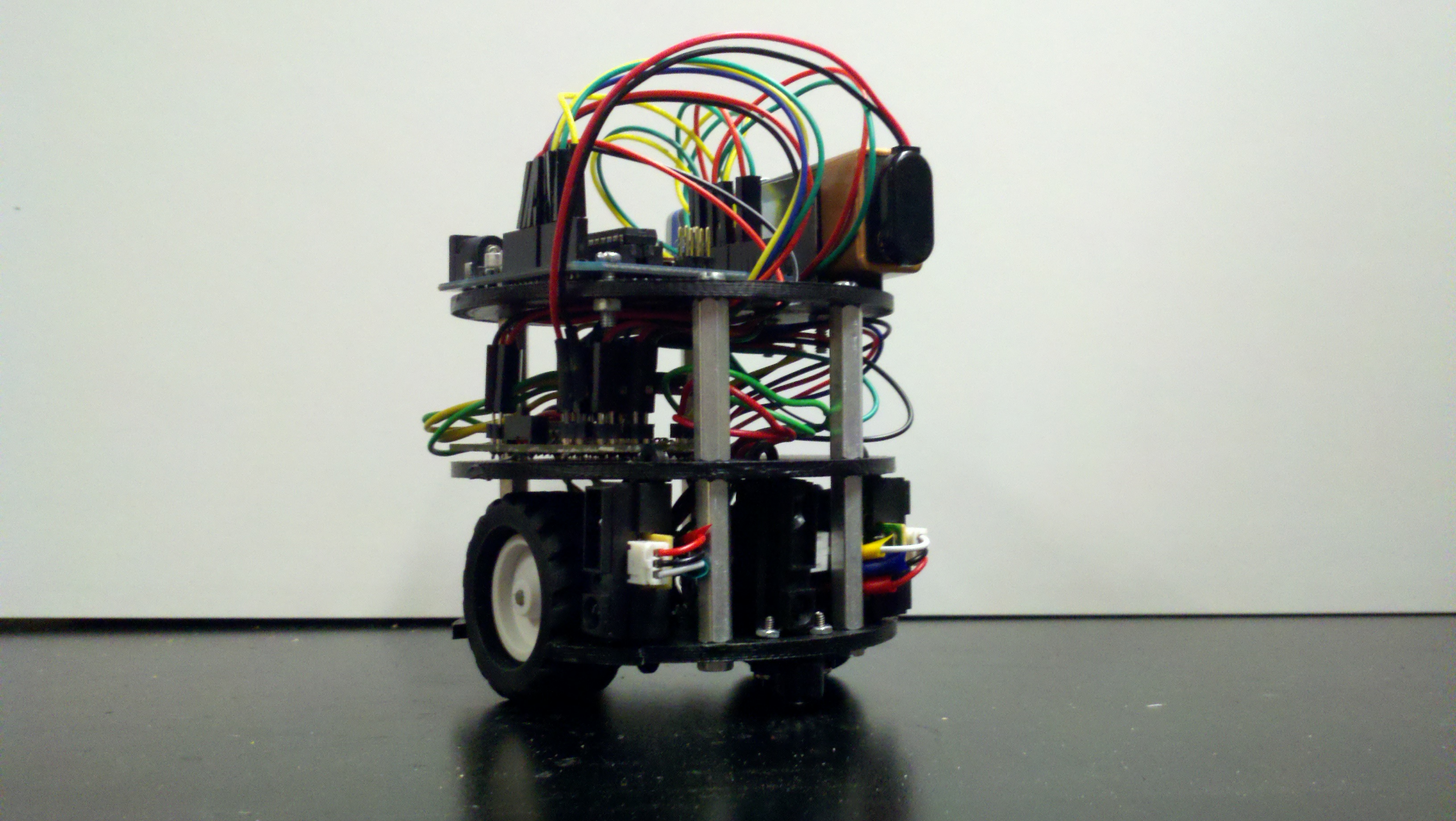
Department of Electrical Engineering

Presents





*A micromouse design project for Robotics and Embedded Systems Design Track of Senior Design*

