



Control Applications of DSP Microprocessors
(MEM-800/380-003)

Instructor: Dr. B.C.Chang

Final Project Report

**Design, Analysis & Implementation of a QuadBot Using the Leap Motion
Sensor**

Date of Submission: Friday, 13 June 2014

Daniel Ehala

Hadi Hajieghrary

Nester Pereira

Sundar Ram

Table of Contents

I.	ABSTRACT	Page 3
II.	INTRODUCTION	Page 4
	1. Pulse Width Modulation	Page 4
	2. Hobby Servo Motors	Page 5
	3. Motor Control Using PWM	Page 6
	4. QuadBot	Page 7
	5. Arduino Mega 2560	Page 8
	6. Leap Motion Sensor	Page 11
III.	DESIGN & IMPLEMENTATION	Page 13
	A – Building the QuadBot	Page 13
	1. Materials Required	Page 13
	2. Assembly Procedure	Page 14
	B – Programming the Servos	Page 17
	C – Interfacing the Leap Motion Sensor to Arduino	Page 22
IV.	RESULTS & DISCUSSION	Page 24
V.	CONCLUSION & FUTURE WORK	Page 25
VI.	REFERENCES	Page 26
VII.	APPENDIX	Page 27
	I. Materials Required	Page 27
	II. Assembly Procedure	Page 29
	III. Wiring Schematic	Page 31
	IV. Arduino Codes	Page 31
	V. Leap Motion Controller Codes	Page 33

I. Abstract

This report presents the design, analysis and implementation of a QuadBot where the motion command signals are based on gesture recognition using the Leap Motion Sensor (LMS). The Quadbot is a small, lightweight robotic kit that is equipped with 8 micro servo motors. These servos are PWM regulated and were controlled using an Arduino Mega. By interfacing the LMS to the Arduino, the Quadbot executed specific motions corresponding to hand gestures.

The project was executed in three phases. In phase 1, the QuadBot was assembled and mounted with an Arduino Mega. The QuadBot was then programmed to execute different motion such as walking along straight line and performing turns. This was achieved by varying the sequencing & angular position commands of the servos. In phase 2, the gesture recognition data from the LMS was extracted and analyzed. Finally in phase 3, the LMS data and the QuadBot servos were integrated via the Arduino so that the motion of QuadBot can be triggered by the LMS.

The fundamental theory used in this project is PWM control of a motor. The pulse width of the PWM precisely controls the shaft position and by accessing the individual servos, the QuadBot can be programmed to execute specific motions. Various functions corresponding to various motions were created. These functions were then executed when a specific hand gesture was recognized. In this project, the QuadBot was programmed to execute five different motions for five different hand gestures.

II. Introduction

1. Pulse Width Modulation

The term 'Pulse Width Modulation' or simply PWM refers to technique where a command signal (usually a voltage or current signal) is modulated between an off (inactive) or an on (active) position to drive a load such as a motor. The width of the pulse (pulse duration) is determined based on the application.

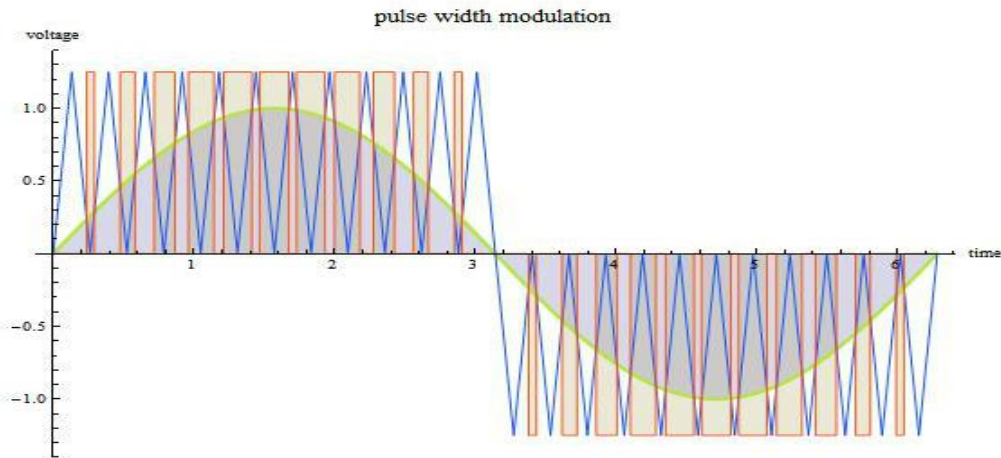


Figure 1: A PWM signal (orange) is generated by the superimposition of a sine wave (green) and a sawtooth wave (blue).

The term 'duty cycle' describes the proportion of 'on' time to the regular interval or 'period' of time. A low duty cycle corresponds to low power, because the power is off for most of the time. Duty cycle is expressed in percent, 100% being fully on.

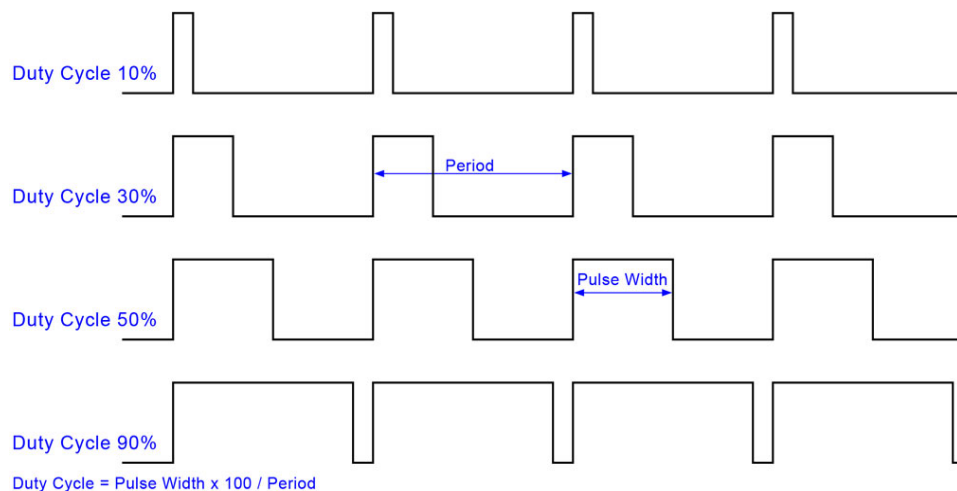


Figure 2: Various duty cycles are shown here. The longer the 'on' period, the higher the duty cycle

2. Hobby Servo Motors

Hobby servos, such as the ones utilized in the QuadBot, are a popular and inexpensive method of motion control. Hobby servos eliminate the need to custom design a control system for each application. Since the early 1990's servos have used a standard pulse width modulation (PWM) technique to control the position of the output shaft. Internally, these servos can be thought of as a direct current (dc) motor (which rotates an external motor shaft but provides no way to determine the amount of rotation) with a built-in controller.

The control circuitry compares an angular position, determined by a control signal, to the current position of the motor shaft (as shown in Fig. 3). The motor shaft's angular position is often determined by a potentiometer, which is rotated by the motor shaft. A potentiometer is a three-terminal resistor whose center connection has variable resistance, usually controlled by a slider or dial. The potentiometer acts as a variable voltage divider. The voltage from the center connection of the potentiometer represents the angular position the motor shaft is in.

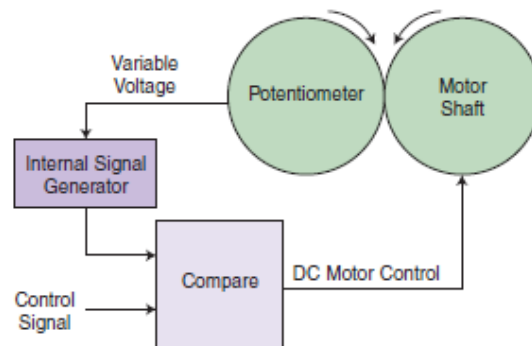


Figure 3: Servo flow diagram

The control signal does not supply power to the motor directly. It is an input to a control chip inside the servo and as such it does not have to supply much current to the servo. A separate power wire supplies the power to the servo. The ground for power is also used as the ground for the control line. This can be noticed on the QuadBot motors as each motor comes with 3 wires, one for control, one for power & one common ground as seen in figure 4. The center red wire is for power, the yellow wire on the right is for control and the brown wire on the left is ground.

