Project 1: Prototype Base Station and Remote Station

Doug Cox
Department of Computer Science
University of Victoria
dacox@uvic.ca

Richard Muir
Department of Electrical Engineering
University of Victoria
rnmuir@gmail.com

Yaju Jadeja
Department of Electrical Engineering
University of Victoria
yjadeja@uvic.ca

February 18, 2011
# Table of Contents

Table of Contents

List of Figures

List of Tables

1 Introduction

2 Lab Equipment and Materials

3 Development
   3.1 Eclipse .................................................. 3
   3.2 Google Code SVN ........................................... 3
   3.3 Minicom and Realterm .................................... 3

4 Design

5 Components
   5.1 L298 Motor Driver ........................................ 7
      5.1.1 Problems and Solutions ................................ 9
   5.2 SRF04 Ultrasonic Range Finder ............................ 10
      5.2.1 Hardware ............................................. 10
      5.2.2 Software ............................................. 11
   5.3 Logitech Game Pad ........................................ 12
      5.3.1 Problems and Solutions ............................... 12

6 Base Station

7 Remote Station
   7.1 Hardware .................................................. 15
   7.2 Software ................................................... 19

8 Results

9 Conclusion

References

A Full Source Code
   A.1 Base Station ............................................. 27
   A.2 Remote Station ........................................... 29
      A.2.1 Seeeduino Mega ....................................... 29
      A.2.2 Arduino Uno .......................................... 35
List of Figures

1  Block diagram of prototype base/remote stations ......................... 5
2  L298 Motor Driver .................................................................. 7
3  Simple PWM loop .................................................................. 8
4  PWM being generated with analogWrite() ............................... 8
5  SRF05 ultrasonic range finder .................................................. 10
6  Timing diagram for the SRF05 Ultrasonic Rangefinder ............... 10
7  Sonar trigger function ............................................................ 11
8  Reading gamepad data with start byte ..................................... 12
9  Base station code .................................................................. 14
10  Power supply and emergency switch circuit .............................. 15
11  Configuration for fixed 5V LM7805 regulator ............................ 15
12  Power supply and emergency switch circuit: Photo .................. 16
13  Switching a mechanical relay with NPN BJT ............................ 17
14  The remote station from behind .............................................. 17
15  Control loop for Seeeduino on Remote Station ....................... 19
16  The poll input function .......................................................... 20
17  State diagram for control loop of remote station ..................... 21
18  Manual mode ...................................................................... 21
19  Auto mode .......................................................................... 22
20  Formatting strange analog gamepad output .............................. 22
# List of Tables

1. Thorough breakdown of equipment and materials ........................................ 2
2. The logic table for the L298 motor driver ................................................ 7
3. Gamepad Data Encoding ........................................................................... 12
4. Buttons for state transitions ..................................................................... 20
5. Motor control functions ............................................................................ 20
1 Introduction

The purpose of this project was to develop familiarity with the required components and to gain insight into the design process of the final project for SENG 466, an autonomous hovercraft. The hovercraft to be designed and built this semester must be capable of carrying a payload of up to a 1.5kg while consuming no more than 35W of power. Using vertically-positioned propellers, the craft will be lifted by the air captured inside a skirt attached to the underside of the machine.

The craft will be remote-controlled via a joystick connected to a base station. In addition, the hovercraft will also be capable of detecting and avoiding objects in its path autonomously. Eventually, all instrumentation data such as current draw, orientations, and the distance to possible collisions should be relayed wirelessly from the machine itself to a base station to be read on a computer terminal.

This report details the first project milestone. A pair of prototype base and remote stations were designed with the ability to operate in two modes: manual and automatic. The base station was intended to remain near a PC to receive input from a USB gamepad as well as to transmit this data to the remote station via radio. The remote station was set atop a freely rotating platform where it would receive and parse the gamepad data sent from the base station.

In manual mode, the system was designed to respond to human input from a USB gamepad, allowing for manual control of the DC motors in both forward and reverse operation to control the rotational speed of the platform. When running in automatic mode, a simple feedback controller was designed to orient the platform parallel to an anterior surface sufficiently close to the sonar modules; when the closest anterior surface was beyond this threshold, the system was to maintain its rotational position while still engaging both DC motors to provide an equal amount of propulsion and thus, zero net torque about the pivot point.

Both modes incorporated an emergency stop function which would cut the power supply to the higher-current DC motor circuit while shielding the lower-current signal circuit from the back-EMF generated by the sudden stop of the DC motors. The implementation chosen to attain this goal was through the use of an N-channel MOSFET (IRF740) switching a 5V DC driven relay (833H-1C-C) put in place to break the high-current circuit powering the two DC motors.
## 2 Lab Equipment and Materials

The following list of equipment was used to implement the prototype base and remote stations in preparation for our autonomous hovercraft:

