with information about the joystick velocities, and the push button's state. Servo data has also been included for the group's who added a servo to the top of their roomba.

4.3 Roomba Interface

We interfaced with the roomba using the onboard serial port which is documented on Neil's website nrqm.ca. The roomba designers also have an open interface specification which explains the different commands that can be sent location here. Ultimately we used the provided `roomba.cpp` and `roomba.h` to create command packets and send them to the roomba. The serial port also provides power to the ATMega2560, however, the power provided is 15-17V. In order to lower the voltage to 5V we a 7805 power regulator circuit which is described here.

The UART communication occurred through serial port 1 (TX1/RX1) on the ATMega2560. The provided `uart.cpp` and `uart.h` was used by the `Roomba_Init()` to set a baud rate of 115,200. This function uses the START, and SAFE commands to initialize the roomba as defined in `roomba_sci.h`, and when the communication is finished the STOP command is sent in `Roomba_Finish()`.

4.3.1 Roomba Commands

Once communication has been initialized, commands can be sent to the roomba. The commands include movement, retrieving sensor data, and music. First an OPCODE byte is sent followed by any data necessary to perform the command. The OPCODE can be referenced from the Open Interface specification mentioned above.

4.3.2 Movement

The drive command takes the drive OPCODE followed by 16 bits for the velocity and 16 bits for the rotation. The velocity is a value between [-500, 500] and the rotation is a value between [-2000, 2000].

4.3.3 Sensors

We query the sensors using the sensors OPCODE and a group number. The group represents a number of sensors together as it is not possible to ask for data from only a single sensor. The `roomba_sensor_data_t` struct provided by the `Roomba_UpdateSensorPacket()` is used to get the sensor data.

4.3.4 Ignoring Roomba Failsafes

The roomba automatically detects problems such as stairs or when it is lifted and stops operation. However, this is not desired when a human player is controlling the roomba so we have gotten around this by sending the SAFE command at the beginning of every other roomba command. This makes the roomba ignore the failsafes until it is re-enabled or it is triggered again.

4.4 IR Emitter and Receiver

The IR emitter and IR receiver are used to shoot our gun and be hit by enemy roombas.

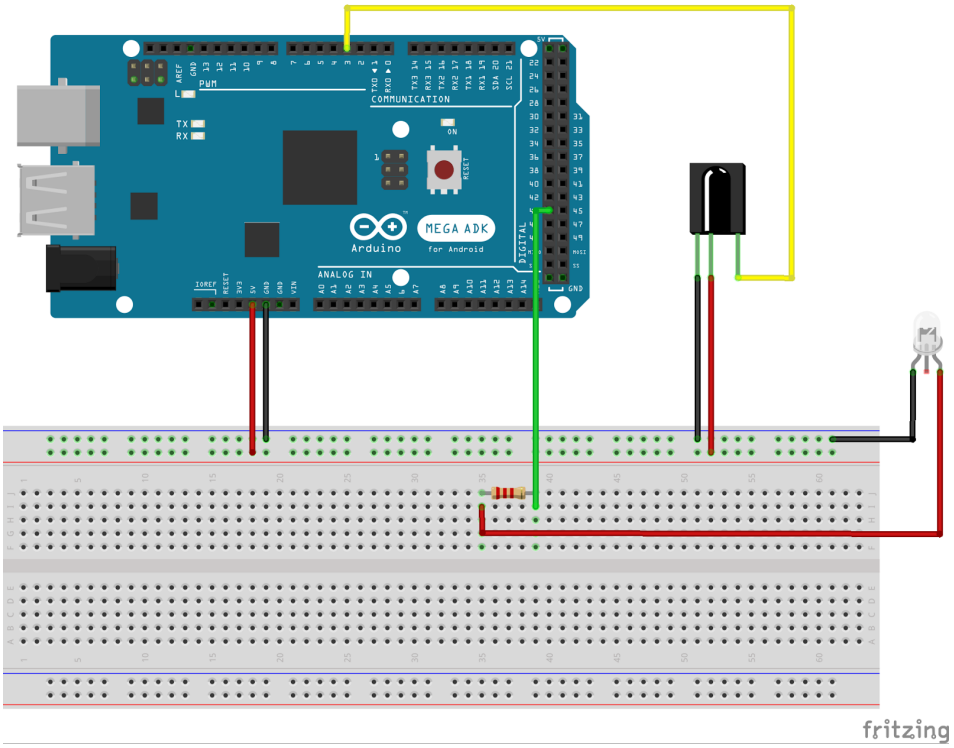


Figure 12: IR Emitter and Receiver layout of the Roomba Controller

Pin Label	Pin Connected	Pin Type
(Receiver) Source	Digital Pin 3	Input
(Receiver) GND	GND	Ground
(Receiver) VCC	5V	Power
(Emitter) Source	2.2 KΩ ± 5% Resistor	Output
(Emitter) GND	GND	Ground
2.2 KΩ ± 5% Resistor	Digital Pin 44	Resistor

Table 3: IR Emitter and Receiver Pin Layout

4.4.1 IR Emitter

To tag the opponent we used the IR emitter to hit an opponent's Roomba IR receiver. The libraries for the IR emitter is located in `ir.cpp` and `ir.h` that was provided to us by Daniel. The IR emitter uses the `IR_transmit()` function located in `ir.cpp`, and when called it will send a 8 bit integer over infrared to the IR receiver. This function is called after every 5 packets from the base station. The IR emitter is found in a Round Robin task for the current_radio_state() function.