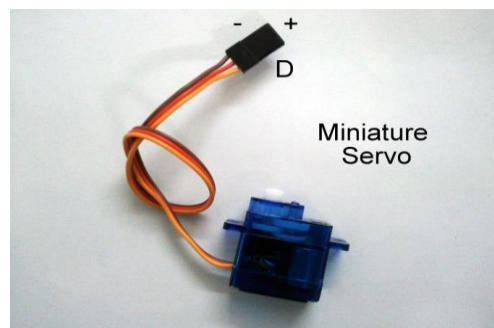
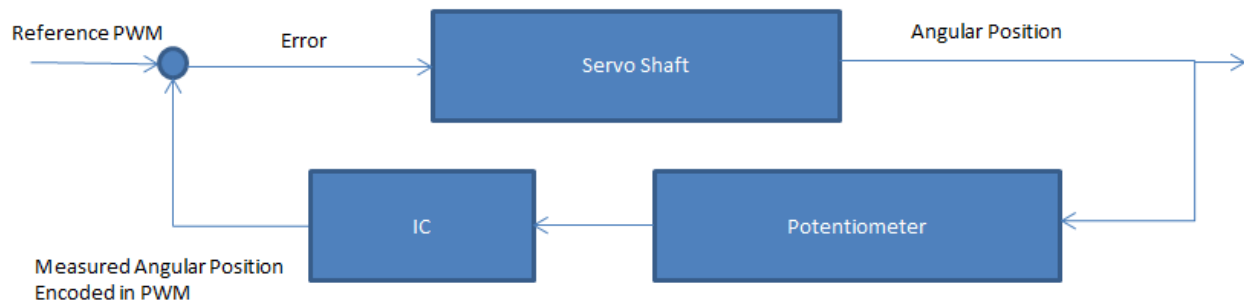


Figure 4: Servo wiring

3. Motor Control Using PWM

The angular position of the servo is determined by the pulse width of the PWM sent through the control line. The servo has internal circuitry where the input PWM signal is compared to an internally generated signal whose pulse width is controlled by the potentiometer which determines the shaft angle and matches the pulse widths by rotating the motor shaft. A control block diagram is shown below.



This internal circuit consists of a pulse width comparator, which compares the incoming signal with a one-shot timer whose period depends on the resistance of a potentiometer connected to the servo's drive shaft. This feedback is what provides the stability for the tracking control circuit. The difference between the control signal and the feedback signal is the error signal. This error signal is used to control a flip-flop that toggles the direction the current flows through the motor. The outputs of the flip-flop drive an H-Bridge circuit that handles the high current going through the motor. The M51660L servo motor control chip contains all the electronics needed to decode the signal and control a motor. A block diagram is shown in figure 5.

BLOCK DIAGRAM

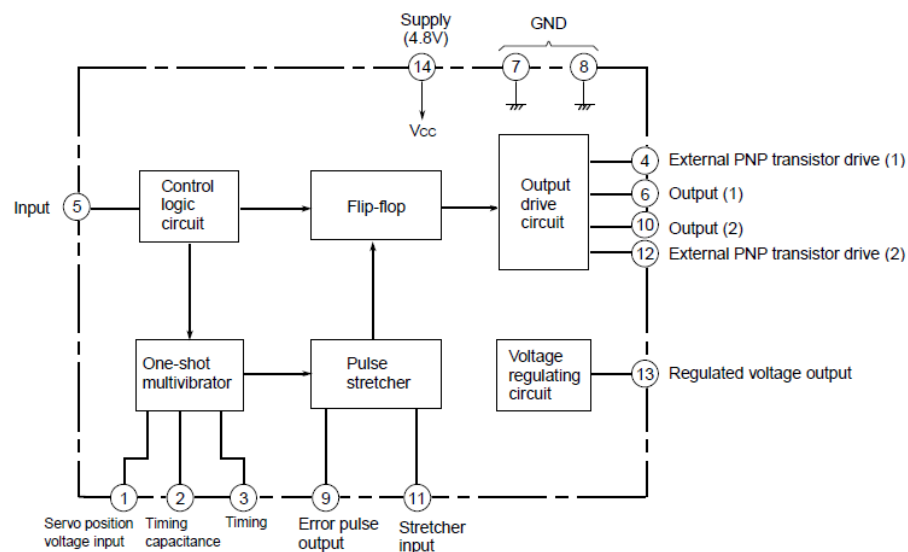


Figure 5: Servo motor block diagram

The control part of the signal is broken down into the 1ms minimum time, the 1ms PWM signal, and a roughly 40ms delay. This delay is not as critical as the other parts of the timing signal. It is essentially the dead time between control signals. If the control signals are repeated too quickly (i.e. 10ms delay) the servo will buzz and jitter. If the control signals are repeated too slowly (i.e. 70ms delay) the servo will shut off between signals and its position will not remain constant.

4. QuadBot

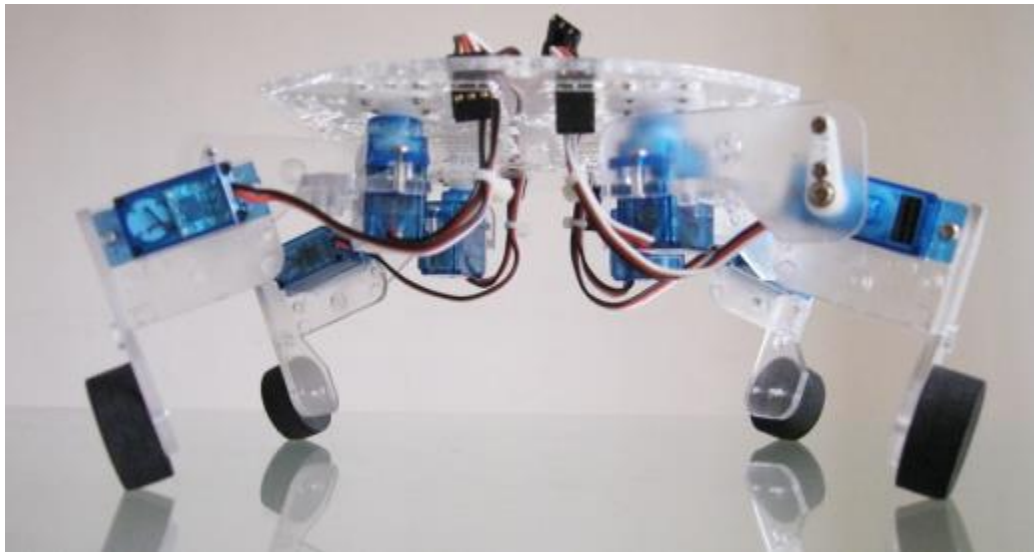


Figure 6: The QuadBot

The QuadBot (shown in figure 6) is a lightweight, simple 4 legged robot chassis ideal for students and hobbyist wanting to experiment with walking robots. It has 8 servo motors (specifications are shown in table 1) which are controlled independently. The servos can be controlled by any controller capable of PWM output. For this project, an Arduino Mega was used whose details are covered in the next section. A SPIDER Robot Controller could equivalently be used.

Specification Parameter	Value
Weight	8 grams
Torque	1.5 Kg/cm @ 6V
Speed	0.1 sec / 60 degrees
Max. Voltage	6V
Travel	180 degrees

Table 1: Servo motor specifications

5. Arduino Mega 2560

Overview

The Arduino Mega 2560 (seen in figure 7) is a microcontroller board based on the ATmega2560. It has 54 digital input/output pins (of which 15 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller.

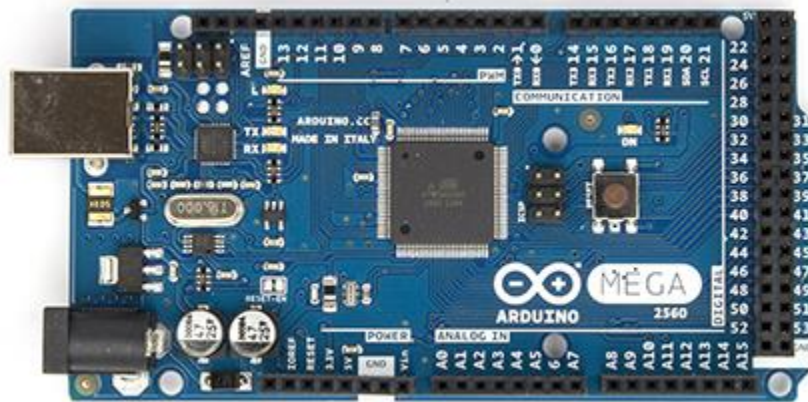


Figure 7: Arduino Mega 2560

Power

The Arduino Mega can be powered via the USB connection or with an external power supply. The power source is selected automatically. External (non-USB) power can come either from an AC-to-DC adapter (wall-wart) or battery. The adapter can be connected by plugging a 2.1mm center-positive plug into the board's power jack. Leads from a battery can be inserted in the Gnd and Vin pin headers of the POWER connector.

The board can operate on an external supply of 6 to 20 volts. If supplied with less than 7V, however, the 5V pin may supply less than five volts and the board may be unstable. If using more than 12V, the voltage regulator may overheat and damage the board. The recommended range is 7 to 12 volts.