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Seeestudios Seeeduino Mega (Atmel ATmega 1280 chipset)</td>
</tr>
<tr>
<td>2</td>
<td>Arduino Uno (Atmel ATmega328P chipset)</td>
</tr>
<tr>
<td>1</td>
<td>Solarbotics L298 Compact Motor Driver Kit, preassembled</td>
</tr>
<tr>
<td>2</td>
<td>DC motors with propellers</td>
</tr>
<tr>
<td>2</td>
<td>Devantech SRF04 Ultrasonic Range Finders</td>
</tr>
<tr>
<td>2</td>
<td>Nordic Semiconductor nRF24L01+ MiRF-v2 2.4GHz Radio Modules, with antennae</td>
</tr>
<tr>
<td>1</td>
<td>Logitech USB Gamepad</td>
</tr>
<tr>
<td>1</td>
<td>piece of .060' styrene board, cut into a 25cm x 25xm square</td>
</tr>
<tr>
<td>2</td>
<td>Styrofoam pieces, cut to form ballasts for motors and sonar modules</td>
</tr>
<tr>
<td>1</td>
<td>3.5 hard-drive, disassembled to reveal the freely rotating ball-bearing platter</td>
</tr>
<tr>
<td>1</td>
<td>8HH3-1C-C 5V DC Relay</td>
</tr>
<tr>
<td>1</td>
<td>Red LED</td>
</tr>
<tr>
<td>1</td>
<td>2N3904 NPN Transistor</td>
</tr>
<tr>
<td>1</td>
<td>1N4004 Diode</td>
</tr>
<tr>
<td></td>
<td>Various resistors</td>
</tr>
<tr>
<td>1</td>
<td>+7.2V DC battery pack</td>
</tr>
<tr>
<td>1</td>
<td>LM7805 3-Terminal +5V DC regulator</td>
</tr>
<tr>
<td>1</td>
<td>9V DC battery, regulated to 5V DC</td>
</tr>
<tr>
<td>3</td>
<td>Standard 830 point solderless-breadboard</td>
</tr>
<tr>
<td>1</td>
<td>Soldering iron, 60/40 rosin core solder</td>
</tr>
<tr>
<td>1</td>
<td>Glue gun</td>
</tr>
<tr>
<td>1</td>
<td>Styrofoam cutter</td>
</tr>
<tr>
<td></td>
<td>Various lengths of solid wire</td>
</tr>
</tbody>
</table>

Table 1: Thorough breakdown of equipment and materials
3 Development

3.1 Eclipse

The Integrated Development Environment (IDE) selected to aid in the design of the software required to drive the project was Eclipse:

http://www.eclipse.org

Eclipse allowed for the creation of several different project types; for the purposes of this project, cross-target applications were required for the AVR-based Arduino microcontrollers. Creating, compiling and executing the code for the various software-related components of this project was performed through the use of the Eclipse IDE.

In addition to the IDE on its own, the AVR-Plugin for Eclipse was also utilized as it allowed the end-user to easily build and flash the raw .HEX machine code to the Arduino microcontrollers from inside the Eclipse IDE. The Arduino core libraries were also necessary as they contained all the necessary header and function files required to permit the use of the pre-built Arduino sketch functions.

In order to reduce the filesize of the compiled binaries, the static libraries for both the Uno and Seeeduino microcontrollers were created first and then linked to their respective projects; this reduced the filesize of the Seeeduino binaries by approximately 30% and that of the Uno by roughly the same amount by allowing the compiler to take only what was needed from the Arduino core as opposed to including everything whether it was necessary or not.

3.2 Google Code SVN

Subclipse is an Eclipse Plug-in that provides access to Subversion (SVN) software source code repositories. SVN repositories allow developers to maintain current and historical versions of files as well as allow multiple team members to commit changes without overwriting another members’ work.

Google provides free project source code hosting through its Google Code service. Using the Subclipse Plug-in, revisions to the project hosted by Google Code could be made from within Eclipse quickly and easily. The current project source code can be viewed at:

http://code.google.com/p/seng466p1

3.3 Minicom and Realterm

Minicom is a text-based terminal emulator for use on Unix-based operating systems, while RealTerm is intended for use on Windows-based machines. minicom was chosen for its simplicity and ease of use; once a configuration has been saved for a particular serial port it can be opened with relative ease to receive and transmit serial data.
When operating on a Windows-based operating system, RealTerm was already available on the laboratory PC’s and provides all the required functionality needed for the scope of this project; one very useful feature of RealTerm is the ability to easily select the format data received is displayed in such as ANSI, ASCII, hexadecimal, etc. RealTerm also allows the user to spy on communications performed over serial ports by other applications, which proved to be very useful in monitoring the gamepad data sent between the PC and the base station’s Uno microcontroller.
4 Design

In order to implement the Arduino microcontrollers, the electronic components and the sensors required in the design of the project prototype, a thorough analysis and understanding of the related technical documents was required. Specifically, reference material from the Arduino web page and Neil MacMillan’s laboratory guide was researched in order to develop a solution. The final design consisted of two aspects: software and circuit design.

The base station was centrally controlled by an Arduino Uno microcontroller. The USB gamepad was connected to a PC running a Python script which parsed and transmitted the data via a serial port to the Arduino Uno located on the base station. The remaining Arduino Uno and Seeeduino microcontrollers were both placed on the remote station; the Seeeduino was selected as the primary controller for the system due to its larger array of input/output (IO) pins. Both the base and remote stations were linked via the nRF24L01+ radio modules, each communicating with their respective Arduino Uno microcontrollers via their Serial Peripheral Interface (SPI) busses.

![Block diagram of prototype base/remote stations](image)

Figure 1: Block diagram of prototype base/remote stations

The Arduino Uno placed aboard the base station transmitted the parsed USB gamepad data to the remote stations Uno in packets of 18 bytes using the nRF24L01+ radio modules.
Once the gamepad data had been received at the remote station, the Uno forwarded the information to the primary controller, the Seeeduino microcontroller. This was accomplished via the Universal Asynchronous Receiver/Transmitter (UART) interfaces through the Rx/Tx pins present on each board.

