Fuzzy Logic Control of an Inverted Pendulum Robot

By

Kyle Reid

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

2010

Table of Contents

Acknowledgement

This report and the project itself would not have been possible without the support of my father and mother who supported me financially and emotionally.

Abstract

The inverted pendulum is a classical problem in controls. The inherit instabilities in the setup make it a natural target for a control system. Over the years it has been the benchmark for testing new and innovative control theories such as Fuzzy Logic. During the course of this project a software fuzzy logic controller was implemented using a PIC microcontroller. At the conclusion of the experiment an interesting realization was made about the nature of Fuzzy Logic and its applicability in the world of controls.

I. Introduction

Control systems have a variety of uses in society today from the control of the temperature in house to life saving devices such as anti-lock brakes. One of the main concerns in any control system is stability for without this the system will cease to function in the prescribed manner. One of the best experiments in classical controls is the inverted pendulum. Due to the extreme instability of the plant this problem covers a wide spectrum of the problems and solutions common in the world of controls. This project will investigate the control of an inverted pendulum using an accelerometer, gyroscope and a software Fuzzy Logic controller.

II. Background

The theory of Fuzzy Logic was developed by Lotfi A. Zadeh in the mid-1960s. It was originally created as a way of handling data rather than control systems.

> Fuzzy technology, Zadeh explained, is a means of computing with words-*bigger*, *smaller*, *taller*, *shorter*. For example, *small* can be multiplied by a *few* and added to *large*,or *colder* can be added to *warmer* to get something in between." (Perry, Jun 1995)

This loose approach to problems in the world of engineering and mathematics was blasphemy to the established ideas of the time. To think that a complex engineering problem could be solved with approximations and intuitive thinking was something most people in the industry didn't accept. The Japanese were the first to accept the idea of Fuzzy Logic as a means for controlling a plant in a control system. Their successful uses of this idea lead to some positive press coverage and mainstream acceptance of the validity of Fuzzy Logic. The concept of fuzzy logic in controls centers on many IF-THEN statements and of course the use of membership functions. In the system built for this project there are two input membership functions and one output membership functions. The three basic steps for any Fuzzy Logic controller are fuzzification, inference, and defuzzification. In fuzzification, membership functions are created to categorize the information the

inputs are conveying. For example, if the input to the controller for a thermostat was temperature, the membership functions may be warm, cold and hot. Figure 1 below shows how membership functions for this example might be organized.

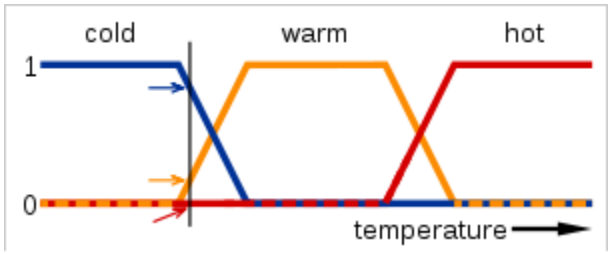

Figure 1: Membership function example

The line shows where a value for the input may lie, the intersection of the line with the membership functions determines the degree of membership the input has to each function. In this example the input may be 0.7 cold, 0.3 warm and 0.0 hot. Once this is determined it is necessary to infer what each degree of membership means for the output of the controller. Rule sets are used to dictate possible outputs for the various input conditions. IF-THEN statements are used to determine the rules. For example a rule for the thermostat might be:

IF the temperature is cold THEN turn on the heater.

It is important to note that more than one rule may be applicable for each input value. All rules are evaluated each run through the controller and the degree of membership determines the weight each rule will have in the determination of the output. A rule matrix such as the one in Figure 2 is used to determine all outcomes

possible from the information gathered in the fuzzification step for more complex situations involving multiple inputs and more than three membership functions.

Angular Velocity (theta_dot) \ Angular Position Error (theta)	VNeg	Neg	Neu	Pos	VPos
VNeg	VNeg	VNeg	VNeg	Neg	Neu
Neg	VNeg	VNeg	Neg	Neu	Pos
Neu	VNeg	Neg	Neu	Pos	VPos
Pos	Neg	Neu	Pos	VPos	VPos
VPos	Neu	Pos	VPos	VPos	VPos

Figure 2: Rule Matrix Example

Once the applicable rule sets have been determined it is time to convert this information into a crisp output in the defuzzification step. To accomplish this, methods such as center of gravity, center of singleton, and maximum methods are employed to establish a viable output for the controller. Each method has its own pros and cons and it is up to the designer to determine which method is right for his application.

III. Requirements

- Robot must maintain stability between negative four degrees and positive four degrees of inclination.
- Once initialization is complete the robot must stand on its own without human involvement.
- Robot must be powered by a single 12v supply and a single USB connection to the microcontroller board.
- Control algorithm must be accomplished using fuzzy logic.

IV. Design

Design of the robot naturally centered on the basic idea of an inverted pendulum. There would be a weight at the top of a long shaft and there would be some sort of mechanism attached to the bottom of this shaft that would be used to balance the weight at the top. Unlike most inverted pendulum experiments that consist of a cart on a track that limits mobility, this project was constructed much like a Segway to enable the robot to have a greater range of mobility. Figure 3 below shows the design for the robot from the initial proposal of the project.

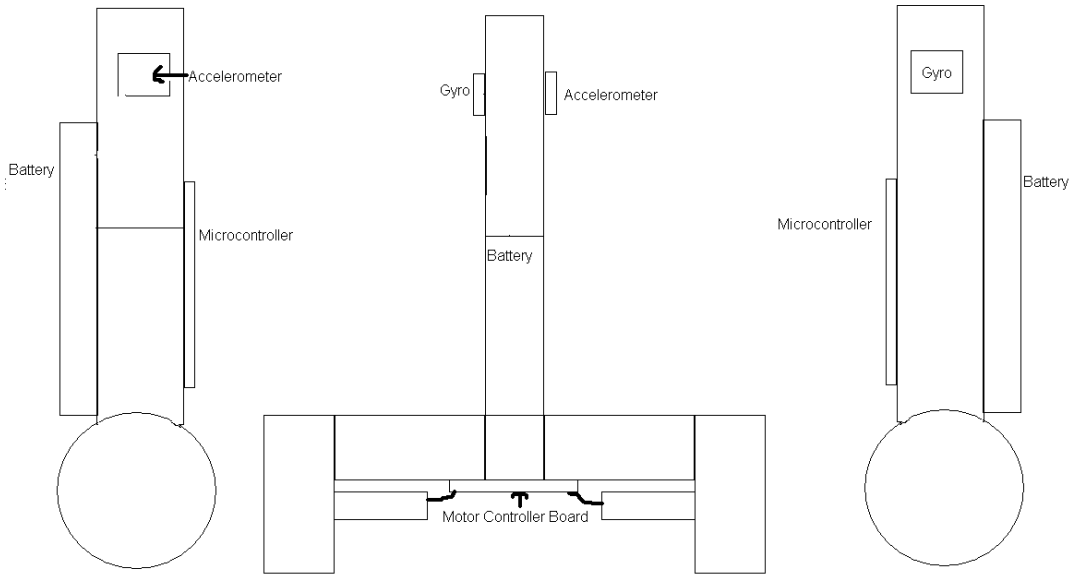

Figure 3: Preliminary Sketch of Robot

The main control of the robot is accomplished using PIC32MX460 Development Board from Sparkfun. Due to the fact this board would be the only

computational center for the robot it needed to be fast enough and have enough memory to handle several computations and a complex control algorithm. The 32bit PIC would be more than enough to satisfy this qualification. In addition, it was desired that the board be easy to learn how to use, have open source software and be easily programmed using a computer. This particular development board was created by a third party, and its design breaks out all 72 input and output pins of the PIC32 microcontroller. In addition, the microcontroller has been configured with a USB bootloader to allow for convenient programming directly through any USB port, using the bootloading software provided with the board. PIC microcontrollers are made by Microchip, a well established company that provides a wealth of resources that can be utilized to allow for easy programming. MPLAB IDE is a program used to compile software programs for the PIC microcontrollers. It is available as a free trial with slightly limited capabilities. Microchip also provides an extensive library of code to help simplify the use of on-chip peripherals. The availability of peripherals was also a main concern in selection of the microcontroller. It was necessary to have access to 3 ADC channels with above average resolution for conversion, several timers for time keeping in the project, and at the minimum I^2C and serial interfaces. The development board was able to fulfill all requirements including a 10 channel ADC with 10 bit resolution as well as several timer counters, SPI, serial and I^2C interfaces.