The power pins are as follows:

- **VIN:** The input voltage to the Arduino board when it's using an external power source (as opposed to 5 volts from the USB connection or other regulated power source). You can supply voltage through this pin, or, if supplying voltage via the power jack, access it through this pin.
- **5V:** This pin outputs a regulated 5V from the regulator on the board. The board can be supplied with power either from the DC power jack (7 - 12V), the USB connector (5V), or the VIN pin of the board (7-12V). Supplying voltage via the 5V or 3.3V pins bypasses the regulator, and can damage your board. We don't advise it.

- **3V3:** A 3.3 volt supply generated by the on-board regulator. Maximum current draw is 50 mA.
- **GND:** Ground pins.
- **IOREF:** This pin on the Arduino board provides the voltage reference with which the microcontroller operates. A properly configured shield can read the IOREF pin voltage and select the appropriate power source or enable voltage translators on the outputs for working with the 5V or 3.3V.

Input & Output

Each of the 54 digital pins on the Mega can be used as an input or output, using

1. pinMode()
2. digitalWrite()
3. digitalRead()

They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50 kOhms. In addition, some pins have specialized functions:

- **Serial:** 0 (RX) and 1 (TX); Serial 1: 19 (RX) and 18 (TX); Serial 2: 17 (RX) and 16 (TX); Serial 3: 15 (RX) and 14 (TX). Used to receive (RX) and transmit (TX) TTL serial data. Pins 0 and 1 are also connected to the corresponding pins of the ATmega16U2 USB-to-TTL Serial chip.
- **External Interrupts:** 2 (interrupt 0), 3 (interrupt 1), 18 (interrupt 5), 19 (interrupt 4), 20 (interrupt 3), and 21 (interrupt 2). These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value. See the attachInterrupt() function for details.
- **PWM:** 2 to 13 and 44 to 46. Provide 8-bit PWM output with the analogWrite() function.
- **SPI:** 50 (MISO), 51 (MOSI), 52 (SCK), 53 (SS). These pins support SPI communication using the SPI library.
- **LED: 13:** There is a built-in LED connected to digital pin 13. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off.
- **TWI: 20:** (SDA) and 21 (SCL). Support TWI communication using the Wire library. Note that these pins are not in the same location as the TWI pins on the Duemilanove or Diecimila.

The Mega2560 has 16 analog inputs, each of which provide 10 bits of resolution (i.e. 1024 different values). By default they measure from ground to 5 volts, though it is possible to change the upper end of their range using the AREF pin and analogReference() function.

Communication

The Arduino Mega2560 has a number of facilities for communicating with a computer, another Arduino, or other microcontrollers. The ATmega2560 provides four hardware UARTs for TTL (5V) serial communication. An ATmega16U2 on the board channels one of these over USB and provides a virtual com port to software on the computer

Programming

The Arduino Mega can be programmed with the Arduino software. The ATmega2560 on the Arduino Mega comes preburned with a bootloader that allows you to upload new code to it without the use of an external hardware programmer. It communicates using the original STK500 protocol.

Summary

The Arduino Mega specifications can be summarized as seen in table 2

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

Table 2: Arduino Mega Specifications

6. Leap Motion Sensor

The Leap Motion sensor is a small USB peripheral device (seen in figure 8) which is designed to recognize hand features such as hand palm orientation, fingers' length, width and orientation, hand opening and other non-hand features including tools and computer screen location.



Figure 8: Leap Motion Sensor (LMS)

The Leap Motion works with two monochromatic IR cameras and three infrared LEDs as a depth sensor in a limited field of view (FOV) of 8 cubic feet (approximately 61 cubic centimeters). The LEDs generate a 3D pattern of dots of IR light and the cameras generate almost 300 frames per second of reflected data, which is then sent through a USB cable to the host computer, where it is analyzed by the Leap Motion controller software synthesizing 3D position data by comparing the 2D frames generated by the two cameras.

The controller's field of view is an inverted pyramid centered on the device. The effective range of the controller extends from approximately 25 to 600 millimeters above the device (1 inch to 2 feet). The controller itself is accessed and programmed through Application Programming Interfaces (APIs), with support for a variety of programming languages, ranging from C++ to Python. The positions of the recognized objects are acquired through these APIs. The Cartesian and spherical coordinate systems used to describe positions in the controller's sensory space are shown in figure 9. However, it should be noted that the sampling frequency is not stable, cannot be set, and varies significantly.

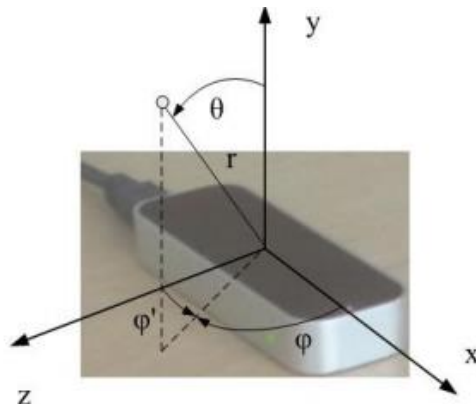


Figure 9: LMS field of view

Although it has a limited space to work with, the hand tracking is very precise. The Leap Motion Software Development Kit (SDK) comes with three different presets of tracking: Robust mode, Low resource mode and Auto orientation device mode. Furthermore, it has also three different performance sets: High precision, High Speed and Balanced tracking. In this project, the balanced tracking mode was used.

In the High Precision mode, the dataflow provides circa 50 data frames per second (data fps), representing about 20 milliseconds (ms) of delay. With the Balanced tracking its data fps increases by a factor of two, reducing the delay to 10 ms combined with a still good precision. By choosing the High Speed mode, it loses a perceptible amount of precision with tracking but reduces the delay to 5 ms (approximately 200 fps).

The five gestures used in this project are

1. Swipe Left
2. Swipe Right
3. Circle Clockwise
4. Circle Anticlockwise
5. Screen Tap

III. Design & Implementation

A – Building the QuadBot

1. Materials Required

Listed below are the materials required to build the QuadBot chassis. A detailed pictorial description is provided in the appendix (refer to section VII - I).

Part No.	Item	Quantity
1	Aligner	1
2	Laser-cut mounting plate	1
3	Miniature Servos	8
4	Servo Horns (Linear + circular)	8
5	Foam rubber feet	4
6	Leg Segments	8
7	3 mm x 12 mm Pan head screws	4
8	2 mm x 8 mm Pan head screws	8
9	2 mm x 6 mm Pan head screws	16
10	2 mm x 8 mm Self Tapping screws	16
11	2 mm x 6 mm Self Tapping screws	8
12	4 x AA battery holder	1
13	20 mm Spacers	4
14	Nuts	4
15	Bolts	4
16	Breadboard	1
17	Arduino Mega 2560	1

Table 3: Required Materials

The power to the motors will be fed from the controller (Arduino board). In order to power the Arduino, a battery pack can be installed on the chassis. For the purposes of testing, the Arduino was powered externally using a power cable. The breadboard is required to connect the PWM output pins of the Arduino to the eight servos of the QuadBot. Equivalently a motor shield could be used such as the one shown in figure 10.



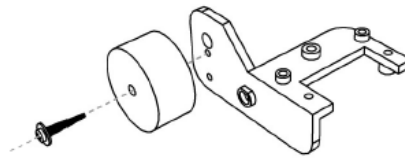
Figure 10: Servo shield

2. Assembly Procedure

Using the materials from above, the Quadbot was assembled using the following instructions.

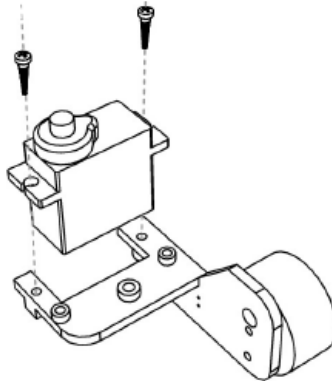
Step 1:

Mount the 4x foam rubber feet to leg segments as shown using 3mm x 12mm pan head screws.



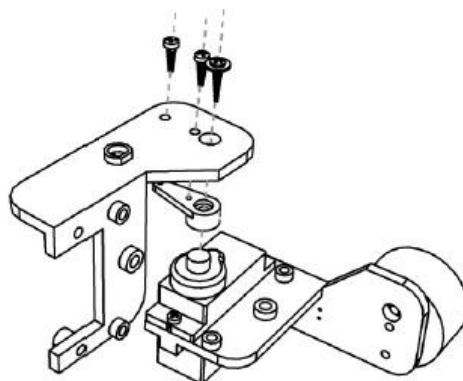
Step 2:

Mount a servo on each of the leg segments fitted with a foot as shown using 2.3mm x 8mm self-tapping screws.