Once the gamepad information was received by the Seeeduino microcontroller it was parsed accordingly and commands were issued to adjust each of the two motors speed through pulse width modulation (PWM) and in turn, the platforms rotational speed. In addition to controlling motor speed, an emergency stop command could also be issued which would break the higher current circuit controlling the motors. A signal originating from the Seeeduino being amplified by a 2N3904 transistor was used to activate a relay which would open the motor circuit and thus stop the motors without affecting the signal circuit.

A feedback controller was designed to orient the platform parallel to a surface in front of the sonar modules. The distance reading from each sonar module was compared and the corresponding motors were triggered by the Seeeduino microcontroller accordingly to bring the distance readings closer to an equivalent value, thus rendering the platform parallel to the anterior surface detected by the sonar modules. When the distance to the closest surface the sonar range finders could detect was greater than 50cm, the platform’s rotational speed was kept constant at zero by producing an equal amount of propulsion from each DC motor.

The block diagram in figure 1 illustrates the interconnection of all components:
## 5 Components

### 5.1 L298 Motor Driver

<table>
<thead>
<tr>
<th>ENABLE</th>
<th>L1</th>
<th>L2</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>OFF</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>H</td>
<td>OFF</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>L</td>
<td>OFF</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
<td>OFF</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
<td>BRAKE</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>H</td>
<td>FORWARD</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
<td>BACKWARD</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
<td>BRAKE</td>
</tr>
<tr>
<td>PWM</td>
<td>L</td>
<td>L</td>
<td>PULSE-BRK</td>
</tr>
<tr>
<td>PWM</td>
<td>L</td>
<td>H</td>
<td>FWD-SPD</td>
</tr>
<tr>
<td>PWM</td>
<td>H</td>
<td>L</td>
<td>BCK-SPD</td>
</tr>
<tr>
<td>PWM</td>
<td>H</td>
<td>H</td>
<td>PULSE-BRK</td>
</tr>
</tbody>
</table>

Table 2: The logic table for the L298 motor driver

The L298 motor driver is capable of simultaneously running two DC motors. Three digital I/O pins from the microcontroller are required to drive each DC motor, at least one of which should have PWM (pulse width modulation) capabilities. The logic table for each side of the motor driver is as follows:

![L298 Motor Driver](image)

Figure 2: L298 Motor Driver

PWM is when a digital I/O pin is used to fake an analog voltage by rapidly switching output from HIGH to LOW. The percentage of time a pin is HIGH versus the time it is LOW is referred to as the duty cycle. There are a variety of methods for using PWM on the Arduino:
1. Creation of a timed loop which manipulates the output of a PWM pin
2. Use of the Arduino library function analogWrite()
3. Direct manipulation of the microcontrollers registers

Our first step was to see if we could successfully use PWM to change the speed by pulsing the ENABLE pin. To test this we wrote a simple loop that sets a digital pin high and low continuously, with a set delay after each write to the pin. Please see figure 3 for details.

```cpp
int ENABLE = 13;
pinMode(ENABLE, OUTPUT);

for (;;) {
    digitalWrite(ENABLE, HIGH);
    delay(9);
    digitalWrite(ENABLE, LOW);
    delay(1);
}
```

Figure 3: Simple PWM loop

Obviously, this is not an optimal solution. By using delay() to generate a PWM signal, we are making it so that no other work can be done in between digitalWrite() operations. This is not good when the goal is to make a system that can react in real time. However, the experiment served its purpose and we were able to ascertain that PWM was indeed possible with our hardware.

```cpp
int ENABLE = 13;
pinMode(ENABLE, OUTPUT);

... void write_duty_cycle(int pin, double percent) {
    analogWrite(pin, (int)(255*percent));
}
```

Figure 4: PWM being generated with analogWrite()

This brings us to the second method of generating a PWM signal, the analogWrite() function. this function allows us to generate a PWM signal without wasting precious clock cycles waiting. The first argument to analogWrite(int arg1, int arg2) is an integer specifying the pin we are writing to (which must be set as OUTPUT with the pinMode() call), like digitalWrite(). The second argument is an integer in the range of 0 to 255. When this argument is 0, the function performs like digitalWrite(OUTPUT, LOW), and when it is 255 it behaves like digitalWrite(OUTPUT, HIGH). The analogWrite() function is defining the duty cycle of the pin as the second argument’s percentage of 255.
\[ \text{duty cycle} = \frac{\text{arg}^2}{255} \]

For an example of a PWM signal being generated by this method, see figure 4.

The third way of achieving PWM is to directly manipulate the registers that the analogWrite() function is modifying behind the scenes. This method gives the programmer complete control over the PWM signal being output. We opted to use analogWrite() to control our PWM signal due to its ease of use and simplicity. In future projects we will investigate using the third method for more advanced control.

5.1.1 Problems and Solutions

Many groups noticed an annoying hum when using the analogWrite() function to output a PWM signal. The fix for this was to modify the registers used by the function to increase the frequency that the microcontroller uses for PWM. Our group did not make this change as the hum never seemed to be an issue for us.

However, PWM is where we made our first mistake—not paying attention to the documentation. After our initial success with the PWM code in figure 3 we tried using analogWrite() to generate the PWM signal, and were dismayed to find that it would not work. After some time, we finally figured out that this was because we had not called the init() function, thinking that it was a suggested function much like setup(). As it turns out, init() is actually a very important Arduino function that, among other things, initializes the timer registers on the microcontroller which the analogWrite() function depends on.
5.2 SRF04 Ultrasonic Range Finder

5.2.1 Hardware

We tried using the LV-MaxSonar-EZ4 and SRF05 sonar modules for range detection. Initially we selected EZ4 because it drew less current, was easy to interface using Analog input or UART, and provided continuous free run and triggered operation for range reading. Even though the EZ4 had easier interfaces and more features, it lacked the close range accuracy and provided a low resolution of only 2.5cm step.