4.4.2 IR Transmitter

The IR receiver is to indicate to the players roomba that it has been hit by another roomba and it tells the roomba whether it should change its current state based on the other roomba it hit. It was added by our group members Scott Lowe and Simon Diemert to read the IR value transmitted by IR emitter and determine if the appropriate roomba hit our roomba. When our roomba is hit an interrupt occurs and calls the function `ir_rxhandler()` located in our `main.cpp`. As they implemented this functionality more can be found on their report.

4.5 Roomba Music

We added music to our roomba to play on initialization of the game as well as when the roomba restarts. The music it plays is depends on the team that the roomba is on. The music API is located in the files `roomba_music.cpp` and `roomba_music.h and` was provided by Paul Moon and Jordan Yu's team from our group. More information on it can be found in their report located at https://github.com/paulmoon/csc460. We load the music notes and play the song on the Roomba after it receives a packet from the base station. A flag is used to ensure it only does it once. On initialization a human roomba will play Zelda's Lost Woods and a zombie roomba will play Mario's Death Theme. When a human turns into a zombie it will also play Mario's Death Theme. The figure below shows our block of code to determine what song to load and play.

https://www.youtube.com/watch?v=tz0UCcf1tv8

```
void load_death_music() {
      roomba_music_song_t death_song;
      death_song.len = 0;
      death_song.song_num = 1;
      if (roomba.team == ZOMBIE) {
      Roomba_Music_add_note( & death_song, 83, 9);
      Roomba_Music_add_note( & death_song, 90, 18);
      Roomba_Music_add_note( & death_song, 89, 9);
      Roomba_Music_add_note( & death_song, 89, 9);
      Roomba_Music_add_note( & death_song, 88, 9);
      Roomba_Music_add_note( & death_song, 86, 9);
      Roomba_Music_add_note( & death_song, 84, 9);
      Roomba_Music_add_note( & death_song, 88, 9);
      Roomba_Music_add_note( & death_song, 88, 9);
      Roomba_Music_add_note( & death_song, 84, 9);
      } else {
      Roomba_Music_add_note( & death_song, 89, 9);
      Roomba_Music_add_note( & death_song, 81, 9);
      Roomba_Music_add_note( & death_song, 83, 9);
      Roomba_Music_add_note( & death_song, 89, 9);
      Roomba_Music_add_note( & death_song, 81, 9);
      Roomba_Music_add_note( & death_song, 83, 9);
      Roomba_Music_add_note( & death_song, 89, 9);
      Roomba_Music_add_note( & death_song, 81, 9);
      Roomba_Music_add_note( & death_song, 83, 9);
      Roomba_Music_add_note( & death_song, 84, 9);
      Roomba_Music_add_note( & death_song, 88, 9);
      Roomba_Music_add_note( & death_song, 83, 9);
      Roomba_Music_add_note( & death_song, 84, 9);
      Roomba_Music_add_note( & death_song, 83, 9);
      Roomba_Music_add_note( & death_song, 91, 9);
      Roomba_Music_add_note( & death_song, 88, 9);
      }
      Roomba_Music_load_song( & death_song);
}
```

Figure 13: Playing Music on the Roomba

4.6 Roomba Autonomous Features

In our roomba we added some basic autonomous features that involve using collision detection of the bumpers. Using `roomba.cpp` and `roomba.h` which Neil provided us, we checked the external sensor data of the roomba to determine if the roomba's bumpers have contacted anything. Using the Open Interface Serial Protocol, we receive back from the rooba ten bytes of sensor data. Looking through the first byte of data we see that the first three bits

contain no data while the last five contain the data for Wheeldrop Castor, Wheeldrop Left, Wheeldrop Right, Bump Left, and Bump Right. When any of these sensor get activated the bit corresponding to the sensor is flipped to a 1, the below diagram shows the sensor byte.

Roomba Wheel and Bumper
Sensor Byte

Bit	7	6	5	4	3	2	1	0
Sensor	N/A	N/A	N/A	Wheeldrop Castor	Wheeldrop Left	Wheeldrop Right	Bump Left	Bump Right

Figure 14: Roomba Bumper Sensor Byte

We set a periodic task in our code to check the sensor data every 80 ticks with a worst case execution time of 60 ticks. Our task simply checks the sensor data to see if whether the right or left bumper bit is flipped or not and if it is the user will lose control of the roomba and the roomba will perform its autonomous task. During its autonomous task it will rotate in place for about three radio packets, this way if it collides with anything will turn until it is no longer facing the object. For the roomba to perform its autonomous task we added different modes in the roomba logic, USER mode which allows the users of the roomba to control it and when not in this mode the roomba will act on its own based on its program logic. The video below shows the roomba performing its autonomy.

https://www.youtube.com/watch?v=p1DSJ8STgmE

Special thanks to Mark and Curtis' team for helping us as the code initially not return sensor data through UART library. Thanks to their discovery that the function `Roomba_UART_Init()` in `uart.cpp` allowed us to only send and not receive through the serial port. To remedy this issue we added the following register flags: `UCSR1B = (1<<RXEN1)|(1<<TXEN1)|(1<<RXCIE1); in uart.cpp`.

4.6.1 Autonomous Testing

Manual tests were performed with the roomba by allowing it to hit object and seeing if it would perform the action and disable the input. A logic analyzer was used to see if the periodic task were running properly at the scheduled intervals.