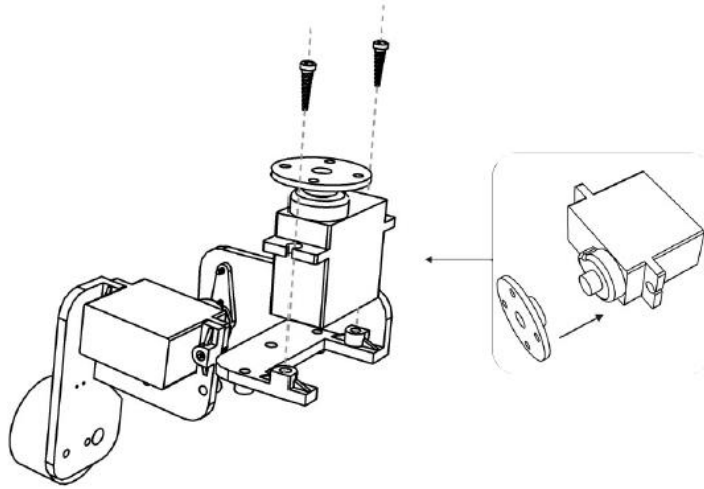
Step 3:

Mount a servo horn to an unused leg segment as shown using 2mm x 6mm self-tapping screws. Center the servo and then fit the leg segment to the servo using a 2mm x 8mm pan head screw as shown. Gently turn the servo by hand to check the range of movement and adjust if necessary.



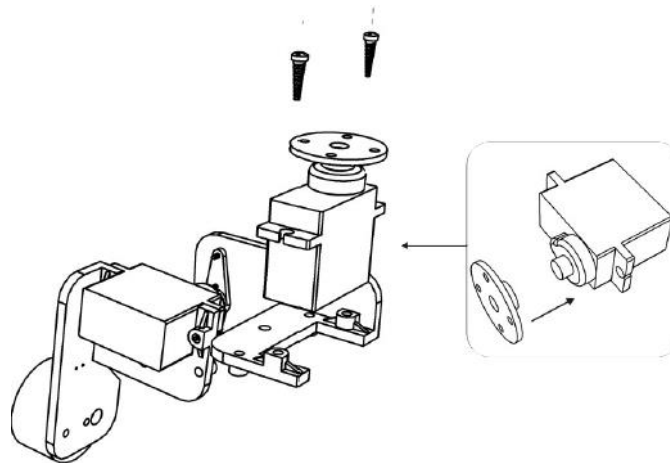
Step 4:

Now two legs for the right side of the chassis are to be assembled. Mount two servos as shown in the diagram with 2.3mm x 8mm self-tapping screws. Note the orientation of the servo.



Step 5:

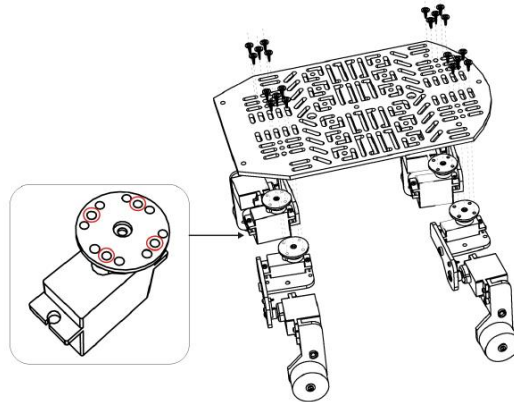
Make two legs for the left side of the robot in a similar manner to step 4 except that the servo is mounted the opposite way.



Step 6:

Center the servos and mount the legs onto the mounting plate using 2mm x 6mm pan head screws. Use a 2mm x 8mm pan head screw in the center of the servo horn to secure the servo to the mounting plate.

When mounting the servo horn to the mounting plate the holes marked in red are recommended. Gently move the legs by hand to check their range of movement and adjust the servo horns as required.



Finally, the breadboard and Arduino are mounted on the QuadBot chassis. The boards are stuck on with double sided tape to ensure the boards do not move when the QuadBot is in motion. The final QuadBot is seen in figure 11. A more detailed assembly procedure can be found in the appendix (refer to section VII – II). A wiring schematic is also attached in the appendix. (refer to section VII-III).

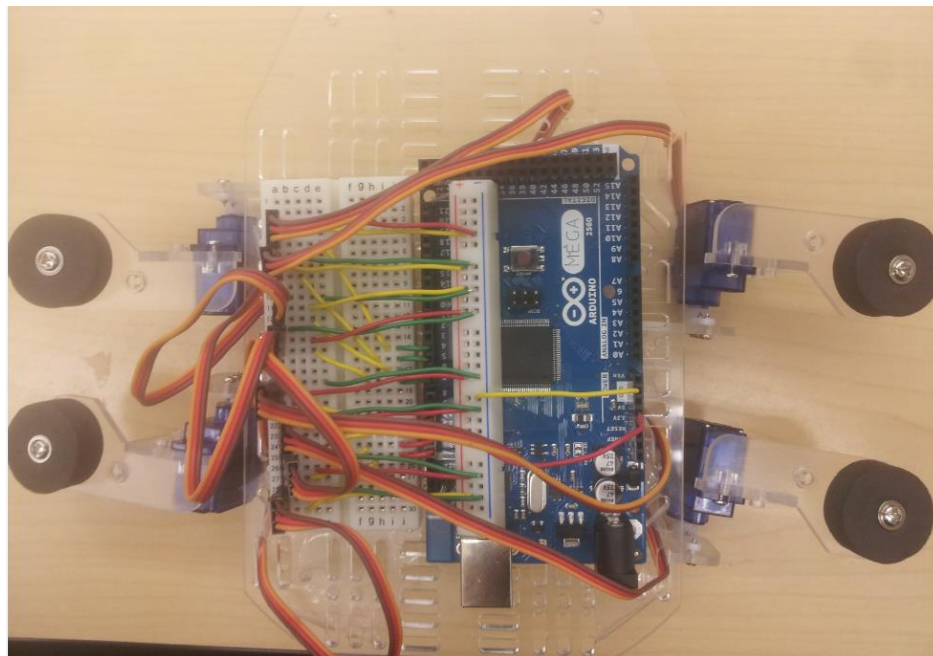


Figure 11: QuadBot with the Arduino & breadboard added on.

B – Programming The Servos

After the chassis is assembled, it is essential to label each servo in order to specify the sequence of operation. It is important to note here that the servos are limited to 0-180 degrees of displacement. The labeling scheme is shown in figure 12. The circular edge of the mounting plate is the front of the QuadBot and the flat edge is the rear.

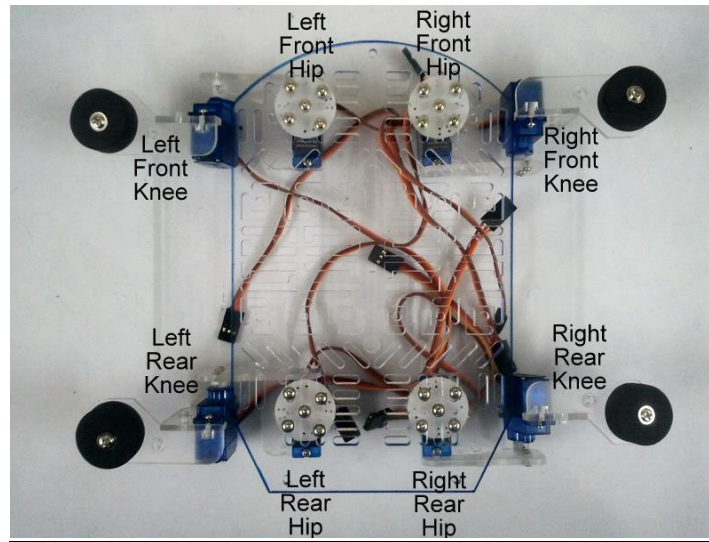


Figure 12: Labeling the QuadBot servos

The servos pin assignment on the Arduino and array indices are shown in table 4.

Servo Location	Pin Assignment on Arduino	Array Index
Left Front Knee	2	0
Right Front Knee	11	1
Right Rear Knee	13	2
Left Rear Knee	4	3
Left Front Hip	3	4
Right Front Hip	10	5
Right Rear Hip	6	6
Left Rear Hip	5	7

Table 4: Servo Pin & Index Assignment

Now that access each servo individually can be achieved, various functions can be defined which commands the QuadBot to execute desired motions. Arduino has an inbuilt servo library which allows control over servo motors. As discussed in section II. 2 & 3, precise control over the servos is achieved by controlling the shaft. Standard servos allow the shaft to be positioned at various angles, between 0 and 180 degrees.

The servo library comes with 6 functions as listed in table 5.

Function	Functionality
attach()	Attach the Servo variable to a pin. The Servo library supports only servos on only two pins: 9 and 10.
write ()	Writes a value to the servo, controlling the shaft accordingly. On a standard servo, this will set the angle of the shaft (in degrees), moving the shaft to that orientation. On a continuous rotation servo, this will set the speed of the servo (with 0 being full-speed in one direction, 180 being full speed in the other, and a value near 90 being no movement).
writeMicroseconds()	Writes a value in microseconds (uS) to the servo, controlling the shaft accordingly. On a standard servo, this will set the angle of the shaft. On standard servos a parameter value of 1000 is fully counter-clockwise, 2000 is fully clockwise, and 1500 is in the middle.
read()	Read the current angle of the servo (the value passed to the last call to write()).
attached()	Check whether the Servo variable is attached to a pin.
detach()	Detach the Servo variable from its pin. If all Servo variables are detached, then pins 9 and 10 can be used for PWM output with analogWrite().

