HEXAPOD ROBOT

Final Year Report 2013

10/31/2013

Ben Dabin (0946177)

Contents Page

Introduction

The purpose of the project was to design and develop a robot hexapod controller for controlling 18 servos for a robot spider kit. The materials were supplied on behalf of ***AUT (Auckland University of Technology).*** A custom made ***PCB (Printed Circuit Board)*** will be designed and built for controlling the servos. The 18 servo robot will have 4 different walk modes allowing for ***Forwards***, ***Reverse***, ***Left*** and ***Right.***
There will also be other additional hardware features included such as an ***LCD*** screen for displaying menu functions, control buttons for selecting user choices and an on-board ***Bluetooth module*** for wireless ***remote communication***. Wireless ***remote communication*** will be controlled via an ***Android Application GUI (Graphical User Interface)***, programmed on a ***Samsung Galaxy S3 Smartphone***.

Powering the system will be done from a single ***on board battery source***, so choosing the right battery to do the job was an essential part of the project.

The report will be divided up into various sections for the project each detailing different areas and challenges that had to be overcome and resolved, which will be discussed in more detail later on. The report will mention in full detail regarding the ***research***, ***design*** and ***development*** stages for both the ***software*** and ***hardware*** aspects of the project. The project comprised of almost ***50% software*** and ***50% hardware***.

The report will begin by detailing the H***ardware Research aspects*** of the project which will briefly explain some of the important hardware features that were considered and or used. This will then be followed by a section on the ***Power Requirements*** that will mention about how the robot will be powered, and explain other important features that was considered for final ***PCB design***.

Next will be a section on the ***software aspects,*** for the ***Android App Development*** section. This section will go into full-detail about the ***GUI and Bluetooth Code*** that was developed on the ***S3 Smartphone***, and the developing software used.

The next sections onwards will mention information about the ***Final PCB design and schematics,*** and for the ***embedded programming code.*** There will also be a section detailing the ***Faults and Obstacles*** that had been encountered with the project in general. Both the software and hardware faults will be mentioned. The project will conclude with a small section listing the materials used and then finish with a conclusion.

Theory and Applications of a Robot Hexapod

The term meant by hexapod robot is where there are six legs, which provide for six different axis of movement. The most common and useful applications for a robot hexapod are most useful in the hobby and recreational industries. This could also be useful for scientific and research purposes. For example ***NASA*** has sent the ***Mars Rover*** robot to explore the terrain of the planet ***Mars***. The mars rover robot has six legs which provide for increase stability on navigating and exploring uneven terrains. However this robot differs to the robot that is being developed for this project in that it has no wheels, significantly smaller size, and use low budget materials.

Another useful application for the robot hexapod is that it can be used to emulate the behaviour of an insect such as a spider or ant. This can be particularly useful in the science and research industry for studying and learning the behaviour of insects and then implementing this on a real life model.

The project is more about the learning and researching aspects of the project as opposed to developing a product that can be useful in the real world.

Hardware Research

Summary

This section will mention about the important hardware aspects that were considered for the final development of the hexapod robot controller ***PCB.*** The ***microcontroller*** types used for the project will also be mentioned. The ***microcontroller*** is the most important hardware feature because it is the brains for the whole system that controls everything that the hexapod robot does. This hardware device essentially bridges the gap between the hardware and the software so therefore a suitable **microcontroller** with a sufficient on-board hardware features had to be chosen.

The servos on the servo kit provided which was ***Tower Pro MG995*** servos will be mentioned in more detail.

Microcontroller

It was important that the appropriate ***microcontroller*** for the project be chosen one that has sufficient hardware features such as ***PWM (Pulse Width Modulation) channels***, ***USART (Universal Asynchronous Receiver Transmitter), ADC's***, etc.

The project used 2 different microcontroller types were which were both manufactured by ***ATMEL***. The ***ATMEGA16*** microcontroller was used for prototyping and testing. The reason for using this microcontroller is that it can easily be built on breadboard, as shown in ***Figure 1.2*** and also comes available in a ***PDIP*** package as shown in ***Figure 1.2.1.***

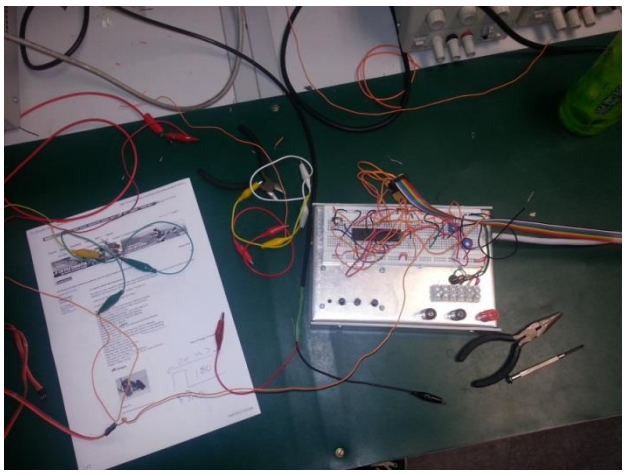

Figure 1.2

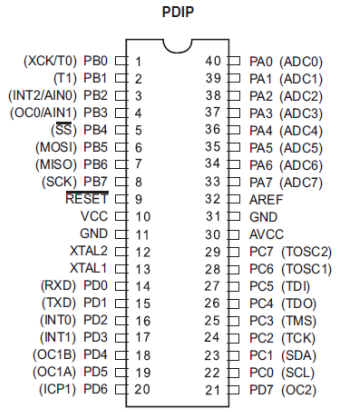

Figure 1.2.1

The other microcontroller that was used and considered for the final PCB design was the ***ATMEGA1280.*** This microcontroller was chosen because it has ***86 I/O ports*** and ***15 PWM channels,*** making it the idea component for controlling the ***18 servos,*** and other hardware features. This ***microcontroller*** also comes equipped with ***4 USART (Universal Asynchronous Receiver Transceiver)***, where one will be required for interfacing with the ***Bluetooth Module,*** and the other remaining USARTS could possibly be used for connecting other optional features such as sensors, voice recognition modules, etc. The main drawback with this component is that it is only available in a ***100 pin quad package*** (**TQFP**) in a surface mount package, ***Figure 1.2.2*** which basically means that it cannot be easily prototyped or tested on breadboard.

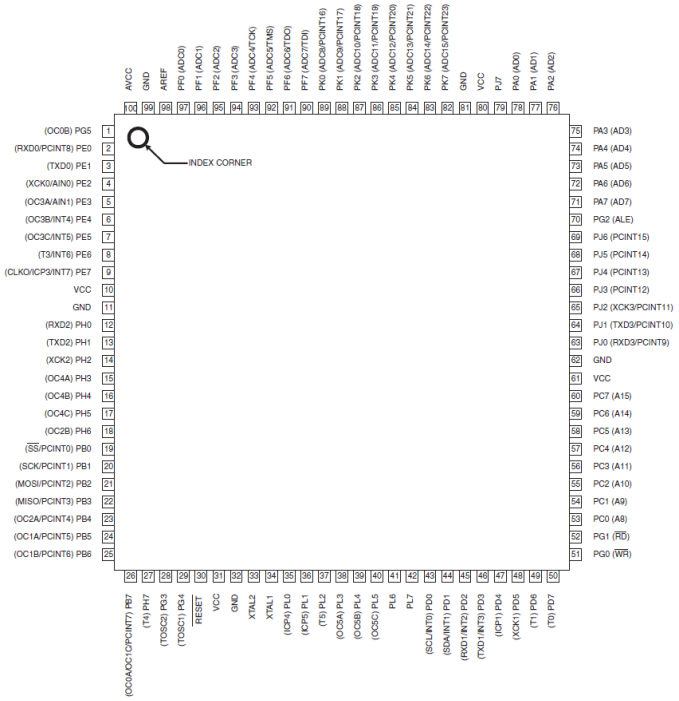

Figure 1.2.2

Considering both microcontroller types, the ***ATMEGA16*** provided the main basis for building all the prototyping circuits used. The other reason was that it was readily available and had similar but reduced amount of critical features that the ***ATMEGA1280*** had. Both microcontrollers were also equipped with a ***JTAG*** programming interface and can be programmed the same way as each other.

JTAG requires four signals of communication which are TCK, ***TDO***, ***TMS*** and ***TDI*** as show in ***Figure 1.2.3.***

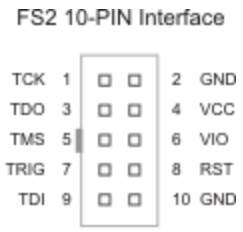

Figure 1.2.3

Figure 1.2.1 shows the ***AVR dragon*** that is the programmer used for programming both the ***ATMEGA16*** and ***ATMEGA1280*** micro-controllers which is programmed in JTAG mode.

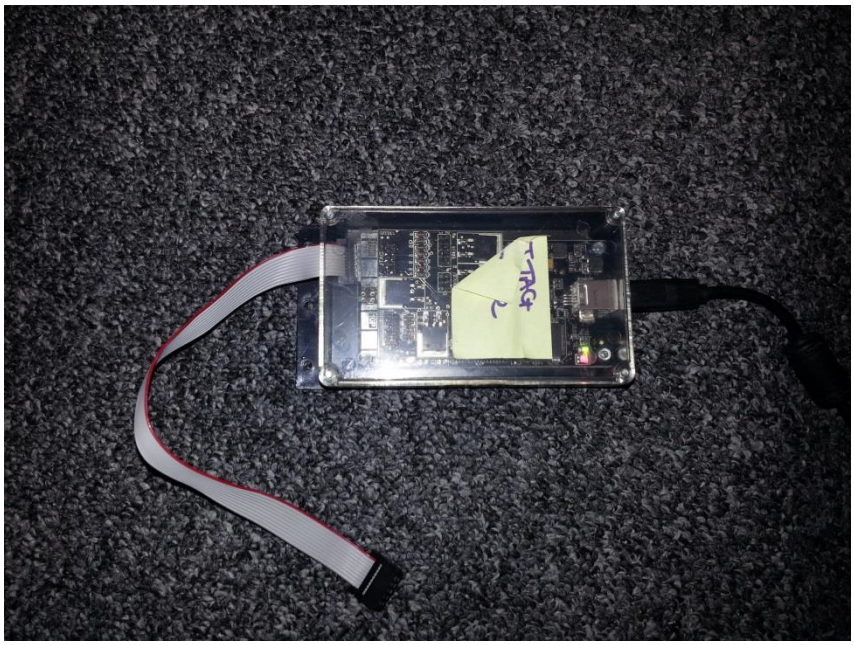

Figure 1.2.4

Servo Kit

The ***18 servo leg kit*** was the main item supplied by ***AUT*** for which the ***controller PCB*** will be designed for, as shown in ***Figure 1.3.*** The top aluminium base plate is to house the ***PCB*** and the ***bottom base plate*** will be for the battery module which will be discussed in more detail in the next section.

Figure 1.3

Figure 1.3.1

Figure 1.3.1 shows a close-up image of a servo leg, where there are 3 servos on each leg which each provide for different axis of movement, more information about this will be explained later on.

It was also important to understand the brand of Servos that was provided with the servo kit. ***Figure 1.3.2*** shows a ***Tower Pro MG995 servo***, which are the servos that are attached to the kit.

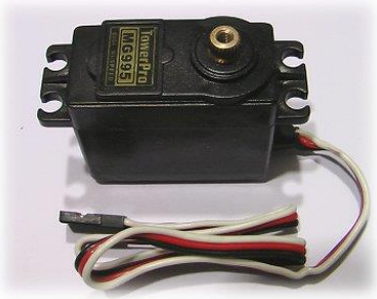

Figure 1.3.2

For the programming aspects of the servos which will be mentioned later on, it was important to gain some insight and understanding into the ***timing requirements*** for each servo as shown in ***Figure 1.3.2.*** The diagram shows the required timing pulses to rotate a servo at a particular angle.

This information was useful for prototyping purposes because when a testing circuit was built on breadboard consisting of connecting one servo, ***PWM*** values could be sent to that servo, and the angle of rotation for that value could be determined. Another advantage of this was that the optimal PWM frequency could be found as well. Experimenting and research showed that to be ***50HZ*** which equates to a period of ***20ms***. The power requirements for that servo was also recorded where an Ammeter was connected in series with the supply voltage pins. The servo was also rotated at an angle and current was recorded when pressure was applied to the servo leg. The current for a servo under a load was shown to be ***2 peak Amps***.

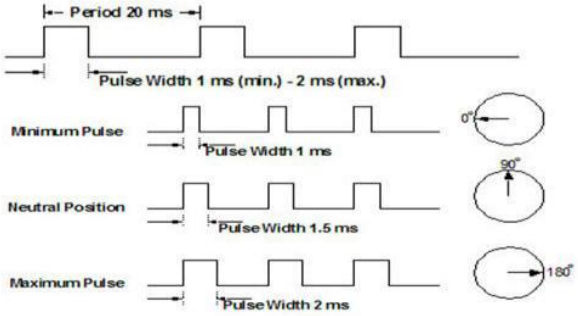

Figure 1.3.2

The pinout for a servo connector are shown in ***Figure 1.3.3***

Red Wire is +6V
Brown Wire is Ground
Orange Wire is Signal (PWM)

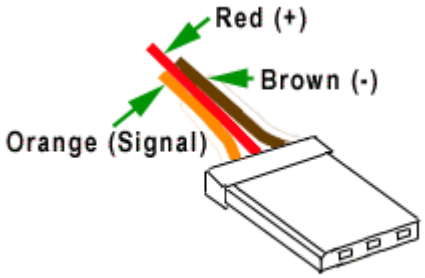

Figure 1.3.3

Bluetooth Module

The chosen Bluetooth module for the project is the ***BlueSmirf HID Bluetooth module***. The ***Bluetooth Module PCB*** is manufactured by ***Sparkfun Electronics (Figure 1.7)***. This module has a wireless range of ***10 metres*** and can be configured in 2 different profiles, which are called ***HID*** and ***SPP*** mode. ***HID*** stands for ***Human Interface Device***, which is mainly used for wireless Bluetooth device communication such as keyboards and mice, etc. The mode that is most useful for the required purpose is ***SPP (Serial Port Profile)*** because this allows raw data to be sent directly from a ***Smartphone.*** The raw data can be sent and received to and from the ***Bluetooth Module*** and then decoded through the microcontroller USART as indicated by the diagram in ***Figure 1.6***.

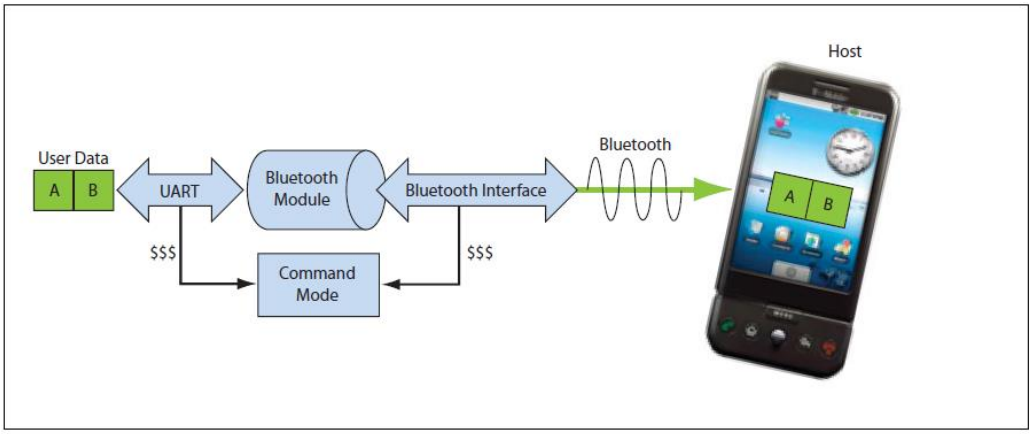

Figure 1.6

The desired baud rate for the purpose of the project is ***9600 baud (100us/bit).*** The default configuration for the Bluetooth module is 115.2K baud, 8 bits, no parity and 1 stop bit. This can be changed using ***Tera Terminal***, which is a ***DOS window program*** on a computer that can be used to configure the module by sending commands wirelessly over a Bluetooth connection, more information about this will be explained later.

Figure 1.7

Power Requirements

Summary

Power supplying was important because power had to be provided from a single ***on-board power source***. The first considerations that had to be taken into account were, to decide the type of battery to use and the battery voltage required. A ***+6V DC*** power source was decided on because the voltage was low enough to power all the 18 servos directly without burning them. Another reason for choosing the ***+6V*** energy source was to avoid the need for adding additional voltage regulating components to lower the voltage drop to ***+6V*** for all the 18 servos. This would be the case if a higher battery voltage rating was used because each servo draws ***1-2amps peak amps*** of current when tested under a load. This factor was considered when the final PCB was designed because this had to take into account for a ***20Amp*** current draw as worst case for all 18 servos being turned on at once. Therefore finding a voltage regulator to supply that much current proved too difficult and complicated because most voltage regulators types that were looked at were unable to provide that much current.

A **+5V** energy source was needed to provide power to the ***microcontroller, LCD*** and ***Bluetooth module***. The ***+6V*** power source presented some other levels of complications. The complications were that a suitable voltage regulator had to be chosen, one that would provide a low enough dropout voltage to transform into ***+5V DC***. The term dropout voltage is the difference between the input and the output. In this case the dropout voltage was ***+1V***. Most voltage regulator types that was looked at were rated higher than ***+1V***. A ***Buck Boost Regulator*** was chosen because it was the most efficient regulator of choice and has the advantage of converting ***input voltages*** lower than its ***output voltage*** which was ideal for the required purpose.

A series of different battery types were researched and looked into because the ***battery*** footprint size requirements had to be sufficient enough to mount on underside of the metal base frame and have a sufficient ***AH hour capacity***. The batteries types looked at were ***lead acid***, ***Lithium poly***, and ***NiMH (Nickel Metal Hydroxide)***.

Lithium poly, was considered but not used because the closest voltage to the required voltage was ***+7.2V*** which was considered too high for the servos.

Voltage regulators

Linear Regulator

The ***LT1529-5*** linear voltage regulator type was previously looked at and considered. The reasons for not choosing this regulator is that , these can be very inefficient and dissipate an excess amount of heat and heat sinking would be required.

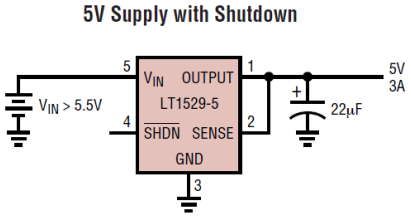

Figure 1.8

Buck boost regulator

The schematics shown in ***Figure 1.8.1,*** LM5118 *is a* Buck boost regulator chip which is manufactured by Texas Instruments (TI). This will also be the final voltage regulator of choice. The circuit shows and recommends the following components; however some of the components had to be substituted for other component values because most of the values were not readily available in New Zealand and the footprints were too complicated to solder by hand.

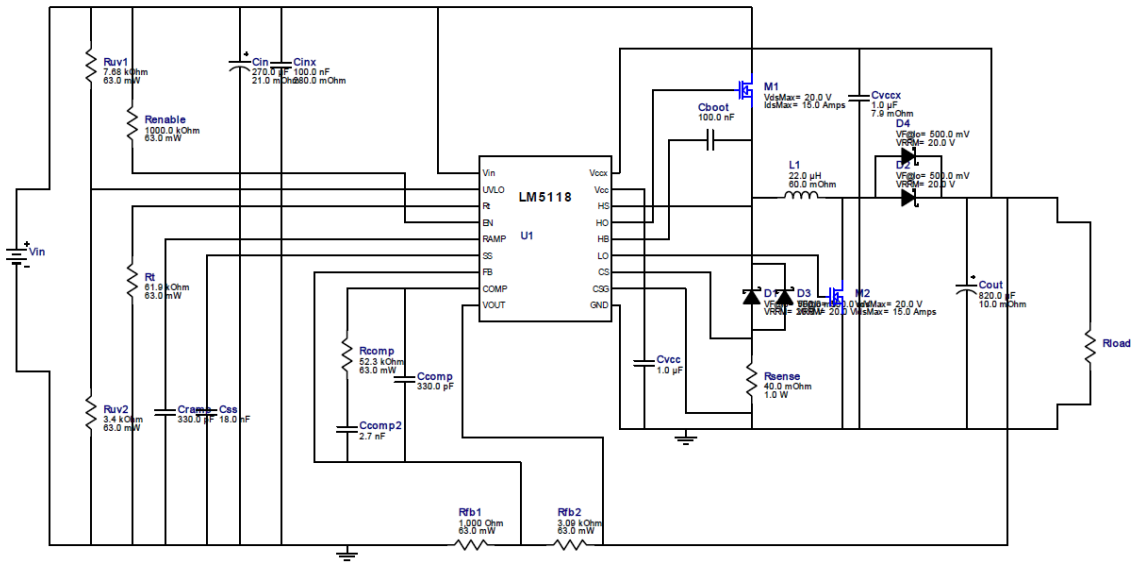

Figure 1.8.1

Batteries

Lead Acid

This was used for the testing and debugging the project because it had the advantage of supplying a large amount of current to the PCB controller and servos. The only disadvantage of this is that the battery was too large and too heavy to be mounted on the frame. Therefore, when the battery was used for testing and debugging, power cables were made longer to provide for sufficient slack. Having longer wires can also create other issues such as input impedances which could potentially contribute to power loss, and voltage drops in wires.