![SRF05 ultrasonic range finder](image)

Figure 5: SRF05 ultrasonic range finder

In contrast, the SRF05 modules draws more current, does not offer UART or Analog output or a continuous running mode. But what it lacks in features, it makes up in terms of...
providing accuracy with closer detection range and higher resolution. It offers 3cm to 3m
detection through a TTL pulse with output width proportional to the range. The SRF04
will output an echo pulse on pin (2) after the trigger pulse input pin(3) is held high for more
than 10 uS and brought low.

5.2.2 Software

To achieve this protocol, we trigger each sonar for 11 $\mu$s and then measure the length of the
response from the sonar. To take care of the pulse measuring we used the Arduino function
pulseIn(echoPin, HIGH). This function waits while echoPin is LOW, and then returns with
the amount of time it goes HIGH for. To avoid any interference, each sonar is triggered
and measured consecutively, rather than simultaneously.

```
void sonar_cm ()
{
digitalWrite(TR1, HIGH);
delayMicroseconds(11);
digitalWrite(TR1, LOW);
DUR1 = pulseIn(ECH1, HIGH) / 58;
digitalWrite(TR2, HIGH);
delayMicroseconds(11);
digitalWrite(TR2, LOW);
DUR2 = pulseIn(ECH2, HIGH) / 58;
}
```

Figure 7: Sonar trigger function

We had discarded the alternative sonar module early on due to poor performance at a close
range. Upon initial testing, this sonar module was much more accurate. However, as the
project neared its completion, it became clear that our remote station had problems sensing
distances of more than about 20cm, a far cry from the advertised 3 meters.

Other groups seemed to be doing quite well at long range—we are still unsure of what caused
the issue with our system. However, we do have some theories. If the two sonars were not
perfectly aligned, it is possible that as the distance grows, the angular discrepancy between
each sonar module also grows. At some point, this would exceed the allowed tolerance
value. Also, apparently there is an input capture pin on the Seeeduino which can improve
performance greatly.
5.3 Logitech Game Pad

<table>
<thead>
<tr>
<th>Gamepad Input</th>
<th>Byte(s)</th>
<th>Python Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buttons 1-12</td>
<td>1-12</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>X and Y axes</td>
<td>13-16</td>
<td>[-99, 99]</td>
</tr>
<tr>
<td>Directional Hats</td>
<td>17-18</td>
<td>[-1, 0, 1]</td>
</tr>
</tbody>
</table>

Table 3: Gamepad Data Encoding

River Allen had previously written a Python-based joystick monitor that incorporates the pySerial and pyGame libraries in order to facilitate the capture of gamepad data. More specifically, each of the 12 buttons are represented as Boolean types (1 for pressed, 0 for not pressed), the two axes as analog values ranging from -99 to 99 for both the X and Y directions, and the "hat" (directional-pad) as signed integers ranging from -1 to 1. (See 3).

The script sends the packed gamepad data to the wireless base station over UART on a virtual COM port.

5.3.1 Problems and Solutions

The gamepad represented an interesting set of problems. The python script written by River Allen quite specifically packs up the 18 bytes of gamepad data followed by the two dummy characters "\r\n"
in the final packet.

We would then have the wireless base station open a serial connection to the PC to read in the packet, terminated by these two characters. However, this did not seem to be working at all. Frustrated, we tried printing out the packets we were receiving where we noticed a kind of cascading effect in the stream of print statements, indicative of the packet length being off by one or two bytes every iteration.

```
int data[PACKET_LEN];
if (Serial.available() >= PACKET_LEN) {
    while (Serial.read() != START_BYTE);
    for (int i = 0; i < PACKET_LEN; i++) {
        data[i] = Serial.read();
    }
    Serial.flush();
}
```

Figure 8: Reading gamepad data with start byte

We reached the conclusion that the packet is most definitely 18 bytes long, and this resolved our problem. However, now that our packets had no identifying terminator characters, we
would need a way to recognize the start of a packet. We decided to send the byte FF as a start byte immediately before sending the gamepad packet. Our code for reading the packets can be seen in figure 8.

Later in the project, we noticed that the robot would react as if the emergency stop button had been pushed even though it had not. We eventually realized that the hats on the gamepad were actually sending the values 255 sometimes instead of the expected -1, 0, or 1 (a signed value problem). This meant that if we ever pressed a hat button, the microcontroller now started reading in a new packet from the middle of a packet, which would in turn cause random values to be interpreted as the stop button.
6 Base Station

The purpose of the base station was to accept gamepad data transferred over UART by the python script and relay it to the remote station via the wireless radio. The base station sits in a continuous loop waiting for input from the gamepad, and as soon as it has a packet it sends it to the remote station, where it is decoded and acted upon.

```c
for (;;) {
    while (Serial.available() < 18);
    for (int i = 0; i < 19; i++) {
        data[i] = Serial.read();
    }
    sum = 0;
    for (int i = 0; i < 18; i++) {
        sum += data[i];
    }
    if (sum > 0) {
        packet.type = MESSAGE;
        memcpy(packet.payload.message.address, my_addr, RADIO_ADDRESS_LENGTH);
        packet.payload.message.messageid = 42;
        memcpy(packet.payload.message.messagecontent, data, 18);
        Radio_Transmit(&packet, RADIO_WAIT_FOR_TX);
    }
}
```

Figure 9: Base station code
7 Remote Station

7.1 Hardware

Initially we used the regulated +5V output from the L298 Motor driver to power the Seeeduino and the Uno base station. When both motors are running at an idle speed (about 60% of maximum), the system is drawing about 0.5A from the 7.2V main battery.