**Senior Design Final Report**

**Fall 2010- Spring 2011**

Project Advisor: *Prof. Xiao, J.*

Team Members:

*David Sohmers*

*Ivan Mechkanski*

*FaradjineLaurore*

*Abdullah Jubayer*

*Rahul Saggi*

**Table of Contents**

**Chapter 1: Hardware Design**

Introduction\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Microcontroller\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Motor\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Encoder\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Sensors\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Control\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Chassis\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

H-Bridge\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Printed Circuit Board\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Chapter 2: Algorithm Development**

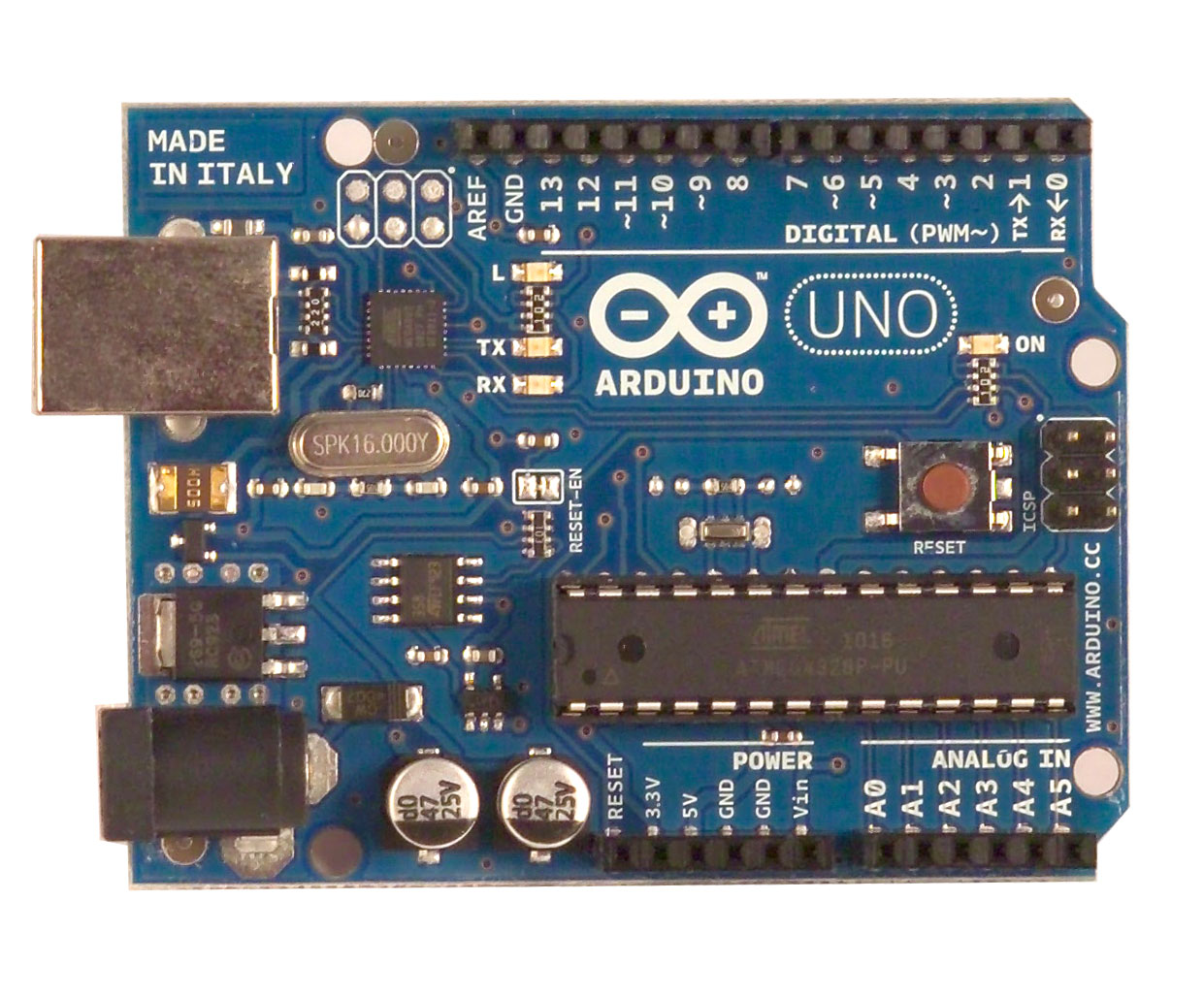
**Chapter 3: Maze Construction**

**Chapter 4: Testing and Implementation**

**Chapter 1: Hardware Design**

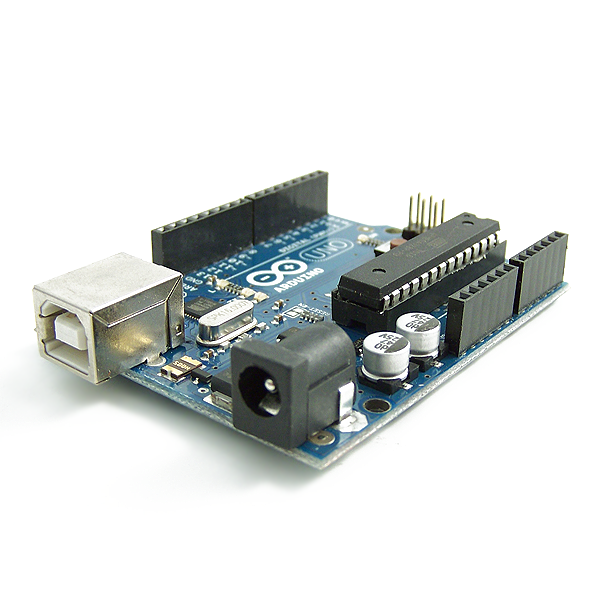
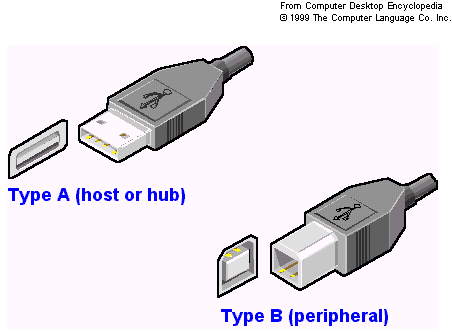
**Microcontroller**

Selecting an appropriate microcontroller was crucial in starting the Micromouse design project. Open-source microcontrollers available in the market provide community-based support from online forums. Example code and tutorials provided an easier learning experience. Arduino microcontrollers are relatively low-cost, about $30, and there were many online support sites and tutorials. The Arduino Uno microcontroller was selected for use in the CityMouse (**Fig M1.**).



**Fig M1.** The Arduino Uno microcontroller.

The Arduino Uno microcontroller includes a free software development kit allowing C programmed language syntax to upload directly to the board via Universal Serial Bus (USB Type B).



**USB Type B Connector**

**Fig M2.** Microcontroller USB port and cable connection.

Online tutorials and forums were critical in the set-up and implementation of the Arduino Microcontroller. The Arduino webpage, Arduino.cc, provided a rich and dynamic user experience by providing example codes to pre-existing projects and user forums for guides/how-to’s and frequently asked question and answers.