For testing the robot a ***DiaMec***, +6V 12AH lead acid battery was used purchased from ***JayCar Electroncs*** as shown in ***Figure 1.9.*** A special charger was also purchased so that the battery could be charged. The advantage of using this battery is that it removes the need for having additional power supplies for testing the robot and it can supply a large amount of current. Also shown in the figure is the ***+5V*** adapter which only provides power to the ***microcontroller circuit, Bluetooth Module***, and ***LCD***. The PCB used in this photo is the first PCB. The reason for using the ***+5V*** adapter in the following example will be explained in more detail later on.

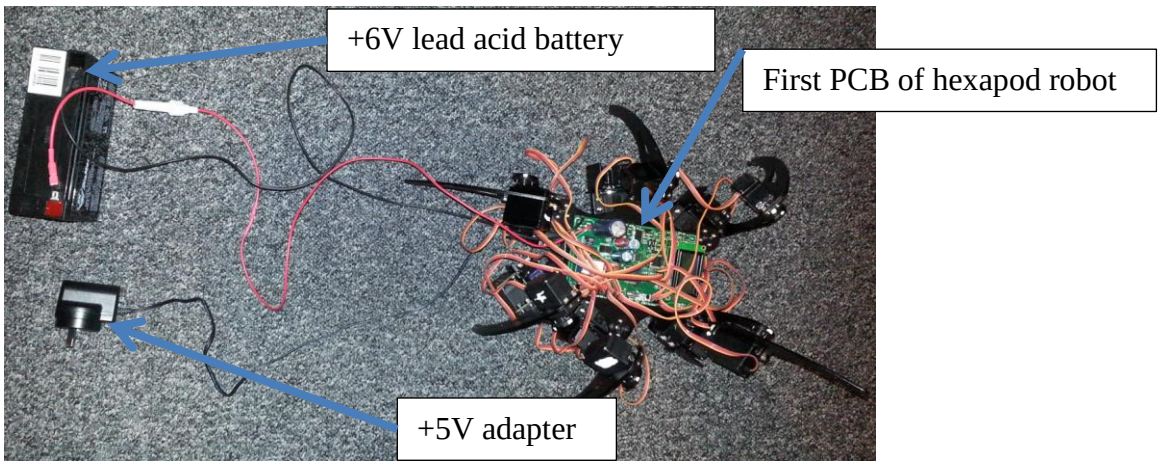

Figure 1.9

NiMH(Nickle Metal Hydroxide)

This was the preferred battery of choice for the final product. The advantage of using this type of battery was that it was light weight, compact and could be mounted on the robot. Another advantage for using the battery was that it is very powerful and can supply a lot of current as it has a rating of ***5.5AH***. The batteries are made up of ***five*** 1.5V cells, which were shrink wrapped to form a single ***+6V*** battery. The battery was manufactured by a power company called ***SimPower. Figure 1.9.1*** Shows the battery mounted on the underside of the finished product of the hexapod robot.

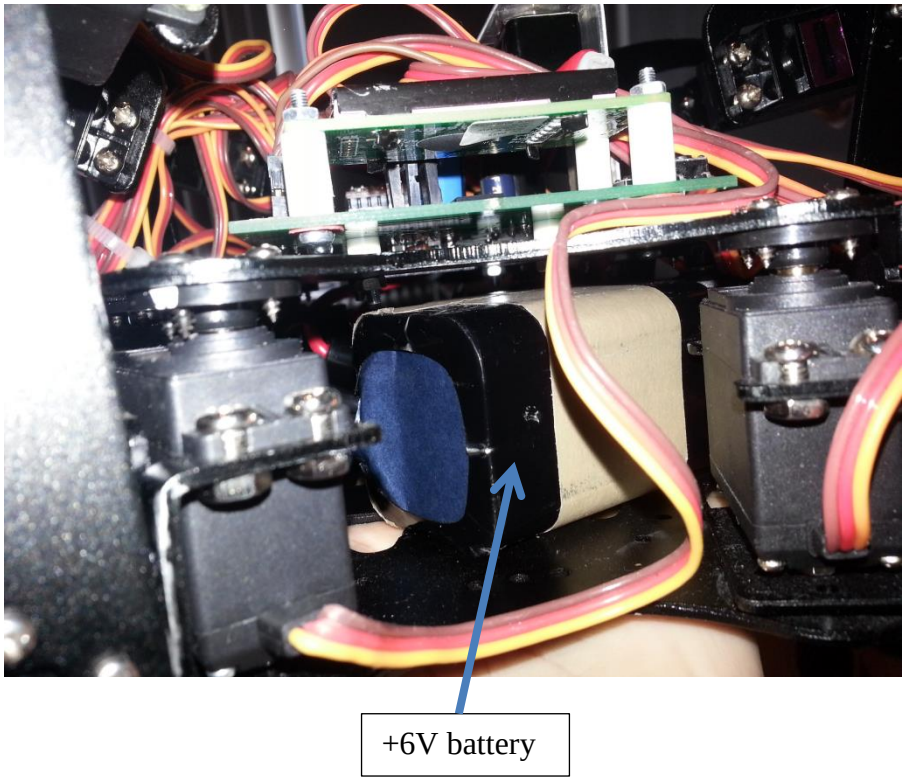

Figure 1.9.1

The existing wires on the battery had to be modified with thicker wires. This is so that more current can be drawn when needed.

Android App Development

Summary

This section explains the ***Android Application GUI*** *and* ***Bluetooth Code*** that was developed and programmed on a ***Samsung Galaxy S3 Smartphone***. The ***GUI*** consists of touch screen buttons that allow the user to select different walk mode options for the robot. The ***Bluetooth Code*** is the essential link that connects the ***GUI*** to the Smartphones ***Bluetooth Adapter***. This application was written in the ***Java*** programming language. The development environment known as ***Eclipse*** was the software used for developing the app.

Before the PCB was developed, the prototyping testing circuit shown in ***Figure 2.0*** with the Bluetooth module connected was used to test the operation and functionality of the ***GUI.*** The reason why the circuit was necessary was because the microcontroller had to receive characters being sent from the ***GUI***, while receiving data from the Bluetooth module through the USART. This was an important factor for the final PCB.

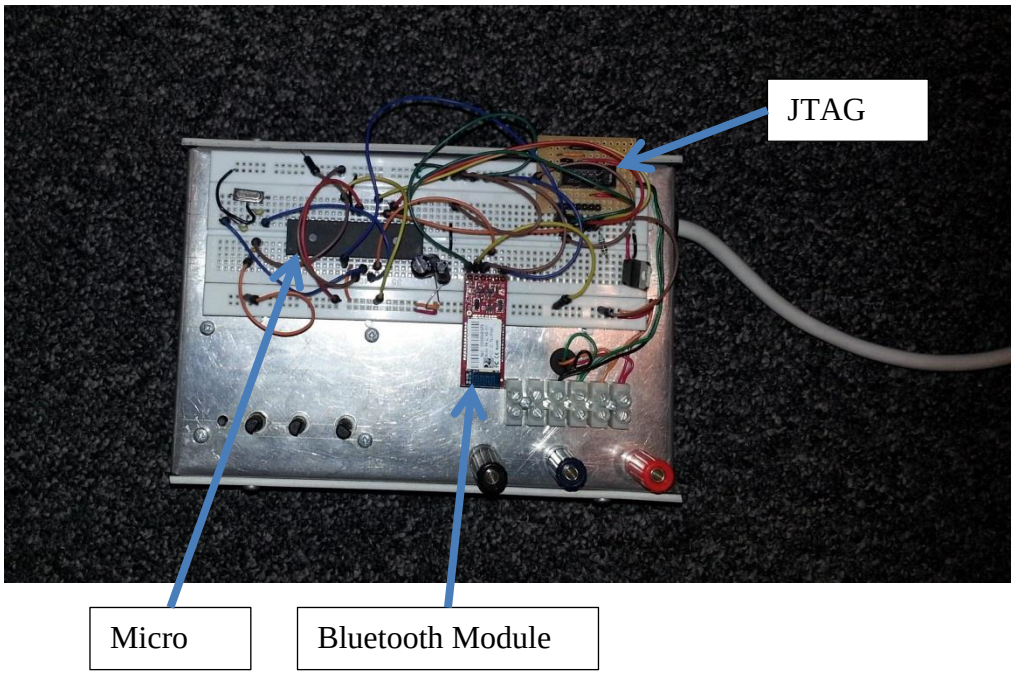

Figure 2.0

Eclipse Software IDE

This section will briefly explain the ***Eclipse software IDE*** and some other important features that were used for developing the ***Android App***. ***Eclipse*** is open source software program that is readily available to download from the internet. ***Figure 2.1*** shows a screen capture of the ***Eclipse*** IDE which was used for writing the app. In the following screen capture the IDE, the main coding window shows the ***Bluetooth Code*** which is known by the filename, ***MainActivity.Java.***

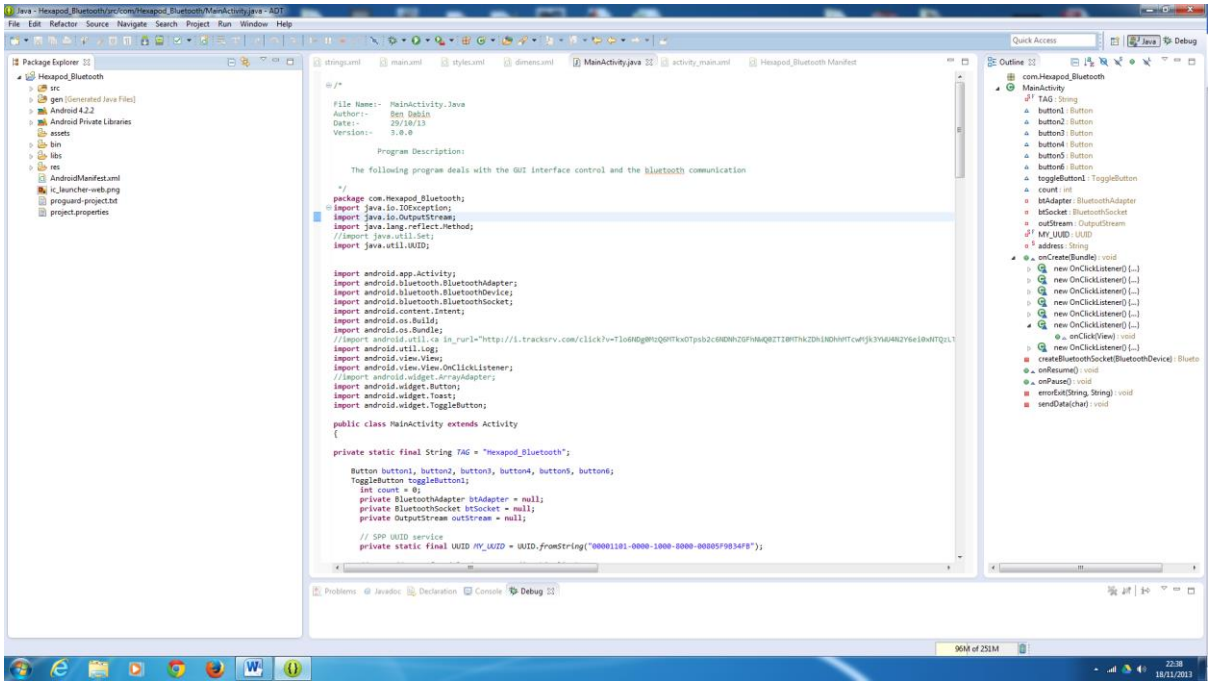

Figure 2.1

Figure 2.2, shows the ***graphical user interface (GUI)***, and the palette to the left allows graphical features such as buttons, widgets, etc to be added to the main layout. The behaviour of the buttons and or widgets can be programmed in the ***MainActivity.Java*** file, which is represented by an assigned function name. Code can be written to the assigned function to perform various types of behaviours. In the case of ***GUI*** that was developed for the project, the buttons displaying the walk modes can be assigned to send unique identifiers when pressed by the user, which will be explained in more detail later on.

Once the ***GUI*** has been developed and the programming code associated with it, the application can be downloaded onto the smartphone. This can be done by launching the ***run*** tab option and then clicking ***run*** as shown in ***Figure 2.3***, alternatively ***Ctrl+F11*** can also be done. Once the ***run*** tab has been launched, the ***Android Device Chooser*** dialogue box appears which informs the user about the desired hardware target for downloading the App. In the case of the screenshot example the ***Samsung S3 smartphone*** has showed up as the target. This is because the phone was connected to the computer at the time this example was taken. If the phone had not been connected, the following option would not have been available to the user.

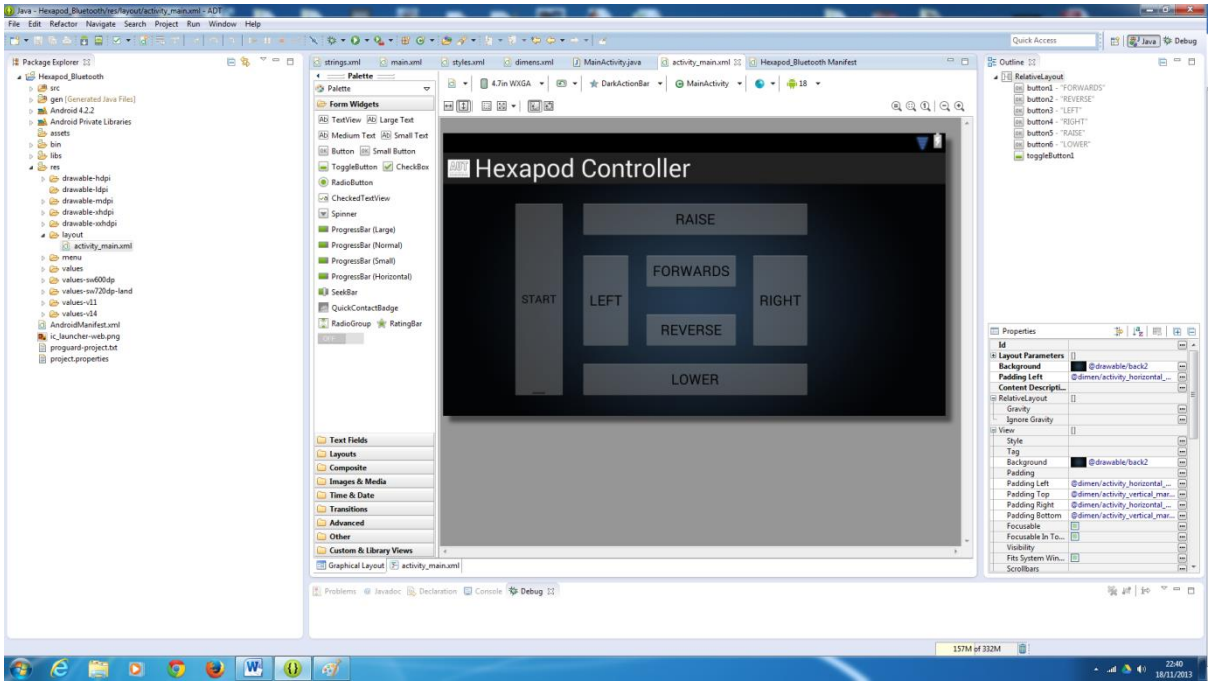

Figure 2.2

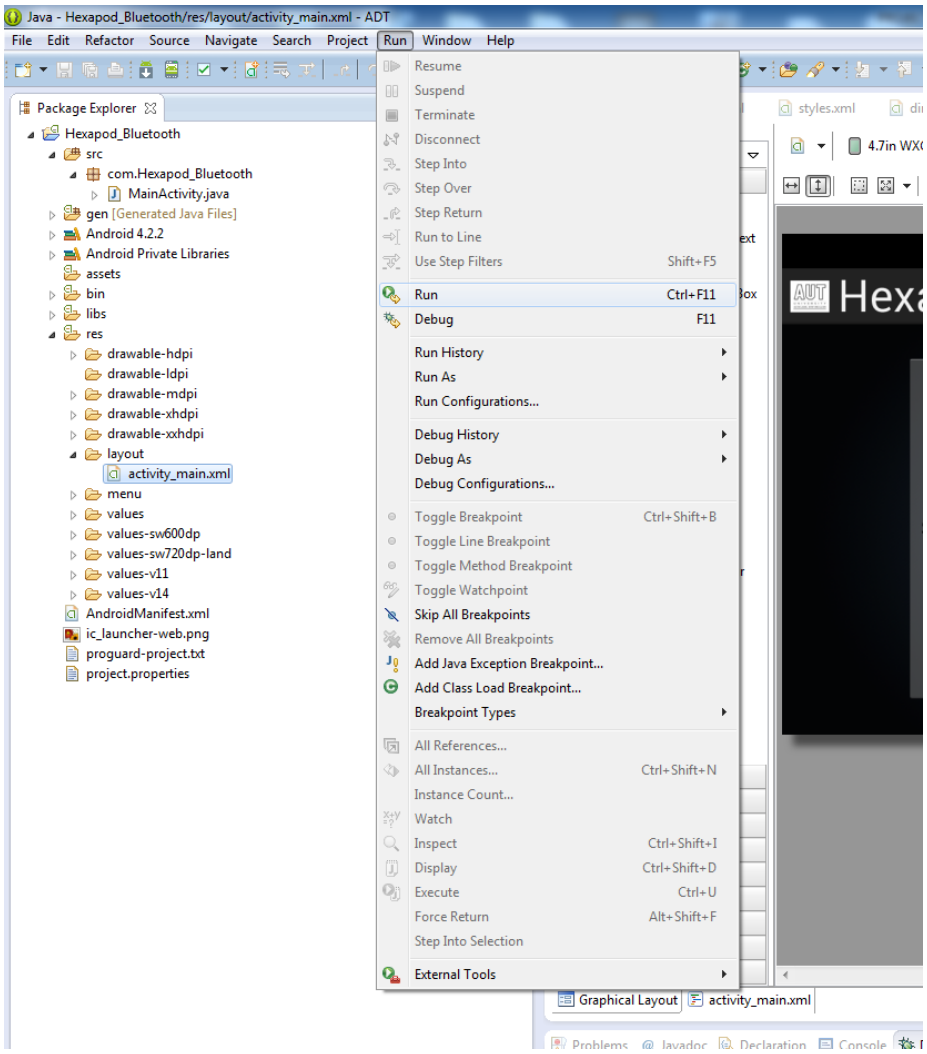

Figure 2.3

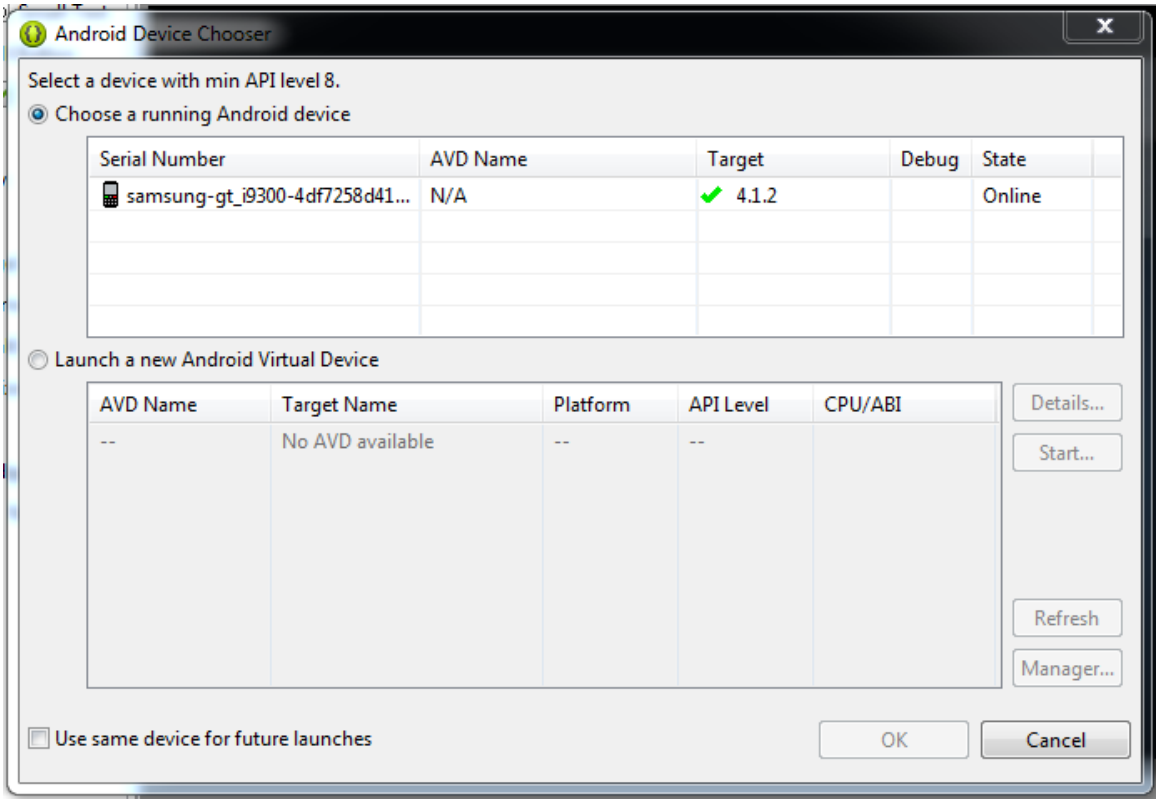

Figure 2.4

Once programmed and downloaded the user can launch the app. More information on launching the app will be explained in detail later on.

Bluetooth Code

This section will explain in further detail about how the ***Java*** code for the ***Android Application*** functions. The coding was an essential part because this allows for communication between the ***smartphone*** and the ***Bluetooth module*** to be made possible.