The next crucial component for the robot was a mechanism to determine the angle of inclination. Several different ideas for approaching this problem were researched in the design phase such as, inclinometers, potentiometers and simple contact switches. However, the solution that best combined cost effectiveness and accuracy was the combination of a two axis accelerometer and a one axis gyro. Both a gyro and an accelerometer were necessary in order to allow each inertial measurement unit to compensate for the weaknesses of the other. The accelerometer, as its name suggests, is a device that measures acceleration. Ideally for this application it would be necessary to measure only the acceleration due to gravity. However, the accelerometer measures all accelerations that it experiences, including mechanical vibrations and any type of jarring motion. As a result it provides a very noisy measurement of the angle of the robot. The gyroscope in contrast measures the rate of change in the angle of the robot; it is possible through the use of trigonometric formulas to glean from this information a very steady measurement of the angle of inclination. Unfortunately the gyro has a problem with drift, meaning, as the robot is rotated away from its upright position and then back again it may not display the same reading each time it reaches upright. Therefore, in order to solve this problem of unreliable sensors, it is necessary to use both sensors and implement a filter that uses only the best attributes of each sensor in the measurement of the angle.

The third main component for the robot is the drive system. This system needed to consist of two gear motors with enough speed and torque to be able to right the robot if it rotates away from vertical, wheels that had enough grip to be able to maintain traction during operation, and encoders on the wheels to be used for optional position control of the robot. In addition, it was necessary to have a motor control board that would be easy to access and control and provided a great deal of customization. For this task the RD02 12v Robot Drive system from Devantech was selected. It contains two 30:1 gear motors with 360 encoder counts per revolution, rated torque of 1.5kg/cm and a rated speed of 170rpm. It also comes with a control board that is accessible by both Serial and I^2C interfaces, it also has 16 onboard registers for full customization and can operate off of one 12v supply and is capable of providing 5v to the rest of the project. Due to the increased challenge of adding a 12v battery to the system and the limited time available for completion, a 12v power supply was used instead. A complete system was selected over the option of creating from scratch a motor drive board to maximize the amount of time available to tune the control algorithm.

The final consideration for design of the robot was the main structure. This was the last component to be decided and was designed to be able to handle the more important components described above. The body was constructed form 1.24” PVC tubing to accommodate the diameter of the motors which would be housed inside of the tubes. In addition, this material was selected because it is easy to cut, easy to connect to other pieces using joints and glue and because it is a rather light and inexpensive material.

V. Integration and Testing

Due to the complex nature of the system being designed, there needed to be incremental testing of the various components followed by small scale and then large scale integration. In all embedded systems it is essential to become familiar with the microcontroller and development board that you are using. In this instance the board came equipped with several LEDs so the first test was to create a program to make one of them blink. This would prove competence in the coding style for PIC microcontrollers as well as verify that the board worked. Using a program that came with the development board that turned on a specific LED it was simple to both become familiar with the coding style for PIC microcontrollers and cause that same LED to blink at a specific interval. To further understand the intricacies of how the board operated and also to gain a valuable feedback device, a 16x2 character LCD display was added to the project. The display is controlled by 8 bit signals sent through the general purpose input and output (GPIO) on the board. There is a strict order that signals must be sent in and a precise wait time in between consecutive signals. As a result it was necessary to incorporate two timer counters into the project. One of the timers would create an interrupt every millisecond and the other every microsecond, timing intervals were tested using a stop watch and a

blinking LED. Once the operation of the timers was verified the display was programmed to display the ever popular “Hello World!” message. The successful display reconfirmed the accuracy of the timers and demonstrated that the GPIO and LCD display were operational. Next the various on-chip peripherals needed to be tested. In addition to the timers previously tested this project required I^2C and serial interfaces and an analog to digital converter (ADC). It became instantly apparent that the addition of a serial interface to the project was not going to be a trivial procedure. In order to communicate with a COM port on a computer via an RS232 cable certain adjustments needed to be made to the serial lines coming from the development board. An RS232 shifter was purchased in order to shift the voltage levels from what is provided on the development board to what is necessary for serial communication with the computer. When the first test of the serial communication was attempted a complete failure was observed with nothing being displayed on the HyperTerminal window on the computer. Investigation into this problem revealed that the shifter that was purchased had been incorrectly labeled and that transmissions had unintentionally been passed to the receiving pin of the device. Once the wires were switched communication began to function normally and the familiar “Hello World!” message was displayed on the computer. Having the LCD display was essential in testing the accuracy and functionality of the ADC. After the ADC was configured for 10 bit interrupt driven conversion on three channels a simple test was conducted to determine if all channels were working properly.

Using a 12v variable DC power supply voltage in the range of 0-3.3v was applied to the inputs of the ADC; after conversion was complete results were displayed on the LCD display as a 10 bit binary number. As the voltage was increase this number also increased until the ADC was saturated at 3.3v, this not only verified that the ADC was working but also that the appropriate range had been selected for conversion.

The next major test was of the motor drive circuitry, the motors and the communication interface between this circuitry and the development board. Accessing the registers on the motor drive board through I^2C is similar to accessing a RAM chip except that on the motor drive board each register has a specific effect on the operation of the motors. All write request to the board must include the address of the board the location of the register being written to and the value to be written to that register. In order to verify that this communication was working properly two signals were sent to the board. The first was a signal to change the mode to mode 2 which would cause the speed1 register to control the speed of both motors with 0 being full reverse and 255 being full forward. Then the drive signal was send setting speed1 register to 255. With the board powered externally by a 12v power supply, the above commands caused the wheels to spin forward rapidly and the motor control circuitry was determined to work.

Next the sensors for the robot needed to be tested individually. Testing of the accelerometer was a fairly strait forward process. Power was applied to the inertial measurement unit (IMU) board and then the voltage on the x and y outputs was measured with a multimeter, as the accelerometer was rotated the voltage output would either rise above or fall below 1.5v depending on the direction of rotation. Similarly, to test the gyro power was applied and the output voltage measured, the faster the rotation the further the voltage was from 1.5v. As a benchmark for the sensor fusion a qualitative baseline test of the accelerometers ability to measure angle without the help of the gyro was conducted. In order to accomplish this, the sensor was connected to channels 1 and 2 of the ADC and by taking the inverse tangent of the quotient of the two angles the angle of inclination of the robot was determined. Starting at zero degrees of inclination (the robot standing upright) the robot was tilted forward to approximately 11 degrees and then backward to approximately 11 degrees and then returned to zero and the middle of the robot was tapped three times with a screw driver. Figure 4 below is a screen shot taken from the Serial Chart that displays the results of the test.

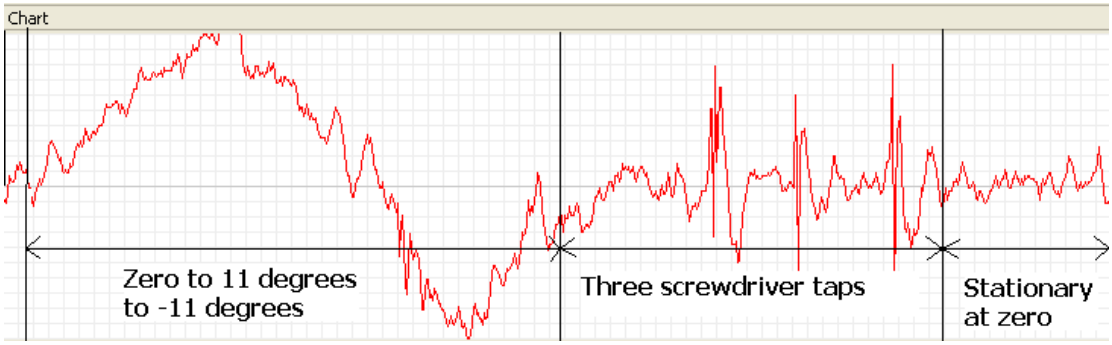

Figure 4: Chart of unfiltered angle measurement

It was determined that the noise this signal exhibited was unacceptable for control of the robot and that the addition of the gyro data and a filter would be necessary. A Kalman-like filter was created to utilize the data from the gyro to predict where the next angle measurement would occur. A gain was used to control how much influence the gyro measurement had on the final angle. Figure 5 below shows the result of the filtering.