Coding the Arduino required sufficient knowledge of C high level programming language through prior coursework. At the top of syntax were headers, used to call libraries, followed by initialization of variables. A setup loop was used to properly setup output pins and data read/display rates. Then a loop is written to perform the desired action of the microcontroller. A simplified outline of the code is as such:

*#include <header\_file.h> //header*

*int x = 0; // initialize variable x to zero*

*void setup() {*

*pinMode(13, OUTPUT); // initialize the digital pin as an output.*

*}*

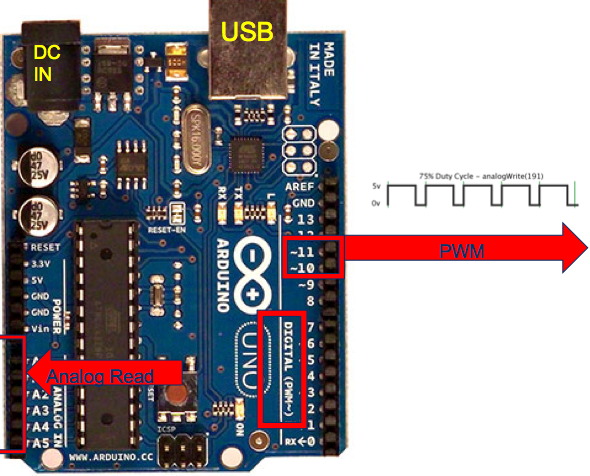
*void loop() {*

*//desired code here;*

*}*

Proper control for any robotic system requires that you slow and speed up speed moving parts. For this to be accomplished, Pulse Width Modulation, PWM, was needed. Fortunately, the Arduino Uno microcontroller has several ports on board for this very function. Such ports allow the microcontroller to send out binary 1’s or 0’s at a desired rate. This feature is crucial for DC Motor speed control. The Arduino Uno PWM ports were port number 3, 5, 6, 9, 10, and 11 and provided 8-bit Pulse Width Modulation with the analogWrite() function.

Additionally, the microcontroller is capable of reading analog data from several ports. These ports are crucial for control and feedback loops to be established throughout the system. For example, sensor values could be sent to the analog read ports so the robot could interpret the data and make decisions based on the data received at any given time. The figure below outlines the PWM port locations in addition to the analog read on the Arduino Uno board:



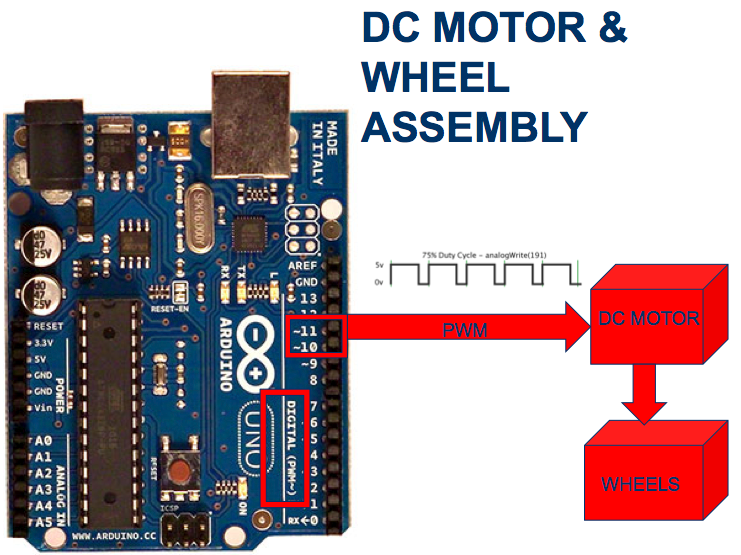
**Fig M3.** Pulse Width Modulation and analog read ports.

The microcontroller was initially supplied power to its DC barrel jack adapter input. However, after finding an appropriate rechargeable battery and developing a printed circuit board, the team decided to power it through the Vin and GND ports. The microcontrollers board supplies 5 volts with 40mA to each of its output pins and its recommended input voltage was 7 to 12 volts. The finished CityMouse robot ran on 7.4 volt, 1200 mAh, rechargeable Lithium Ion battery.

The Arduino Uno board runs on a high performance 8-bit ATmega328 microcontroller. The ATmega328 is a RISC-based microcontroller capable of executing one million instructions per second (1MIPS). This allows many instructions, such as read sensor values and output motor speed, to be performs quickly and efficiently.