To make ***Bluetooth*** communication possible the first step was to configure the ***Bluetooth module*** in the correct user profile as mentioned in previous sections, the correct profile being ***SPP*** mode which stands for ***Serial Port Profile.*** In this mode this allows for raw data to be streamed directly from the smartphone which is basically what is required. More information into correctly configuring the ***Bluetooth module*** will mentioned later on.

The next step was acquiring 2 critical pieces of information, such as a ***128 bit UUID*** **(*Universally Unique Identifier*)** which is a unique identifier for the ***Android App***, and the ***MAC address*** of the ***Bluetooth*** module. ***Figure 2.6,*** shows a screenshot example of the Java programming code in ***Eclipse***. The first line of code shows the ***128 bit UUID*** and the second line shows the Mac address of the ***Bluetooth Module***, which are both represented as a ***string*** variable.

```
// SPP UUID service
private static final UUID MY_UUID = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB");

// MAC-address of module (you must edit this line)
private static String address = "00:06:66:4E:06:F6";
```

Figure 2.6

After ***Bluetooth*** communication was established with the following code example, the next step was to configure the behaviour of the code for ***GUI*** control buttons which will be mentioned in the next section.

GUI(Graphical User Interface)

The graphical user interface of the Android Smartphone consists of 4 different control buttons which determine the walk mode choices that the hexapod robot will perform. ***Figure 2.7,*** was taken from ***Eclipse*** shows the layout view.

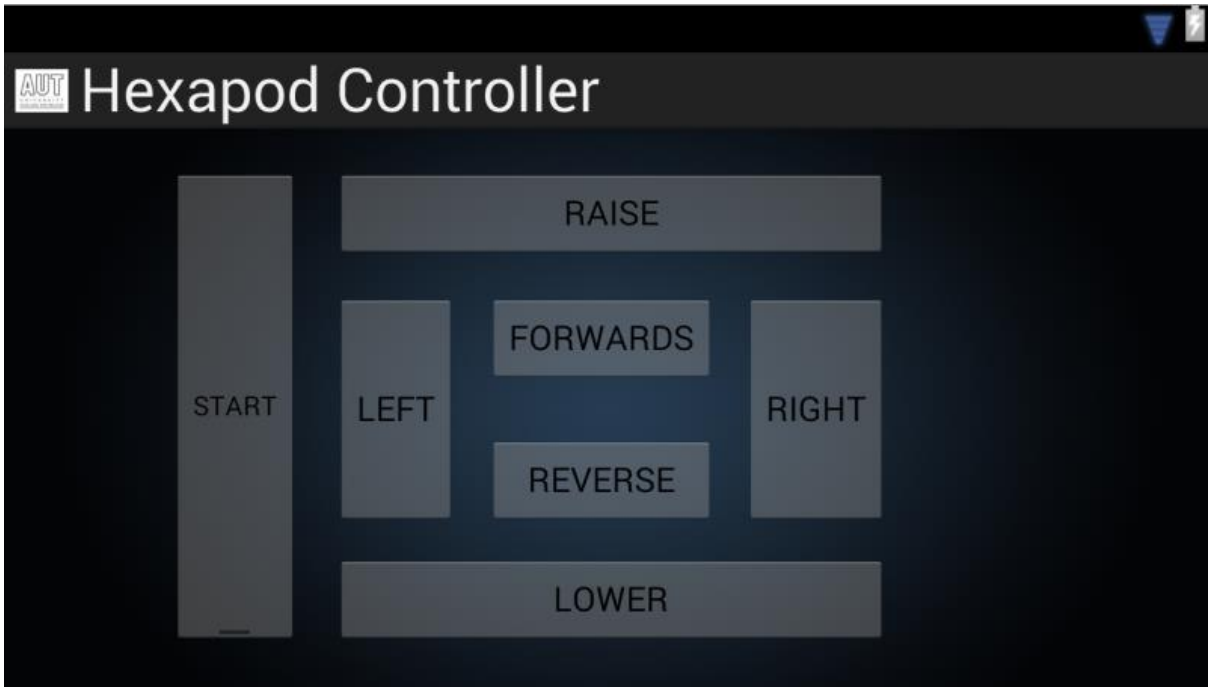

Figure 2.7

Figure 2.8 shows a screenshot view of the code that was taken from the ***MainActivity.Java*** file. This code describes the behaviour for each of the control buttons shown for the ***GUI*** in ***Figure 2.7.*** The code for each button is group together in individual functions. For example when the user presses the ***FORWARDS*** button, the **button1.setOnClickListner(new OnClickListener())** function is executed. The following code sends an identifying character ***'F'*** to the smartphone's ***Bluetooth*** adapter, which will then be wirelessly sent to the ***Bluetooth Module.*** The circuit consisting of the ***microcontroller*** and ***Bluetooth Module*** will receive the character information and then decode this as a ***Forward*** mode walking sequence**.** More information regarding the walking sequences will be mentioned later on. The exact same thing happens for the other buttons in the coding layout.

There are also other additional buttons for ***Raise*** and ***Lower*** and the coding behaviour for these buttons are same as for the 4 other walk mode buttons. The ***Raise*** and ***Lower*** buttons allow the frame to be lowered and raised; this will be explained in more detail later on.

```
//Display message for button 1 press

          button1.setOnClickListener(new OnClickListener()
         {
          public void onClick(View v)
          {
                sendData('F');      //Lift up all
         }
                                          });

         button2.setOnClickListener(new OnClickListener()
         {
          public void onClick(View v)
          {
                sendData('B');      //Lift up all servos
         }
                                          });

         button3.setOnClickListener(new OnClickListener()
         {
          public void onClick(View v)
          {
                sendData('L');      //Lift up all servos
         }
                                          });
         button4.setOnClickListener(new OnClickListener()
         {
          public void onClick(View v)
          {
                sendData('R');      //Lift up all servos
         }
                                          });

         button5.setOnClickListener(new OnClickListener()
         {
          public void onClick(View v)
          {
                sendData('U');      //Lift up all servos
         }
                                          });

         button6.setOnClickListener(new OnClickListener()
         {
          public void onClick(View v)
          {
                sendData('D');      //Lift up all servos
         }
                                          });
```

Figure 2.8

For the ***STOP/START*** toggle button this has a different programming behaviour when the button is pressed. When the following button is pressed the button changes to a ***STOP*** state as shown in ***Figure 2.9*** and a small blue strip at the bottom of the button appears. This indicates to the user that the ***STOP*** button has been enabled. The purpose of the ***Stop/Start*** button is to halt and resume the movement actions of the robot.

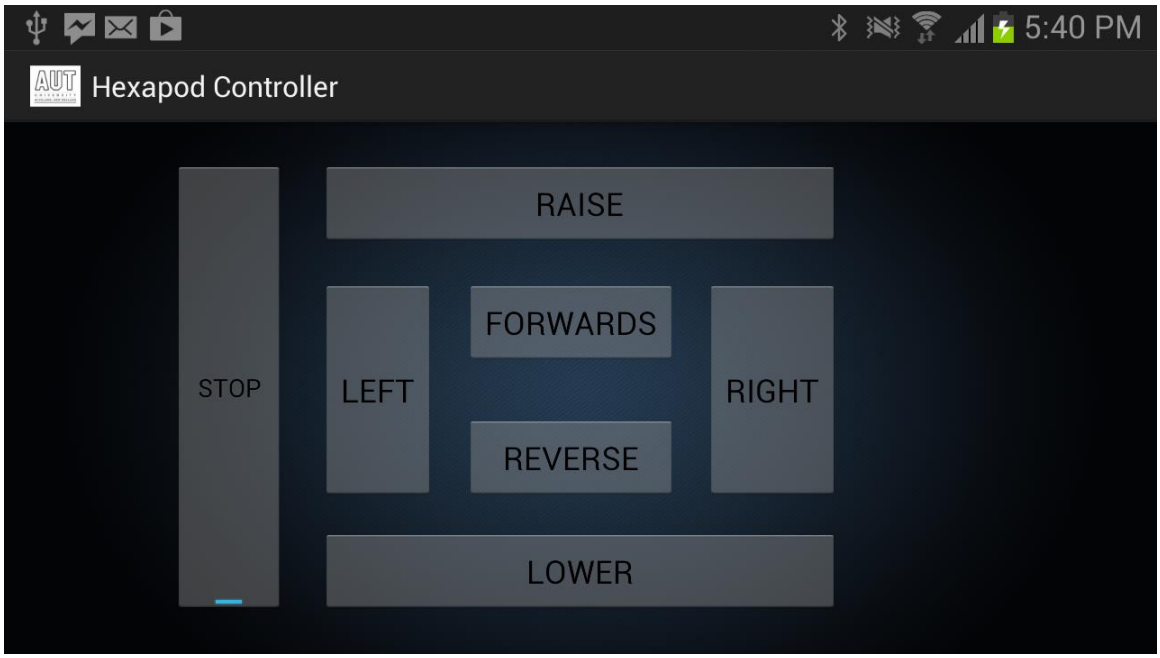

Figure 2.9

Coding for the ***STOP/START*** button is shown in ***Figure 2.9.1***. This button has 2 different modes of operations, depending on the type of mode that the toggle button has been pressed in, different characters will be sent to the ***Bluetooth*** module. If the ***STOP*** state has been checked, as shown in ***Figure 2.9*** then an ***'O'*** Identifier is sent, otherwise the ***'C'*** identifier is sent.

```
toggleButton1.setOnClickListener(new OnClickListener()
{
    @Override
    public void onClick(View v) {
    if (toggleButton1.isChecked())
    {
        sendData('O');      //reset hexapod
    }
    else{
        sendData('C');      //stop hexapod
    }

    }
});
```

Figure 2.9.1

Launching the App

Launching the app is mentioned in this section. ***Figure 2.9.2*** shows an actual screenshot of the ***GUI application*** launch icon programmed on the ***smartphone***. When the user presses the icon, indicated by the ***AUT logo*** named ***Hexapod Controller***, the ***GUI*** is automatically loaded as shown in ***Figure 2.9.3.*** Before the app is launched, the app checks whether the ***smartphone Bluetooth adapter*** is turned on. If the Bluetooth is not switched on then the following message will show up as shown in ***Figure 2.9.3.*** Pressing the yes button switches on the Bluetooth.

Hexapod Controller launch icon

Figure 2.9.2

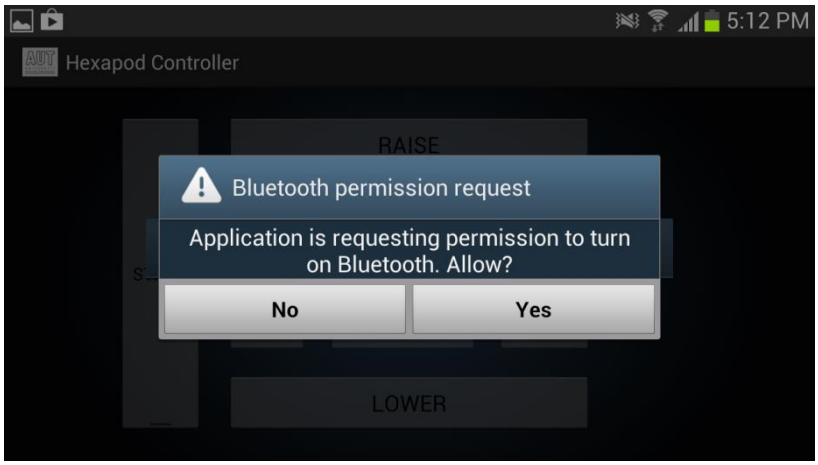

Figure 2.9.3

After the ***Bluetooth adapter*** has been turned on, the Bluetooth icon just above the GUI turns on as shown in ***Figure 2.9.4.***

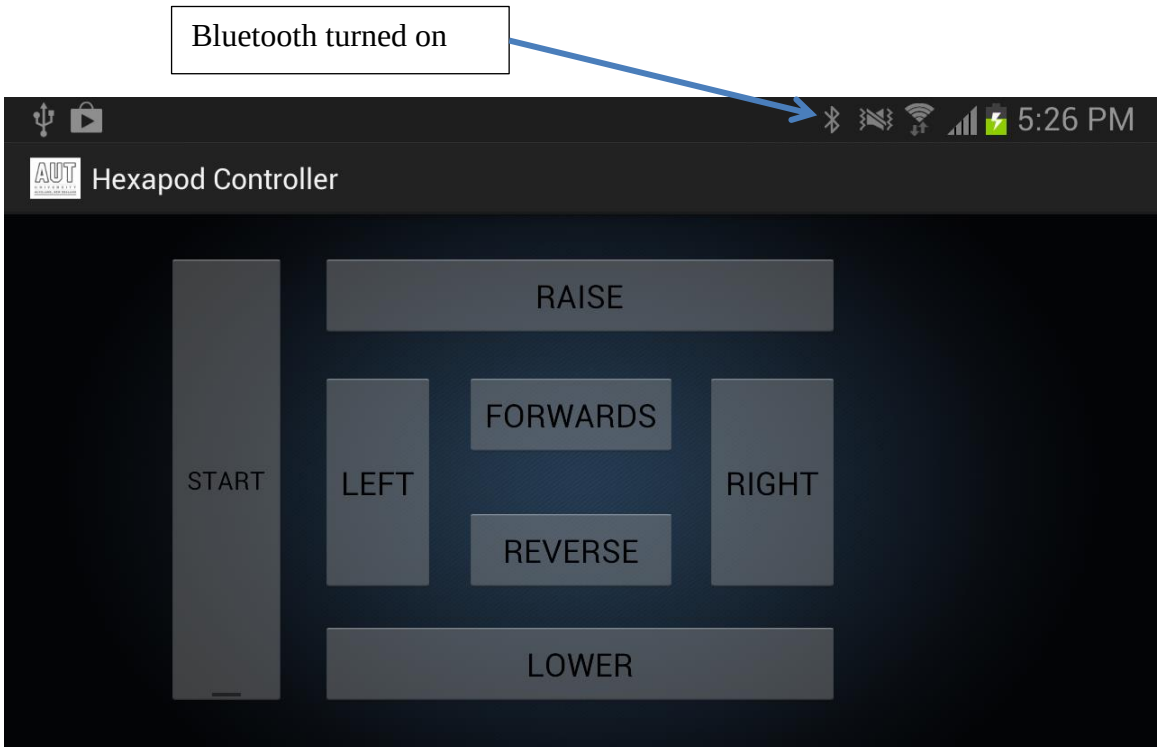

Figure 2.9.4

When trying to connect the ***smartphone GUI*** with the ***Bluetooth Module,*** the status light in ***Figure 2.9.5*** will change colour, from red to green, which indicates a successful connection. This will only occur if all of the steps mentioned previously for launching the app have been met. If the smartphone's Bluetooth was too suddenly be switched off or the if user terminated the app, the status light will change from green to red, which indicates that the connection has been terminated and therefore data can no longer be sent to the module.

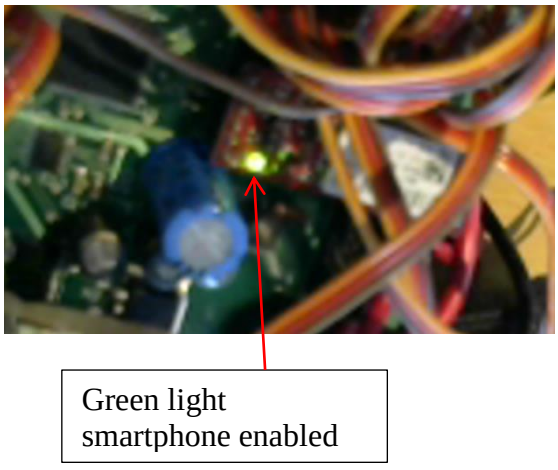

Figure 2.9.5

Summary

This section will explain the schematics diagram and choice of components used. For the ***Buck Boost regulator*** circuit some of the recommended component values from the datasheet were substituted for different values which will be mentioned in more detail later on.

The other things that will be mentioned in this section are the fact that 2 controller PCB's had to be developed for the robot. The reason for this being is that the first PCB had some design issues and flaws.

Schematics

Figure 3.0, shows the final schematics diagram for which the ***First*** and ***Final*** PCB layout are based on.

The diagram clearly shows the ***ATMEGA1280*** microcontroller which connects to all of the 18 servos which are indicated by the component symbols ***P1 to P18***. The symbols represent a 3 pin connector. Pin 1, of the connector is where the control signal is applied to, driven directly from the microcontroller. The control signal mentioned in a previous section is ***PWM (Pulse Width Modulation)***.

For the ***buck boost regulator*** circuit chip ***LM5118***, some of the surrounding components that were recommended from the components datasheet had been substituted with replacements, the MOSFETS indicated by the symbol ***M1*** and ***M2,*** were replaced with ***IRLZ24NSPBF – MOSFET's*** because the footprints were easier to solder and the components were readily available from ***Element 14***. The ***20mH***, ***Inductor L1*** was substituted for a different brand because the footprint was easier to solder and the fact that it was cheap and available from Element14. The resistor values for the buck boost circuit were not changed but the recommended footprints were changed for ones that are readily available.

A large *22000uF electrolytic* capacitor ***C18*** was put on the schematics, across the input terminals of the battery. The reason for having this was to smooth out the voltage rail and provide a means of decoupling the voltage noise spikes that the servos will generate when they are drawing current.

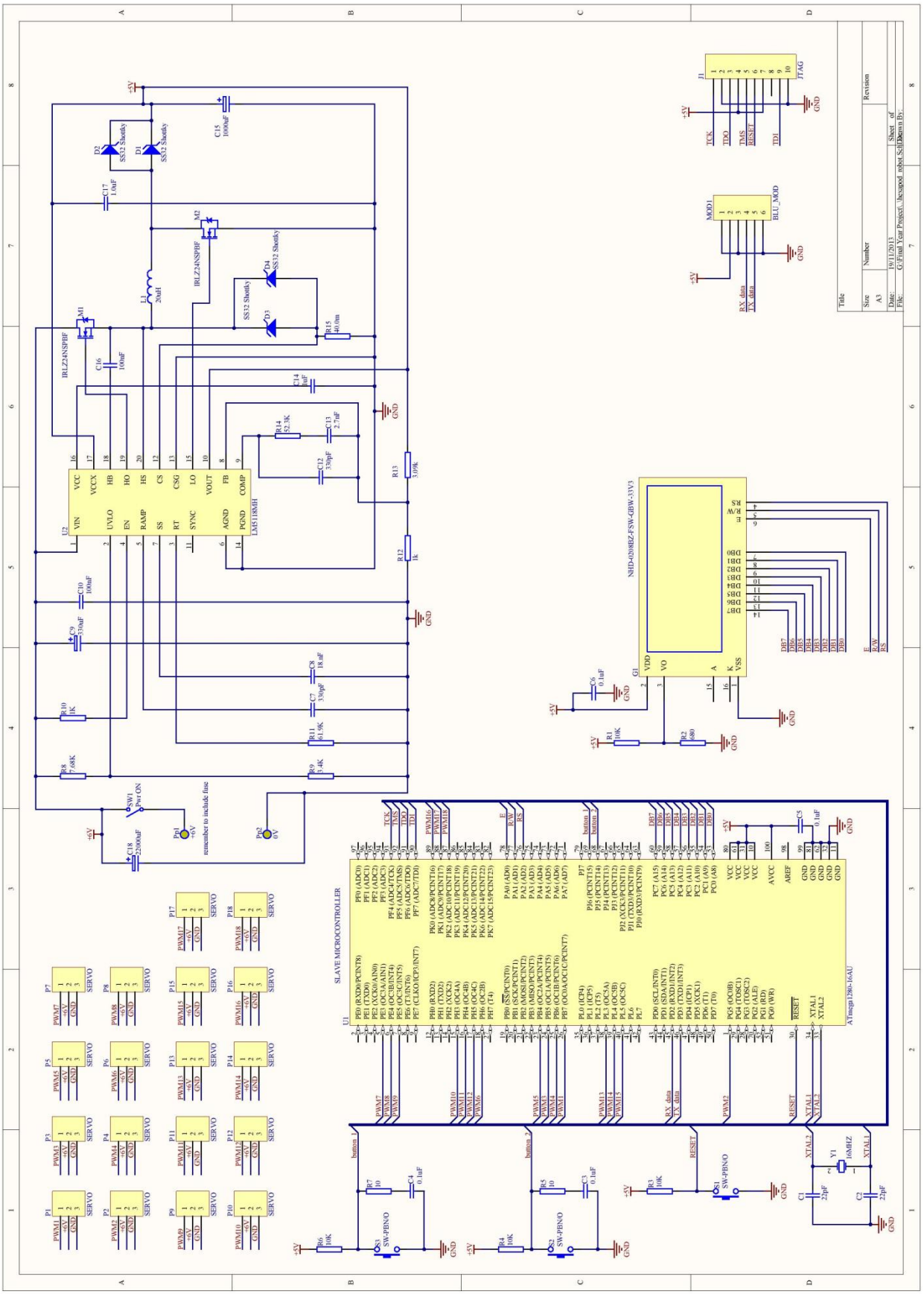

Figure 3.0

First PCB

Here is the design layout attempt of the First PCB, which was developed using the ***Altium Designer*** software as shown, in ***Figure 3.1.*** The PCB layout is based on the schematics diagram in ***Figure 3.0***. The servo connections are represented by the connectors P1 to P18.