![Power supply and emergency switch circuit](image1)

A simple switching circuit was added to shut off the power to the L298 in case of an emergency. As that would also shut off the onboard regulator, we added a LM7805 regulator to supply the +5V to Seeeduino and Arduino Uno remote station.

![Configuration for fixed 5V LM7805 regulator](image2)
Minimum of 70mA is required to energize the input coil of the SRD-S-105D relay and this was too much to source from the Seeeduino. Switching circuit was initially constructed using a NPN BJT (2N3904) to provide the required current to the relay input. The input coil resistance of the relay is $70\,\Omega$ at $20^\circ\,\text{C}$. The particular relay was tested to energize at a minimum current of 61mA. Therefore the collector current can be calculated as:

$$I_C = \frac{V_{CC} - V_{CE\,\text{SAT}}}{R_{COIL}} = \frac{5.0V - 0.3V}{70\,\Omega} = 67.14\,mA$$

The power dissipated by the transistor is calculated as

$$0.3V \times I_C = 20\,mW$$

To calculate the base resistor, we used the minimum $\beta(hfe)$ of 100 and $I_C = 70\,mA$

We chose a lower base resistor of $5k\Omega$ to increase the $I_B$ even further. This ensured that the 2N3904 operated well within the saturated region. Also a $5k\Omega$ resistor was much easier to find in the parts bin. A $100k\Omega$ pull down resistor was also added across the base and emitter to keep the base low when the digital output pin from the Seeeduino is floating.

$$R_B = \frac{V_{CC}}{I_C} = \frac{5V}{70mA} = 71.2\,\Omega$$
Figure 13: Switching a mechanical relay with NPN BJT

A Freewheeling snubber diode 1N40001 was added to absorb the voltage spike caused by the input coil to protect the transistor. The diode is reverse biased during normal current flow and at the moment the transistor turns off, the kickback current gets absorbed back into the coil via the forward biased diode. L298 is connected to the N/O terminal of the relay and is electrically isolated from low current +5V digital side.

To further enhance our learning experience and test all the components in our parts kit, we tried using the n-channel mosfet (IRF740) to do the switching. It was much easier to use the IRF740 because there were almost no calculations required and the datasheet was quite straightforward. Unlike the BJT, where we calculated the base current required to saturate the transistor, MOSFET could be simply switched by controlling the gate voltage. MOSFET gate could directly be connected to the digital control pin which eliminated the need for R3 (base resistor used for BJT).

Figure 14: The remote station from behind

Apart from the cost disadvantage, we found many advantages of using the MOSFET as a switch compare to a BJT transistor. The 2N3904 BJT used earlier consumed more power
(20mW) in the on-state. It could not be saturated with less than 0.3 voltage drop, and it took roughly 10mA of base current to achieve it.

Even with IRF740s poor on state $R_{DS}$ of $0.48\Omega$, the power consumed is only

$$I_D^2 \times R_{DS} = (70mA)^2 \times 0.48\Omega = 2.3mW$$

Also the IRF740 gate will draw negligibly low amount of current from the Seeeduino to accomplish this task. Even though the switching times were not critical for this application, the MOSFET also provided faster switching times compare to the BJT.
7.2 Software

The remote station is implemented on two microcontrollers—the Arduino Uno, and the Seeeduino Mega. The Seeeduino Mega is responsible for running the main control loop of the program, while the Uno is responsible for receiving wireless communications from the base station and then relaying them to the Seeeduino.

```c
int main(void)
{
    init();
    setup();
    _delay_ms(5000);
    STATE = AUTO;
    LAST_STATE = AUTO;
    for (;;) {
        poll_input();

        if (STATE == AUTO) {
            if (STATE != LAST_STATE)
                Serial.println("AUTO MODE");
            auto_mode();
        }
        else if (STATE == MANUAL) {
            if (STATE != LAST_STATE)
                Serial.println("MANUALMODE");
            manual_mode();
        }
        else if (STATE == HALT) {
            if (STATE != LAST_STATE)
                Serial.println("EMERGENCY STOP");
            emerg_stop();
        }

        LAST_STATE = STATE;
    }

    return 0;
}
```

Figure 15: Control loop for Seeeduino on Remote Station

The remote station is controlled by a simple polling loop. At each iteration of the loop, the Uno that is relaying the wireless communication is polled for new data by calling poll_input().

The function poll_input() checks if there is new data from the gamepad, and if there is, reads a whole packet. Depending on which buttons have been pressed, the variable STATE changes. The possible states and their corresponding buttons are:
```c
void poll_input(void)
{
    int data[PACKET_LEN];
    if (Serial.available() >= PACKET_LEN) {
        while (Serial.read() != START_BYTE);
        for (int i = 0; i < PACKET_LEN; i++) {
            data[i] = Serial.read();
        }
        Serial.flush();
    }
    if (data[0] == 1)
        STATE = AUTO;
    else if (data[2] == 1)
        STATE = MANUAL;
    else if (data[1] == 1)
        STATE = HALT;
    else
        STATE = LAST_STATE;
    DRIVEL = data[13];  //ANALOG INPUT
    DRIVER = data[15];
}
```

Figure 16: The poll input function

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>1</td>
</tr>
<tr>
<td>EMERGENCY STOP</td>
<td>2</td>
</tr>
<tr>
<td>MANUAL</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: Buttons for state transitions

The program starts in automatic mode, and switches to either manual control or emergency stop mode. When the user specifies emergency stop mode, the Seeeduino sends a signal to the relay circuit seen in 12 which cuts the power to the l298 and the DC motors.