Table 5: Arduino Servo library functions

For this project, the write() and read() functions were used extensively. The function 'move_servo' was created which reads the current position of the servo and then commands it to be displaced by a desired amount.

```
void move_servo(int x,int y)
{
  // Read the current position of Servo 'x' and move it by 'y' degrees
  servo[x].write(servo[x].read()+y);
}
```

By placing this function into a for loop, we can displace the servos by a desired amount. The sequence of execution determines the motion performed by the QuadBot.

```
for(int i=0;i<displacement;i++) // Move the Left Front Forward and Right Rear Hip Backward
{
    move_servo(4,+1);
    move_servo(6,-1);
    delay(delay_time);
}
```

When this for loop is executed, servo 4 and 6 will be displaced by +30 & -30 degrees meaning the left front hip will move forward while the right rear hip will move backward.

Now that access & control to each servo has been achieved, it is important to understand the range of motions of the knee and hip servos as this will determine the motion executed by the QuadBot.

Knee Servos

When the knee servos are at 0 degrees, the position of the QuadBot leg position is seen in figure 13.

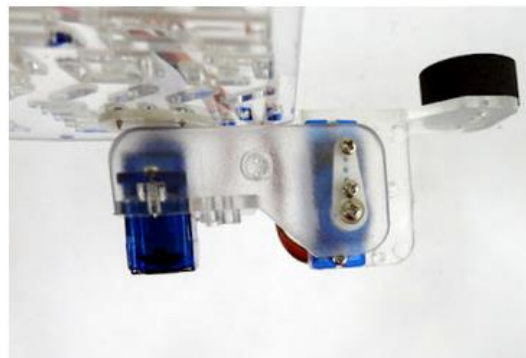


Figure 13: Knee Servo at 0 degree displacement

When the knee servos are displaced, they move as shown in figure 14. Therefore when all four knee servos are displaced, the QuadBot will standup. This is the idea implemented in the stand_up () function.

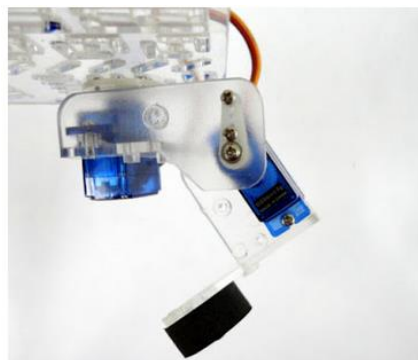


Figure 14 Knee Servo at 180 degree displacement

A knee servos angle diagram is shown in figure 15.

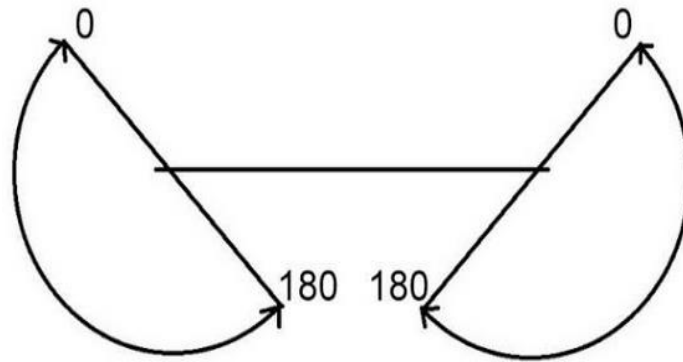


Figure 15: Knee Servos Angle Diagram.

Hip Servos

The hip servos angle diagram is shown in figure 16.

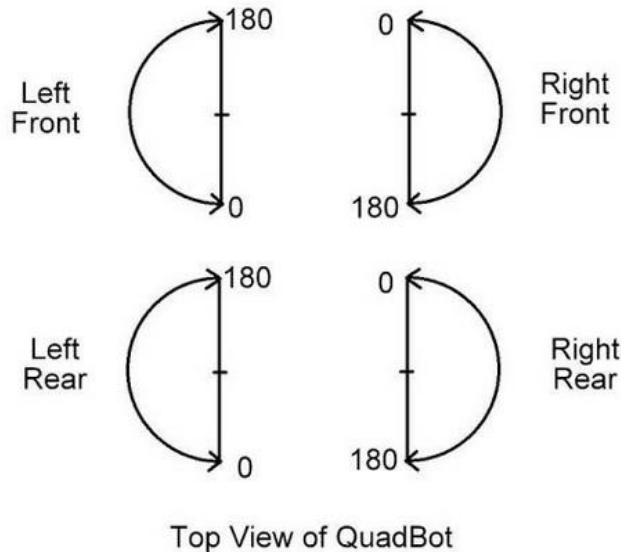


Figure 16: Hip Servos Angle Diagram.

Together with the knee servo angle diagram, the forward motion algorithm can be described as seen below. The QuadBot is assumed to be standing.

1. Raise Left Front & Right Rear Knee.
2. Move Left Front Hip forward & Right Rear Hip backward
3. Drop Left Front & Right Rear Knee.
4. Move Left Front Hip backward & Right Rear Hip forward
5. Raise Right Front & Left Rear Knee.
6. Move Right Front Hip backward & Left Rear Hip forward
7. Drop Right Front & Left Rear Knee.
8. Move Right Front Hip forward & Left Rear Hip backward

The detailed C code for the forward () function is provided in the appendix (refer to section VII – IV).

Following the same idea, seven different functions were created which are listed below in table 6.

Function	Motion Performed By QuadBot
stand_up ()	Stand up
forward ()	Move forward by one step
backward ()	Move backward by one step
step_right()	Turn right by one step
step_left()	Turn left by one step
continous_forward()	Move forward continuously until commanded to stop
stretch()	Stretches the legs forward, backward, left and right

Table 6: QuadBot motion functions

C –Interfacing the Leap Motion Sensor to Arduino (LMS)

In order to achieve gesture recognition, the first step is to setup the Leap Motion sensor. For proper set up, the shiny side of the controller should face up and the green light should face the user. The next step is to install the Leap software development kit (SDK) from the leap motion website.

One of the key features of the project is to integrate the leap motion sensor with the Arduino in order to achieve the following objectives:

1. Recognize the various gestures made by the hands
2. Communicate the gesture data to the Arduino via Serial Port
3. Execute the motion associated with the gesture

There are two key algorithms to achieve these objectives. The first is the 'Processing sketch' which is in charge of processing the input from the Leap Motion Sensor and mapping it to the corresponding gesture. After processing, a unique signal (integer value) corresponding to the gesture is sent to the Arduino via USB. Processing 2.2.1 and LeapMotion 2.0 library were utilized for executing this algorithm.

The leap motion library has many inbuilt gestures such as:

1. Swipe Left
2. Swipe Right
3. Circle Clockwise
4. Circle Anticlockwise
5. Screen Tap
6. Palm Orientation

By identifying & tracking the fingertips of the hand, the LMS can detect when a gesture is performed. So when a gesture is completed, say the clockwise rotation, it immediately identifies the circle gesture. There are a certain parameters associated with each gesture. Once a gesture is recognized, parameters such radius of the circle, point entry of the circle, duration of gesture and direction of the circle (clockwise or anticlockwise) are identified.

Next, the gesture data is sent to the Arduino. The Arduino is connected to the computer via a serial port. For every gesture, a unique integer value is sent to the Arduino. The processing sketch is programmed to serially write a different integer for each gesture.

```

if(clockwiseness == "clockwise") //clockwiseness is the parameter for direction of gesture

{

    myPort.write(1);           //myPort is the Serial Port

}

```

This algorithm assigns the integer value 1 to the clockwise gesture. When the Arduino receives this value, it is programmed to execute the forward () function which makes the QuadBot move forward. The complete gesture recognition code can be seen in the appendix (refer to section VII – V).

The second algorithm is executed in the Arduino platform.

The Arduino has been programmed to execute 5 different functions corresponding to 5 gestures as listed in table 7.

Gesture	Integer Representation	Function Executed	QuadBot Motion
Swipe Left	4	step_left ()	Move left by one step
Swipe Right	3	step_right ()	Move right by one step
Circle Clockwise	1	forward ()	Move forward
Circle Anticlockwise	2	backward ()	Move Backward
Screen Tap	5	stretch ()	Stretch Exercise

Table 7: Gesture to QuadBot Motion

Once the Arduino reads the integer sent to it by the Leap Motion Sensor, it executes that particular motion associated with that integer.

```

if (input == 1)           //Input stores the serial data sent from the Leap motion Sensor

{

    forward();           //Function to perform forward motion

    stand_up();          // Function to perform standing up motion

}

```

At the end of each motion, the QuadBot is programmed to return to the stand up position. This ensures stability and helps it to perform the next motions efficiently.