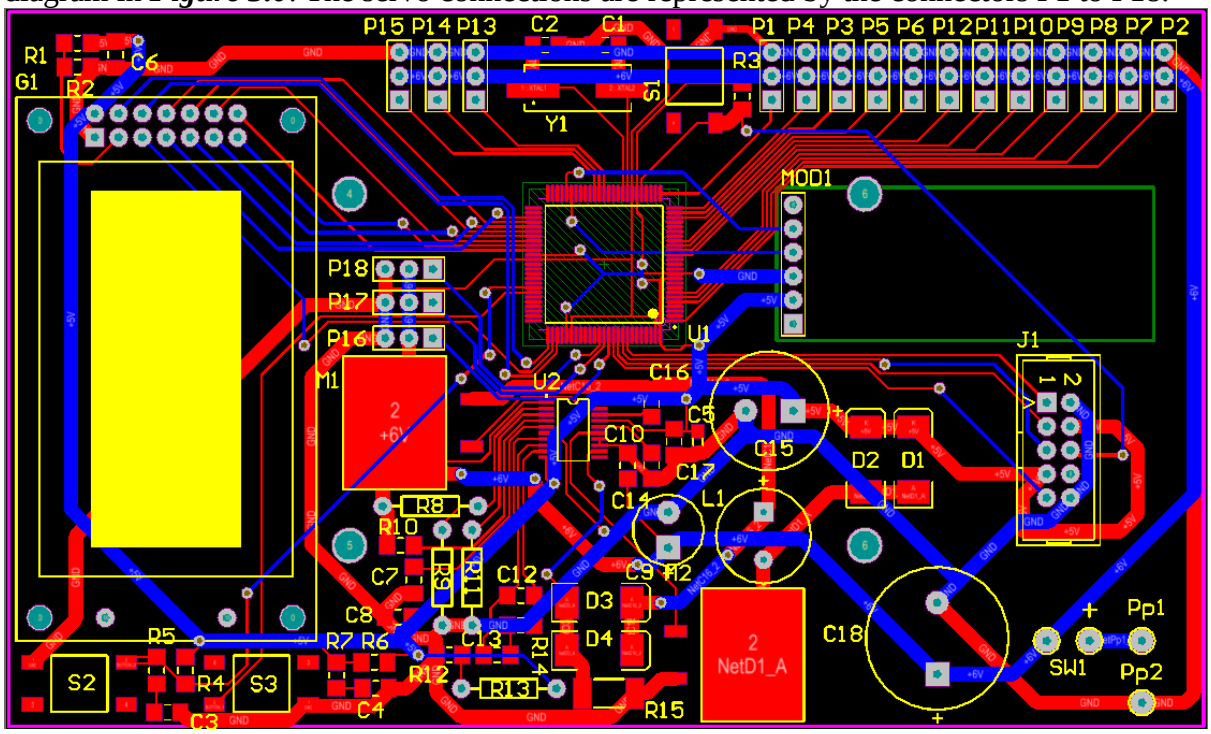

Figure 3.1

Figure 3.2 shows the manufactured PCB layout for ***Figure 3.1***. Most of the components had already been soldered to the board at the time this image was taken. Temporary wires for ***+6V*** and ***0V*** were soldered to the board to allow for testing. During the testing of this board many faults had been encountered which made programming the microcontroller cumbersome, which will be explained in more detail in other sections.

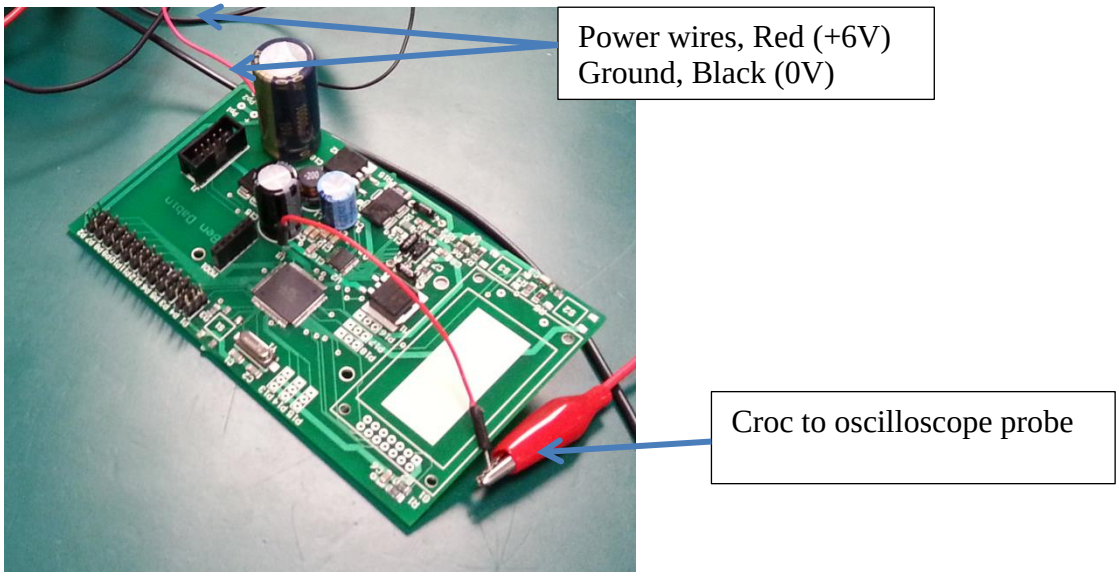

Figure 3.2

Design flaws in the ***PCB*** required extensive modification to be made. The existing tracks, going to the servo connectors were not large enough to handle the amount of current that will be required for when all of the servos are turned on. To solve the issue the silkscreen on the bottom layer of the track was carefully removed using a scalpel to reveal the copper. The next step was to bulk up the existing track by reflowing solder onto the copper as shown in ***Figure 3.3.***

Additional loop wires were also soldered to the underside of the board, for the 3 other servo connectors, ***P16, P17***, and ***P18***. The reason for this being is that the existing power and ground tracks for the following connectors previously used a longer current path from the battery terminal input.

Other modifications made to the existing PCB was for a ***+5V*** adapter, when the current voltage regulator on the PCB stopped functioning, so appropriate tracks were cut in various locations on the board, this will be explained in more detail in the faults and obstacles section.

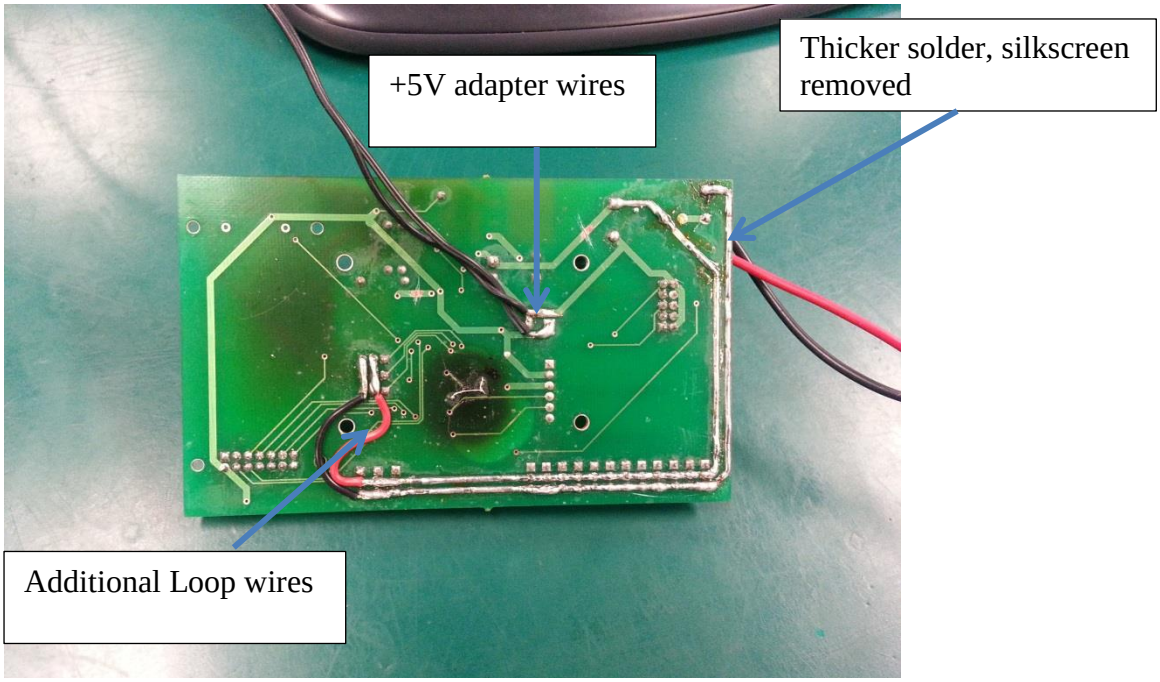

Figure 3.3

Figure 3.4 shows the top view layout of the ***First PCB.*** The modifications made to this were that the existing pads were too small to connect the size of wire that that is shown in the image, so drilling was required. Once drilling was done the wire could therefore be fed through the hole. It was necessary to have a 10A fuse holder in series with the battery wire, because if the existing board shorts out, the safety fuse is in place to prevent further damage.

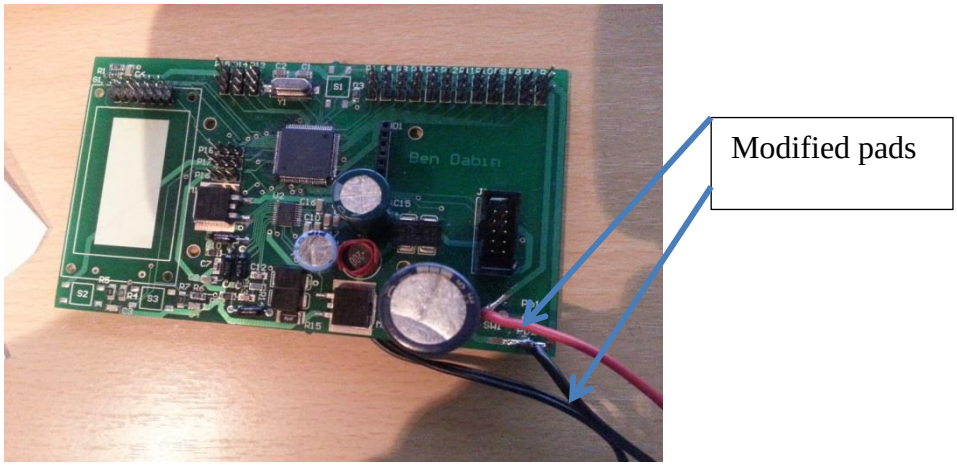

Figure 3.4

The following modifications were necessary in order to have a working ***PCB*** that is capable of controlling all of the 18 servos, until a 2nd improved PCB layout was made.

Improved PCB

The final PCB was designed, which removes all of the design flaws that were prevalent in the first PCB, as shown in ***Figure 3.5.*** Some extensive improvements had been made to the power and ground tracks, which are now much thicker and capable of powering all the 18 servos at the same time. The track layout of this PCB is much tidier which has the advantage of reducing track impedances. The microcontroller's ground path no longer shares the same path as the servos use; this was removed to prevent further noise problems.

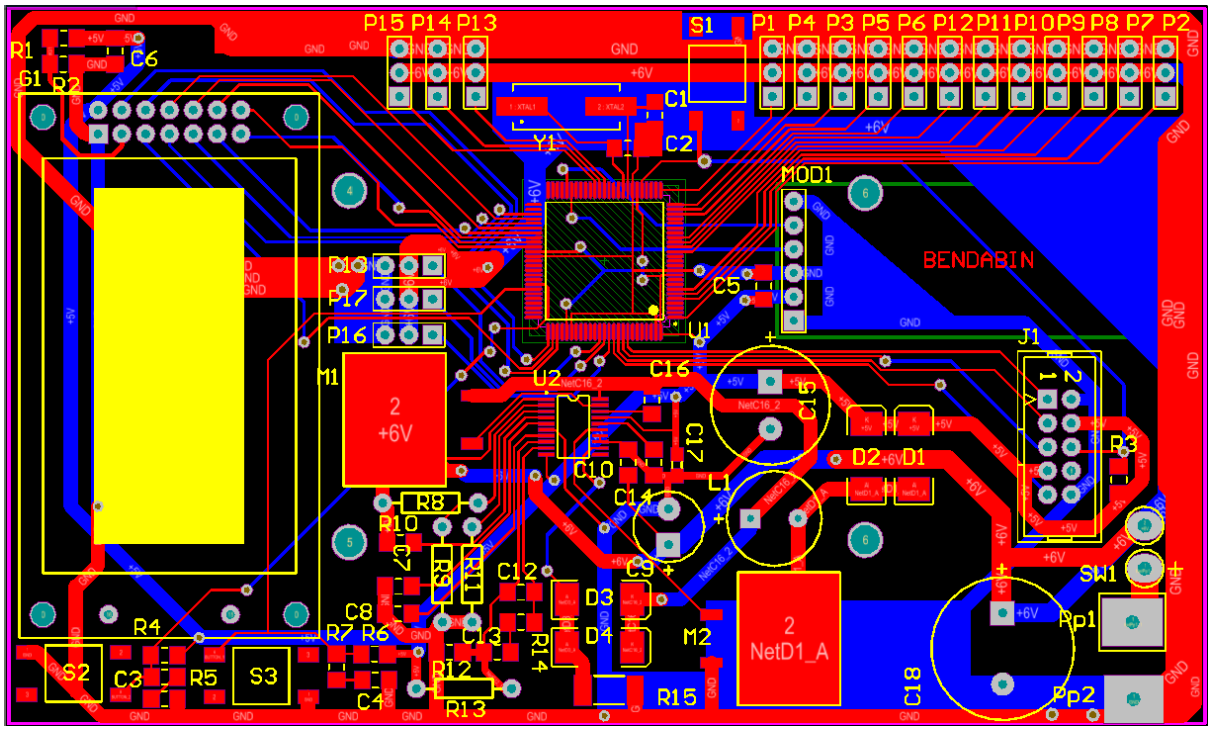

Figure 3.5

The battery connector pads were also made thicker and replaced with square pads. This allows for thicker cable to be connected. The thicker cable also allows for more current flow

and reduces track impedances. Another advantage of using thicker cable is that longer cable can be used. This was important when using the lead acid battery was connected to the PCB.

Figure 3.6 shows the manufactured ***PCB layout***. Most of the components had already been soldered to the board at the time this image was taken. Also a temporary loop wire was soldered in place where the switch terminal contacts were, to allow for the ***PCB*** to be tested.

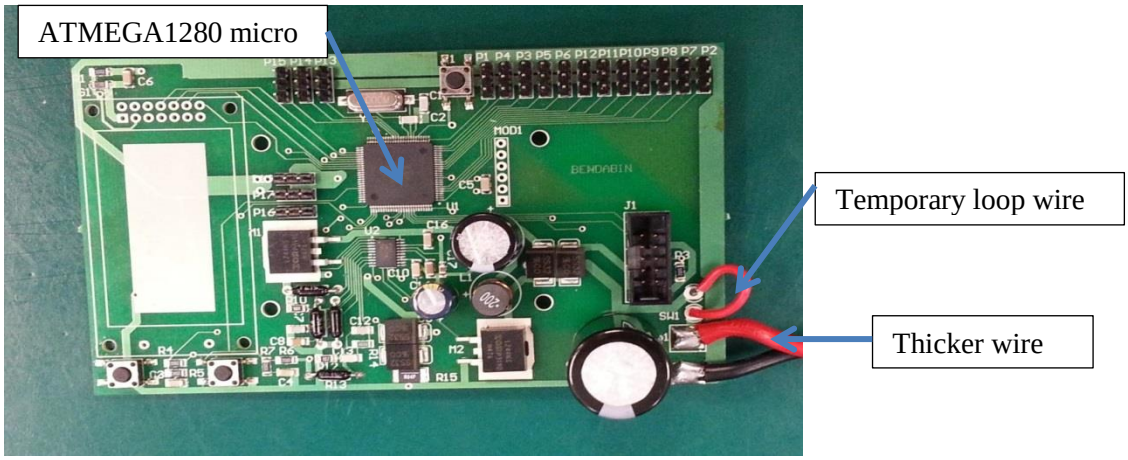

Figure 3.6

Embedded Programming code

Summary

This section will explain the embedded software programming part that was essential for controlling the robots walking and user menu functions. The programming code used in this section was written in the ***C programming language,*** which was programmed on the ***ATMEGA1280 microcontroller.***

This section will start off by discussing a brief overview of how the embedded system functions as a whole, and the hardware that is to be controlled. This will then be followed by a detailed explanation of the ***IAR Embedded Workbench*** programming and debugging software used for writing and testing the code. The ***AVR dragon programmer*** will also be mentioned as well.

Also, other important things that will be mentioned are information regarding the robots walking sequence, and how this was determined and what it consists of. The robots walking sequence consists of multi hierarchical levels of coding. These levels of coding are further broken down into individual, ***Step Cycle's*** for each leg and then when formed together they make up the ***walk sequence***, this will be explained in more detail later on.

Lastly this section will end by discussing the menu function features. The menu function features comprise of an on-board control system featuring ***2 push buttons*** and an ***LCD (Liquid Crystal Display)***. Another item that works in conjunction with the on-board is the system that deals with accepting user the input being received from the ***smartphone.***

Project code diagram

Here is a diagram which explains how the embedded systems controls all of the following hardware features shown.

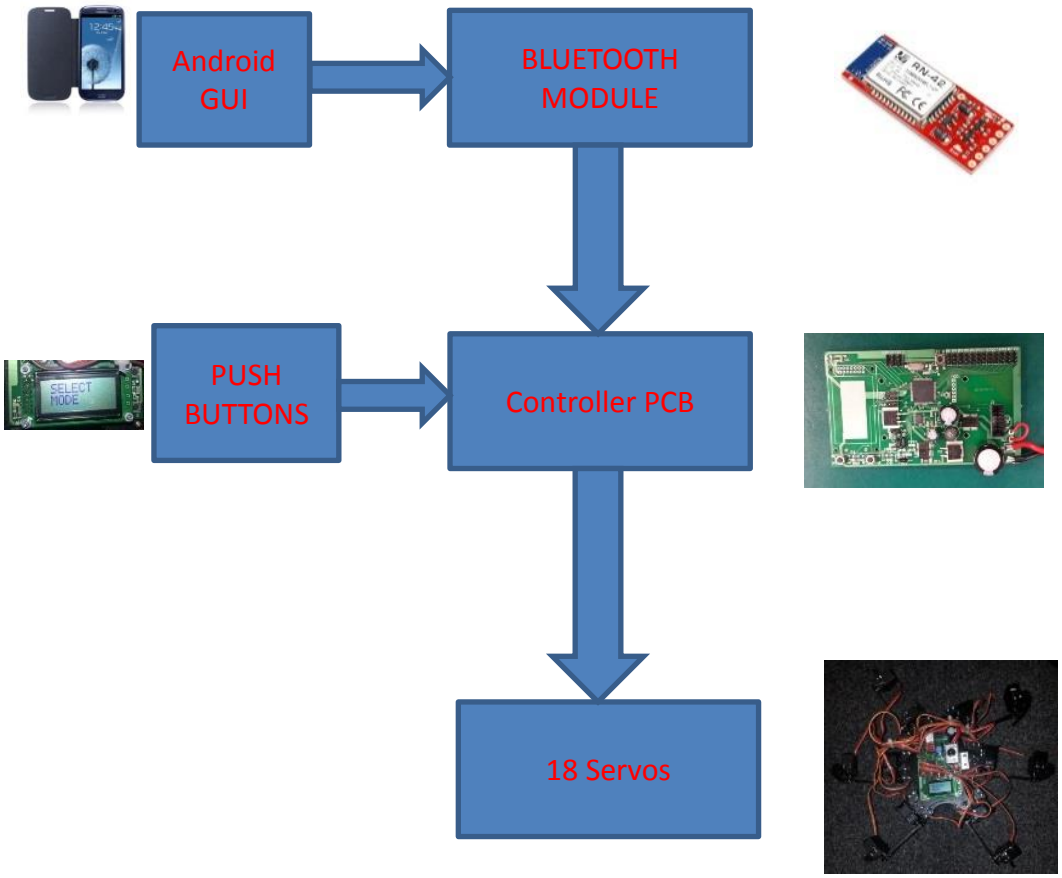

Figure 4.0.0

As the following diagram shows, ***Figure 4.0.0,*** the ***Controller PCB*** is the brains for the whole system as it contains the ***ATMEGA1280*** microcontroller that has the embedded programming software. The system starts off by accepting user input coming from either the ***Android smartphone*** or the push button menu systems. The user gets to choose the type of walking desired. Once the ***Controller PCB*** receives this information it can therefore begin the next sequence that is to perform the preprogramed walking sequences for the ***18 servos***.