The following functions have been implemented to handle the differential drive of the two DC motors connected to the L298 motor driver:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>void motor_for(int dir);</td>
<td>Forward speed</td>
</tr>
<tr>
<td>void motor_idle(int dir);</td>
<td>Idle speed</td>
</tr>
<tr>
<td>void motor_acc(int dir, int duty);</td>
<td>Acceleration</td>
</tr>
</tbody>
</table>

Table 5: Motor control functions

These functions are used to specify the behaviour of the manual mode and automatic mode states.
The variables DRIVEL and DRIVER from the poll_input() function hold the value of the left and right analog sticks for manual mode, and are positive integers less than 100. Manual mode uses these values to calculate the duty cycle as the percent of the difference between the manual idle speed and the manual forward speed.

```
void manual_mode(void)
{
    digitalWrite(RELAY, HIGH);

    if (DRIVEL >= 0 && DRIVEL < 100 && DRIVER >= 0 && DRIVER < 100) {
        motor_acc(LEFT, (DRIVEL*5)/2);
        motor_acc(RIGHT, (DRIVER*5)/2);
    }

delay_ms(100);
}
```

Figure 18: Manual mode
For auto mode, we implemented a simple feedback loop controller. Every iteration of the program, the sonar is queried to see if the difference between the two is within the acceptable range. Based on the cardinality of the difference of the two distances returned by the sonar, the motors independently switch between their idle speed and their full speed to correct the rotation of the platform.

For future projects we plan on looking into a PID (proportional, integral, derivative) controller. This much more advanced form of control will help the system prevent overshoot and settle faster.

The other part of the remote station is the Arduino Uno, which receives gamepad transmis-
sions from the base station. The Uno then parses the incoming packets for useful information, and passes it along to the Seeeduino, the main controller of the base station. One problem we ran into is that the values for the analog sticks did not match what was supposed to come out of the python script. While pressing the analog sticks up, to speed up each of the motors, instead of building from 0 to 100, it goes from 0 to 255 to 155. While this seems suspiciously like a problem with signed and unsigned integers, we could not resolve this problem under that assumption. Instead, we used some math to make the outputs from the gamepad match the expected outputs before being passed to the Seeeduino.

If an analog is being pressed up, calculate the actual value as follows:

\[ val = 100 - (val - 155) \]

Otherwise, the value is zero. This way, the gamepad only affects the motors if the analog sticks are being pressed forwards, or if a button is pressed to change the state.
8 Results

While the entire prototype system designed for this phase of the hovercraft was tested exhaustively, certain elements of our design proved to our group that we will need to redesign some of the key aspects of our assembly. In particular, focus will be directed upon the parsing of human input, motor control and the detection of foreign objects within the line-of-sight of our craft.

Technical difficulties involving the gamepad data caused a minor setback on the day of our demonstration; we noticed that the remote station based on the rotating platform was unexpectedly powering down as well as intermittently ceasing to propel itself from one of our DC motors. In addition to this, the left-hand motor appeared to have lost a significant amount of power and was unable to cancel out the propulsion from the right-hand motor, even when being instructed to operate at full-power while the right-hand side was left idling.

Upon closer inspection, the wiring for the motor in question was determined to be faulty and was re-soldered with stranded wire to increase the Controller/Motor connections surface area and thus its reliability. Although this provided a slight boost in morale and fixed the issue of the intermittent operation of this motor, it did not explain the poor performance of that half of the platforms propulsion.

To compensate for the lack of power from the left-hand motor, the allowable range for its duty-cycle was increased by approximately 40%. This value was chosen after several trial-and-error test runs of trying to keep the platform stable when idling in automatic mode while permitting the left-hand side to overtake the right-hand sides idle speed via manual control; higher values would result in instability in automatic mode while lower values would not allow the manual control of the left-hand motor to overtake the idle propulsion of the right-hand motor.

After several hours of debugging and testing, an issue in the design of the gamepad parsing brought itself to our attention. We had chosen a start byte of value 255 to inform the remote station to begin examining the parsed gamepad data transmitted via the radio link; coincidentally, this value represents the value -1 in twos complement form. Since the hat on the gamepad transmitted a value of -1 on each axis when either down or left was depressed, any erroneous input, either human or electrical in nature, would put the microcontroller out of sync and the array of 18 bytes sent as instructions was not properly handled. This explained the random trips of our relay circuit which resulted in the craft powering itself down unexpectedly.
9 Conclusion

At the end of the day, our group was satisfied that we had reached the goals set out for us in this project. The main purpose of this task was to obtain familiarity and become comfortable with the tools provided in order to become prepared for our final project. While our demonstration of this project milestone may not have been perfect, its purpose was as a learning stage and that is precisely what occurred.

The lessons learned in this phase of the autonomous hovercraft design have shown our group which areas we need to focus on in order to successfully create our final product. Our group will take the knowledge we have gained through the problems we discovered and put it to use in finding a more robust solution to our design problem.