IV. RESULTS & DISCUSSION

During the execution of this project, two major problems were encountered.

1. Power supply to the motors

Once the hardware was set up, the QuadBot was initially tested by executing a few simple test functions. During this phase of testing, issues with power supplied to the motors were encountered due to which the QuadBot was not moving as desired. The problem was encountered as 8 servo motors were being driven with one Arduino. The total current these motors drain from it effected the USB operation.

Every motor drains about 4.8mA and 150mA in its idle and no load operating condition. Once all these motors are connected to an Arduino, they drain the required current from just the USB port. The theoretical current limit on a bus-powered USB port (as in a personal computer) is 100mA. It is like short-circuiting the USB port, and the protective circuit of computer will shut it down consequently, and hence it would not be able to transmit any data. According to the specification of the Arduino, the recommended input voltage the Mega is between 7 – 12 volts, when the input voltage limit is between 6 – 20 volts. However, when a 12 volt dc adapter was used, Arduino Mega board got burnt. When a 5 volts external supply along with a USB connecting to the computer at the same time was implemented, the QuadBot worked well. Hence, this set up was continued to be used.

2. Continuous forward motion of the QuadBot

Next, the gesture recognition of the LMS was tested. During this phase, different gestures were performed and the recognition and extraction of data was analyzed. The LMS recognized various gestures and the data was successfully extracted. This data was also successfully communicated to the Arduino via the serial port.

Initially, six distinct motions were programmed to be executed corresponding to six gestures. However, the QuadBot was able to execute only five of these motions successfully. The sixth motion, where the QuadBot should have moved forward continuously did not run properly. This motion was associated to the swipe up gesture. Although the gesture was successfully recognized and sent to the Arduino, the QuadBot was not able to execute the motion. A possible reason of this failure is the calibration and sequencing of motion of the motors.

Although some problems were encountered, the QuadBot was able to perform the desired objectives successfully. A demo of the project can be viewed by using the link below:

<https://www.youtube.com/watch?v=jrgyrCeQPd4&feature=youtu.be>

V. CONCLUSION & FUTURE WORK

This project is one of many examples of utilizing PWM signals generated by a microcontroller to control motors. The microcontroller used in this project was the Arduino Mega which was additionally interfaced with the Leap Motion Sensor (LMS). Using the capability of the LMS to recognize hand gestures such as swipe, circles, palm orientation etc. the QuadBot was programmed to execute various motions. In this project, five hand gestures were utilized and each gesture was associated with a specific QuadBot motion.

In addition to the PWM control, another important realization of this project was the use of microcontrollers such as the Arduino. These controllers offer a simple and inexpensive solution for controlling servo motors & other such devices for similar robotic and electronics projects.

In the future, the following goals will be pursued

1. Wireless operation of the QuadBot

Due to the power issues faced, the QuadBot was operated with an external power supply connected to it. A suitable power supply should be designed, assembled and installed on the QuadBot so that it can operate wirelessly. This would allow for the QuadBot to move more freely.

2. Increasing the number of hand gestures

Currently, the QuadBot is programmed to respond to 5 gestures. A possible improvement is increasing the number of gestures the QuadBot will respond to. Another possible extension to design is for motion sensitivity of the QuadBot. An example would be slower motion of the QuadBot when a larger circular gesture is made.

3. Duration of Motion

The QuadBot is currently programmed to perform a movement for a fixed duration (one step) and then it returns to idle. A possible extension to this is programming the QuadBot to continue its motion until a new gesture is recognized. This will entail detailed calibration and sequencing of the motors such that the QuadBot does not deviate from a desired path.

4. Speed of the QuadBot

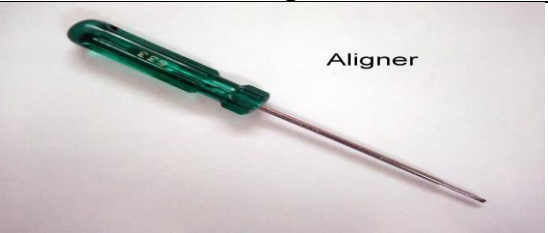
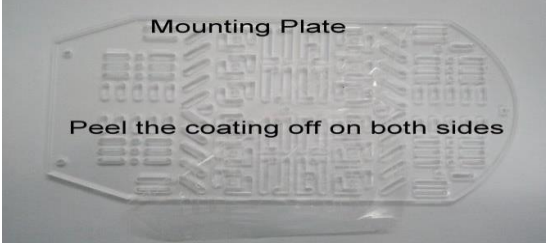
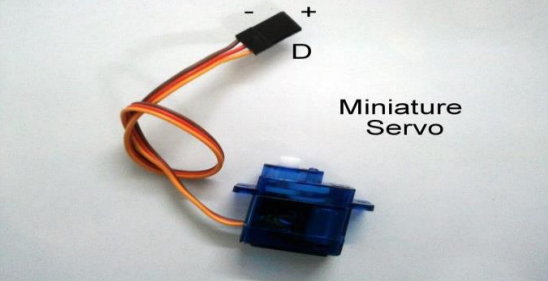

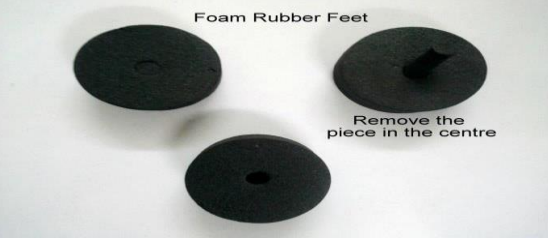
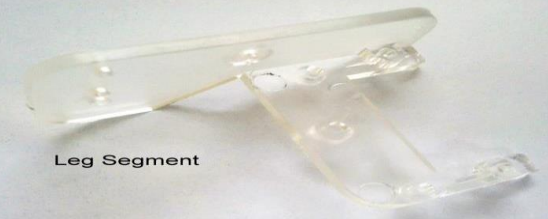
Since we are only using position control of the servos, the servos are always operating at full speed. By controlling the power line of the servos, the speed can be altered. This would require further investigation into the power distribution from the Arduino.

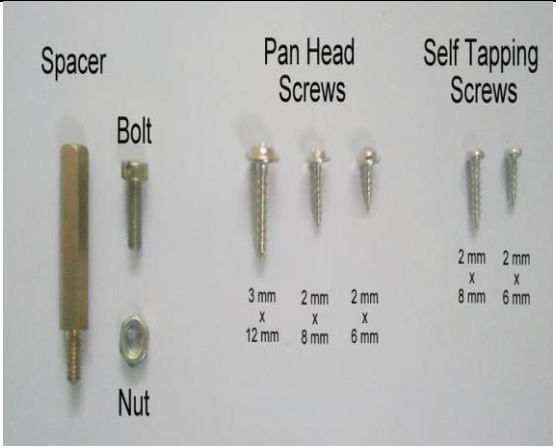
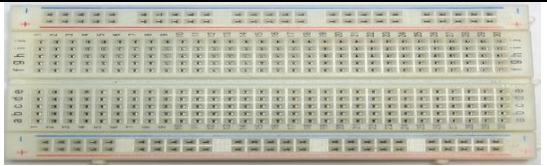
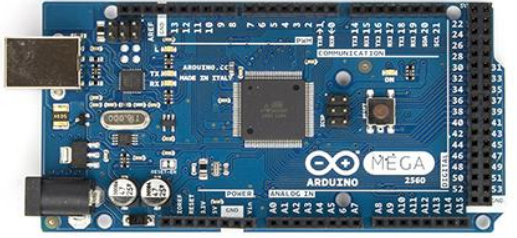
VI. REFERENCES

1. Arduino - HomePage. (n.d.). Retrieved June 11, 2014,
<http://www.arduino.cc/>
2. Leap Motion | Mac & PC Motion Controller for Games, Design (n.d.)
<https://www.leapmotion.com/>
3. Simple Labs. (n.d.). . Retrieved June 11, 2014
<http://www.simplelabs.co.in/>
4. PINCKNEY, N. Pulse-width modulation for microcontroller servo control.
IEEE POTENTIALS, 0278-6648/06/, 27-29.
5. Lab Manual 1 “Control Applications Using DSP” B.C.Chang and Mishah Salman.
Department of Mechanical Engineering and Mechanics, Drexel University.
6. Lab Manual 3 “Open-Loop DC Motor Control” B.C.Chang and Mishah Salman.
Department of Mechanical Engineering and Mechanics, Drexel University.
7. Lab Manual 4 “Modeling & Simple Feedback Control” B.C.Chang and Mishah Salman.
Department of Mechanical Engineering and Mechanics, Drexel University.
8. Guna, J., Jakus, G., & Pogacnik, M. An Analysis of the Precision and Reliability of the Leap
Motion Sensor and Its Suitability for Static and Dynamic Tracking. Sensors, 3702-3720,
www.mdpi.com/journal/sensors