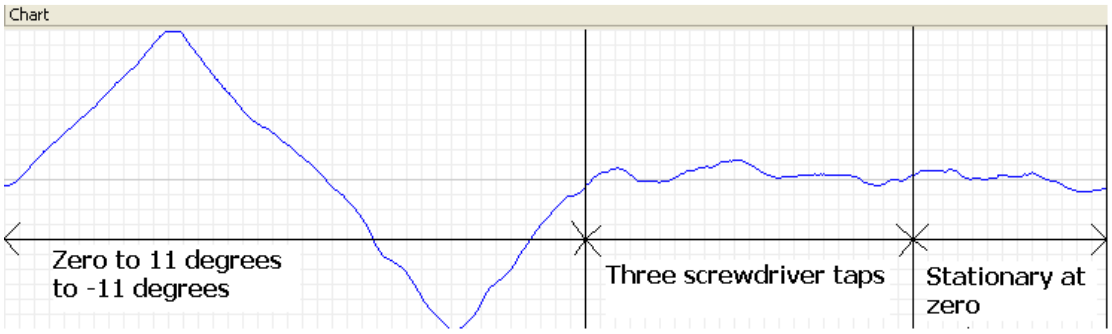

Figure 5: Graph of filtered angle measurement

As you can see the noise that was present on the previous chart has been significantly decreased. It is also worth noting that in the section with the three screwdriver taps the three spikes from the previous chart are no longer present.

The use of the gyro data allows these anomalous angles to be discarded as inaccuracies. To highlight the differences in the charts the two are shown together below in Figure 6.

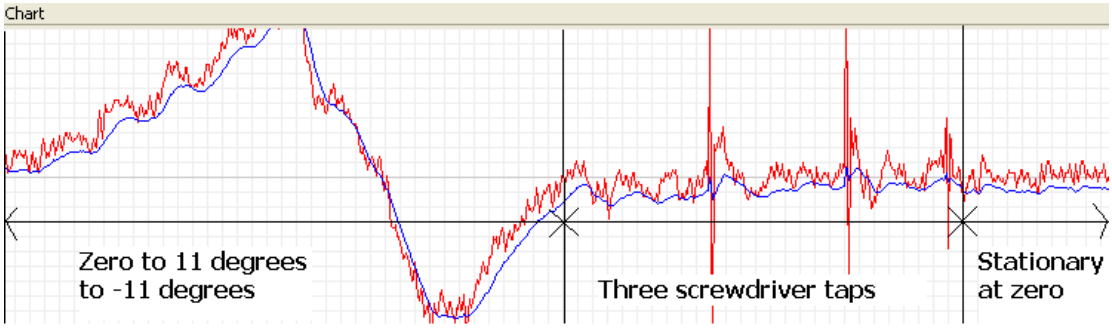

Figure 6: Unfiltered and Filtered chart of measured angle

Clearly the filtered angle, represented by the blue line, is far less noisy and susceptible to vibrations than the red unfiltered angle.

Finally with all the necessary circuitry assembled and tested it was time for the final integration. The components were mounted to the frame of the robot using a combination of screws, zip ties and electrical tape to allow for easy access and manipulation of their location during the final testing of the project. A picture of the finish robot is below in Figure 7.

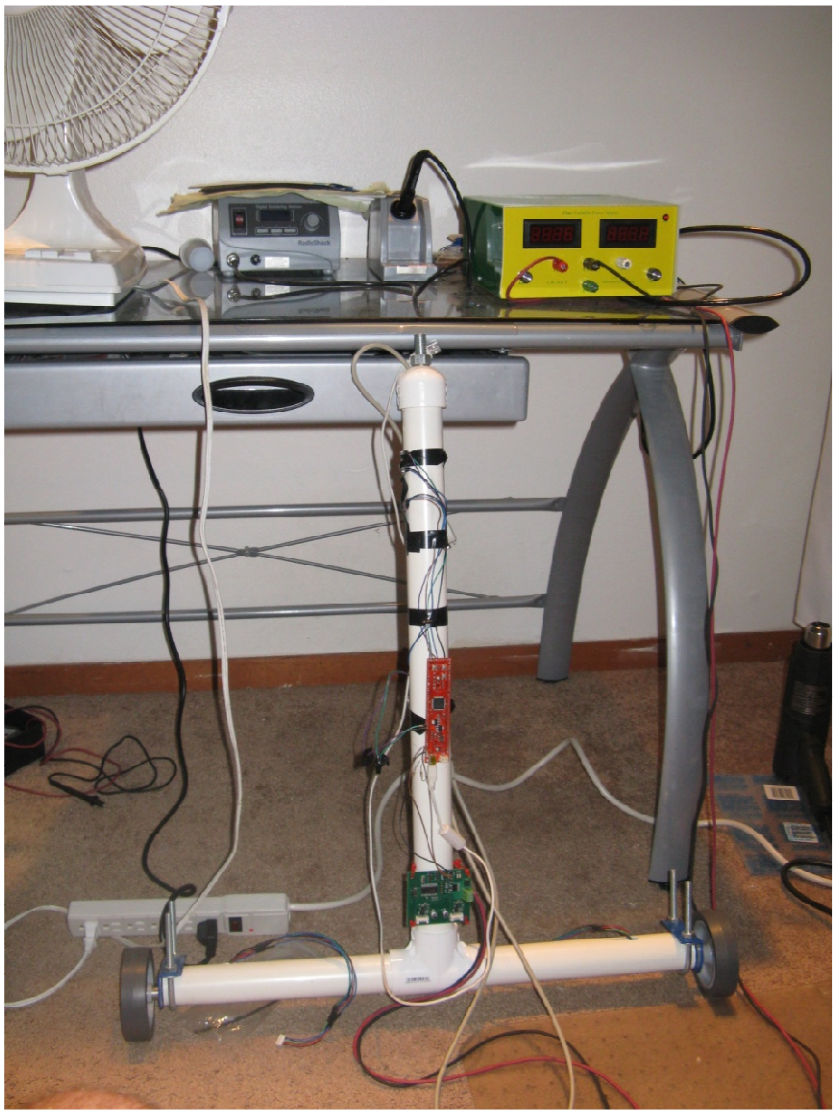

Figure 7: Photo of final robot construction

The initial test of the robot would involve a fuzzy type proportional control algorithm that would basically case the robot to move slowly under itself as the user moved the top of the robot. This test demonstrated that all systems were working together harmoniously and that during assembly no damage was done to any of the components. With this test passed it was time to develop the fuzzy logic algorithm that would hopefully result in a stable balancing robot. To attempt this, a scheme of 5 equal triangles was used as show in Figure 8. This design was used for both inputs and the output of the fuzzy logic controller.

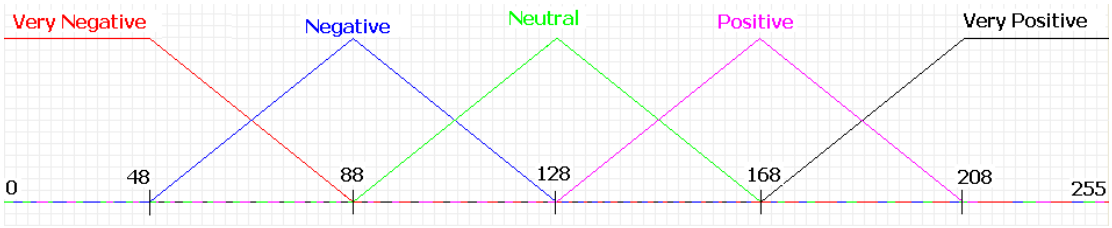

Figure 8: Input and output membership function design

To increase the accuracy of the control the inclination error θ and the angular velocity $\frac{d\theta}{dt}$ were used as the inputs. Modifications to the inclination error and the angular velocity needed to be made in order to have them fit into the range of 0 to 255. To accomplish this they were multiplied by a gain either Ki (inclination error) or Ka (angular velocity) and then 128 was added to the product. The gains could be changed during the tuning to achieve better results. The output did not need to be altered because in mode two for the motor drive board full forward is 255 and full reverse is zero with full stop residing at 128. The rule matrix is identical to the one

displayed in the background section and the method for defuzzification was root sum squared. Initially the performance was severely unstable due to the fact the motors didn't have enough torque to overcome inclination errors greater than about four degrees. Incrimental changes to the gains Ka and Ki were made until an acceptable level of stability was achieved. In an attempt to achieve better results the membership function design was edited to look like Figure 9.

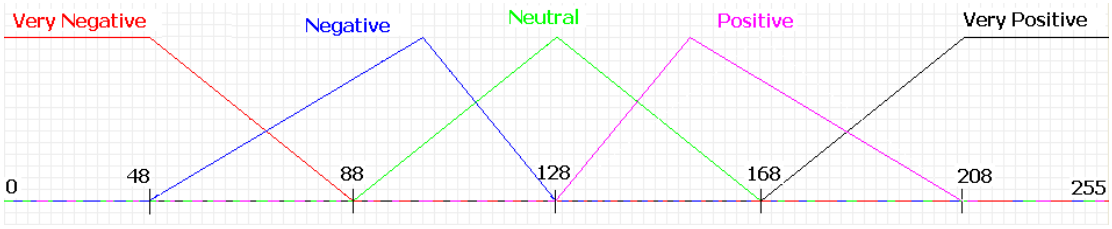

Figure 9: Membership function design with center biased triangles

Having the negative and positive triangles biased more toward the neutral position was hypothesized to increase the stability of the robot when it was in the upright position. Unfortunately after numerous tests and adjustments it was determined that no appreciable improvement was made.