Trace tests were created to mock the bumper sensor byte and see if the roomba responded correctly. A correct response was shown by logging the drive inputs the roomba used for subsequent drive commands. In all cases the correct answer is to stop forward movement and rotate in place away from the obstacle.

4.7 RTOS Task Scheduling

When the ATMega2560 is first powered on we go through the process of initializing all of the components. First the bluetooth radio is initialized by setting up the PIN and giving it a name. Next we initialize the radio by power cycling it and setting the radio's address. Finally we initialize the roomba with `Roomba_Init()` and we turn on the LEDs. All of these operations occur in the first system task `r_main()`.

Next we create three services: `radio_receive_service`, `bluetooth_receive_service`, and `ir_receive_service`. These services encapsulate the process of retrieving packet data from the rest of the system, thus modularizing it. Tasks only need subscribe to these services to be awoken when new data is ready to be processed.

Finally we create three round robin tasks to enable radio, bluetooth, and IR functionality. When a radio packet has arrived it is awoken by the publishing service. The radio immediately begins updating the game state, and if bluetooth is not currently connected, uses the joystick data as input. Like the radio task, the bluetooth task handles bluetooth data using the bluetooth_service. This is done by moving the received velocity and rotation values into global variables which are used as input the next time a roomba drive command is issued. Lastly the IR task is used to determine if we hit another roomba, and whether to fire the gun or not. The IR_receive_service awakens the IR_task when new IR data is available, and if the value is non-zero we know it is the ID of the enemy roomba which we hit with our gun.

4.7.1 Radio Task Overview

The following figure shows the states that the task goes through as it processes a packet.

First it waits for a packet to be published to the radio service and reads the packet data. It retrieves the joystick values and calls Roomba_Drive() with the corresponding velocity and rotation. If a servo is attached then the servo can also be controlled over PWM, however, our roomba does not use a servo.

Finally if any IR fire data is present in the packet, the IR service will be published to so it can provide the data to the IR task.

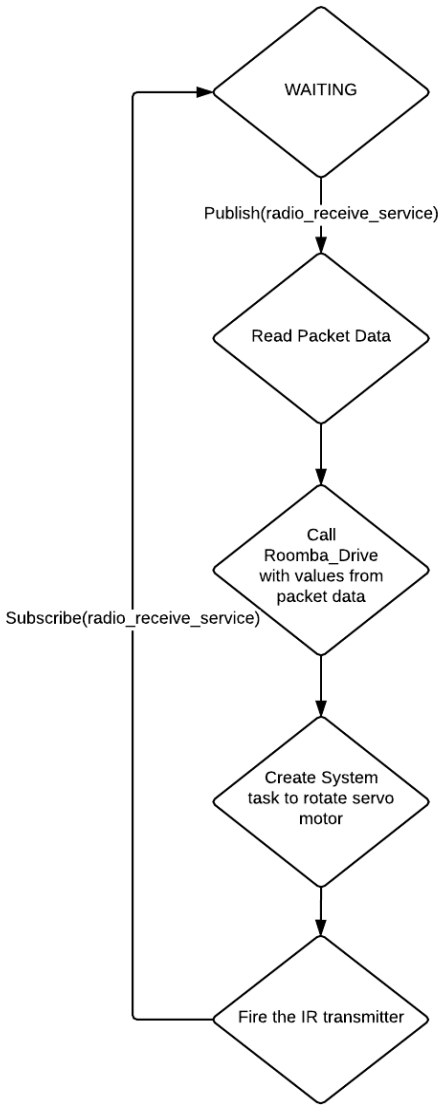


Figure 15: Radio Task State Machine

4.8 HC-06 Bluetooth Radio

A bluetooth radio was added to our roomba controller so that it could receive inputs not from the base station. Ultimately an Android application was written to control the roomba with the player's finger. The bluetooth radio was wired using two resistors to divide the voltage so that it could operate at 3.3V instead of 5V. We used the Serial2 (RX2/TX2) hardware UART port to send and receive data from the bluetooth radio.