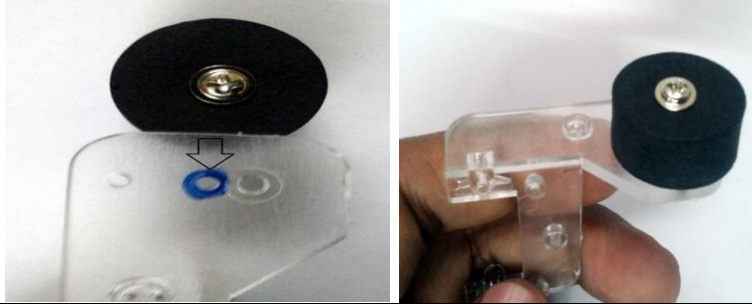
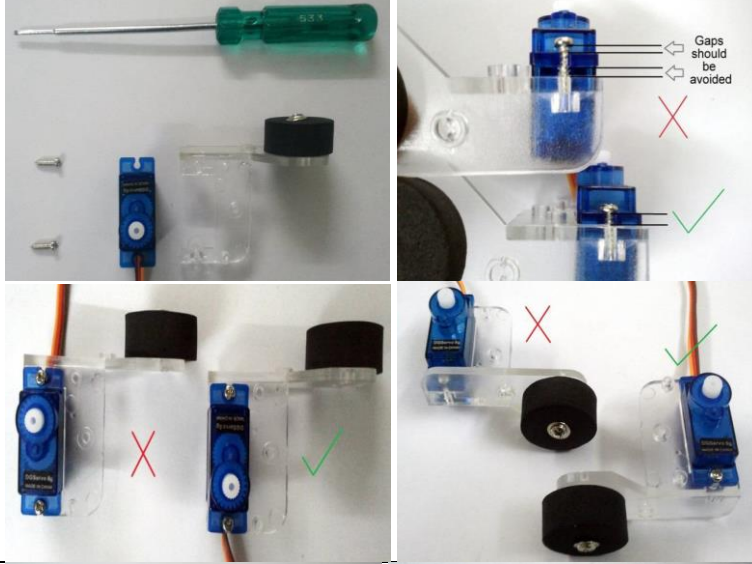

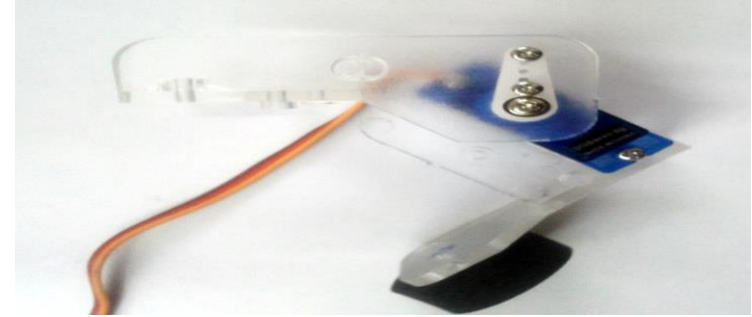
VII. APPENDIX

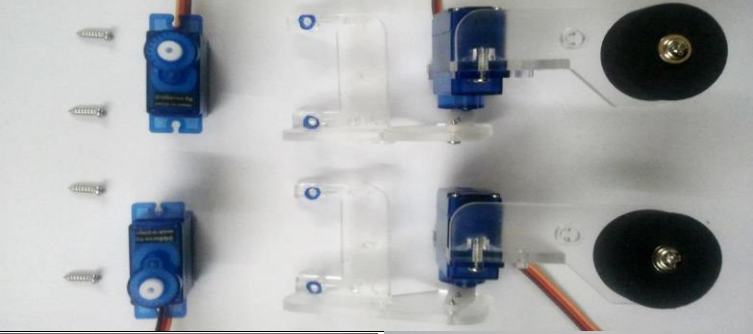
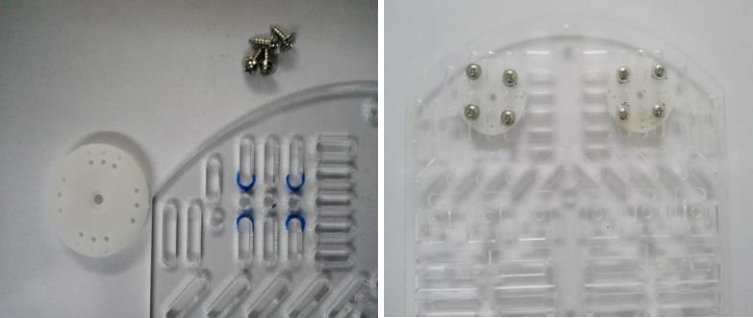
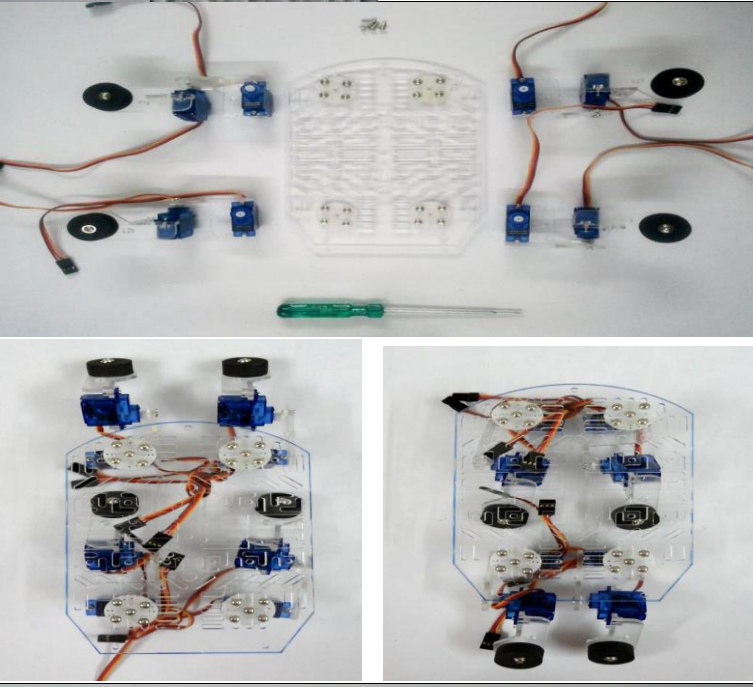
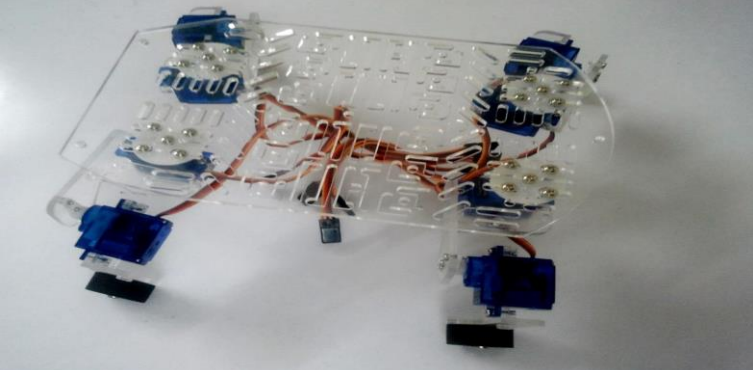
I. Materials Required

Part #	Item	Quantity	Image
1	Aligner	1	 <p>Aligner</p>
2	Laser-cut mounting plate	1	 <p>Mounting Plate</p> <p>Peel the coating off on both sides</p>
3	Miniature Servos	8	 <p>Miniature Servo</p>
4	Servo Horns (Linear +circular)	8	 <p>Servo Horns</p> <p>Circular</p> <p>Linear</p>
5	Foam rubber feet	4	 <p>Foam Rubber Feet</p> <p>Remove the piece in the centre</p>
6	Leg Segments	8	 <p>Leg Segment</p>