VI. Conclusions and Reccomendations

In the end perfect balance in the upright position was not achieved with fuzzy logic control. In a study on the use of fuzzy control versus PID control of a muscle-like actuated arm it was determined that

> Better control results were obtained in the PID controller compared to the fuzzy controllers. The PID controller quickly recovered the position error that was initially present. (Lee & Gonzolas, 2008)

Clearly this illustrates the one common problem associate with the use of fuzzy logic control, a lack in steady state performance. Unfortunately for this project it is essential that it perform well in steady state, the robots upright position. As a result when the robot is powered on it will initially have good response in the upright posisition, however, as soon as inclination error is introduced even by the effect of gravity and the robot attempts to correct itself an position error occurs in the steady state condition. As the robot continues to correct for this inaccuracy it passes the 4 degree threshold for balance and it becomes unstable. In addition to the known problems with steady state stability, fuzzy logic suffers from a lack of a predefined method of tuning. Unlike the classical control theory such as root locus method or Routh Hurwitz stability criterion that allow for ideal simulation of PID type systems, fuzzy logic provides no method of estimation for gain values. Users without prior

teaching in the intuition of tuning fuzzy type systems are left with a seemingly infinite number of possibilities to increase stability. Based on the information in this report and the experience accrued during this project the best method for fixing the issue of instability in the robot would be to incorporate a PID type controller. Although this method would involve a good deal more work up front to determine the transfer functions necessary to model the non-linear behavior of the robot. The eventual stability would more than warrant that investment. In essence, although there is great promise in using fuzzy logic to conquer some complex non-linear control problems, the use of a PID type controller is a much better established and easily tuned method for controlling an inverted pendulum robot.

VII. Bibliography

Lee, C., & Gonzolas, R. (2008). Fuzzy logic versus a PID controller for position control of a muscle-like actuated arm. *Journal of Mechanical Science and Technology* , 1475-1482.

Perry, T. (Jun 1995). Lotfi A. Zadeh [fuzzy logic inventor biography]. *Spectrum, IEEE , vol.32, no.6* , 32-35.