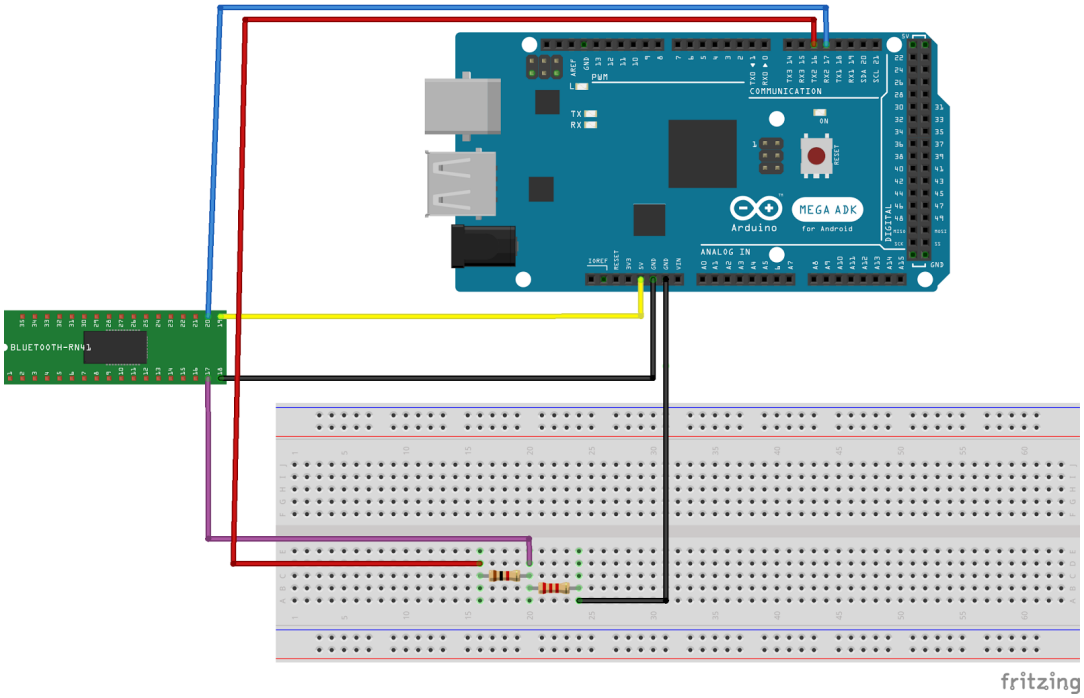


Figure 16: Bluetooth Radio layout of the Roomba Controller

Pin Label	Pin Connected	Pin Type
RXD	Digital Pin 17 (RX2)	Input Pin
TXD	Between the resistors	Output Pin
2.2 KΩ ± 5% Resistor	GND	Resistor
1.0 KΩ ± 5% Resistor	Digital Pin 16 (TX2)	Resistor
GND	GND	GND Pin
VCC	5V	Power Pin

Table 4: Bluetooth Pin Layout

4.8.1 Configuring the Bluetooth Radio

The HC-06 bluetooth radio accepts commands in the form of *AT+COMMAND*. For example, *AT+PIN8888* sets the PIN to 8888, and *AT+NAMERoombaBluetooth* sets the radio's name to RoombaBluetooth when it is discovered by other devices. If the command is successful the radio returns the string *OKsetcommand* such as *OKsetpin*.

The three commands we send to initialize the bluetooth radio are:

1. **AT+BAUD4** - set the baud rate to 9600, returns *OK9600*
2. **AT+NAMERoombaBluetooth** - sets the name, returns *OKsetname*
3. **AT+PIN1111** - sets the pin to 1111, returns *OKsetPIN*

One annoying consequence of this format is that if the bluetooth radio is already connected to a device during the initialization process it will simply send the command as is to the other device instead of configuring itself. However, this can be easily solved by setting the power for the bluetooth radio to a pin other than VCC and power cycling the bluetooth radio when the ATMega2560 first turns on.

4.8.2 Communicating Over Bluetooth

The following tutorial was used to begin our `bluetooth.cpp` and `bluetooth.h` files. It uses SoftwareSerial to communicate with the bluetooth radio on any port that can receive change interrupts.

https://github.com/aron-bordin/Android-with-Arduino-Bluetooth

As per the documentation, not all pins can be used with SoftwareSerial:

Only the following can be used for RX: 10, 11, 12, 13, 14, 15, 50, 51, 52, 53, A8 (62), A9 (63), A10 (64), A11 (65), A12 (66), A13 (67), A14 (68), A15 (69).

However, we ended up switching to HardwareSerial (Serial2) on pins 16, and 17 because we were unable to compile and link the SoftwareSerial library alongside Arduino. Furthermore, the tutorial's string handling was deprecated and produced several warning, so we changed the functions to use char* arrays instead. Please see the appendix for more information on how we added Arduino to our Atmel Studio project.

Reading from the bluetooth radio occurs one byte at a time. In the bluetooth task we check if the bluetooth radio is available and read the latest data if it is. The radio can store up to 64 bytes, so it is important to read the data as soon as possible. Each pair of bytes the radio receives corresponds to the velocity and rotation component of the Roomba_Drive() command. No data is ever sent over the bluetooth radio because there is not currently any information that the roomba needs to send in our system, however, there is opportunity for future work here.

4.8.3 Reading Velocity and Rotation

The velocity and rotation are both read in as byte values from the radio. The bluetooth task alternates between setting the corresponding global variables for velocity and rotation with the

value read by the radio. This enforces a restriction of [0, 255] range for values sent to the radio. To map these values to their proper ranges (velocity: [-500, 500], rotation: [-2000, 2000]) the bluetooth task simply multiplies the recieved value by the correct multiplier. See section 5.2 for more information on what values the Android application sends and how the roomba maps them to the correct range.

4.8.4 Testing the Bluetooth Received Values

Using an Android app called BlueTerm to send a receive bluetooth data, and an Arduino application on our github created to test the bluetooth radio. BlueTerm allows the user to use the Android keyboard and send characters manually. We connected the phone to the roomba using the BlueTerm application and manually sent ascii characters to the roomba and logged the values received over the Serial interface to the Arduino log.

For example, when we send “abcd” in BlueTerm the following log was printed by the ATMega2560 in figure 17.