IAR Workbench

The ***IAR IDE*** was used for writing the ***C code*** programming embedded software. ***Figure 4.1*** shows a typical layout view of the ***IAR Workbench IDE.***

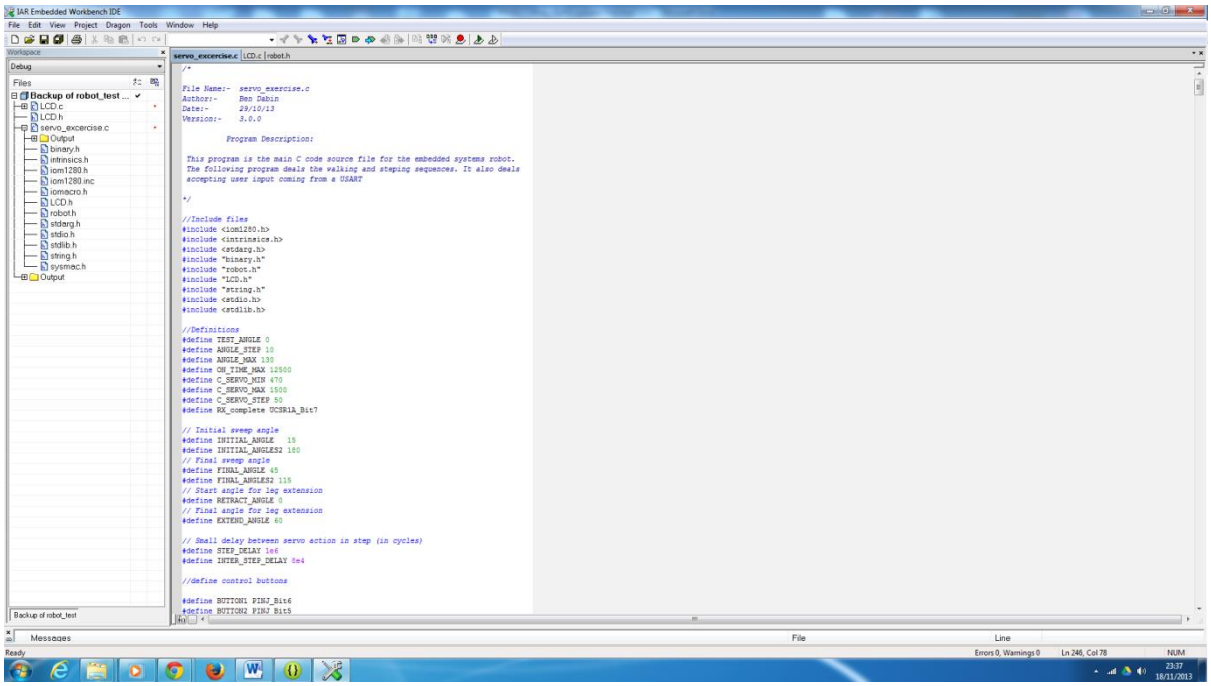

Figure 4.1

Programming the ***ATMEGA1280 microcontroller*** was done using the ***AVR dragon*** as shown in ***Figure 4.3.*** The image in ***Figure 4.2*** shows the download/debugger button option launching this button will send the ***IAR IDE*** into another debugger window and this will therefore allow the robot to be programmed with the software as shown in ***Figure 4.3***. Additional icon options will also appear after the IAR is in debugging mode as shown in ***Figure 4.2.1***. Also notice that there are several options available using the following icons the icon indicated by a red cross will cause the IAR complier to exit the debugger, the other icon next to the cross will run the complied programming code. The debugger is a very convenient way for checking the performance of the programming code to make sure that the code is doing the job it is required to do.

Figure 4.2

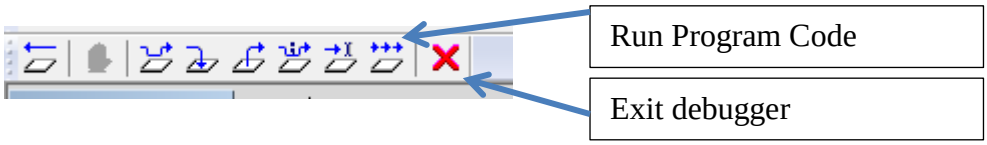

Figure 4.2.1

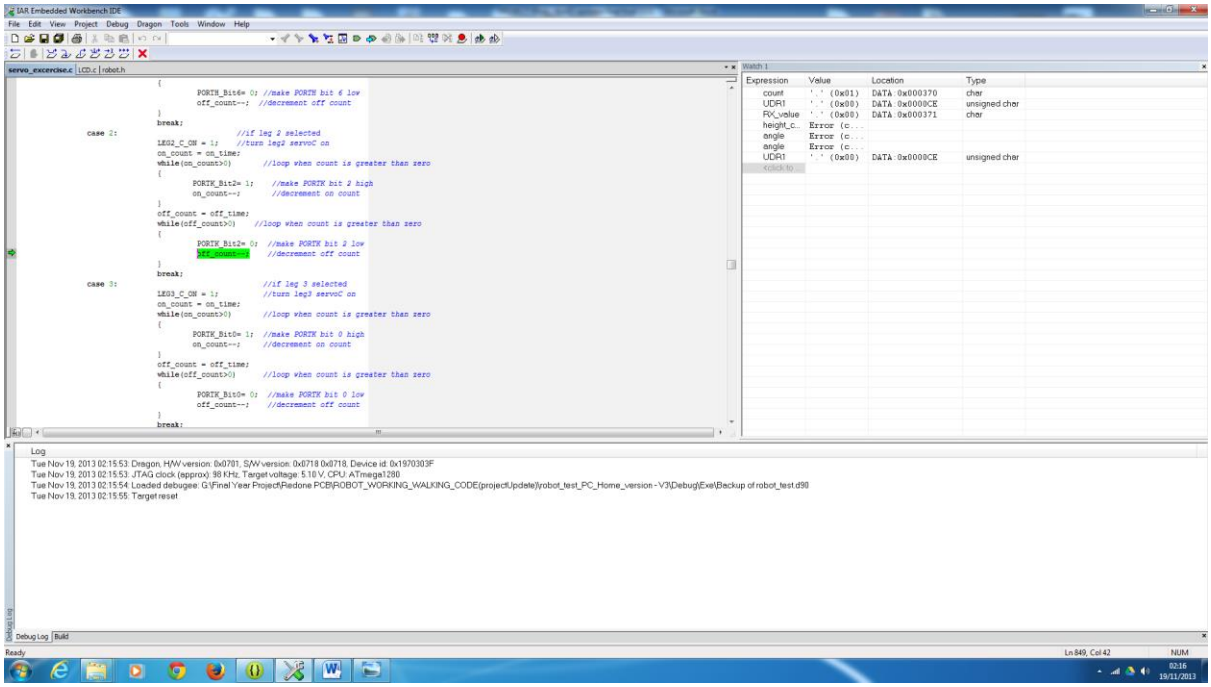

Figure 4.3

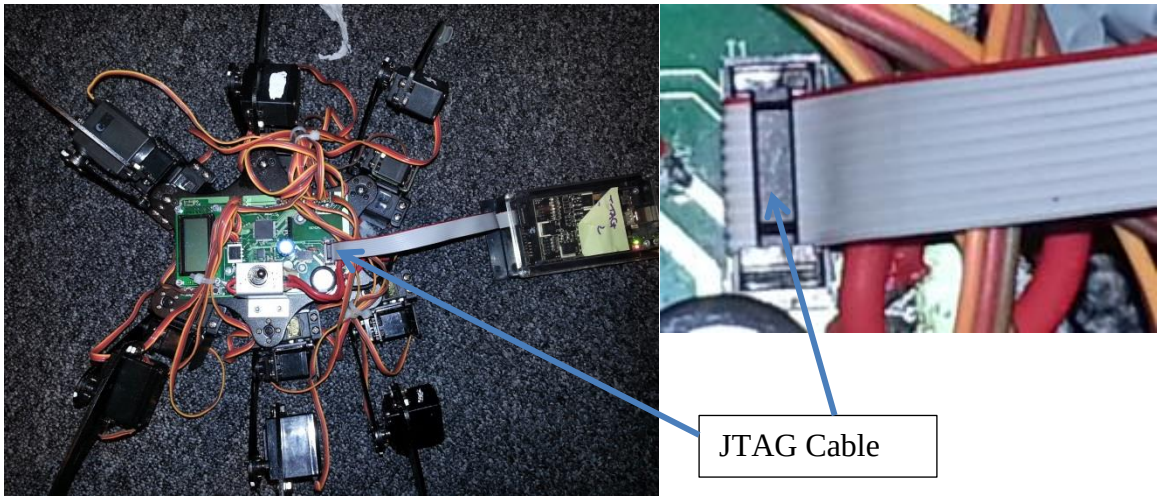

Figure 4.4

Leg Step Summary

A step cycle comprises of providing programming control to an individual servo leg on the hexapod robot kit. In ***Figure 4.5,*** this shows a diagram, showing a break down of a servo leg which comprises of controlling 3 different servos. ***Figure 4.6*** shows an actual servo leg that is to be controlled. All of the servos use ***PWM (Pulse Width Modulation)*** for rotating the servos at a particular angle value. Servos ***A and B*** use the microcontroller's ***16 bit timer registers*** as the ***PWM*** signal source. ***Servo C*** uses a software generated ***PWM*** that is controlled by varying ***ON and OFF*** times.

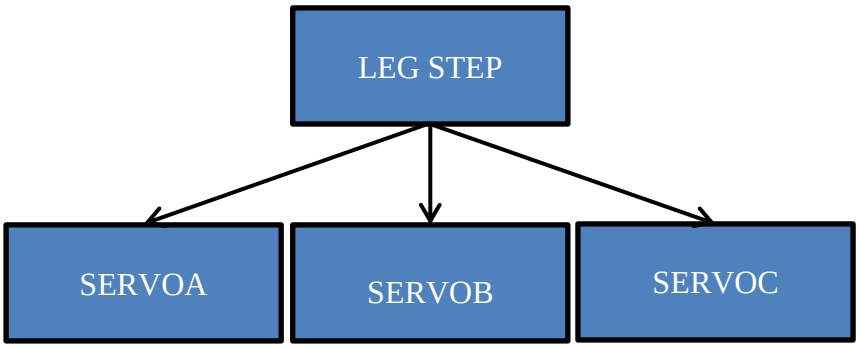

Figure 4.5

A servo leg step is made up of the following servos:

Servo A: This controls the extending and the lifting actions of the hexapod robot. The leg segment is directly connected to the ground.

Servo B: This servo has much the same behaviour that servo A does, as it acts as another joint extension and the leg segment of this is connected directly to the hexapod frame.

Servo C: This servo controls the sweeping and horizontal movement actions that are required to move the legs from side to side.

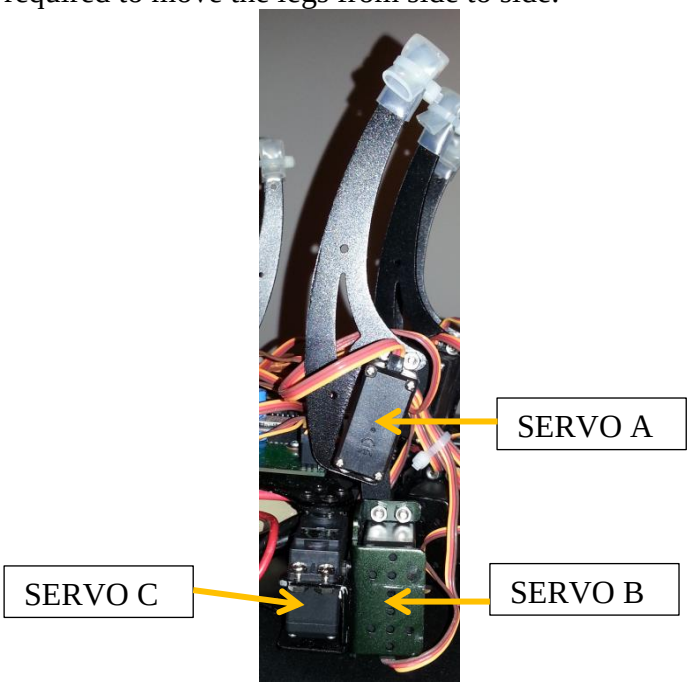

Figure 4.6

Servo A and B coding

As mentioned in the summary servos ***A and B*** use the ***ATMEGA1280's*** built in ***16 bit Timer*** counter registers as the ***PWM*** signal source. ***Figure 4.8*** shows a ***16 bit OCR register (Output Compare Register)*** which generates ***PWM*** on an output pin by writing programming code into the following register. There are a total of ***4, 16bit timer counter registers*** on this microcontroller. Each timer counter register has ***3 OCR registers*** which make up a total of 12. There are also 12, ***A*** and ***B*** servos, which use these registers.

17.11.17 OCR1AH and OCR1AL – Output Compare Register 1 A

Bit	7	6	5	4	3	2	1	0	
(0x89)	OCR1A[15:8]								OCR1AH
(0x88)	OCR1A[7:0]								OCR1AL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 4.8

Figure 4.9, shows the main setup code for the ***16 bit*** counter registers used on the microcontroller.

```
TCCR1A= b10101010;                  //setup 16 bit PWM timer 1
TCCR1B= b00011011;                  //prescale 8, set top as

TCCR3A= b10101010;                  //setup 16 bit PWM timer 2
TCCR3B= b00011011;                  //prescale 8, set top as

TCCR4A= b10101010;                 //setup 16 bit PWM timer 3
TCCR4B= b00011011;                 //prescale 8, set top as

TCCR5A= b10101010;                 //setup 16 bit PWM timer 4
TCCR5B= b00011011;                 //prescale 8, set top as

ICR1= PWM_MAX_TIME;                //setup PWM time to 20ms
ICR3= PWM_MAX_TIME;                //setup PWM time
ICR4= PWM_MAX_TIME;                //setup PWM time
ICR5= PWM_MAX_TIME;                //setup PWM time
```

Figure 4.9

The setup configuration for ***Figure 4.9*** for the ***16 bit timers*** is done by configuring binary values in the ***TCCR1A*** and ***TCCR1B*** setup register. Here in the current setup, the timers use the ***ICR1 (Input Capture)*** register as the ***TOP*** value for determining the maximum timer count which is 2500. The timer counters, counting rate is determined by using a pre-scaled version of the microcontrollers, 8MHZ external crystal clock source. The code suggests that the crystal has been pre-scaled by 8 to a clock frequency of 1MHZ, which means that the timer will count every 1us.

The following formula in 4.9.1, calculates the desired frequency ***PWM*** which is determined by the setup code mentioned earlier. In this case, the desired frequency of choice is 50HZ, which produces a period of 20ms, because as mentioned earlier the servos need the frequency to be this value.

The PWM frequency for the output can be calculated by the following equation:

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \cdot (1 + TOP)}$$

The N variable represents the prescaler divider (1, 8, 64, 256, or 1024).

Figure 4.9.1

A more detailed explanation, for describing how the PWM register operating is shown in the following diagram, ***Figure 4.9.2,***which was taken directly from the ***ATMEGA1280*** datasheet. This shows that the ***waveform generation*** block is connected directly to an output pin of the micro. The ***OCRnA*** registers are the registers of interest as these will be used for writing the values that will generate the PWM pulses in terms of ***MARK:SPACE*** over a period ***20ms***.

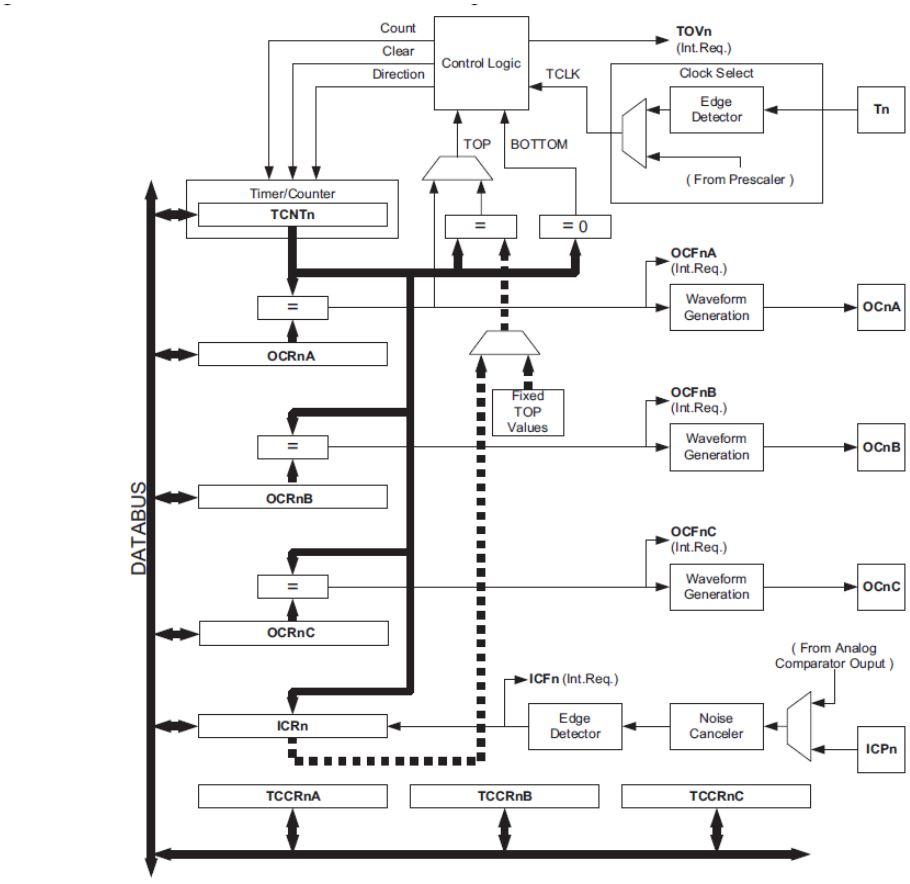

Figure 4.9.2

Figure 4.9.3, shows the actual OCR register definitions, which have been given a name in terms of a single servo. This makes programming more simple.

```
//defines for PWM
#define LEG1_A OCR1A
#define LEG1_B OCR1B

#define LEG2_A OCR3A
#define LEG2_B OCR3B

#define LEG3_A OCR5A
#define LEG3_B OCR5B

#define LEG4_A OCR4B
#define LEG4_B OCR4C

#define LEG5_A OCR3C
#define LEG5_B OCR4A

#define LEG6_A OCR1C        //P1
#define LEG6_B OCR5C        //P15
```

Figure 4.9.3

The code shown in ***Figure 4.9.4***, is the ***angle2OCR()*** function**,**which converts an angle value into a corresponding OCR register value. This is very useful because it allows servo angles to be determined in terms of real angle values instead of coding directly in OCR values.

```
int angle2OCR(int angle)
{
        //figure out what OCR value corresponds to this angle (in degrees)
        return (int)(angle * 1.44) + 100;
}
```

Figure 4.9.4

Once the OCR value from a given angle value is retrieved, this information can be passed onto the servo drive function shown in ***Figure 4.9.5.***This function is capable of controlling any of the ***1*** to ***12 A*** and ***B*** servos of the robot. The following example shows that servos ***1*** and ***2*** will be controlled by writing the following indexes to that function.

```
servo_drive(1, angle2OCR(EXTEND_ANGLE));     //extend angle of servo 1A
servo_drive(2, angle2OCR(EXTEND_ANGLE));     //extend angle or servo 2A
```

Figure 4.9.5

Figure 4.9.6, which shows what the ***servo_drive()*** function comprises of. The way this function works, is that in the previous examples index values of ***1*** and ***2*** are passed onto the following function. This information is passed onto the following ***switch(servo)*** and this determines what OCR register value will be controlled when the index cases are matched. The full code version will show cases of ***1 to 12*** instead of ***1 to 8,*** shown in the following example.

```
/*
    The following code controls the angle of the 12, A and B servos and has
    1 to 12 cases which decide the servo too be performed
*/
void servo_drive(char servo, int ocr)
{
        switch(servo)                           //what servo will be performed
        {
                case 1:
                        LEG1_A_ON= 1;           //servo 1A will be performed
                        LEG1_A = ocr;           //P3
                        break;
                case 2:
                        LEG1_B_ON= 1;           //servo 1B will be performed
                        LEG1_B = ocr;           //P4
                        break;
                case 3:
                        LEG2_A_ON= 1;           //servo 2A will be performed
                        LEG2_A = ocr;           //P7
                        break;
                case 4:
                        LEG2_B_ON= 1;           //servo 2B will be performed
                        LEG2_B = ocr;           //P8
                        break;
                case 5:
                        LEG3_A_ON= 1;           //servo 3A will be performed
                        LEG3_A = ocr;           //P13
                        break;
                case 6:
                        LEG3_B_ON= 1;           //servo 3B will be performed
                        LEG3_B = ocr;           //P14
                        break;
                case 7:
                        LEG4_A_ON= 1;             //servo 4A will be performed
                        LEG4_A = ocr;             //P11
                        break;
                case 8:
                        LEG4_B_ON= 1;             //servo 4B will be performed
                        LEG4_B = ocr;             //P12
                        break;
```

Figure 4.9.6

Servo C coding

Servo C uses a software bit banging technique as the PWM signal source which works by varying the ***on and off*** times of the port. This part of the code converts an angle value into 2 individual ***on*** and ***off*** times that allow the servos to rotate at a given angle.