A. Source Code

Below is included copied source code. For better formatting please consult the provided CD.

```
/********************************************************************
 *                                                                  *
 * This file contains the main loop for the Inv_Pen project         *
 *                                                                  *
 *                                                                  *
 ********************************************************************
 *                                                                  *
 *   Author:       Kyle Reid                                        *
 *   Filename:     main.c                                           *
 *   Date:         8/3/2010                                         *
 *   File Version:  1.00                                            *
 *   Other Files Required: GenericTypeDefs.h                        *
 *                  Compiler.h                                      *
 *                  HardwareProfile.h                               *
 *                  microcontroller.h                               *
 *                  main.h                                          *
 *                  timers.h                                        *
 *                  adc.h                                           *
 *                  iic.h                                           *
 *                  accelerometer.h                                 *
 *                  gyro.h                                          *
 *                  my_uart.h                                       *
 *                  imu.h                                           *
 *                  string.h                                        *
 *                  stdio.h                                         *
 *                  math.h                                          *
 *                                                                  *
 ********************************************************************
 *                                                                  *
 * Other Comments:                                                  *
 ********************************************************************/


#include "GenericTypeDefs.h"
#include "Compiler.h"
#include "HardwareProfile.h"
```

```
#include "Headers\microcontroller.h"
#include "Headers\main.h"
#include "Headers\timers.h"
#include "Headers\adc.h"
#include "Headers\iic.h"
#include "Headers\accelerometer.h"
#include "Headers\gyro.h"
#include "Headers\my_uart.h"
#include "Headers\imu.h"
#include <string.h>
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int byteTimingTest;

int main(void){
  int very_negative_ie, negative_ie, neutral_ie, positive_ie, very_positive_ie,
      very_negative_av, negative_av, neutral_av, positive_av, very_positive_av,
      very_negative_1, very_negative_2, very_negative_3, very_negative_4,
very_negative_5,
      very_negative_6, negative_1, negative_2, negative_3, negative_4, neutral_1,
neutral_2,
      neutral_3, neutral_4, neutral_5, positive_1, positive_2, positive_3, positive_4,
      very_positive_1, very_positive_2, very_positive_3, very_positive_4, very_positive_5,
      angular_vel, very_positive_6, very_negative, negative, neutral, positive, very_positive,
rate,
      sample_counter, error;
  double inclination_error, past_angle, angle;

  byteTimingTest=0;

  //Initalize System
  Microcontroller_init();

  //Initalize timers
  Timer_init();

  // LED init
  //mInitAllLEDs();

  //ADC setup
  ADC_init();
  EnableADC10();
```

```
//UART setup
uart_init();

//I2C Setup
I2C_init();

//Motor Control setup
mode(2);
accl_rate(5);

//Prepare for IMU estimation
setup();


double Ka=.48;
double Ki=12.0;
double zero=0.0;
int initalize_counter=0;
angle=0.0;
rate=0.0;

while(1){
  g_iWaitForInterrupt = 1;

        //Determination and scaling of inclination error and angular velocity
  //values of interest are between 0 and 255
  angle=Get_Inclination(RwEst[0], RwEst[1]);
  if(initalize_counter>=1000){
    if(initalize_counter==1000){
      zero=angle;
    }
    if(firstSample==1){
      past_angle=0.0;
      angular_vel=0.0;
    }else{
      angular_vel=(((angle-past_angle)*100)*Ka)+128;
    }
    past_angle=angle;
    inclination_error=((angle-zero)*Ki)+128;


          //-------------------------------------------------------------
          // Inclination Error
          //-------------------------------------------------------------
```

```
        //Very negative membership determination for inclination error
if(inclination_error<48){
          very_negative_ie=80;
        }else if(inclination_error>48 && inclination_error<88){
          very_negative_ie=(-2*inclination_error)+176;
        }else if(inclination_error>88){
          very_negative_ie=0;
        }

        //Negative membership determination for inclination error
        if(inclination_error<48 || inclination_error>128){
          negative_ie=0;
        }else if(inclination_error>48 && inclination_error<88){
          negative_ie=(2*inclination_error)-96;
        }else if(inclination_error>88 && inclination_error<128){
          negative_ie=(-2*inclination_error)+256;
        }

        //Neutral membership determination for inclination error
        if(inclination_error<88 || inclination_error>168){
          neutral_ie=0;
        }else if(inclination_error>88 && inclination_error<128){
          neutral_ie=(2*inclination_error)-176;
        }else if(inclination_error>128 && inclination_error<168){
          neutral_ie=(-2*inclination_error)+336;
        }

        //Positive membership determination for inclination error
        if(inclination_error<128 || inclination_error>208){
          positive_ie=0;
        }else if(inclination_error>128 && inclination_error<168){
          positive_ie=(2*inclination_error)-256;
        }else if(inclination_error>168 && inclination_error<208){
          positive_ie=(-2*inclination_error)+416;
        }

        //Very positive membership determination for inclination error
        if(inclination_error<168){
          very_positive_ie=0;
        }else if(inclination_error>168 && inclination_error<208){
          very_positive_ie=(2*inclination_error)-336;
        }else if(inclination_error>208){
          very_positive_ie=80;
        }
```

```
//------------------------------------------------------------
// Angular Velocity
//------------------------------------------------------------

//Very negative membership determination for angluar velocity
if(angular_vel<48){
  very_negative_av=80;
}else if(angular_vel>48 && angular_vel<88){
  very_negative_av=(-2*angular_vel)+176;
}else if(angular_vel>88){
  very_negative_av=0;
}

//Negative membership determination for inclination error
if(angular_vel<48 || angular_vel>128){
  negative_av=0;
}else if(angular_vel>48 && angular_vel<88){
  negative_av=(2*angular_vel)-96;
}else if(angular_vel>88 && angular_vel<128){
  negative_av=(-2*angular_vel)+256;
}

//Neutral membership determination for inclination error
if(angular_vel<88 || angular_vel>168){
  neutral_av=0;
}else if(angular_vel>88 && angular_vel<128){
  neutral_av=(2*angular_vel)-176;
}else if(angular_vel>128 && angular_vel<168){
  neutral_av=(-2*angular_vel)+336;
}

//Positive membership determination for inclination error
if(angular_vel<128 || angular_vel>208){
  positive_av=0;
}else if(angular_vel>128 && angular_vel<168){
  positive_av=(2*angular_vel)-256;
}else if(angular_vel>168 && angular_vel<208){
  positive_av=(-2*angular_vel)+416;
}

//Very positive membership determination for inclination error
if(angular_vel<168){
  very_positive_av=0;
```

```
}else if(angular_vel>168 && angular_vel<208){
  very_positive_av=(2*angular_vel)-336;
}else if(angular_vel>208){
  very_positive_av=80;
}

//------------------------------------------------------------
// Output membership determination
//------------------------------------------------------------

//Rule #1
if(very_negative_ie!=0 || very_negative_av!=0){
  very_negative_1=min(very_negative_ie, very_negative_av);
}else{
  very_negative_1=0;
}
//Rule #2
if(negative_ie!=0 || very_negative_av!=0){
  very_negative_2=min(negative_ie, very_negative_av);
}else{
  very_negative_2=0;
}
//Rule #3
if(neutral_ie!=0 || very_negative_av!=0){
  very_negative_3=min(neutral_ie, very_negative_av);
}else{
  very_negative_3=0;
}
//Rule #4
if(positive_ie!=0 || very_negative_av!=0){
  negative_1=min(positive_ie, very_negative_av);
}else{
  negative_1=0;
}
//Rule #5
if(very_positive_ie!=0 || very_negative_av!=0){
  neutral_1=min(very_positive_ie, very_negative_av);
}else{
  neutral_1=0;
}
//Rule #6
if(very_negative_ie!=0 || negative_av!=0){
  very_negative_4=min(very_negative_ie, negative_av);
}else{
  very_negative_4=0;
```

```
}
//Rule #7
if(negative_ie!=0 || negative_av!=0){
  very_negative_5=min(negative_ie, negative_av);
}else{
  very_negative_5=0;
}
//Rule #8
if(neutral_ie!=0 || negative_av!=0){
  negative_2=min(neutral_ie, negative_av);
}else{
  negative_2=0;
}
//Rule #9
if(positive_ie!=0 || negative_av!=0){
  neutral_2=min(positive_ie, negative_av);
}else{
  neutral_2=0;
}
//Rule #10
if(very_positive_ie!=0 || negative_av!=0){
  positive_1=min(very_positive_ie, negative_av);
}else{
  positive_1=0;
}
//Rule #11
if(very_negative_ie!=0 || neutral_av!=0){
  very_negative_6=min(very_negative_ie, neutral_av);
}else{
  very_negative_6=0;
}
//Rule #12
if(negative_ie!=0 || neutral_av!=0){
  negative_3=min(negative_ie, neutral_av);
}else{
  negative_3=0;
}
//Rule #13
if(neutral_ie!=0 || neutral_av!=0){
  neutral_3=min(neutral_ie, neutral_av);
}else{
  neutral_3=0;
}
//Rule #14
if(positive_ie!=0 || neutral_av!=0){
```

```
  positive_2=min(positive_ie, neutral_av);
}else{
  positive_2=0;
}
//Rule #15
if(very_positive_ie!=0 || neutral_av!=0){
  very_positive_1=min(very_positive_ie, neutral_av);
}else{
  very_positive_1=0;
}
//Rule #16
if(very_negative_ie!=0 || positive_av!=0){
  negative_4=min(very_negative_ie, positive_av);
}else{
  negative_4=0;
}
//Rule #17
if(negative_ie!=0 || positive_av!=0){
  neutral_4=min(negative_ie, positive_av);
}else{
  neutral_4=0;
}
//Rule #18
if(neutral_ie!=0 || positive_av!=0){
  positive_3=min(neutral_ie, positive_av);
}else{
  positive_3=0;
}
//Rule #19
if(positive_ie!=0 || positive_av!=0){
  very_positive_2=min(positive_ie, positive_av);
}else{
  very_positive_2=0;
}
//Rule #20
if(very_positive_ie!=0 || positive_av!=0){
  very_positive_3=min(very_positive_ie, positive_av);
}else{
  very_positive_3=0;
}
//Rule #21
if(very_negative_ie!=0 || very_positive_av!=0){
  neutral_5=min(very_negative_ie, very_positive_av);
}else{
  neutral_5=0;
```

```
        }
        //Rule #22
        if(negative_ie!=0 || very_positive_av!=0){
          positive_4=min(negative_ie, very_positive_av);
        }else{
          positive_4=0;
        }
        //Rule #23
        if(neutral_ie!=0 || very_positive_av!=0){
          very_positive_4=min(neutral_ie, very_positive_av);
        }else{
          very_positive_4=0;
        }
        //Rule #24
        if(positive_ie!=0 || very_positive_av!=0){
          very_positive_5=min(positive_ie, very_positive_av);
        }else{
          very_positive_5=0;
        }
        //Rule #25
        if(very_positive_ie!=0 || very_positive_av!=0){
          very_positive_6=min(very_positive_ie, very_positive_av);
        }else{
          very_positive_6=0;
        }

        //Combination of like rules
        very_negative=sqrt(pow(very_negative_1,2)+pow(very_negative_2,2)
                +pow(very_negative_3,2)+pow(very_negative_4,2)
                +pow(very_negative_5,2)+pow(very_negative_6,2));
        negative=sqrt(pow(negative_1,2)+pow(negative_2,2)
                +pow(negative_3,2)+pow(negative_4,2));