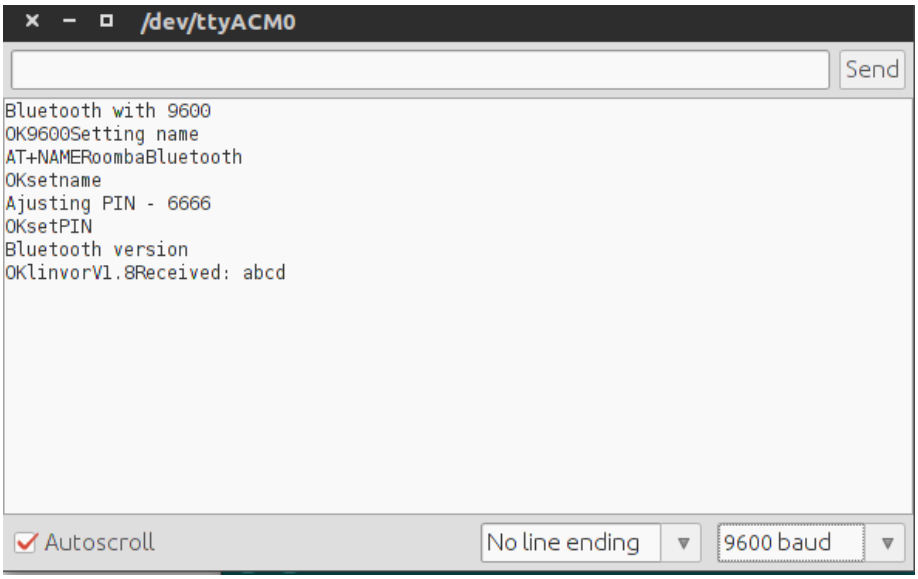


Figure 17: Bluetooth Testing Log

The same procedure was followed using the RTOS and with `trace.cpp`.

4.9 Base Station and Roomba Integration Testing

We used the LEDs on both the base station and on the roomba to test whether packets were being sent correctly or not. Each time the base station sent a packet to an individual roomba it flashed an LED corresponding to roomba it communicated with. The LED only flashes when the packet is acknowledged.

Furthermore, the roomba has its own set of LEDs. Each time it receives a packet it flashes the packet received LED. Likewise each time it transmits a packet it flashes the LED when it is acknowledged by the base station. Finally the IR transmitter LED will blink when the player presses the push button on the controller. This means that the push button was successfully detected as pushed by the base station, the packet arrived at the roomba, and the roomba deciphered the packet correctly to determine that it should fire the IR transmitter. Some of the other LEDs represented whether the zombie or human, using those we could test to see if the base station was properly changing roomba's from human to zombie when the appropriate action was performed.

Using manual methods such as these allowed us to test the system integrated as a whole. It is not possible to write trace() tests with so many moving pieces, and separate systems.

4.10 Completed Roomba

Figure X shows the completed roomba with both radios, the IR transmitter and receiver, and all of the LEDs.

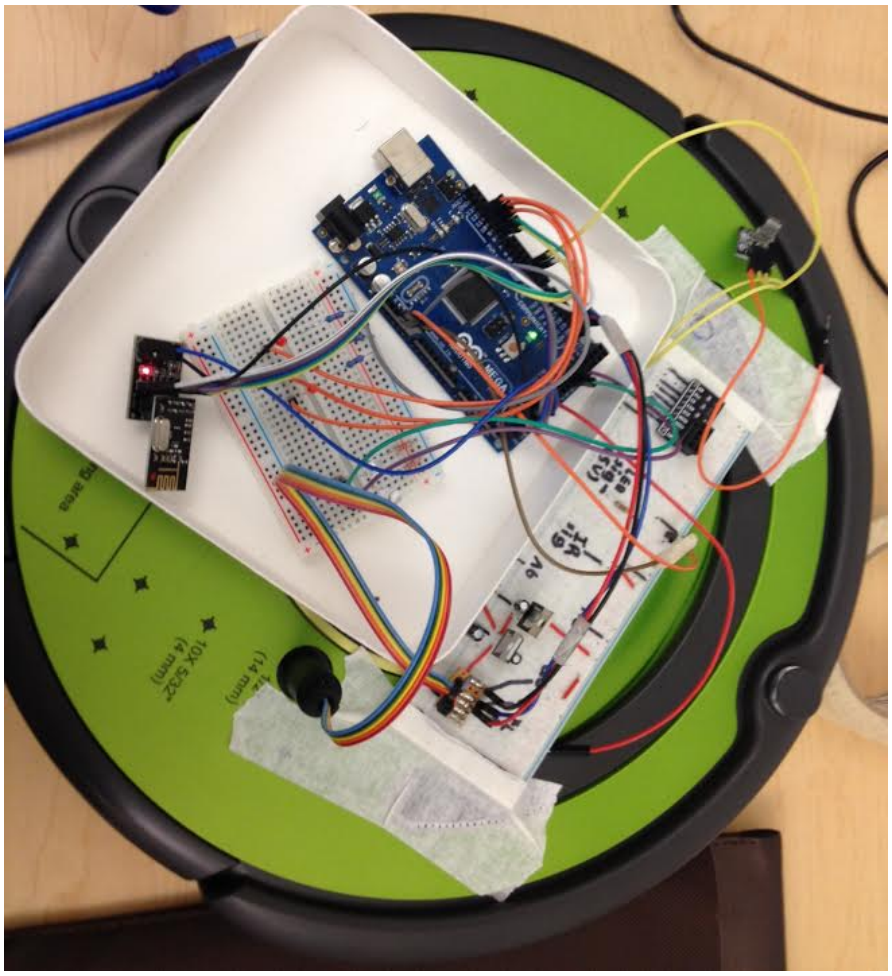


Figure 18: Roomba Setup with Flashing LEDs

5.0 Android App

An Android application was created to send velocity and rotation commands to the roomba. The user drags his/her finger across the screen and the application calculates how many pixels from the center the finger is to the edge of the app's screen. Up/down maps to forwards/backwards and left/right maps to rotate left/rotate right.