Figure 4.9.7 shows the ***angle2count*** function which works by converting an angle value of ***120*** degrees into 2 individual ***on*** and ***off*** time count values that are to be used for the ***servo_drive_c*** function.

```
angle2count((int)120, &on_time, &off_time);
servo_drive_c(1, on_time, off_time);
```

Figure 4.9.7

Figure 4.9.8, shows a detailed view of the ***angle2count*** and what the function is comprised of, where the angle value is passed onto to the following function and then 2 ***on*** and ***off*** counter values are returned. The maximum time count value would be represented by 2500(***on_time + off time)***. Below shows how this algorithm works, the example code shows an angle of 120 degrees, so the following calculations would be as follows:

$$on\_time = 7 * 120 + 70$$
$$= 1310$$
$$off\_time = 2500 - 1310$$
$$= 1190$$

```
/*
    This part converts and angle value into an on and off time count value
*/
void angle2count(int angle, int* on_time, int* off_time)
{
  int on, off;
  on = (7 * angle + 470);
  off = ON_TIME_MAX - on;
  *on_time = on;
  *off_time = off;
}
```

Figure 4.9.8

When the ***servo_drive_c*** function is called the code in ***Figure 4.9.9*** is executed. This function decides which servo leg is to be controlled by passing an index value onto the following functions as well as the ***one and off*** time information. In the example shown in Figure 4.9.7, an index value of ***1*** is passed onto the function, the following case, ***switch(servo)*** determines which C servo will be controlled. In the case of the example it will accept an index of ***1 to 6,*** because there are 6 'C' servos in total. The on and off times values are the delay counter values that will toggle ***PORTH_Bit6*** of the microcontroller port.

```
/*

     This function deals with the bit PWM software bit banging action of
     the C servos determined by recieving the on and off times

*/

void servo_drive_c(char servo, int on_time, int off_time)
{
  int counter, on_count, off_count;
  counter = 15;   //how many pulses

  while(counter>0)
  {

    if(RX_complete)         //Check if USART interrupted
    {
      counter=0;
    }

        switch(servo)           //what servo has been selected?
        {
               case 1:                                  //if leg 1 selected
                                LEG1_C_ON = 1;          //turn leg1 servoC on
                                on_count = on_time;
                                while(on_count>0)  //loop when count is greater than zero
                                {
                                        PORTH_Bit6= 1; //make PORTH bit 6 high
                                        on_count--;    //decrement on count
                                }
                                off_count = off_time;
                                while(off_count>0) //loop when count is greater than zero
                                {
                                         PORTH_Bit6= 0; //make PORTH bit 6 low
                                         off_count--;  //decrement off count
                                }
                                break;
```

Figure 4.9.9

Leg step Cycle coding

The previous section mentions how each individual servo is controlled. This section will mention how the 3 servos actions are combined together to form a single leg step function routine***. Figure 4.9.9.1*** shows the ***stepLeg1*** function and when called will consist of further functions that form the step routine in terms of the A, B and C servos.

```
stepLeg1(STATE);     //move leg1
```

Figure 4.9.9.1

A breakdown of this function is shown in ***Figure 4.9.9.2***. This shows that the ***stepLeg1*** function is made up of several step routines for one servo leg. Each servo step has different routines for moving ***Forwards, Backwards***, ***Left***, and ***Right***. The ***if(STATE)*** reads the value that has been stored in the ***STATE*** variable, depending on the character received or passed onto the following function, this will determine which section of code is executed.

```
void stepLeg1(char STATE)       //This function deals with one leg step
{

        int on_time, off_time;
        on_time = 0;
        off_time = 0;

  if(STATE == 'F')          //If F recieved do Forwards step
  {
      angle2count((int)120, &on_time, &off_time);  //convert angle to counts
      servo_drive_c(1, on_time, off_time);         //move servoc leg1C
      __delay_cycles(STEP_DELAY);
      servo_drive(1, angle2OCR(EXTEND_ANGLE));     //extend angle of servo 1A
      servo_drive(2, angle2OCR(EXTEND_ANGLE));     //extend angle or servo 2A
      __delay_cycles(STEP_DELAY);                  //
      angle2count((int)85, &on_time, &off_time);
      servo_drive_c(1, on_time, off_time);
      servo_drive(1, angle2OCR(RETRACT_ANGLE));
      servo_drive(2, angle2OCR(RETRACT_ANGLE));
      __delay_cycles(STEP_DELAY);
      servo_drive(1, angle2OCR(35));
      __delay_cycles(STEP_DELAY);
      angle2count((int)120, &on_time, &off_time);
      servo_drive_c(1, on_time, off_time);
  }
  else if(STATE == 'B')      //If B recieved do Backwards step
  {
      angle2count((int)85, &on_time, &off_time);
      servo_drive_c(1, on_time, off_time);
      __delay_cycles(STEP_DELAY);
      servo_drive(1, angle2OCR(EXTEND_ANGLE));
      servo_drive(2, angle2OCR(EXTEND_ANGLE));
      __delay_cycles(STEP_DELAY);
      angle2count((int)120, &on_time, &off_time);
      servo_drive_c(1, on_time, off_time);
      servo_drive(1, angle2OCR(RETRACT_ANGLE));
      servo_drive(2, angle2OCR(RETRACT_ANGLE));
      __delay_cycles(STEP_DELAY);
      servo_drive(1, angle2OCR(35));
      __delay_cycles(STEP_DELAY);
      angle2count((int)120, &on_time, &off_time);
      servo_drive_c(1, on_time, off_time);
  }
```

Figure 4.9.9.2

Walk Sequence

In the previous section, this explained how the individual elements of the code functions operate in terms of leg steps and servo angle control, but when a group of ***leg step*** functions are joined together they form a walk sequence routine. A walk routine consists of moving ***Forwards, Reverse, Left,*** and ***Right.***as indicated by the following diagram in ***Figure 4.9.9.3.***

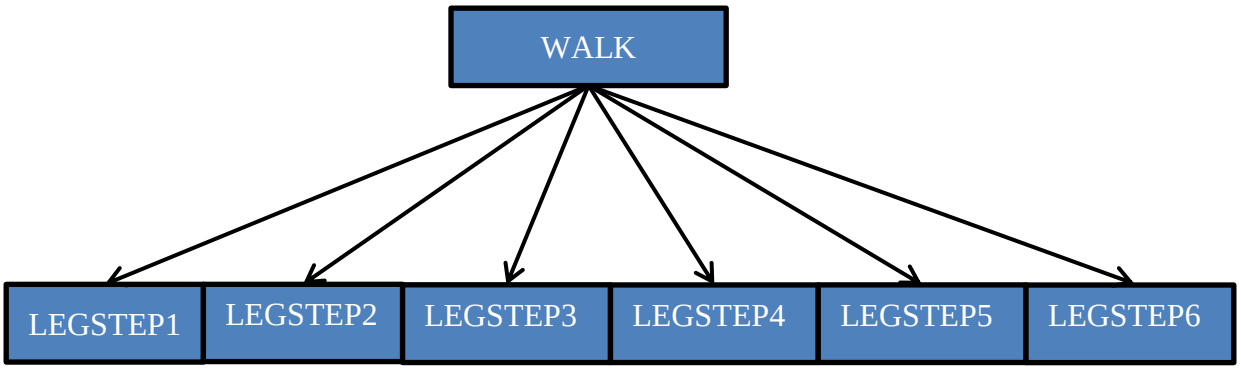

Figure 4.9.9.3

The following section of code in ***Figure 4.9.9.4*** shows walk routines for a ***forwards*** walking motion. Notice that there are 6 individual servo step leg functions that make up this sequence called ***stepLeg3, stepLeg4,etc***. There is also an ***_delay_cycles(INTER_STEP_DELAY);*** function between each leg step as this determines the speed that the servos move and operate at between each leg steps. The INTER_STEP_DELAY is a modifiable value and varying this value will make the delay between steps faster. This will result in a faster moving robot.

```
__delay_cycles(INTER_STEP_DELAY);  //legstep delay
```

Also notice that there is an ***if(RX_complete)*** followed by a list of statements inside the brackets. The purpose of this statement is that it allows immediate halting of the *current* walking routine, when data from the ***USART*** is received. This was added to allow on-board smartphone control to immediately check when a different walk mode option has been selected by the user; this will be explained in more detail later on. There is also a another function called ***reset_position();*** that will bring the robot into a default state so that the servo legs will not collide with each other when another walking routine has been selected.

```
if(RX_complete)  //check if USART STATE interrupted
{
    STATE= UDR1;        //change USART to never state
    reset_position();  //reset robot leg position
    break;             //break out of current mode
}
```

```
switch(STATE)           //selects state based on character received
{
      case  'F'  :            //forwards walk mode

        if(RX_complete)       //check if USART STATE interrupted
        {
          STATE= UDR1;        //change USART to newer state
          reset_position();   //reset robot leg position
          break;              //break out of current mode
         }
          stepLeg3(STATE);    //move leg3
          __delay_cycles(INTER_STEP_DELAY); //legstep delay
       if(RX_complete)        //check if USART STATE interrupted
        {
          STATE= UDR1;        //change USART to newer state
          reset_position();   //reset robot leg position
          break;              //break out of current mode
        }
           stepLeg4(STATE);    //move leg4
          __delay_cycles(INTER_STEP_DELAY);  //legstep delay
        if(RX_complete)       //check if USART STATE interrupted
        {
          STATE= UDR1;        //change USART to newer state
          reset_position();   //reset robot leg position
          break;              //break out of current mode
        }
         stepLeg2(STATE);     //move leg2
        __delay_cycles(INTER_STEP_DELAY); //legstep delay
          if(RX_complete)      //check if USART STATE interrupted
        {
          STATE= UDR1;        //change USART to newer state
          reset_position();   //reset robot leg position
          break;              //break out of current mode
        }
          stepLeg5(STATE);    //move leg5
          __delay_cycles(INTER_STEP_DELAY);  //legstep delay
        if(RX_complete)       //check if USART STATE interrupted
        {
          STATE= UDR1;        //change USART to newer state
          reset_position();   //reset robot leg position
          break;              //break out of current mode
        }
```

Figure 4.9.9.4

On board Menu System

This section describes how the on board menu system function works. There are basically 2 control buttons, ***BUTTON1*** and ***BUTTON2*** and a status ***LCD***, which displays the on-board interface**.**

The following Flowchart in ***Figure 5.5*** shows a basic break diagram of how the on-board menu system for when ***BUTTON1*** is pressed functions. The user choices are determined by the amount of time ***BUTTON1*** pressed, which causes a counter to be incremented.

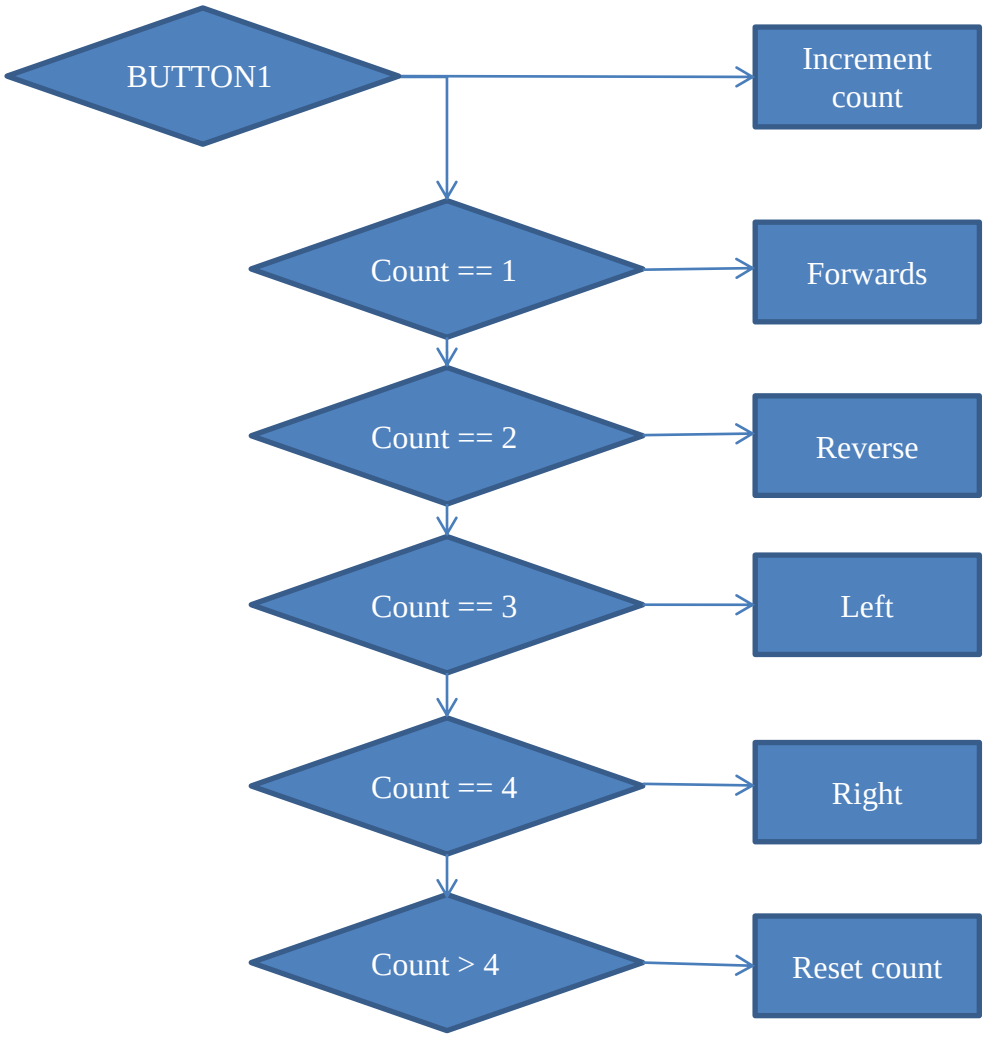

Figure 5.0

The menu system works by first beginning with the message shown in ***Figure 5.1,*** which appears when the robot is first switched on. During this time an initialisation sequence will be performed.

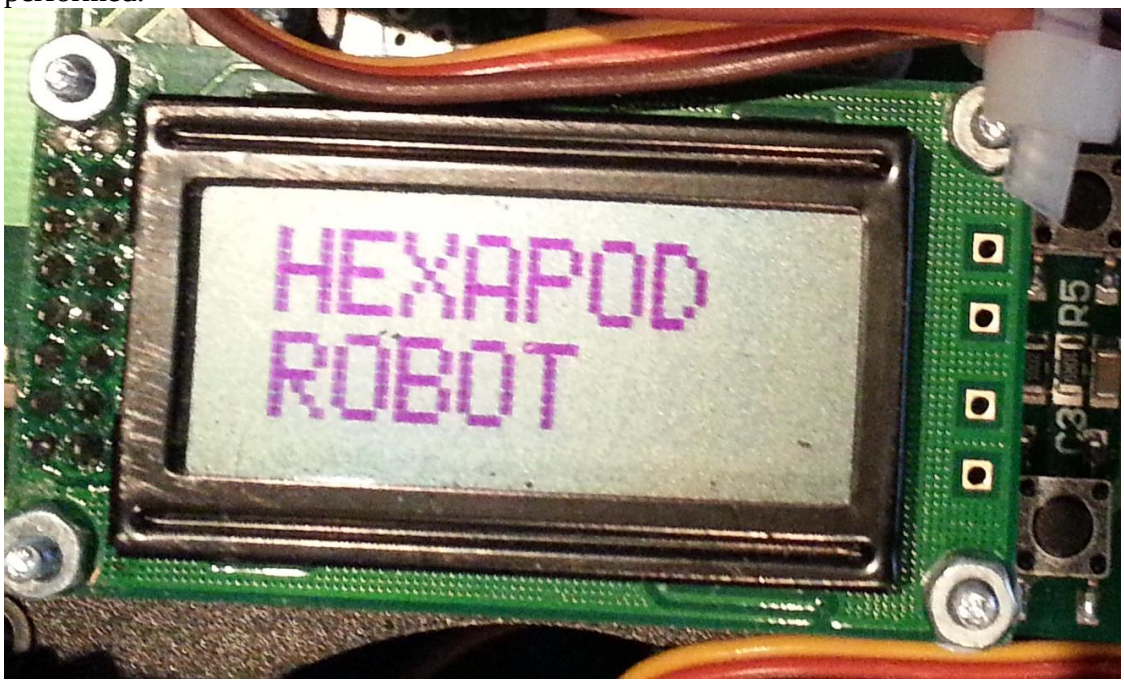

Figure 5.1

After the initialisation sequence has finished, another message will be displayed to the user, shown in ***Figure 5.2*** that enables the user to select the walk mode of choice, by pressing ***BUTTON1*** .

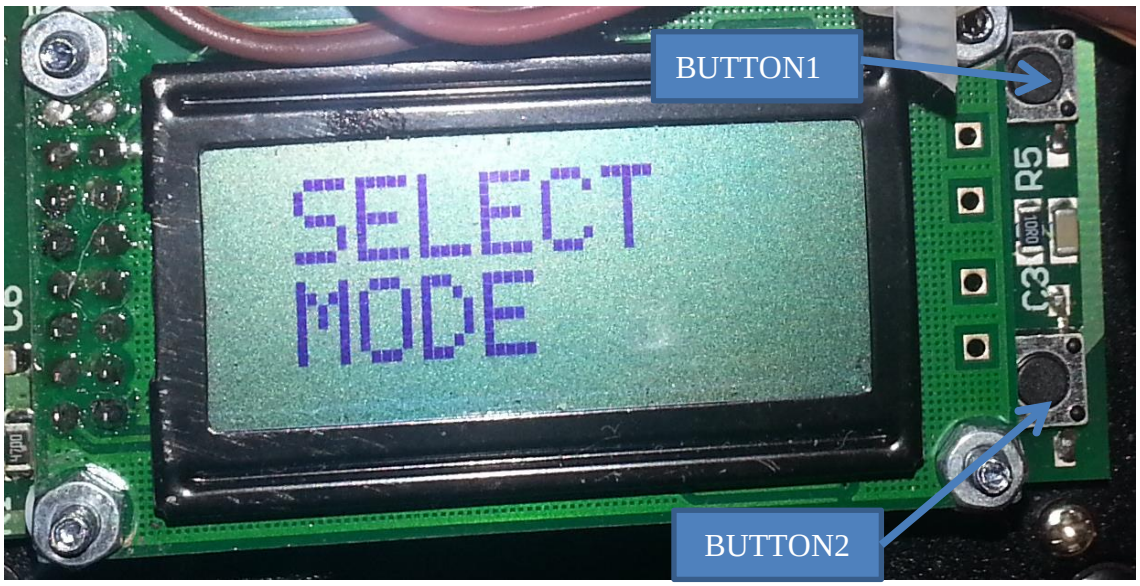

Figure 5.2

The first walking mode choice will show up on the LCD, which is ***FORWARDS,***as shown in ***Figure 5.3***.

Figure 5.3

If the user wants to select another walk mode, pressing ***BUTTON1*** again will change the message on the ***LCD*** and allow the user to select the next mode as shown in ***Figure 5.4***.

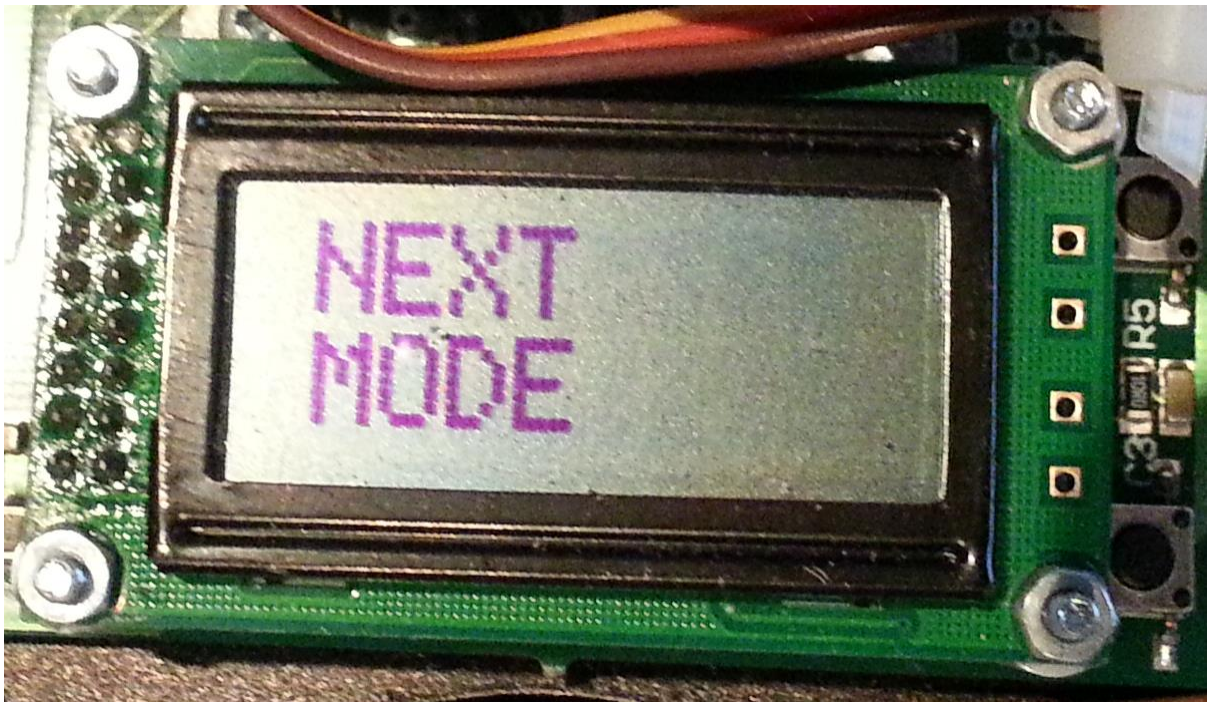

Figure 5.4

Pressing BUTTON1 again will change the message to the next walk mode which in this case will be ***BACKWARDS.***

If ***BUTTON2*** was pressed instead the message would be as shown in ***Figure 5.5.*** This informs the user that ***FORWARDS*** has now been enabled. As soon as the message is displayed the robot will perform the forwards moving sequence. Pressing ***BUTTON2*** confirms to the user that the walk mode of choice has been enabled and the robot performs the associated function calls for the ***Forwards*** walk motion.