        neutral=sqrt(pow(neutral_1,2)+pow(neutral_2,2)
                +pow(neutral_3,2)+pow(neutral_4,2)+pow(neutral_5,2));
        positive=sqrt(pow(positive_1,2)+pow(positive_2,2)
                +pow(positive_3,2)+pow(positive_4,2));
        very_positive=sqrt(pow(very_positive_1,2)+pow(very_positive_2,2)
                +pow(very_positive_3,2)+pow(very_positive_4,2)
                +pow(very_positive_5,2)+pow(very_positive_6,2));

        //Determination of the drive signal for the motors
        int num =
((very_negative*48)+(negative*88)+(neutral*128)+(positive*168)+(very_positive*208));
        int den = (very_negative+negative+neutral+positive+very_positive);
        rate= num/den;
```

```
      if(initalize_counter>1100){
                drive(rate);
              }
    }
          Estimate_Components();


    printf("%1.2f",angle);
    printf(",");
    printf("%1.2f",angular_vel);
    printf(",");
    printf("%1.2f",inclination_error);
    printf(",");
    printf("%d",initalize_counter);
    printf(",");
    printf("%1.2f\n\r",rate);

    sample_counter=0;

    if(initalize_counter<=1200){
      initalize_counter++;
    }

    do{
      sample_counter++;
            if( byteTimingTest == 0 ){
                    if( g_iWaitForInterrupt == 0 ){
                              error=1;
                            }
                            byteTimingTest = 1;
                  }
          }while( g_iWaitForInterrupt == 1 );
  }
}
```

```
/*****************************************************************************
 *                                                                           *
 * This file contains the implementation of accelerometer.h for the    *
 * PIC32 device.                                                        *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Author:        Kyle Reid                                            *
 *   Filename:      accelerometer.c                                      *
 *   Date:          8/3/2010                                             *
 *   File Version:  1.00                                                 *
 *   Other Files Required: adc.h                                         *
 *                  accelerometer.h                                      *
 *                  math.h                                               *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 * Other Comments:                                                       *
 *****************************************************************************/

#include "Headers\Accelerometer.h"
#include "Headers\ADC.h"
#include <math.h>

#define ADC_ACC_X 0     //w = x-axis
#define ADC_ACC_Y 1     //w = y-axis
#define VREF 3.6        //Reference voltage for the ADC
#define SENSITIVITY 1.0  //Sensitivity of accelerometer in V/g
#define VZEROG 2.6      //Voltage the acclerometer displays at 0g
#define PI 3.14159265

/*
 * Used for debug has same function as adc_read(0 or 1)
 */
int Get_AdcRw(int axis){
  return adc_read(axis);
}

double Get_RwAcc(int axis){
  if(axis==0){
    return -(adc_read(axis) * VREF / 1023 - VZEROG) / SENSITIVITY;
  }else{
    return (adc_read(axis) * VREF / 1023 - VZEROG) / SENSITIVITY;
  }
}
```

```
/*****************************************************************************
 *                                                                           *
 * This file contains the implementation of adc.h for the PIC32              *
 * device.                                                                   *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Author:       Kyle Reid                                                 *
 *   Filename:     adc.c                                                     *
 *   Date:         8/3/2010                                                  *
 *   File Version:  1.00                                                     *
 *   Other Files Required: adc.h                                             *
 *                  timers.h                                                 *
 *                  HardwareProfile.h                                        *
 *                  plib.h                                                   *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 * Other Comments:                                                           *
 *****************************************************************************/


#include "Headers\ADC.h"
#include "Headers\Timers.h"
#include "HardwareProfile.h"
#include <plib.h>

/*****************************************************************/
/* PARAM1:  Turns on ADC module, Formats the ouput as a 16bit   */
/*        integer, uses the a timer to trigger conversion in  */
/*        the case of this project timer 3 is used for 10ms   */
/*        sample period and a sample freq of 100hz, turns on  */
/*        auto sampling.                                      */
/* PARAM2:  Sets the reference voltage to AVDD=3.3V and       */
/*        AVSS=GND, Disables offset calibration, scan muxed   */
/*        inputs and convert multiple inputs, Do not use      */
/*        alternate buffer, do not use alternate input,       */
/*        take 3 samples per interrupt                        */
/* PARAM3:  Skips the scanning of all channels except for     */
/*        channels 0,1 and 2                                  */
/*****************************************************************/


#define PARAM1 ADC_MODULE_ON | ADC_FORMAT_INTG16 | ADC_CLK_TMR | \
  ADC_AUTO_SAMPLING_ON
```

```
#define PARAM2 ADC_VREF_AVDD_AVSS | ADC_OFFSET_CAL_DISABLE | \
  ADC_SCAN_ON | ADC_ALT_BUF_OFF | ADC_ALT_INPUT_OFF | \
  ADC_SAMPLES_PER_INT_3
#define PARAM3 ADC_CONV_CLK_INTERNAL_RC | ADC_SAMPLE_TIME_12
#define PARAM4 ENABLE_AN0_ANA | ENABLE_AN1_ANA | ENABLE_AN2_ANA
#define PARAM5 SKIP_SCAN_AN3 | SKIP_SCAN_AN4 | \
  SKIP_SCAN_AN5 | SKIP_SCAN_AN6 | SKIP_SCAN_AN7 | \
  SKIP_SCAN_AN8 | SKIP_SCAN_AN9 | SKIP_SCAN_AN10 | SKIP_SCAN_AN11 |\
  SKIP_SCAN_AN12 | SKIP_SCAN_AN13 | SKIP_SCAN_AN14 | SKIP_SCAN_AN15

//Buffer to store value converted in the ADC
unsigned int adc_buf[16];

//Temp buffers used for averaging
unsigned int adc_buf1[3];
unsigned int adc_buf2[3];

unsigned int current_buf = 1;
double dt;

void ADC_init(void){

  //Ensure there isn't another ADC open
  CloseADC10();

  //Set channel 0 as negetive reference
  SetChanADC10( ADC_CH0_NEG_SAMPLEA_NVREF);

  //Open ADC with above parameters
  OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5);

  //Configure interrupt
  ConfigIntADC10(ADC_INT_PRI_2 | ADC_INT_ON);
}

void __ISR(_ADC_VECTOR, ipl2) _ADCHandler(void){
  dt=timer_dt();
  unsigned int buffer = 0;

  // clear the interrupt flag
  mAD1ClearIntFlag();

  //Average over 2 samples
  if (current_buf == 1){
          for (buffer=0; buffer < 3; buffer++){
```

```
        adc_buf1[buffer] = ReadADC10(buffer);
      }
      current_buf = 2;
    }
    else{
      for (buffer=0; buffer < 3; buffer++){
        adc_buf2[buffer] = ReadADC10(buffer);
      }
      current_buf = 1;
    }
    for (buffer=0; buffer < 6; buffer++){
      adc_buf[buffer] = (adc_buf1[buffer]/2 + adc_buf2[buffer]/2);
    }
}
```

```
/************************************************************************
 *                                      *
 * This file contains the implementation of gyro.h for the PIC32      *
 * device.                                  *
 *                                      *
 ************************************************************************
 *                                      *
 *   Author:        Kyle Reid                   *
 *   Filename:      gyro.c                      *
 *   Date:          8/3/2010                    *
 *   File Version:  1.00                        *
 *   Other Files Required: adc.h                    *
 *                  gyro.h                  *
 *                                      *
 ************************************************************************
 *                                      *
 * Other Comments:                              *
 ************************************************************************/

#include "Headers\Gyro.h"
#include "Headers\ADC.h"

#define ADC_GYRO 2
#define GVREF 3.6
#define GSENSITIVITY 0.0125
#define VZERORATE 2.59

/*
 * Used for debug has same function as adc_read(2)
 */
int Get_AdcGyro(void){
  return adc_read(ADC_GYRO);
}


double Get_RateAyr(void){
  return (adc_read(ADC_GYRO) * GVREF / 1023 - VZERORATE)/GSENSITIVITY;
}
```

```
/*****************************************************************************
 *                                                                           *
 * This file contains the implementation of iic.h for the PIC32              *
 * device.                                                                   *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Author:        Kyle Reid                                                *
 *   Filename:      iic.c                                                    *
 *   Date:          8/3/2010                                                 *
 *   File Version:  1.00                                                     *
 *   Other Files Required: iic.h                                             *
 *                   plib.h                                                  *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 * Other Comments:                                                           *
 *****************************************************************************/

#include "Headers\iic.h"
#include <plib.h>

#define BRG_VAL 498                 //Value used for baud rate
#define IC2PARAM I2C_ON | I2C_7BIT_ADD  //Turns on IC2 and uses 7 bit address

char i2cData[10];
int DataSz, Index;


void i2c_wait(unsigned int cnt)
{
  while(--cnt)
  {
    Nop();
    Nop();
  }
}

void I2C_init(void)
{

  OpenI2C1(IC2PARAM , BRG_VAL);
}

void mode(int mode)
```

```
{

  i2cData[0] = 0xB0;//Device Address and WR Command
  i2cData[1] = 15;//mode register
  i2cData[2] = mode;//data to write
  DataSz = 3;

  StartI2C1();//Send the Start Bit
  IdleI2C1();//Wait to complete
  int Index = 0;
  while( DataSz )
  {
    MasterWriteI2C1( i2cData[Index++] );
    IdleI2C1();//Wait to complete
    DataSz--;
    //ACKSTAT is 0 when slave acknowledge. if 1 then slave has not acknowledge
    //the data.
    if( I2C1STATbits.ACKSTAT )
    break;
  }
  StopI2C1();//Send the Stop condition
  IdleI2C1();//Wait to complete
  // wait to complete write process. poll the ack status
  while(1)
  {
    i2c_wait(10);
    StartI2C1();//Send the Start Bit
    IdleI2C1();//Wait to complete
    MasterWriteI2C1( i2cData[0] );
    IdleI2C1();//Wait to complete
    if( I2C1STATbits.ACKSTAT == 0 )//acknowledged
    {
       StopI2C1();//Send the Stop condition
       IdleI2C1();//Wait to complete
       break;
    }
    StopI2C1();//Send the Stop condition
    IdleI2C1();//Wait to complete
  }
}

void drive(int speed)
{

  i2cData[0] = 0xB0;//Device Address and WR Command
```

```
  i2cData[1] = 0;//speed register
  i2cData[2] = speed;//data to write
  DataSz = 3;

  StartI2C1();//Send the Start Bit
  IdleI2C1();//Wait to complete
  int Index = 0;
  while( DataSz )
  {
    MasterWriteI2C1( i2cData[Index++] );
    IdleI2C1();//Wait to complete
    DataSz--;
    //ACKSTAT is 0 when slave acknowledge. if 1 then slave has not acknowledge
    //the data.
    if( I2C1STATbits.ACKSTAT )
    break;
  }
  StopI2C1();//Send the Stop condition
  IdleI2C1();//Wait to complete
  // wait to complete write process. poll the ack status
  while(1)
  {
    i2c_wait(10);
    StartI2C1();//Send the Start Bit
    IdleI2C1();//Wait to complete
    MasterWriteI2C1( i2cData[0] );
    IdleI2C1();//Wait to complete