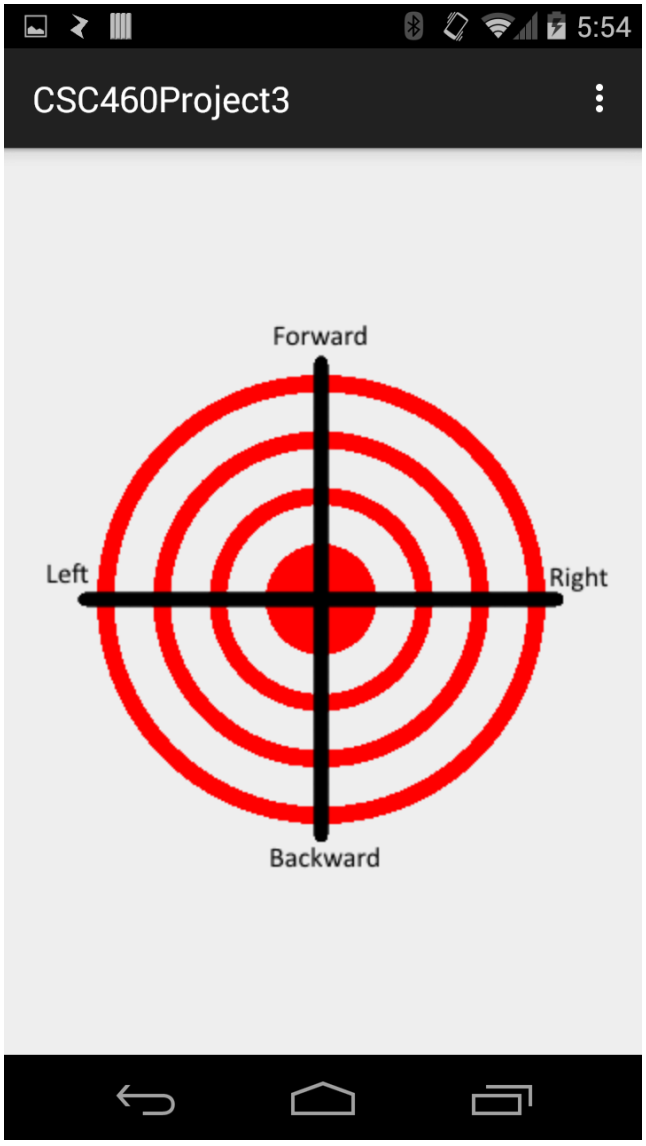


Figure 19: The Completed Android Application's User Interface

The following video demonstrates the Android app controlling the roomba over bluetooth!

https://www.youtube.com/watch?v=0LbMU15FPRY

5.1 Tracking the User's Finger

The main activity in the application implements the View.onTouchListener interface and the corresponding onTouch(View v, MotionEvent event) function. The MotionEvent object provides information about what type of event has occurred:

- ACTION_DOWN
- ACTION_MOVE
- ACTION_UP

On ACTION_DOWN and ACTION_MOVE the app begins tracking the finger and sending roomba commands over bluetooth. The finger's location is mapped to a velocity and rotation such that near the edge is maximum speed/rotation and near the middle means no speed/rotation. A generous space of 50 by 50 pixels at the center is used for stop commands as the finger location is not 100% accurate. Later when the user lifts their finger on ACTION_UP the app also sends roomba stop commands with zero velocity and rotation.

5.2 Mapping the Finger to Roomba Commands

The pixel differences are translated to a value between [65, 85] or the characters ['A', 'U'] in ascii for both roomba velocity and roomba rotation. Since the roomba's bluetooth driver can only read data in 1 byte [0, 256] at a time, it was decided that only ascii characters less than 8 bits should be sent. However, many of the ascii characters are not visible (newlines, bell sound, NUL, etc) so in the interest of ease of debugging and logging, only visible characters were chosen to be sent. In order to map the received byte into the range [-10, 10], 75 was subtracted from the received byte.

Roomba velocity is between [-500, 500]:

```
velocity = (received_byte - 75) * 50;
```

Roomba rotation is between [-2000, 2000]:

```
rotation = (received_byte - 75) * 200;
```

5.3 Using Bluetooth

The app also handles requesting the user to turn on bluetooth if it is not turned on, searching for the roomba's bluetooth radio, and prompting the user to pair with the radio using the correct pin. If the user follows all of the instructions, the app will automatically connect to the

bluetooth radio and begin transmitting roomba commands once the user starts moving their finger. All instructions were created by using Android intents in the main activity.