**Motor Control**

Proper motor control was fundamental to the robots ability to travel in the maze. Appropriate speed control was essential to ensure the robot would not travel too fast or too slow throughout the maze. The microcontroller provided six ports capable of Pulse Width Modulation to accomplish this task.

These ports output an analog value capable of changing the speed at which the motors would turn. Changing a loop instruction inside the code allowed use to pause and start these output values and in turn speed up or slow down motor/wheel speed. Motor control was set as shown in **Fig Motor1**.

**Fig Motor1: DC Motor and Wheel Assembly**

To set up motor control via the Arduino microcontroller, the motor pins had to be set as output in the void setup() loop by using the pinMode() function. The pinMode function required that the selected pin number be designated as either in input or output, such as:

*void setup() {*

*pinMode(pin number, OUTPUT);*

*}*

There were two bi-directional motors used on the robot allowing for forward and backward rotation. Each pin used was declared in a PWM output pin as discussed before. To designate the rotation, two output pins per motor were used such as:

*const int motor\_right\_backwards = 5;*

*const int motor\_right\_forwards = 6;*

*const int motor\_left\_forwards = 11;*

*const int motor\_left\_backwards = 10;*

*void setup() {*

*pinMode(motor\_right\_backwards, OUTPUT);*

*pinMode(motor\_right\_forwards, OUTPUT);*

*pinMode(motor\_left\_forwards, OUTPUT);*

*pinMode(motor\_left\_backwards, OUTPUT);*

*}*

So far the pins were set up as outputs but the output pins are not sending any current to either motor. To send values to the motor to ensure movement, an additional loop, arbitrarily named straight() was used as such:

void straight()

{

analogWrite(motor\_left\_backwards, 0);

analogWrite(motor\_left\_forwards, 160);

analogWrite(motor\_right\_forwards, 160);

analogWrite(motor\_right\_backwards, 0);

}

It is obvious that by sending the same analog value to the PWM pin to the forward designated pins of both motors, the wheels should rotate such that the robot moves forward in a straight path. While this is the ideal case, testing indicated that weight distribution of the chassis/parts and errors in wheel diameter, among other inconsistencies, could force the robot to not move in a completely straight path. This required analog values to be calibrated such that both motor rotate at the same speed and this process is further discussed in the **Testing** section of the report.

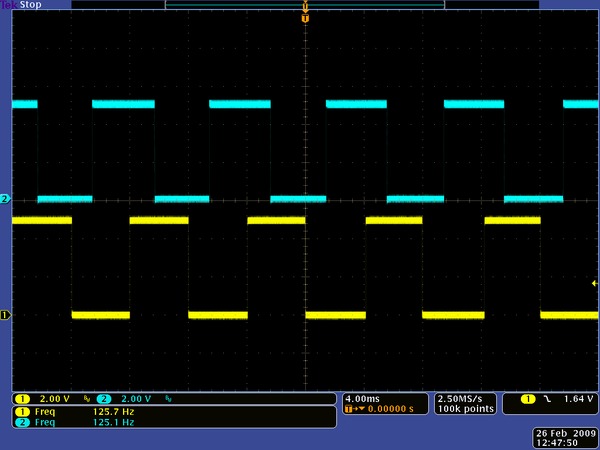
**Encoders**

To keep track of which cell the robot is in, distance traveled by the robot had to be tracked and analyzed. Because the cell dimensions were pre-determined to be 18x18 cm, CityMouse needed to be able to track distance by recording the cell location every time it has traveled 18cm.

To keep track of distance traveled, optical wheel encoders from Pololu.com were used. When the encoder was placed inside a 42x19 millimeter wheel, the encoders could determine both wheel velocity and direction of rotation. The wheels selected had twelve teeth along the inside rim of the wheel.