    if( I2C1STATbits.ACKSTAT == 0 )//acknowledged
    {
       StopI2C1();//Send the Stop condition
       IdleI2C1();//Wait to complete
       break;
    }
    StopI2C1();//Send the Stop condition
    IdleI2C1();//Wait to complete
  }
}

void accl_rate(int rate)
{

  i2cData[0] = 0xB0;//Device Address and WR Command
  i2cData[1] = 14;//acceleration rate register
  i2cData[2] = rate;//data to write
  DataSz = 3;
```

```
  StartI2C1();//Send the Start Bit
  IdleI2C1();//Wait to complete
  int Index = 0;
  while( DataSz )
  {
    MasterWriteI2C1( i2cData[Index++] );
    IdleI2C1();//Wait to complete
    DataSz--;
    //ACKSTAT is 0 when slave acknowledge. if 1 then slave has not acknowledge
    //the data.
    if( I2C1STATbits.ACKSTAT )
    break;
  }
  StopI2C1();//Send the Stop condition
  IdleI2C1();//Wait to complete
  // wait to complete write process. poll the ack status
  while(1)
  {
    i2c_wait(10);
    StartI2C1();//Send the Start Bit
    IdleI2C1();//Wait to complete
    MasterWriteI2C1( i2cData[0] );
    IdleI2C1();//Wait to complete
    if( I2C1STATbits.ACKSTAT == 0 )//acknowledged
    {
       StopI2C1();//Send the Stop condition
       IdleI2C1();//Wait to complete
       break;
    }
    StopI2C1();//Send the Stop condition
    IdleI2C1();//Wait to complete
  }
}

void command(int command)
{

  i2cData[0] = 0xB0;//Device Address and WR Command
  i2cData[1] = 16;//acceleration rate register
  i2cData[2] = command;//data to write
  DataSz = 3;

  StartI2C1();//Send the Start Bit
  IdleI2C1();//Wait to complete
```

```
  int Index = 0;
  while( DataSz )
  {
    MasterWriteI2C1( i2cData[Index++] );
    IdleI2C1();//Wait to complete
    DataSz--;
    //ACKSTAT is 0 when slave acknowledge. if 1 then slave has not acknowledge
    //the data.
    if( I2C1STATbits.ACKSTAT )
    break;
  }
  StopI2C1();//Send the Stop condition
  IdleI2C1();//Wait to complete
  // wait to complete write process. poll the ack status
  while(1)
  {
    i2c_wait(10);
    StartI2C1();//Send the Start Bit
    IdleI2C1();//Wait to complete
    MasterWriteI2C1( i2cData[0] );
    IdleI2C1();//Wait to complete
    if( I2C1STATbits.ACKSTAT == 0 )//acknowledged
    {
      StopI2C1();//Send the Stop condition
      IdleI2C1();//Wait to complete
      break;
    }
    StopI2C1();//Send the Stop condition
    IdleI2C1();//Wait to complete
  }
}
```

```
/*********************************************************************
 *                                                                   *
 * This file contains the implementation of imu.h for the PIC32      *
 * device.                                                           *
 *                                                                   *
 *********************************************************************
 *                                                                   *
 *   Author:       Kyle Reid                                         *
 *   Filename:     imu.c                                             *
 *   Date:         8/3/2010                                          *
 *   File Version:  1.00                                             *
 *   Other Files Required: adc.h                                     *
 *                  imu.h                                            *
 *                  accelerometer.h                                  *
 *                  math.h                                           *
 *                  gyro.h                                           *
 *                                                                   *
 *********************************************************************
 *                                                                   *
 * Other Comments:                                                   *
 *********************************************************************/

#include "Headers\IMU.h"
#include "Headers\adc.h"
#include "Headers\accelerometer.h"
#include "Headers\gyro.h"
#include "Headers\timers.h"
#include <math.h>

#define PI 3.14159265


int wGyro;          //Smoothing factor for the gyro measurement


double Ayr;         //Angle between R and the Y axis (deg)

void setup(void){
  firstSample=1;
  wGyro=75;
}

void Estimate_Components(void){
  static int i,w;
  double tmpf;
```

```
  //get accelerometer readings in g, gives us RwAcc vector
  for(i=0;i<2;i++){
    RwAcc[i]=Get_RwAcc(i);
  }

  //normalize vector
  normalize_Vector(RwAcc);


  if (firstSample){
    for(w=0;w<2;w++){
      RwEst[w] = RwAcc[w];    //initialize with accelerometer readings
    }
  }else{
    //evaluate RwGyro vector
    if(fabs(RwEst[1]) < 0.1){
      //Ry is too small and because it is used as reference for computing Axy
      //it's error fluctuations will amplify leading to bad results
      //in this case skip the gyro data and just use previous estimate
      for(w=0;w<2;w++){
        RwGyro[w] = RwEst[w];    //Skip Gyro update results invalid
      }
    }else{
      //get angles between projection of R on ZX/ZY plane and Z axis, based on last RwEst
      tmpf = Get_RateAyr();                    //get current gyro rate in deg/s
      tmpf *= dt;                              //get angle change in deg
      Ayr = atan(RwEst[0]/RwEst[1]) * 180 / PI;     //get angle and convert to degrees
      Ayr += tmpf;                             //get updated angle according to gyro movement

      //reverse calculation of RwGyro from Awz angles
      RwGyro[0] = tan(Ayr * PI/180);
      RwGyro[0] /= sqrt( 1 + pow(tan(Ayr * PI/180),2));
      RwGyro[1] = sqrt(1-pow(RwGyro[0],2));
    }

    //combine Accelerometer and gyro readings
    for(w=0;w<2;w++){
      RwEst[w] = (RwAcc[w] + wGyro* RwGyro[w]) / (1 + wGyro);
    }
    normalize_Vector(RwEst);
  }
  firstSample = 0;
}
```

```
void normalize_Vector(double* vector){
  static double R;

  R = sqrt(pow(vector[0],2) + pow(vector[1],2));
  vector[0] /= R;
  vector[1] /= R;
}

double Get_Inclination(double Rx, double Ry){
  return (atan(Rx/Ry) * (180.00 / PI));
}
```

```
/*****************************************************************
 *                                       *
 * This file contains the implementation of my_uart.h for the PIC32    *
 * device.                                   *
 *                                       *
 *****************************************************************
 *                                       *
 *   Author:       Kyle Reid                   *
 *   Filename:      my_uart.c                     *
 *   Date:         8/3/2010                    *
 *   File Version:   1.00                      *
 *   Other Files Required: my_uart.h                   *
 *                  microcontroller.h              *
 *                  HardwareProfile.h                *
 *                  plib.h                 *
 *                                       *
 *****************************************************************
* Other Comments:                               *
 *****************************************************************/

#include <plib.h>
#include "Headers\my_uart.h"
#include "HardwareProfile.h"
#include "Headers\microcontroller.h"

/*****************************************************************/
/* PARAM1:  Turns on UART module, UART does not power down when */
/*        idle, UART can be used for TX and Rx, set up for    */
/*        no parity and 8 bit data TX and RX, setup for 1 stop*/
/*        bit, set for simplex mode                  */
/* PARAM2:  Sets TX pin as active low, enables transmission,    */
/*        enables reception                  */
/*****************************************************************/
#define PARAM1 UART_EN | UART_IDLE_CON | UART_RX_TX | \
  UART_NO_PAR_8BIT | UART_1STOPBIT | UART_MODE_SIMPLEX |\
  UART_DIS_BCLK_CTS_RTS | UART_NORMAL_RX | UART_BRGH_SIXTEEN
#define PARAM2 UART_TX_PIN_LOW | UART_TX_ENABLE | \
  UART_RX_ENABLE

#define DESIRED_BAUDRATE     (57600)     // The desired BaudRate

void uart_init(void){
        OpenUART2(PARAM1, PARAM2, pbClk/16/DESIRED_BAUDRATE-1);        // calculate
actual BAUD generate value.
}
```

```
/*****************************************************************************
 *                                      *
 * This file contains the implementation of timers.h for the PIC32     *
 * device.  Timer1 is used for determining the time between samples    *
 * for filtering.  Timer2 is used for creating milisecond delays.      *
 * Timer3 is used to create a 10ms sampling period for the ADC         *
 * Timer4 is used for creating microsecond delays.                     *
 *                                      *
 *****************************************************************************
 *                                      *
 *   Author:        Kyle Reid                      *
 *   Filename:      timers.c                        *
 *   Date:          8/3/2010                       *
 *   File Version:  1.00                           *
 *   Other Files Required: timers.h                        *
 *                  plib.h                    *
 *                  Microcontroller.h
 *                                      *
 *****************************************************************************
 *                                      *
 * Other Comments:                                  *
 *****************************************************************************/

#include "Headers\Timers.h"
#include <plib.h>
#include "Headers\Microcontroller.h"

//10ms sample period for adc
#define SAMPLE_PERIOD 33333

//Clock ticks required to make one ms
#define MS_TICK 20000
//Clock ticks required to make one us
#define US_TICK 20

int g_iWaitForInterrupt;
static volatile int ms_delay, us_delay;
static const double ticks_per_second = (double)SYS_FREQ/64.0;

/*
 * Timer 1: Turn on, internal clock source, prescale 64, period set well
 * above sample period
 * Timer 2: Turn on, internal clock source, prescale 4, period set to one
 * milisecond
 * Timer 3: Turn on, internal clock source, prescale 8, period set to one
```

```
 * sample period
 * Timer 4: Turn on, internal clock source, prescale 4, period set to one
 * microsecond
 */

void Timer_init(void){
  OpenCoreTimer(800000);
  mConfigIntCoreTimer(CT_INT_ON | CT_INT_PRIOR_2);
  OpenTimer1(T1_ON | T1_SOURCE_INT | T1_PS_1_64, 65536);
  OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_4, MS_TICK);
  ConfigIntTimer2(T2_INT_ON | T2_INT_PRIOR_1);
  OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_8, SAMPLE_PERIOD);
  EnableIntT3;
  OpenTimer4(T4_ON | T4_SOURCE_INT | T4_PS_1_4, US_TICK);
  ConfigIntTimer4(T4_INT_ON | T4_INT_PRIOR_1);
}

/*
 * Resets TMR1 to zero when called
 */
void timer_reset(void){
  T1CONbits.ON = 0;    //Turn counter off
  TMR1 = 0;           //Reset counter
  T1CONbits.ON = 1;    //Turn counter on
}

/*
 * Returns the time in seconds since the last call
 * of this function
 */
double timer_dt(void){
  double ticks = (double)TMR1;
  timer_reset();
  return ticks/ticks_per_second;
}

/*
 * Function used to dealy by "ms" miliseconds
 */
void delay_ms(int ms){
  ms_delay=ms;
  while(ms_delay>0){
    ;
  }
}
```

```
/*
 * Function used to dealy by "us" microseconds
 */
void delay_us(int us){
  us_delay=us;
  while(us_delay>0){
    ;
  }
}

void __ISR(_CORE_TIMER_VECTOR, ipl2) _CoreTimerHandler(void){
  g_iWaitForInterrupt = 0;
  // clear the interrupt flag
  mCTClearIntFlag();
  // update timer period
  UpdateCoreTimer(800000);
  //toggle interrupt wait flag

}


void __ISR(_TIMER_2_VECTOR, ipl1) _Timer2Handler(void){

  // clear the interrupt flag
  mT2ClearIntFlag();
  //count down miliseconds
  ms_delay--;
}

void __ISR(_TIMER_4_VECTOR, ipl1) _Timer4Handler(void){

  // clear the interrupt flag
  mT4ClearIntFlag();
  //count down microseconds
  us_delay--;
}
```