Turning on Bluetooth **Pairing Prompt**

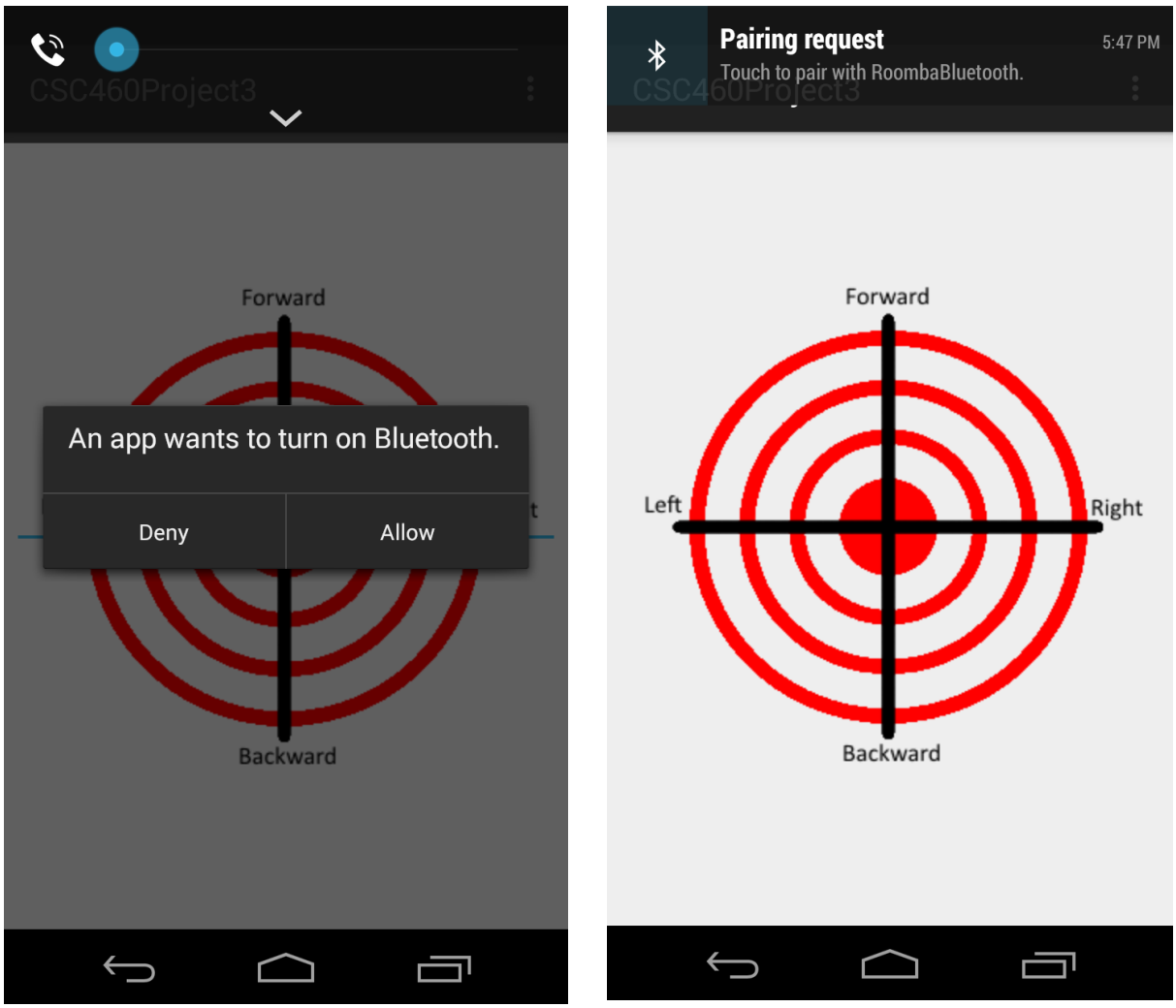


Figure 20: Turning on Bluetooth and Prompting the User to Pair with RoombaBluetooth