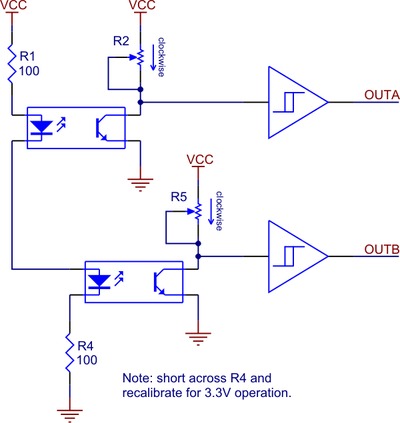
Each time the two sensors indicated that a tooth passed over it, it compared the two values recorded and determined the direction of rotation and increments a counter. The encoder operates such that after 48 counts, the wheel has done a complete rotation. Accordingly, it was determined that after approximately 60 counts, the robot traveled the length of one cell (18cm), after 30 counts approximately half (9cm).

The two encoders each display two square waves on two different channels, channel A and channel B. It was important that all square waves operate at 50% duty cycle. Before robot assembly, the duty cycle was determined by spinning the wheel and displaying the output square wave on the oscilloscope. The duty cycle was measured off the screen. A small screwdriver was used on the board of the encoder to adjust the duty cycle of each sensor. After all 4 square waves read 50%, the encoders were ready to record accurate distance values. The two encoders square waves operated between 0-5 volts like in **Fig E1**. Take note that the square waves have a 90-degree phase shift. This is needed to ensure the direction of the wheel can be determined.



**Fig E1.** Typical square wave outputs from encoder.

**Fig E2** shows the encoder schematic. Notice that the encoder outputs on two separate channels simultaneously and resistors R2 and R5 indicated the potentiometers that must be turned to adjust the duty cycle of each channel.

[[](http://www.pololu.com/picture/view/0J1206)](http://www.pololu.com/picture/view/0J1206)

**Fig E2.** Schematic diagram for the Pololu wheel encoder.

Utilizing the Pololu wheel encoder was a relatively simple process due to the fact that a open-source Pololu wheel encoder library was readily available. Using the library allowed the robot to determine direction and increment wheel count easily. This was accomplished by downloading and installing the Pololu user libraries in the Arduino software development kit located at the company website.

Installing and setting up the library did not come without any problems due to a lack of support from Pololu and other users with interfacing their library with the Arduino microcontroller. Our team determined the proper installation of the Pololu library with the Arduino software development kit in several steps:

|  |  |
| --- | --- |
| **For Windows Operating System:** | **For Mac OSX**: |
| 1. Download the lastest Pololu AVR Library version [http://www.pololu.com/docs/0J20/2](http://www.pololu.com/docs/0J20/2" \t "_blank) 2. Extract the library using an unzipping utility (winzip, dismount, etc.) 3. Open libpololu-avr/src/ directory 4. Copy all of the contents here and place them in arduino-0021//libraries/ directory | 1. Download the latest Pololu AVR Library version 101104 [http://www.pololu.com/docs/0J20/2](http://www.pololu.com/docs/0J20/2" \t "_blank)  2. Extract the library using an unzipping utility (disc-unmount, unrar, etc.) 3. Open libpololu-avr/src/ directory 4. Copy all of the contents here 3. Right click the arduino icon and click "show package contents" 4. Open Contents/Resources/Java/libraries 5. Paste all of the contents from libpololu-avr/src/ directory and place them in the contents/Resources/Java/libraries directory |

The encoder header file, PololuWheeEncoders.h, was called in the header of the Arduino file. After, a variable called *encoders* was declared in the header file by writing: *PololuWheeEncoders encoders*. This was done as a way to pass the pin values through to the library for analysis. Next, in the void setup() function, the pins used to accept the square wave outputs of each encoder channel had to be declared. This ensured the values were passed to the variable *encoders* and used in conjunction with the library to determine the direction of rotation and increment the counter. Each time it was desired to start recording the encoder value, or wheel rotation count, a simple function from the encoder library was used, *encoder.getCountsM1* and *encoder.getCountsM2* (*M1* refers to the first motor and *M2* refers to second motor). The value of the counter could be reset to zero at any moment by calling *encoder.getCountsandResetM1* or *encoder.getCountsAndResetM2*.