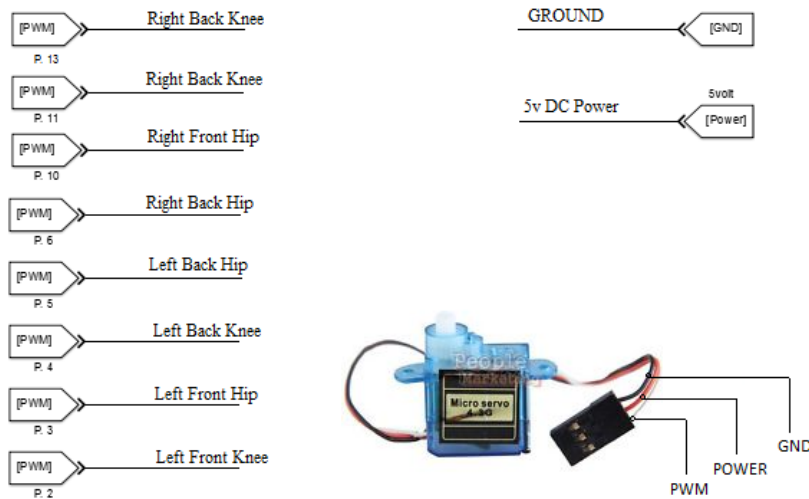
7	3 mm x 12 mm Pan head screws	4	
8	2 mm x 8 mm Pan head screws	8	
9	2 mm x 6 mm Pan head screws	16	
10	2 mm x 8 mm Self Tapping screws	16	
11	2 mm x 6 mm Self Tapping screws	8	
12	20 mm Spacers	4	
13	Nuts	4	
14	Bolts	4	
15	Breadboard	1	
16	Arduino Mega 2560	1	

II. Assembly Procedure

Step #	Procedure	Image
Step 1	Mount the four foam rubber feet to leg segments using the 3 mm x 12mm pan head screws.	
Step 2	Mount a servo on each of the leg segments fitted with a foot using the 2 mm x 8mm self-tapping screws.	
Step 3	Mount the linear servo horns to unused leg segments using 2 mm x 6 mm self-tapping screw.	
Step 4	Mount the leg segments to the servos using a 2 mm x 8 mm pan head screws as shown. Gently turn the servo by hand to check the range of movement and adjust if necessary. The servos should be able to move in between both of them. If not, unscrew and adjust.	

Step 5	<p>Mount two servos as shown in the left image below and two servos as shown in the right image. Note the orientation of the servos. Use 2 mm x 8 mm self-tapping screws.</p>	
Step 6	<p>Mount the 4 circular servo horns to the mounting plate using 2 mm x 6 mm pan head screws in the positions shown below. Look for the pattern of holes marked. There are exactly 4 such patterns on the mounting plate, one each for a servo horn.</p>	
Step 7	<p>Secure the legs to the servo horns, which have been mounted on the mounting plate, using the 2 mm x 8 mm pan-head screws. Note the orientation of the servos and the legs.</p>	
Step 8	<p>The final assembled robot chassis should look as shown.</p>	

III. Wiring Schematic



IV. Arduino Codes

- angles.h

```
int displacement=30;    // Displacement angle of every step
int delay_time=5;      // Time delay between displacement of every angle
```

- pins.h

```
#define RECV_PIN 9 //TSOP pin
```

```
int servo_pin[8]={
    2,11,13,4,    //Knee servos' pins in the order of Left Front, Right Front, Right
Rear, Left Rear
    3,10,6,5};    //hip servos' pins in the order of Left Front, Right Front, Right
Rear, Left Rear
```

- forward()

```
void forward()
{
    /////
    for(int i=0;i<displacement;i++)    // Raise the Left Front and Right Rear legs
    {
        move_servo(0,-1);
        move_servo(2,-1);
```

```

    delay(delay_time);
}
for(int i=0;i<displacement;i++) // Move the Left Front and Right Rear legs forward
{
    move_servo(4,+1);
    move_servo(6,-1);
    delay(delay_time);
}
for(int i=0;i<displacement;i++) // Lower the Left front and Right Rear legs
{
    move_servo(0,+1);
    move_servo(2,+1);
    delay(delay_time);
}
for(int i=0;i<displacement;i++) // Move the Left Front and Right Rear legs back &
// Raise the Right Front and Left Rear legs
{
    move_servo(4,-1);
    move_servo(6,+1);
    move_servo(1,-1);
    move_servo(3,-1);
    delay(delay_time);
}
for(int i=0;i<displacement;i++) // Move the Right Front and Left Rear legs forward
{
    move_servo(5,-1);
    move_servo(7,+1);
    delay(delay_time);
}
for(int i=0;i<displacement;i++) // Lower the Right Front and Left Rear legs
{
    move_servo(1,+1);
    move_servo(3,+1);
    delay(delay_time);
}
for(int i=0;i<displacement;i++) // Move the Right Front and Left Rear legs back
{
    move_servo(5,+1);
    move_servo(7,-1);
    delay(delay_time);
}
}

```


V. Leap Motion Controller Codes

```
import com.onformative.leap.*;
import org.firmata.*;
import cc.arduino.*;
import processing.serial.*;
import com.leapmotion.leap.Gesture;
import com.leapmotion.leap.SwipeGesture;
import com.leapmotion.leap.CircleGesture;
import com.leapmotion.leap.ScreenTapGesture;
import com.leapmotion.leap.Gesture.State;
import com.onformative.leap.LeapMotionP5;
import com.leapmotion.leap.Hand;
LeapMotionP5 leap;
float backgroundColor;

Serial myPort;
PImage bg;
public void setup() {
    size(300, 300);

    //dont generate a serialEvent() unless you get a newline character from arduino:
    myPort.bufferUntil('\n');
    leap = new LeapMotionP5(this);
    leap.enableGesture(Gesture.Type.TYPE_SWIPE);
    leap.enableGesture(Gesture.Type.TYPE_CIRCLE);
    backgroundColor = (0);
}

public void draw() {
    background(backgroundColor);
    for (Hand hand : leap.getHandList()) {
        PVector handPos = leap.getPosition(hand);
        ellipse(handPos.x, handPos.y, 20, 20);
    }
}

public void circleGestureRecognized(CircleGesture gesture, String clockwiseness) {
    if (gesture.state() == State.STATE_STOP) {
        println("////////////////////////////////////////");
        println("Gesture type: " + gesture.type().toString());
        println("ID: " + gesture.id());
    }
}
```

```

println("Radius: " + gesture.radius());
println("Normal: " + gesture.normal());
println("Turns: " + gesture.progress());
println("Clockwiseness: " + clockwiseness);
println("Turns: " + gesture.progress());
println("Center: " + leap.vectorToPVector(gesture.center()));
println("Duration: " + gesture.durationSeconds() + "s");
println("////////////////////////////////////////");
if(clockwiseness == "clockwise")
{
    backgroundColor = 255;
    myPort.write(1);
}
else
{
    backgroundColor = 0;
    myPort.write(2);
}
}
else if (gesture.state() == State.STATE_START) {}
else if (gesture.state() == State.STATE_UPDATE) {}
}

public void swipeGestureRecognized(SwipeGesture gesture) {
if (gesture.state() == State.STATE_STOP) {
    println("////////////////////////////////////////");
    println("Gesture type: " + gesture.type());
    println("ID: " + gesture.id());
    println("Position: " + leap.vectorToPVector(gesture.position()));
    println("Direction: " + gesture.direction());
    println("Duration: " + gesture.durationSeconds() + "s");
    println("Speed: " + gesture.speed());

    println("////////////////////////////////////////");

    if(gesture.direction().getX() > 0)
    {
        myPort.write(3);
        println("Right Swipe");
    }
    else
    {
        myPort.write(4);
    }
}
}

```

```

println("Left Swipe");
}

for (Hand hand : leap.getHandList())
{

    PVector handPos = leap.getPosition(hand);
    println("position hand " + handPos);
    if(handPos.y > 200) {    //myPort.write(2);
    }

}
}
else if (gesture.state() == State.STATE_START) { }
else if (gesture.state() == State.STATE_UPDATE) { }
}

public void screenTapGestureRecognized(ScreenTapGesture gesture) {
if (gesture.state() == State.STATE_STOP) {
    println("////////////////////////////////////////");
    println("Gesture type: " + gesture.type());
    println("ID: " + gesture.id());
    // println("Position: " + leap.vectorToPVector(gesture.position()));
    //println("Direction: " + gesture.direction());
    //println("Duration: " + gesture.durationSeconds() + "s");
    //println("Speed: " + gesture.speed());
    myPort.write(5);
    println("////////////////////////////////////////");

    for (Hand hand : leap.getHandList())
    {
        PVector handPos = leap.getPosition(hand);
        if(handPos.y > 200) {    // myPort.write(4);
        }
    }
}
else if (gesture.state() == State.STATE_START) { }
else if (gesture.state() == State.STATE_UPDATE) { }
}

public void keyTapGestureRecognized(ScreenTapGesture gesture) {
if (gesture.state() == State.STATE_STOP) {
    println("////////////////////////////////////////");

```

```
        println("Gesture type: " + gesture.type());
        println("ID: " + gesture.id());
        // println("Position: " + leap.vectorToPVector(gesture.position()));
        //println("Direction: " + gesture.direction());
        //println("Duration: " + gesture.durationSeconds() + "s");
        //println("Speed: " + gesture.speed());
        myPort.write(6);
        println("////////////////////////////////////////");

    }
    else if (gesture.state() == State.STATE_START) { }
    else if (gesture.state() == State.STATE_UPDATE) { }
    }

    public void stop() {
        leap.stop();
    }
}
```