Specifically, further attention will be made to our software design in parsing the gamepad data, as well as making further attempts to isolate the power and signal circuits from each other as much as possible to avoid interference amongst the components used in our next design.
References

http://nrqm.ca/mechatronics-lab-guide/


[3] Steven F. Barrett, Daniel J. Park,  
*ATMEL AVR Microcontroller Primer*.

http://download.savannah.gnu.org/releases/avr-libc/  
avr-libc-user-manual-1.4.7.pdf.bz2

[5] Stan Gibilisco,  
*Teach Yourself Electricity and Electronics*.


[7] *7805 5V DC Regulator*.  

[8] *833H-1C-C 5V DC Relay*.  
http://www.datasheetdir.com/833H-1C-C+Solid-State-Relays
A Full Source Code

A.1 Base Station

```c
#include <WProgram.h>
#include <stdio.h>
#include "radio.h"
#include "packet.h"
#include "delay.h"

#define TOGGLE 127
#define START 255

uint8_t my_addr[RADIO_ADDRESS_LENGTH] = {0xE5, 0xE5, 0xE5, 0xE5, 0xE5};
uint8_t other_addr[RADIO_ADDRESS_LENGTH] = {0xE6, 0xE6, 0xE6, 0xE6, 0xE6};
radiopacket_t packet;

void radio_rxhandler(uint8_t pipenumber)
{
    // this station doesn’t receive anything.
}

extern "C" void __cxa_pure_virtual()
{
    // cli();   // disable interrupts
    for(;;);   // do nothing until hard reset
}

void setup()
{
    Serial.begin(38400);
    Radio_Init();
    Radio_Configure_Rx(RADIOPIPE0, my_addr, ENABLE);
    Radio_Configure(RADIO2MBPS, RADIOHIGHESTPOWER);
    Radio_Set_Tx_Addr(other_addr);
}

int main()
{
    init();
    _delay_ms(5000);
    setup();

    uint8_t data[18];
    uint8_t buttons[4] = {0, 2, 13, 15};

    int sum;

    for(;;) {
        while (Serial.available() < 18);
```
for (int i = 0; i < 19; i++) {
    data[i] = Serial.read();
}

sum = 0;

for (int i = 0; i < 18; i++) {
    sum += data[i];
}

if (sum > 0) {
    packet.type = MESSAGE;
    memcpy(packet.payload.message.address, my_addr, RADIO_ADDRESS_LENGTH);
    packet.payload.message.messageid = 42;
    memcpy(packet.payload.message.messagecontent, data, 18);
    Radio_Transmit(&packet, RADIO_WAIT_FOR_TX);
}

return 0;
A.2 Remote Station

A.2.1 Seeeduino Mega

```c
#include "WProgram.h"
#include "avr/interrupt.h"
#include "delay.h"

#define AUTO 0
#define MANUAL 1
#define HALT 2
#define LEFT 0
#define RIGHT 1
#define START_BYTE 255
#define PACKET_LEN 18
#define FULL 150
#define IDLE 110
#define MAN_FULL 255
#define MAN_IDLE 0
#define RELAY 4
#define TOL 3
#define CUTOFF 25

//FUNCTION PROTOTYPE
void setup (void);
void sonar_cm (void);
void motor_for (int dir);
void motor_idle (int dir);
void motor_stop (int dir);
void motor_acc (int dir, int duty);
void emerg_stop (void);
void auto_mode (void);
void manual_mode (void);
void poll_input (void);

//GLOBAL PARAMS
int EN1, L11, L12;
int EN2, L21, L22;
int TR1, ECH1;
int TR2, ECH2;
volatile int LAST_STATE, STATE;
volatile unsigned int DRIVEL, DRIVER;
volatile int DUR1, DUR2;

int main (void)
{
    init ();
    setup ();
    _delay_ms (5000);
    STATE = AUTO;
    LAST_STATE = AUTO;
    for ( ; ; ) {
```

poll_input();

if (STATE == AUTO) {
    if (STATE != LAST_STATE)
        Serial.println("AUTO MODE");
    auto_mode();
}
else if (STATE == MANUAL) {
    if (STATE != LAST_STATE)
        Serial.println("MANUAL MODE");
    manual_mode();
}
else if (STATE == HALT) {
    if (STATE != LAST_STATE)
        Serial.println("EMERGENCY STOP");
    emerg_stop();
}
LAST_STATE = STATE;
}
return 0;
}

void setup(void)
{
    Serial.begin(38400);
    pinMode(RELAY, OUTPUT);
    digitalWrite(RELAY, HIGH);
    //Motor 1
    EN1 = 13;
    L11 = 11;
    L12 = 12;
    pinMode(EN1, OUTPUT);
    pinMode(L11, OUTPUT);
    pinMode(L12, OUTPUT);

    //Motor 2
    EN2 = 10;
    L21 = 9;
    L22 = 8;
    pinMode(EN2, OUTPUT);
    pinMode(L21, OUTPUT);
    pinMode(L22, OUTPUT);