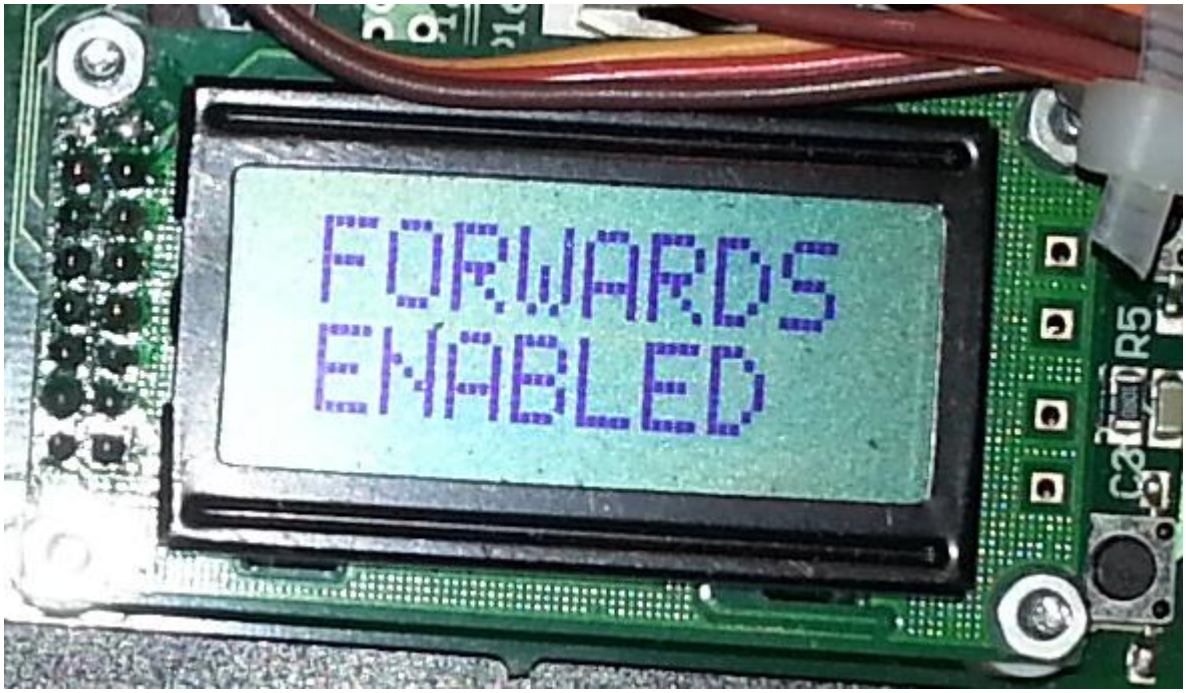

Figure 5.5

Figure 5.6 shows the programming code for the on-board menu system. When ***BUTTON1*** is pressed a ***count*** value is incremented. The amount of times that ***BUTTON1*** is pressed determines the selection of the walking mode of choice. Also notice from the following code that a message will be sent to the LCD when the desired walk mode is selected. The ***send2LCD***(); function is the function that sends the messages out to the LCD. The value of the variable ***STATE*** will also change depending on the walking routine selected.

```
/*
    This section of code enables the menu selection system which
    is controlled using BUTTON1 and BUTTON2

*/

     if(BUTTON1==0)               //if button one selected
     {
             while(BUTTON1==0);   //debounce switch on release
             count++;             //increment counter on button press
             if(count==1)
             {
                sprintf(data, "FORWARDS");      //forward Mode selected
                send2LCD(data,0x01, strlen(data));
                STATE= 'F';                     //change state to Forward
             }
             else if(count== 2)
             {
                sprintf(data, "BACKWARDS");   //Backwards Mode selected
                send2LCD(data,0x01, strlen(data));
                STATE= 'B';                   //change state to backwards
             }
             else if(count== 3)
             {
                sprintf(data, "RIGHT");             //Right Mode selected
                send2LCD(data,0x01, strlen(data));
                STATE= 'R';                         //change state to Right
             }
             else if(count==4)
             {
                sprintf(data, "LEFT");            //Left Mode selected
                send2LCD(data,0x01, strlen(data));
                STATE= 'L';                       //change state to Left
             }
             else if(count>4)
             {
                sprintf(data, "SELECT");          //forward Mode selected
                send2LCD(data,0x01, strlen(data));
                sprintf(data, "MODE");
                send2LCD(data,0xC0, strlen(data));
                count=0;
                STATE= 0;                         //change mode state to 0
             }
```

Figure 5.6

USART Smartphone Communication

This section explains the code part of the robot that accepts user input from the Android Smartphone which thereby determines the walk mode of choice. This works in a similar fashion to that of the on-board menu functions that it changes the case statements, but the only difference being that the cases are determined by what is being received from ***USART*** of the ***ATMEGA1280*** microcontroller. The flowchart in ***Figure 5.7,*** briefly describes how data is sent from an ***Android Smartphone*** and then received through the ***USART***. After receiving the data from the USART the programming code determines which of the following walk modes will be executed. The communication interface between the USART and the S3 Smartphone is received via ***Bluetooth.***

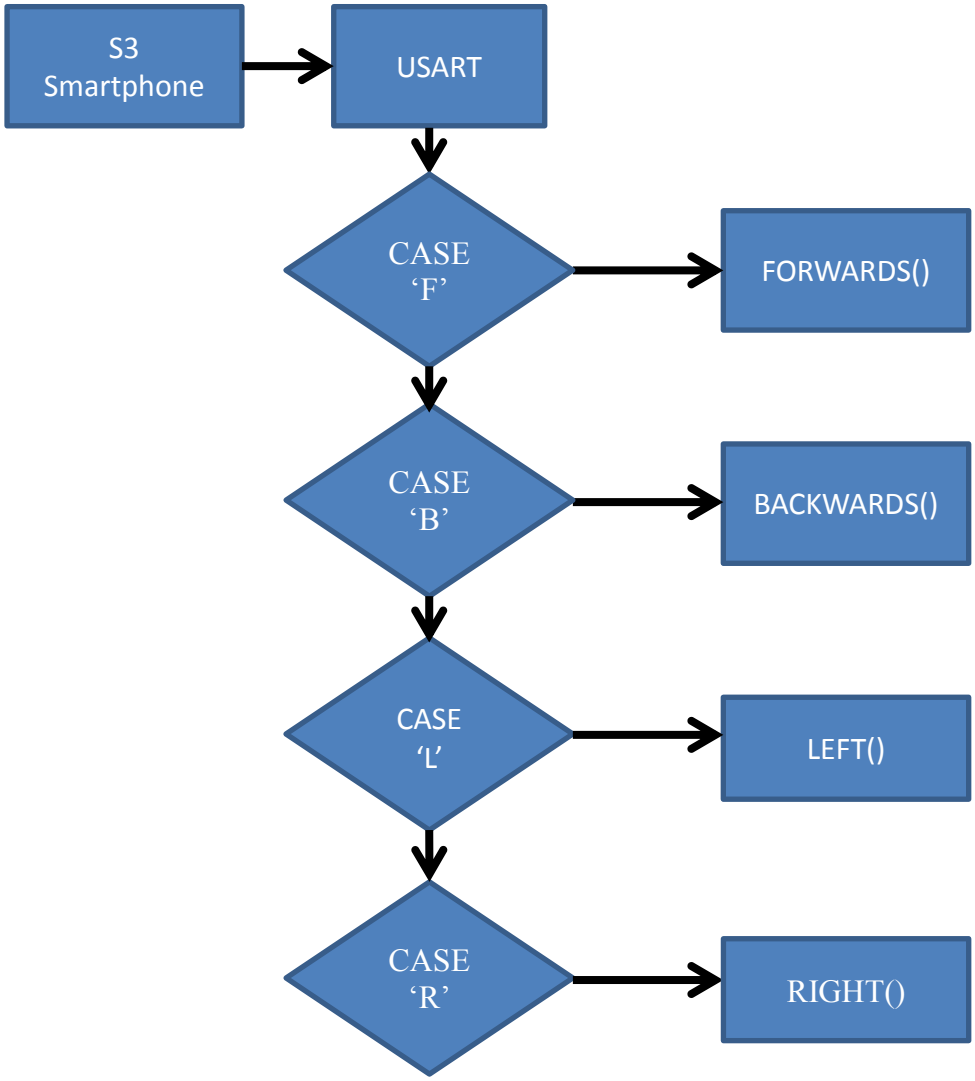

Figure 5.7

Figure 5.8, the code describes how the user information from the USART is received and how this is used to determine the walk mode selected by changing the ***STATE*** variable value of the inside. The characters being sent from the ***smartphone*** are sent to the UDR1 register and this is then stored in the RX_value register for manipulation.

```
/*
       The following code here deals with accepting user input being
       recieved from an Android Smartphone GUI. This also deals with
       the Robots menu functions.

*/

      else if(RX_buffer_full)          //wait for smartphone to recieve data
      {
         while(!RX_buffer_full);
         RX_value= UDR1;
         if(RX_value== 'F')            //If 'F' received Forwards enabled
         {
            while(BUTTON1==0);
            sprintf(data, "FORWARDS"); //forward Mode selected
            send2LCD(data,0x01, strlen(data));
            sprintf(data, "ENABLED");
            send2LCD(data,0xC0, strlen(data));
            STATE= 'F';                //change state to forwards
         }
         else if(RX_value== 'B')       //If 'B' received Backwards enabled
         {
           sprintf(data, "BACKWARDS"); //forward Mode selected
           send2LCD(data,0x01, strlen(data));
           sprintf(data, "ENABLED");
           send2LCD(data,0xC0, strlen(data));
            STATE= 'B';                //change state to backwards
         }
         else if(RX_value== 'C')       //If 'C' recieved Stop mode enabled
         {
           sprintf(data, "STOP");      //Stop Mode selected
           send2LCD(data,0x01, strlen(data));
           sprintf(data, "ENABLED");
           send2LCD(data,0xC0, strlen(data));
           STATE= 'C';                 //change state to C for
         }
         else if(RX_value=='R')        //If 'R' received Right mode enabled
         {
           sprintf(data, "RIGHT");     //Right Mode selected
           send2LCD(data,0x01, strlen(data));
           sprintf(data, "ENABLED");
           send2LCD(data,0xC0, strlen(data));
           STATE= 'R';                 //change state to 'R'
         }
         else if(RX_value=='L')        //If 'L' received Left mode enabled
         {
           sprintf(data, "LEFT");      //Left Mode selected
           send2LCD(data,0x01, strlen(data));
           sprintf(data, "ENABLED");
           send2LCD(data,0xC0, strlen(data));
           STATE= 'L';                 //change state to 'L'
         }
```

Figure 5.8

Faults and Obstacles

Summary

This section will mention the faults and obstacles that had been encountered with the project. As the finding findings suggested that the servos needed replacing. Also there were issues encountered with the power supply circuitry (***the voltage regulator***) and some soldering issues with the PCB. Some programming issues with the embedded software and Android App will also be mentioned in this section.

First PCB Issues

There were many issues with regarding the PCB testing stage, when attempting to program the ***ATMEGA1280*** microcontroller. The first issue was to trying to program the micro controller using the ***"AVR DRAGON"*** using the ***IAR*** complier. An error message generated by the IAR complier would be displayed in the output debugging window, immediately after downloading, programming code into the microcontroller saying, ***"The IDR register was written by target 0xFF".***

The cause of this error could be from possible power supply noise in the ***+5V rail***, which was being generated and supplied directly from the output of the regulator circuit. To determine whether this noise issue was causing the problem, the PCB was connected to a bench power supply and +6V was applied to the input terminals, as shown in the photo below. A wire which was directly connected to a *+5V* output ***veer*** was placed on an oscilloscope probe on ***channel 2***.

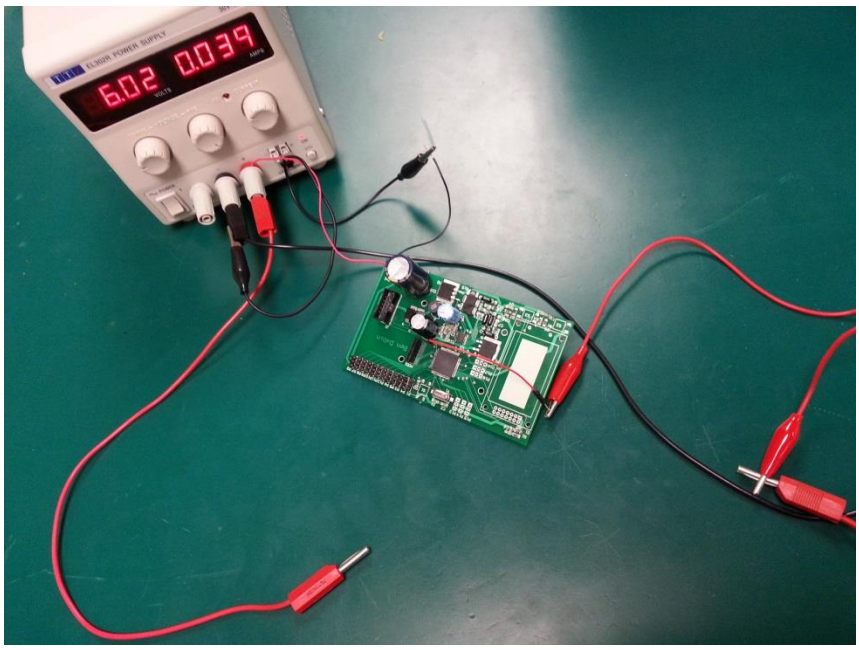

Figure 6.0

The ***Figure 6.1*** shows the output of the +5V veer displayed on the oscilloscope. The reading suggested a noise of ***+1V*** peak to peak, which is considered to be significant for microcontrollers because this could potentially contribute to all different kinds of problems

that could affect the stability of the system. The type of issues that could happen is possible Brown outs, and or other possible intermittent faults. An acceptable level of noise is anything less than ***100mV peak to peak***.

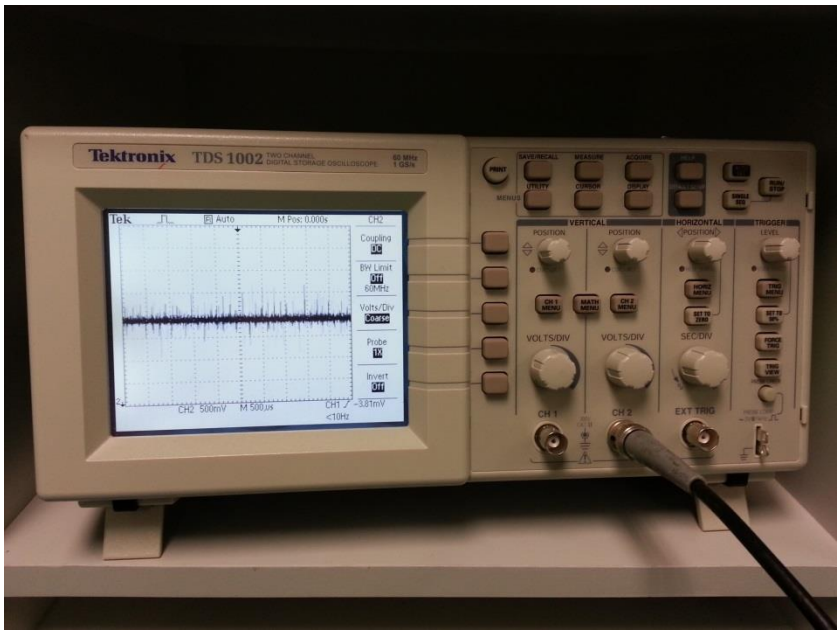

Figure 6.1

A possible solution would be to either re-lay the current PCB, ensuring that the ground and power tracks have minimal possible impedances and making the ground and power tracks thicker.

So in an attempt to rectify the problem, a 100uF Electrolytic capacitor decoupling capacitor was hard soldered and placed on the underside of the board across the output of the +5V in an attempt to smooth the ripple voltage. The smoothing effect that this had on the +5V output was very small. Next a 100nF ceramic (Non electrolytic) capacitor, with a very low ESR rating was also placed in parallel with the 100uF capacitor. This had a slightly better smoothing effect, bringing the ripple voltage down to 500mV peak to peak. Again this had very little effect, for when the ***microcontroller*** was reprogrammed as it would sometimes program properly but then the same error message would be displayed again.

More ***100nF*** decoupling capacitors were placed in parallel with the existing decoupling capacitors in an effort to eliminate the ripple voltage further. The findings from doing this seem to have an even smaller effect on smoothing output voltage. When trying to reprogram the microcontroller again the same error would be displayed. There were also situations where it would program properly but then fail to operate according to the programming code that was inside the device. This type of error could be described as an ***intermittent*** type of error. The next possible steps were to research further and then determine what the possible cause of the problem is.

Again repeating the previous steps as mentioned before, the PCB was connected and configured in the previous configuration where the output of the oscilloscope probe was connected directly across the ***C15*** capacitor to try to smooth the ripple voltage again. This time the power was turned on, this was required in order to see what the effect of adding capacitors were for smoothing the output ripple voltage. As more capacitors were added the smoothing effect was very minimal, but the consequences of this were catastrophic when the

microcontroller was programmed again. The micro would not ***program at all***. The error message that the AVR complier would display back to the user were that the program failed to recognise the device or target the debugger was trying to communicate with. The cause of this could have been due to generating voltage spikes when adding additional capacitors when the power was turned on, which could have possibly contributed to the microcontroller for not functioning where it would send the static sensitive micro controller to blow.

To determine whether the microcontroller was blown, the supply and ground pins were measured using a digital multi-meter. A reading of approximately +5V was displayed on the digital multi-meter LCD display, and 0V for the ground pins, this was evidence that the micro controller was no longer functioning. To be certain the ***fuse bits*** settings in the IAR complier were configured in an attempt for trying to get a response from the ATMEGA1280.

Another possible scenario as to why the microcontroller was not functioning anymore could have been due to trying to setting the ***fuse bits*** in the AVR complier to the incorrect clock frequency thus causing the microcontroller to stop working and preventing further programming.

After the problem was solved and the existing microcontroller was replaced the same error message would appear. After more programming attempts were made to program the micro, an unexpected error messaged arises, which says that it fails to recognise the target, an error message generated by the complier. In an attempt to rectify the problem further a digital multi-meter was used and measured on the microcontroller voltage and ground pins. A reading of ***0V*** was displayed, and this was coming directly from the **+5V supply rail** of the voltage regulator. This fault could only suggest that the regulator circuit had stopped functioning. In another attempt to solve the problem, the schematics diagram was referred too and the components associated with the regulator circuitry were measured. All of the readings surrounding the components were normal, the ***Vin*** pin of the voltage regulator was measured, a ***+6V*** indicated a normal reading, and then the ***0V*** pin was also measured. Next the ***VOUT*** pin was measured, the ***+5V*** reading no longer appeared on the digital multi-meter screen. A conclusion was made where the cause of the problem was that the *buck-boost regulator chip* and that it would need to be replaced.

Instead of removing the regulator chip an easier solutions was made, where a +5V DC supply (AC to DC adapter 250mA plug) was soldered directly to where the ***+5V*** supply line was feeding to the ***micro***, ***LCD*** and ***Bluetooth*** module as a substitute. Some other extensive modifications were also made to the existing PCB tracks where the tracks were cut and removed in appropriate locations. The actual input terminals of the PCB were also modified to handle thicker wires that will come directly from the +6V battery. The current terminals that the board has are not adequate enough to hand the amount of current that the board is intending to supply.

The first PCB now has a system where the **+6V battery** was just being used to power the servos and the **+5V adapter** for the **microcontroller** and other **hardware devices**. This was only a temporary measured but after several program attempts were made this method work as expected and no further problems were reported. This allowed the next phase of the project to be commenced which was to connect all of the **18 servos** and program them. This solution was a good solution for the time being until a second improvement **PCB layout** was made that would take into consideration all of the problems and faults that had been encountered and mentioned in this section. As shown in ***Figure 6.2***.

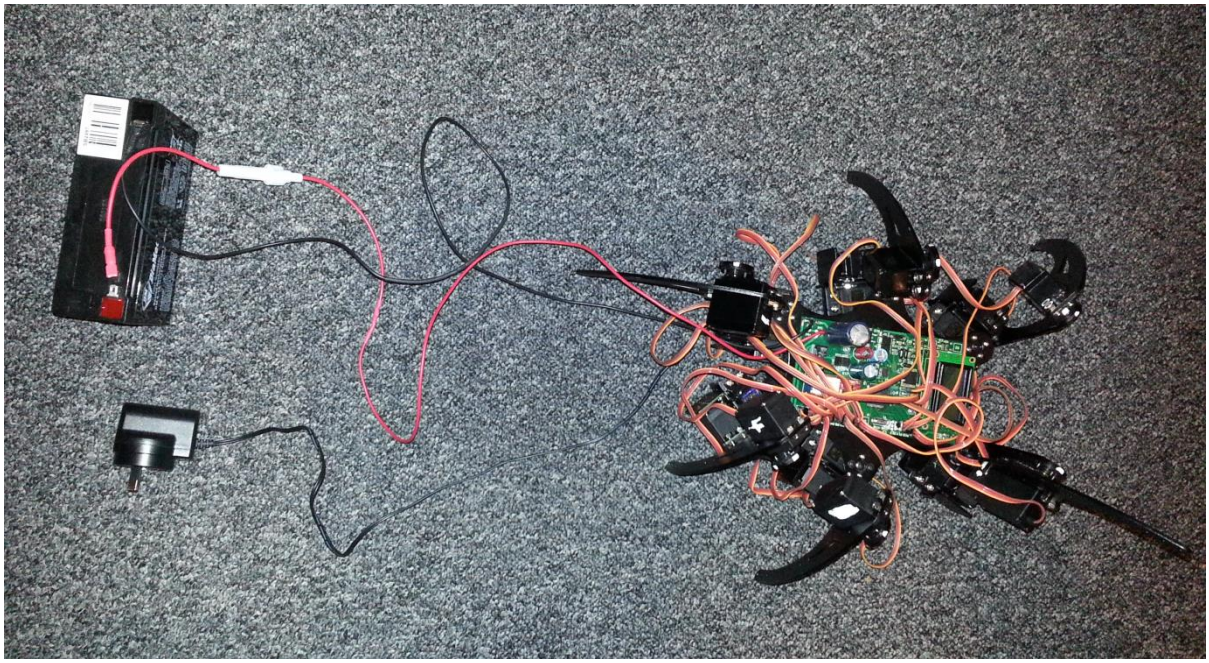

Figure 6.2

Second PCB issues

There were also some issues with the second PCB. The issues were that the same error would show up which was ***The IDR register was written by target 0xFF".*** The initial issue was thought to be with the ***+5V*** supply rail, even after the existing layout was tidied.