The code was constructed such that the value was initialized to zero by calling encoder.getCountsandResetM1, then setting a variable equal to the count number, encoder\_left = encoder.getCountsM1. When the value of the encoder reached 60 counts, the robot has theoretically traveled 18 cm. Example code is as such:

*#include <PololuWheelEncoders.h>*

*PololuWheelEncoders encoder;*

*cellcount = 0;*

*void setup() {*

*encoder.init(12,13,8,9); //pins that accept the square waves of the encoders*

*}*

*void loop(){*

*encoder.getCountsAndResetM1();*

*encoder\_left = encoder.getCountsM1();*

*while(encoder\_left < 60){*

*straight(); //go forward*

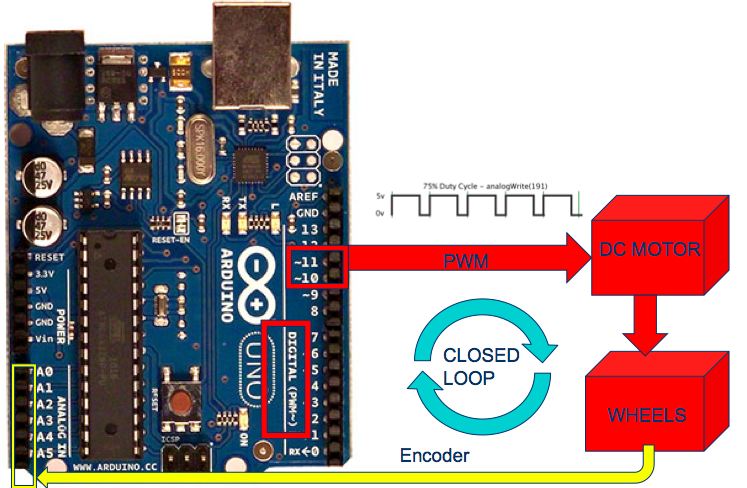
*encoder\_left = encoder.getCountsM1();*

*}*

*cellcount++; //increment which cell the robot is in.*

*}*

At this point, there was enough developed here for the system to be considered a closed loop. Such that the robot travels at a desired speeds, with the ability to develop code for slowing up or speeding down as desired, and incrementing a cell count after a desired distance. Such a system is shown in **Fig E3**.



**Fig E3.** The closed loop feedback system of CityMouse.

**Control**

Proper Control of the robot required extensive process of alterations of code through hours of rigorous testing. This section overviews the process by which the robot went from crashing into walls into a relatively safe and efficient traveler.

Before trying to get the robot to properly travel in a straight path, CityMouse must be able to appropriately make 90-degree left and right turns. This process turned out to be quite difficult and was even continuously tested until hours before the competition start time. The problem was making left and right turns at precise 90 degree turns lies in the fact that while the robot makes these turns, it is completely dependant on the wheel encoders. If the wheel encoder makes a mistake due to errors such as the wheel slippage, the robot might end up at less or greater than 90 degrees with respect to its initial orientation.

Another issue was that sometimes the robot traveled such that before it makes its turn, it ended up on an angle initially. Turning 90 degrees with respect to an initially offset angle means the robot could be headed in a potentially dangerous position and end up crashing. For all intensive purposes, this problem wasn’t of great importance initially in that we first needed to make sure the robot could make turns at precise 90 degree turns to begin with.

To make 90-degree turns about the robot center axis, left wheel must travel in equal and opposite direction to the right wheel. Due to inconsistencies in wheel velocity, wheel diameter, and weight distribution, the values for the PWM output to each wheel after testing might have been plus or minus each other to ensure the robot turns about its center of axis.