    //Sonar 1
    TR1 = 3;
    ECH1 = 2;
    pinMode(TR1, OUTPUT);
    pinMode(ECH1, INPUT);
```cpp
// Sonar 2
TR2 = 6;
ECH2 = 7;
pinMode(TR2, OUTPUT);
pinMode(ECH2, INPUT);
STATE = AUTO;
LAST_STATE = AUTO;

void sonar_cm()
{
digitalWrite(TR1, HIGH);
delayMicroseconds(11);
digitalWrite(TR1, LOW);
DUR1 = pulseIn(ECH1, HIGH) / 58;
digitalWrite(TR2, HIGH);
delayMicroseconds(11);
digitalWrite(TR2, LOW);
DUR2 = pulseIn(ECH2, HIGH) / 58;
}

void motor_for(int dir)
{
if (dir == LEFT) {
analogWrite(EN1, FULL+95);
digitalWrite(L11, LOW);
digitalWrite(L12, HIGH);
} else if (dir == RIGHT) {
analogWrite(EN2, FULL);
digitalWrite(L21, LOW);
digitalWrite(L22, HIGH);
}

void motor_idle(int dir)
{
if (dir == LEFT) {
analogWrite(EN1, IDLE+80);
digitalWrite(L11, LOW);
digitalWrite(L12, HIGH);
} else if (dir == RIGHT) {
analogWrite(EN2, IDLE);
digitalWrite(L21, LOW);
digitalWrite(L22, HIGH);
}

void motor_stop(int dir) {
if (dir == LEFT) {
```
analogWrite(EN1, LOW);
digitalWrite(L11, LOW);
digitalWrite(L12, LOW);
} else if (dir == RIGHT) {
analogWrite(EN2, LOW);
digitalWrite(L21, LOW);
digitalWrite(L22, LOW);
}

void poll_input(void)
{
    int data[PACKET_LEN];
    if (Serial.available() >= PACKET_LEN) {
        while (Serial.read() != START_BYTE);
        for (int i = 0; i < PACKET_LEN; i++) {
            data[i] = Serial.read();
        }
        Serial.flush();
    }
    if (data[0] == 1)
        STATE = AUTO;
    else if (data[2] == 1)
        STATE = MANUAL;
    else if (data[1] == 1)
        STATE = HALT;
    else
        STATE = LAST_STATE;

    DRIVEL = data[13];
    DRIVER = data[15];
}

void auto_mode(void)
{

digitalWrite(RELAY, HIGH);
sonar_cm();

int DIFF = DUR1 - DUR2;

if ((DUR1 > 0 && DUR1 < CUTOFF) && (DUR2 > 0 && DUR2 < CUTOFF)) {
    if ((DIFF > 0 && DIFF < TOL) || (DIFF < 0 && DIFF > (-1)*TOL))
    {
        motor_idle(LEFT);
        motor_idle(RIGHT);
    } else if (DIFF > 0) {

motor_idle(RIGHT);
motor_for(LEFT);
}
else if (DIFF < 0) {
    motor_idle(LEFT);
motor_idle(RIGHT);
}
else {
    motor_idle(LEFT);
motor_idle(RIGHT);
}
_delay_ms(50);

void manual_mode(void)
{
    digitalWrite(RELAY, HIGH);
    if (DRIVE > 0 && DRIVE < 100 && DRIVER > 0 && DRIVER < 100) {
        motor_acc(LEFT, (DRIVE*5)/2);
motor_acc(RIGHT, (DRIVER*5)/2);
    }
    _delay_ms(100);
}

void motor_acc(int dir, int duty)
{
    if (dir == LEFT) {
        analogWrite(EN1, duty);
digitalWrite(L11, LOW);
digitalWrite(L12, HIGH);
    } else if (dir == RIGHT) {
        analogWrite(EN2, duty);
digitalWrite(L21, LOW);
digitalWrite(L22, HIGH);
    }
}

void emerg_stop(void)
{
    //motor_stop(LEFT);
    //motor_stop(RIGHT);
digitalWrite(RELAY, LOW);
}

extern "C" void __cxa_pure_virtual()
{
    cli(); // disable interrupts
for (; ;) : // do nothing until hard reset
A.2.2 Arduino Uno

```c
#include "radio.h"
#include "packet.h"
#include "WProgram.h"
#include "delay.h"

uint8_t station_addr[5] = { 0xE6, 0xE6, 0xE6, 0xE6, 0xE6 };
radio_packet_t packet;
volatile uint8_t rxflag = 0;

extern "C" void _cxapure_virtual()
{
  cli();
  for (;;) ;
}

void radio_rxhandler(uint8_t pipenumber)
{
  rxflag = 1;
}

void setup()
{
  sei();
  Serial.begin(38400);
  pinMode(13, OUTPUT);
  Radio_Init();
  Radio_Configure_Rx(RADIOPIPE0, station_addr, ENABLE);
  Radio_Configure(RADIO_2MBPS, RADIO_HIGHEST_POWER);
}

int main()
{
  init();
  delay_ms(5000);
  setup();
  for (;;) {
    if (rxflag) {
      if (Radio_Receive(&packet) != RADIO_RX_MORE_PACKETS) {
        rxflag = 0;
      }
      if (packet.type != MESSAGE) {
        continue;
      }
    }
  }
```

35
Radio_Set_Tx_Addr(packet.payload.message.address);
Serial.write(255);
for (int i = 0; i < 18; i++) {
    if (i == 13 || i == 15) {
        if ((int)packet.payload.message.messagecontent[i] > 155 &&
            packet.payload.message.messagecontent[i] < 255) {
        } else {
            packet.payload.message.messagecontent[i] = (int)(0);
        }
    } else {
        if (i == 12 || i == 14) {
            packet.payload.message.messagecontent[i] = (int)(0);
        }
        Serial.println((int)packet.payload.message.messagecontent[i]);
    }
} Serial.println(" ");
packet.type = ACK;
if (Radio_Transmit(&packet, RADIO_WAIT_FOR_TX) == RADIO_TX_MAX_RT) {
} else {
};
return 0;