After researching this error from various online forums and on the ***ATMEL*** website, the solution suggested that this error could arise from possible noise issue with the ***JTAG*** programming interface or anything associated with it. Another thing that was noticeable was that when the ***JTAG*** device was plugged into the board JTAG socket when the power was turned on, the LCD would strangely turn off and the same thing would happen with the Bluetooth module.

The output of the regulator was checked under an oscilloscope scope for noise problems, there was some high frequency spikes occurring which could possibly contribute to the noise problem. To solve this problem an extra decoupling capacitor was hard soldered onto the board and the output of the +5V was checked again for noise. This had a really positive effect of eliminating the noise from 200mV pk to pk to 25mV pk to pk.

When the board was re-tested and programmed again, the board performed as expected. The problem only occurred again when the +6V lead acid battery was connected to the input instead of the NiMD battery. Unfortunately the error had occurred again, this time the microcontroller would not program at all.

Next the noise problem was researched into even further where the +5V output from the regulator was checked for noise under the oscilloscope. The results from the Oscilloscope showed high frequency noise spikes with a maximum height of +200mV pk to pk. So in order to solve this problem, 4.7uF decoupling capacitors were place across the +5V and +6V supply rail, one across the microcontroller and another across near where the battery input is

located. Then the output was measured again, this had a much better decoupling effect which effectively eliminated half of the noise.

The noise also got worse when external devices were connected to the board, such as the JTAG programming interface and Bluetooth Module. The previous design did not account for this. The reason for the additional noise was probably due to the Buck Boost Regulator going into a different kind of switching mode when pulling more current from the +5V supply rail and staying in this mode until another change occur. The design had to compensate for this change. So the solution was too more add more decoupling capacitors and filtering to the existing PCB layout. Also a lower ESR 2200uF replaced the 1000uF capacitor in an effort to decouple the noise further. After the changes were made the +5V rail was measured again for noise. This had a much better smoothing effect but the noise was still present in the range of +50mV peak to peak. So to improve the problem, additional wires were soldered onto the board in an effort to lower the impedances of the tracks. This had a very noticeable effect on the successful smoothing of the +5V supply rail. The next step now is to carefully reprogram the micro and carry out additional testing to determine whether the noise and debugger error will appear again.

After one day of careful testing and reprogramming of the micro with the servos connected to the PCB and connecting the external devices, the error no longer appeared on the debugging output window. When the target voltage was measured from the fuse handler in the IAR debugger the voltage fluctuations were a lot smaller compared to the previous tests. The voltage fluctuations were the main reason why the micro kept crashing all of the time because whenever the supply voltage went below a certain value, the micro would shut down in the middle of trying to run the programming code in the debugger. This would cause the micro to be confused and loose its sense of direction.

Solution

The solution to the error problem this time was the JTAG programming ribbon cable, was not properly inserted into the AVR dragon, which contributed to the weird electrical noise that was causing the microcontroller to behave in a strange manner.

Servos

The ***Tower Pro MG995*** servos that were supplied with the kit had to be replaced and repaired many times. This was due to the low quality material that the servos were made of most of the servos that had to be replaced would regularly overheat and overshoot when sending a ***PWM pulse*** to them to get them to move at a certain angle.

Inside the servo its self contains the gears and potentiometer mechanisms that control the servo positioning angles. If the servos are not driven at the correct frequency which is **50HZ**, they can generate an excess amount of noise and heat, which is believed to contribute to the servos breaking, this was discovered when attempting to program them.

Breaking was even notice when the robot was programmed to stand up because eventually over time the servos would break again and then would have to be replaced. This highlights how poor quality the servo material was, for the intended application.

The image in Figure 6.3, shows the inside of a Tower PRO MG995 servo, with its gears showing.

Figure 6.3

The next photo, ***Figure 6.4,*** shows a broken servo top cover, the part that is actually broken is the tiny hole inside the plastic which mounts the gear assembly. The possible cause of damage could have been due to excess weight putting stress on the plastic casing of the servos which caused the servo gears to seize up and stop functioning.

Figure 6.4

As these servos were constantly breaking all the time the ideal solution would be to replace all 18 servos with a better quality brand, made from a much higher quality of plastic but due to the limited budget that the **AUT** has, comprises on the walking sequence had to be made, using the current brand.

Bluetooth module

The Bluetooth module that was used for the project was the ***RN-42 HID BlueSmirf*** Bluetooth Module. The issue that was encountered with this Bluetooth Module was trying to configure the module to be in S***PP (Serial Port Profile)*** mode. The other mode that it has is ***HID*** mode which is only designed for Bluetooth hardware such as wireless keyboards and mice. The problem that had been encountered was when it was accidently set in ***HID*** mode. The real challenging part about this was trying to get it out of that mode.

The baud rate of ***9600*** had to be configured on the Bluetooth Module. To achieve this, a laptop with Bluetooth capability was used and a DOS like terminal window program called ***Tera Term,*** was used to pair with the device. Pairing with the device allows for commands to be sent wirelessly to the module that allows it to be configured. This can only be achieved if the Bluetooth module is set for SPP mode, which fortunately by default is the case.

In order to determine current setup of the Bluetooth module, the following steps had to be done, the ***Bluetooth*** module was be put into command mode where the user enters or types ***$$$*** into the terminal window. The thing that has to be taken note of is that this can only be done within 60 seconds of the Bluetooth module booting up, after 60 seconds if this window is missed the procedure must be repeated again. Once in command mode the Bluetooth module status light starts to flash more rapidly.

Next step was to type ***X<cr>***, this allows the user to view the current setup of the Bluetooth module as shown in ***Figure 6.5***. The user can view the full setup configuration that is useful to the user such as the ***baud rate***, Mac address and even the mode the user is currently in. The factory default baud rate on module is 115.K by default; this was changed to ***9600*** where the data being transmitted and received is being done at a rate of 100us per bit. Both the ***USART*** on the ***ATMEGA1280 and ATMEGA16*** microcontroller and the Bluetooth Module had to be configured with the same Baud rate; otherwise communication would not be possible.

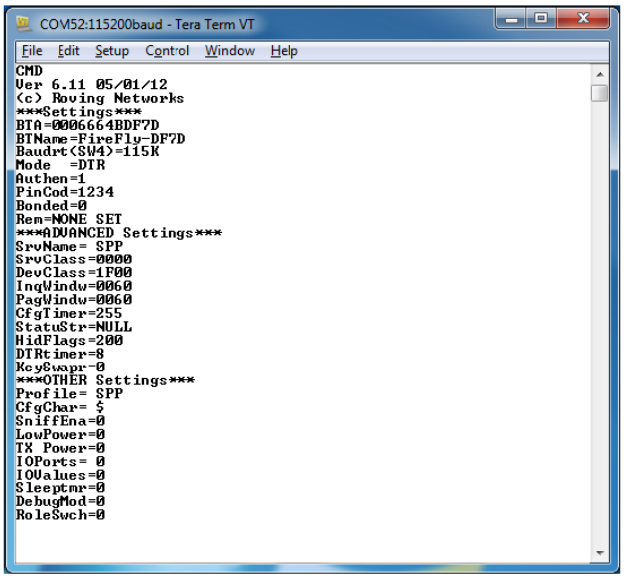

Figure 6.5

Embedded Software Issues with Servos

There were many obstacles encountered with this part of the project that made programming tedious and cumbersome. The difficulty presented, was where each servo would move at different angles or in different directions when programmed with the same OCR values in corresponding registers. The reason for this was due to the way each servo was orientated or positioned on the metal leg frame of the hexapod and the type of servos used. Some of the replacement servos used would also move in opposite directions to the original ones that were provided with the project.

To resolve the issue a specially designed algorithm was designed which allows an angle value to be converted into a corresponding ***OCR*** value. This technique simplifies programming and allows a servo angle position to be written in terms of angles values as opposed to OCR values, see ***Embedded Programming Code*** section.

Also different servo which moved in opposite directions was positioned on the opposite side of the robot in an attempt to get the servos to move in equal opposite positions.

Figure 6.5.1 shows the original ***Tower Pro MG995*** servo that was provided with the robot kit. ***Figure 6.5.2*** shows the replacement servo. They both move in opposite directions of rotations. They differ slightly in appearance, the servo in ***Figure 6.5.1*** has a more dull and rough surface, whereas in ***Figure 6.5.2*** the servo has a shiny smooth black surface.

Figure 6.5.1

Figure 6.5.2

Re-Calibration of Servos

Every servo had to be calibrated and programmed so that they operate synchronously with each other. This presented a major challenge because some of the current and replacement servos moved at different angles of orientation. So certain servos had to be mounted and positioned on a particular side of the robot.

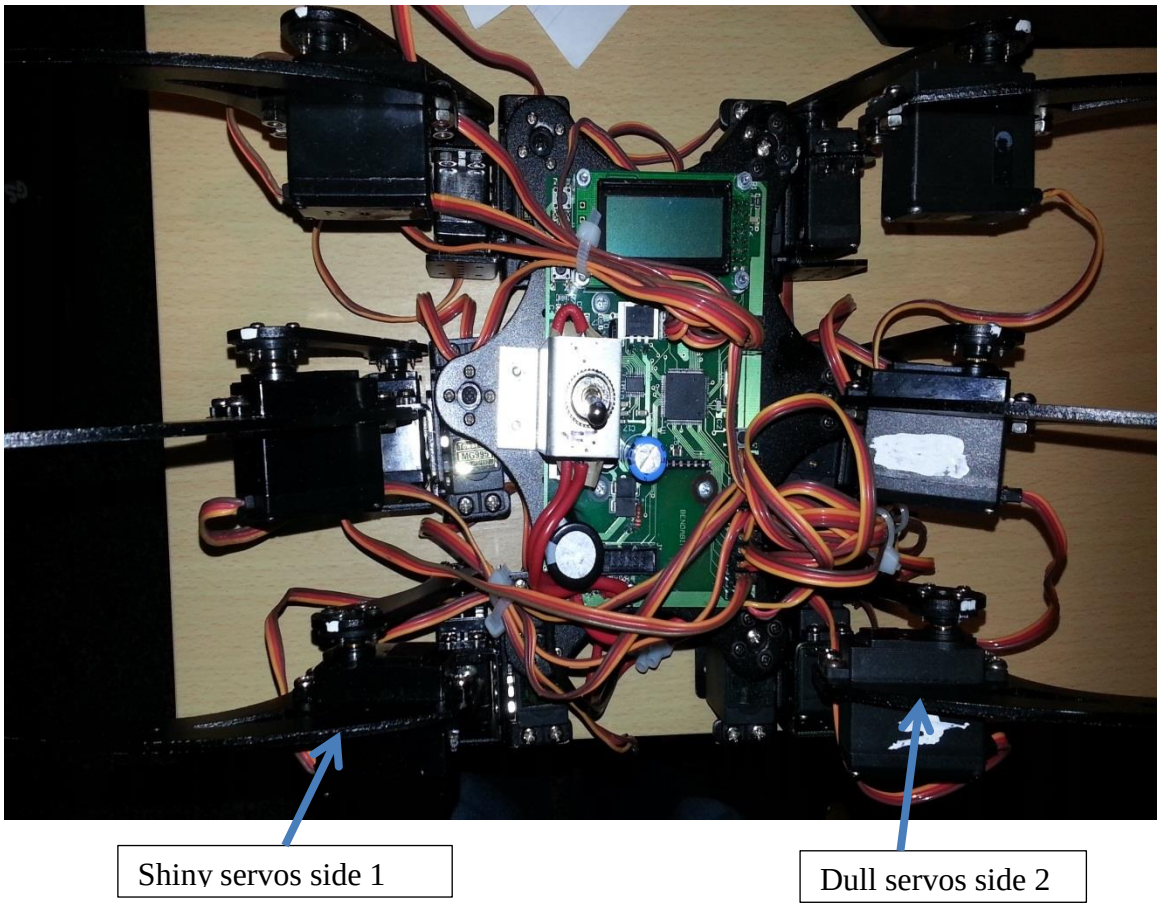

Figure 6.5.3

Programming the robot was made much easier when this modification was made as the servos would be in the same direction on both sides when the same OCR value was sent to them.

A servo exercise function, ***servo_ex()*** as shown in ***Figure 6.6*** was used to determine which servos were faulty or needed replacement. The advantage of this function is that it can be used to sweep through all of the ***1 to 12*** A and B servos. This function tests a range of servos to use and will also sweep through angles 0 to 120, this allows simplified checking for determining which servos need replacing and this also allows servos to be calibrated and aligned.

It was not necessary to write a routine for the ***6 C*** servos because these were not replaced that often and also the ***angle2Count()*** and ***servo_drive_c()*** functions could be used.

```
//// Excercise each servo on the robot. This drives each servo from
////  OCR_MIN to OCR_MAX in steps of OCR_STEP_SIZE

void servo_ex()
{
        int servo, ocr_cur;

                for(servo=1; servo <=12; servo++)
                {
                        // step up from 0
                        ocr_cur = OCR_MIN;
                        while(ocr_cur < OCR_MAX)
                        {
                                servo_drive(servo, ocr_cur);
                                ocr_cur += OCR_STEP_SIZE;
                                //delay here
                                __delay_cycles(OCR_TEST_DELAY);
                        }
                        __delay_cycles(OCR_TEST_DELAY);
                        //step down from 300
                        ocr_cur = OCR_MAX;
                        while(ocr_cur > OCR_MIN)
                        {
                                servo_drive(servo, ocr_cur);
                                ocr_cur -= OCR_STEP_SIZE;
                                __delay_cycles(OCR_TEST_DELAY);
                        }
                }
}
```

Figure 6.6

Previously programming the servos was done at a low hierarchical level for attempting to write the walking sequence and check for faulty servos. The disadvantage with this programming technique is that it proved too cumbersome and challenging, so little progress was made. The advantage behind the ***servo_ex()*** function is that it removes the need writing for programming code directly to the OCR registers and the function can be used to sweep through of the angles without having to consider the low level coding of the robot.

Conclusion

In conclusion to the final year project report, the past year was spent highlighting the important factors and events that had been encountered over the course of the year. The purpose of the project was to design and develop a controller ***PCB*** for an ***18 servo robot kit***.

There were many challenges and situations that made programming tedious and cumbersome. The situations that occurred were with the brand of servos that were supplied with the robot kit, which were Tower ***Pro MG995***. The constant breaking and replacements of these servos made developing the walking routines very challenging. Therefore comprises had to be made in order to get the robot to walk. The initial plan earlier on in the year was to get the robot to stand up and move at any chosen height, but this proved not possible. The reason for this was that every time an attempt was made to get the robot to stand up for long periods of time, many of the servos would break and have to be replaced. For a possible future development a better brand of servos would solve this problem. The limited budget of the ***university*** that supplied the materials prevented the use of a higher quality materials and replacements from being sought. The final walking routine that was developed enabled the robot to walk but it could only walk when the base frame was flat on the ground.

Other challenges that had been encountered and overcome over the course of the year were with the ***2 PCB's*** that were designed for the project. Both PCB's had noise issues that prevented the microcontroller from being programmed. The cause of the problem was with the type of regulator circuit that was used to generate the ***+5V.*** The problem was eventually solved when additional decoupling capacitors were added to the ***second PCB.*** A future reference for this would be to carefully layout the PCB when using this type of regulator and add more decoupling capacitors because the initial design did not account for the regulator changing into different modes of switching when more current was being drawn.

The servo connectors on the PCB could also be changed with connectors that grip the servos wires in place better. This was a common problem that occurred where the connectors would keep coming out of the board. An alternative solution would be to make the wires longer.

Overall the project was a success especially with the ***smartphone GUI*** that was developed. This item worked according to what was initially mentioned in the first part of the project report. There were still some challenges and obstacles that were mentioned earlier on but these had been resolved. A future reference for this would be to include more features such as voice recognition or mapping user interface options.

For the embedded coding part of the project, many attempts were previously made to get the code to control the function of the 18 servos. The code had to be organised in a hieratical fashion for each servo leg. A servo value algorithm was designed to convert angle values into ***PWM values***. This method of operation worked well with the servos because programming the servos could now be thought in terms of angles as opposed to ***OCR values***.

Other improvements that could be made for the project is including on-board hardware features such as proximity sensors for detecting presence of nearby objects. This was initially considered but not used because of the limited time available and the fact that other more important issues with the PCB had to be rectified.

The final product for the ***hexapod robot*** is shown in ***Figure 7.0.*** This shows everything working and up to date. An on/off toggle switch was also mounted and drilled to the existing base frame; this was put in place to allow the robot to be switched off.

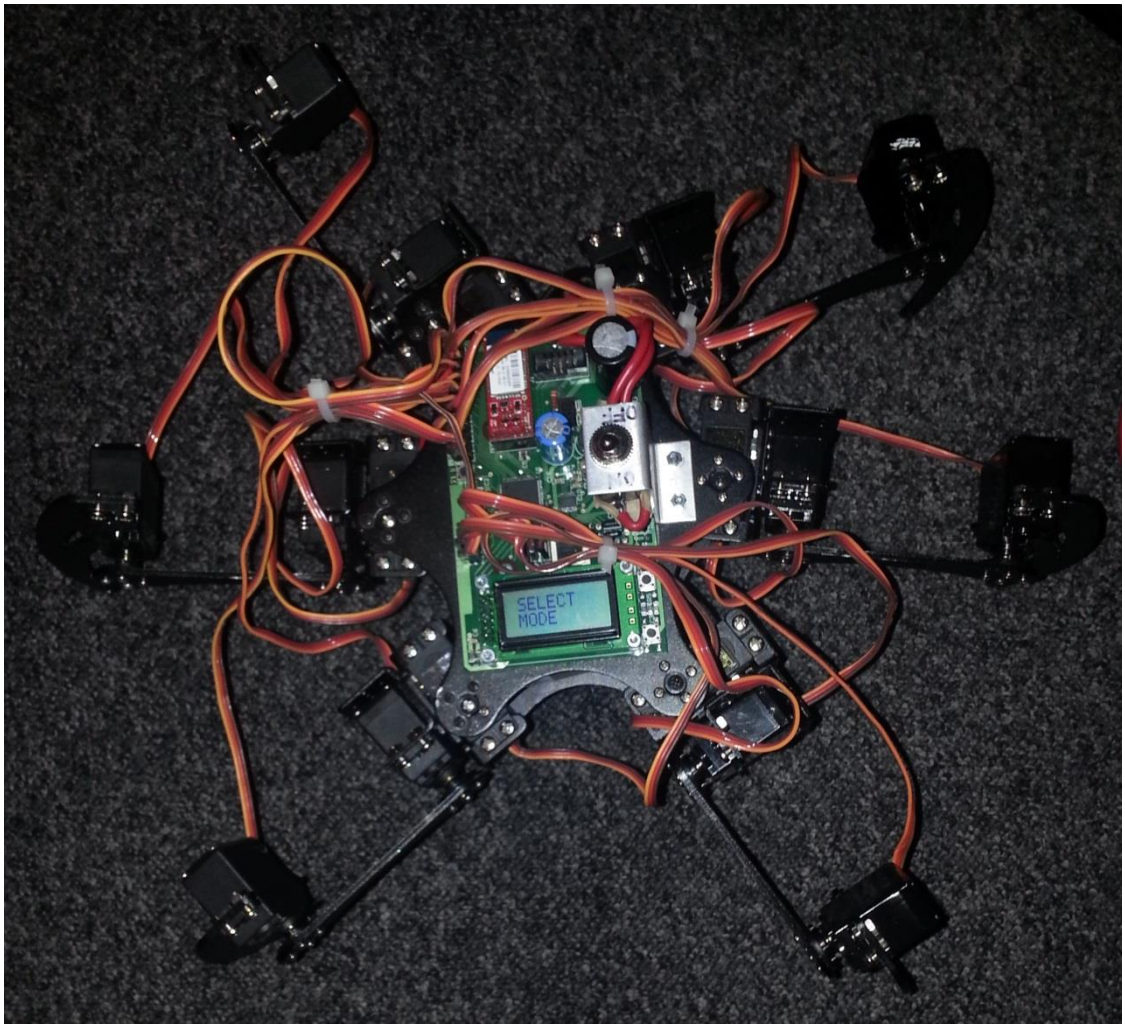

Figure 7.0

Another possible suggested future reference for cosmetic purposes could be to house the robot in some kind of box that could disguise the wires and make the robot look more presentable.

Additional Sheets