To simplify our code a bit, new functions (*left\_turn() & right\_turn()*) were developed and called each time we wanted to make left or right turns at any time in our main loop. The function was constructed such that the outer wheel encoder value was tracked and after 16 counts, or the distance required to turn at least 90 degrees, both motors would be stopped. The outer wheel was smartly chosen to be the measured wheel to ensure that the robot has definitely turned far enough. Left and right turns were desired such as:

*void right\_turn()*

*{*

*encoder.getCountsAndResetM1();*

*encoder\_left = encoder.getCountsM1();*

*Serial.println(encoder\_left);*

*while(encoder\_left < 17){*

*analogWrite(motor\_right\_forwards, 0);*

*analogWrite(motor\_right\_backwards, 60);*

*analogWrite(motor\_left\_backwards, 0);*

*analogWrite(motor\_left\_forwards, 60);*

*encoder\_left = encoder.getCountsM1();*

*}*

*complete\_stop();*

*}*

Notice that in the above example that for turning right, the left encoder value was checked to ensure the robot turned far enough. Designing the process of turning in such a way allows the function be called anywhere in any other part of the entire syntax.

Traveling in a straight path required the robot constantly check the surrounding environment through its sensors to make sure that the robot doesn’t hit anything. An analogy for this is that the robot uses the sensors as its eyes and will always be checking the walls such that it’s not too close.

The design for this was based in a new function called straight\_algorithm(). This algorithm is designed to analyze distance values being constantly read from its left and right sensors and slow down and speed up the appropriate motors. For example, if the left sensor distance is greater then the right sensor distance, the robot it too close to the right wall and the right wheel must be sped up and the left wheel slow down such that:

*left\_wall = ReadSensor\_Left();*

*right\_wall = ReadSensor\_Right();*

*if(left\_wall > right\_wall){*

*analogWrite(motor\_left\_backwards, 0);*

*analogWrite(motor\_left\_forwards, 158); //min is 42*

*analogWrite(motor\_right\_forwards, 163); //min is 45 with full battery*

*analogWrite(motor\_right\_backwards, 0);*

*}*

*else{ //if the robot is in the center*

*straight();*

*}*

*}*

This idea was appropriate but problems quickly began to cascade once the robot make drastic or even slightly inconsistent motion such that it threw the robots orientation off too far form the center of the cell. In such a case, slightly speeding up one motor or slowing down another isn’t enough to fix the orientation of the robot and it will still hit the wall. To fix this, allowable distance thresholds were set up.

If the robot sensor deviated too far from an accepted distance value, the robot would slow down and speed up the appropriate wheels in a much more drastic fashion. Unfortunately, setting up thresholds decreased the smoothness of travel and in some cases slowing the robots overall speed due to it constantly having to rework itself until it dead-center in the cell again. Additionally, setting up thresholds could cause great problems when reaching the edge or the cell where one wall is open. In such a case, a value was established to set up a filtering such that if the sensor distance was greater then 8 cm then that wall is not present and not to take enter the threshold statement. An example of one of the threshold statement is as such:

*left\_wall = ReadSensor\_Left();*

*right\_wall = ReadSensor\_Right();*

*if (right\_wall > 6 && right\_wall<=8){ //this algorithm ensures if the*

*//robot deviates too far away from the right wall*

*analogWrite(motor\_left\_backwards, 0); //it will turn and head*

*//back towards the rightwall or center of the cell*

*analogWrite(motor\_left\_forwards, 160);*

*analogWrite(motor\_right\_forwards, 0);*

*analogWrite(motor\_right\_backwards, 0);*

*}*

Notice that if the distance value of the right sensor is outside of the accepted threshold value of 6 cm away then the robot completely stops the left motor and has the right wheel still on such that the robot will make a much more drastic correction motion then simply slowing down and speeding up the left and right wheels as in the previous correction method.

In the end both methods of correction were used in conjunction with each other all at once. After rigorous testing, a mediating point for the threshold values and wheel speeds were set and these values were agreed upon by what the group felt was stable movement.