Pairing Request **Connecting Bluetooth**

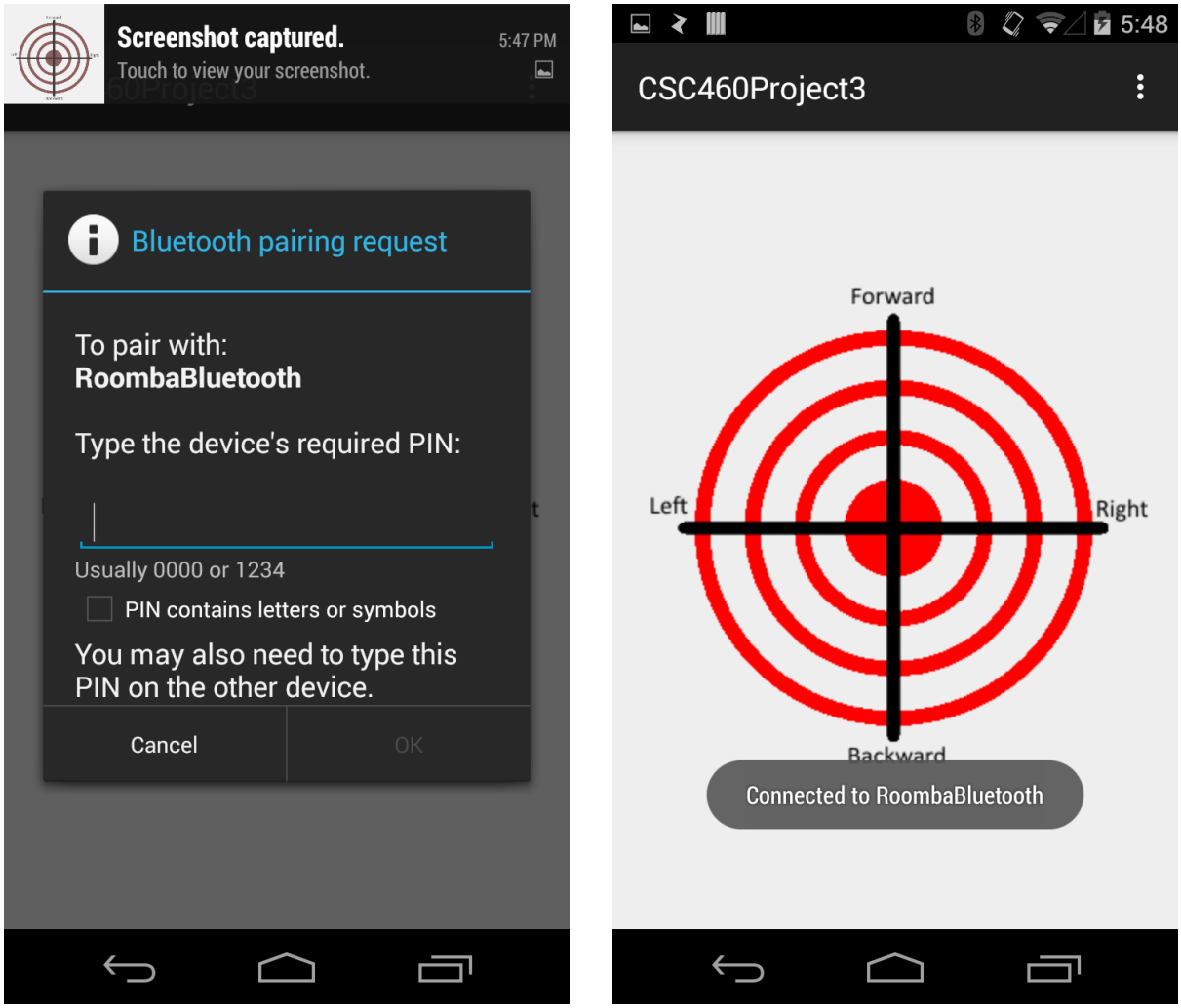


Figure 21: Pairing and Connecting to RoombaBluetooth

5.4 Android Bluetooth Driver

BluetoothChatService.java was used to handle bluetooth communication once the application was connected. It is an open source library under the Apache license provided by developer.android.com for anyone's use.

https://developer.android.com/samples/BluetoothChat/src/com.example.android.bluetoothchat/BluetoothChatService.html

6.0 Conclusion

Project 3 was a very difficult, but fun experience. In the end we managed to build a roomba which played a game of HvZ with three other players. Using an RTOS created in project 2 we built a base station to manage a game of HvZ over a radio, and to control a roomba over the serial port interface. However, from our experience we found several bugs with the RTOS we used and had to make design decisions to avoid periodic tasks which triggered the bugs. For this reason we tried to only use system and round robin tasks. Many times it was difficult to debug whether the base station or roomba was having issues. After we added the three LEDs that showed the current roomba state (receiving a packet, transmitting, or firing IR) it become significantly easier to determine what the issue was when the eventually cropped up. Despite these setbacks we enjoyed the project so much that we even added bluetooth support and created an Android application to control our roomba with our phones. Next we added music that played on initialization depending on the team the roomba player was on, as well as when a human player switched into a zombie. Finally, we added autonomous support so that when the roomba bumped into something it would automatically turn itself around approximately 180 degrees before continuing to play the game.

7.0 Future Work

Now that bluetooth support has been integrated with the roomba, theoretically it is possible to control the roomba over the internet. We have successfully connected to the roomba using bluetooth on a laptop and moved it using the keyboard. The next step would be to add a server to the laptop which a player at home can connect to. The laptop would use the webcam to show the player the “battleground,” and using a REST service on the server and a javascript website the player could use their mouse/keyboard to forward commands to the roomba. Preliminary work has began on this feature and you can track our progress at this github repository.

https://github.com/guand/csc460project3/wiki may provide further documentation on the project.

8.0 Appendix

In order to use Arduino in Atmel Studio there are several steps that need to be taken. This is necessary because we use the HardwareSerial library in Arduino to communicate over UART with the bluetooth radio. Unfortunately the `uart.cpp` file provided for communication with the roomba is hard-coded to only use Serial1 (RX1/TX1). Arduino allows us to use any of the 4 UARTs on the ATMega2560 and we are using Serial2 with the bluetooth radio.

Compiling and Linking Arduino in Atmel Studio

Since Arduino usually provides the main() function, special care needs to be taken to initialize Arduino correctly. “Arduino.h” needs to be included and the init() function must be called before any other Arduino libraries are used. The main reason we wanted Arduino support was to use the Serial libraries: SoftwareSerial and HardwareSerial so that Arduino does the heavy lifting for communication or UART. The following tutorial also assisted us in adding Arduino support to our RTOS.

http://www.engblaze.com/tutorial-using-atmel-studio-6-with-arduino-projects/

Arduino Setup

1. Download and Install the Arduino IDE
2. Compile any Arduino project and copy the core files from the temp directory
   “C:Users\{Your User}\AppData\Local\Temp\buildXXXXXXXXXX.tmp”
3. Find the “core.a” file in this directory and rename is “libcore.a”

Atmel Studio Setup

4. In the C++ Compiler symbol settings, add these symbols
   a. **F_CPU=16000000L**
   b. **ARDUINO=163**
5. In the C++ Compiler directory settings, add these directories for the “Arduino.h” and “pins_arduino.h” files
   a. **{Your Install Location}\Arduino\hardware\arduino\avr\cores\arduino**
   b. **{Your Install Location}\Arduino\hardware\arduino\avr\variants\standard**
6. In the C++ Compiler optimization settings, select “Optimize for size” because Arduino is a large library
   a. Add **-fdata-sections** to the additional compilation flags
7. In the C++ Linker directories settings, add an entry for “core” and add the location of the “libcore.a” file

Congratulations! Arduino can now be used in your Atmel Studio programs.