B. Technical Documentatoin

Technical documentation is included below

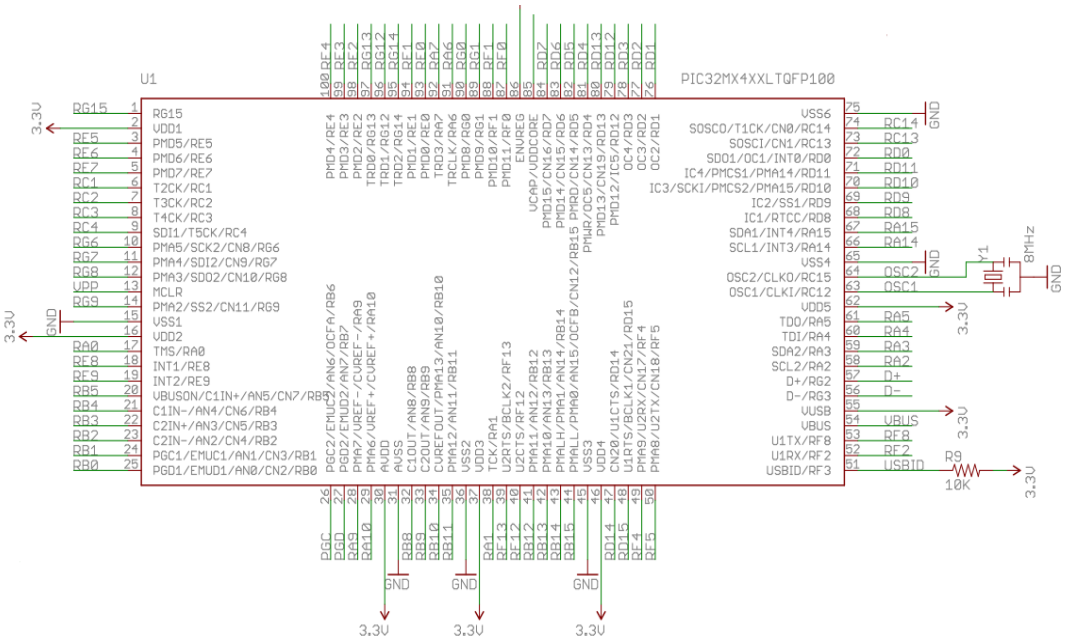

Figure 10: PIC microcontroller pin out

MD25 - Dual 12Volt 2.8Amp H Bridge Motor Drive

I2C mode documentation
(Click here for Serial Mode)

Automatic Speed regulation
By using feedback from the encoders the MD25 is able to dynamically increase power as required. If the required speed is not being achieved, the MD25 will increase power to the motors until it reaches the desired rate or the motors reach there maximum output. Speed regulation can be turned off in the command register.

Automatic Motor Timeout
The MD25 will automatically stop the motors if there is no I2C communications within 2 seconds. This is to prevent your robot running wild if the controller fails. The feature can be turned off, if not required. See the command register.

Controlling the MD25
The MD25 is designed to operate in a standard I2C bus system on addresses from 0xB0 to 0xBE (last bit of address is read/write bit, so even numbers only), with its default address being 0xB0. This is easily changed by removing the Address Jumper or in the software see Changing the I2C Bus Address.

I2C mode allows the MD25 to be connected to popular controllers such as the PICAXE, OOPic and BS2p, and a wide range of micro-controllers like PIC's, AVR's, 8051's etc.
I2C communication protocol with the MD25 module is the same as popular EPROM's such as the 24C04. To read one or more of the MD25 registers, first send a start bit, the module address (0XB0 for example) with the read/write bit low, then the register number you wish to read. This is followed by a repeated start and the module address again with the read/write bit high (0XB1 in this example). You are now able to read one or more registers. The MD25 has 17 registers numbered 0 to 16 as follows;

Register	Name	Read/Write	Description
0	Speed1	R/W	Motor1 speed (mode 0,1) or speed (mode 2,3)
1	Speed2/Turn	R/W	Motor2 speed (mode 0,1) or turn (mode 2,3)
2	Enc1a	Read only	Encoder 1 position, 1st byte (highest), capture count when read
3	Enc1b	Read only	Encoder 1 position, 2nd byte
4	Enc1c	Read only	Encoder 1 position, 3rd byte
5	Enc1d	Read only	Encoder 1 position, 4th (lowest byte)
6	Enc2a	Read only	Encoder 2 position, 1st byte (highest), capture count when read
7	Enc2b	Read only	Encoder 2 position, 2nd byte
8	Enc2c	Read only	Encoder 2 position, 3rd byte
9	Enc2d	Read only	Encoder 2 position, 4th byte (lowest byte)
10	Battery volts	Read only	The supply battery voltage
11	Motor 1 current	Read only	The current through motor 1
12	Motor 2 current	Read only	The current through motor 2
13	Software Revision	Read only	Software Revision Number
14	Acceleration rate	R/W	Optional Acceleration register
15	Mode	R/W	Mode of operation (see below)
16	Command	R/W	Used for reset of encoder counts and module address changes

Speed1 Register
Depending on what mode you are in, this register can affect the speed of one motor or both motors. If you are in mode 0 or 1 it will set the speed and direction of motor 1. The larger the number written to this register, the more power is applied to the motor. A mode of 2 or 3 will control the speed and direction of both motors (subject to effect of turn register).

Speed2/Turn Register
When in mode 0 or 1 this register operates the speed and direction of motor 2. When in mode 2 or 3 Speed2 becomes a Turn register, and any value in this register is combined with the contents of Speed1 to steer the device (see below).

Turn mode
Turn mode looks at the speed register to decide if the direction is forward or reverse. Then it applies a subtraction or addition of the turn value on either motor.

so if the direction is forward
motor speed1 = speed - turn
motor speed2 = speed + turn

else the direction is reverse so
motor speed1 = speed + turn
motor speed2 = speed - turn

If the either motor is not able to achieve the required speed for the turn (beyond the maximum output), then the other motor is automatically changed by the program to meet the required difference.

Encoder registers
Each motor has its encoder count stored in an array of four bytes, together the bytes form a signed 32 bit number, the encoder count is captured on a read of the highest byte (registers 2, 6) and the subsequent lower bytes will be held until another read of the highest byte takes place. The count is stored with the highest byte in the lowest numbered register. The registers can be zeroed at any time by writing 32 (0x20) to the command register.

Battery volts
A reading of the voltage of the connected battery is available in this register. It reads as 10 times the voltage (121 for 12.1v).

Motor 1 and 2 current
A guide reading of the average current through the motor is available in this register. It reads approx ten times the number of Amps (25 at 2.5A).

Software Revision number
This register contains the revision number of the software in the modules PIC16F873 controller - currently 1 at the time of writing.

Acceleration Rate
If you require a controlled acceleration period for the attached motors to reach there ultimate speed, the MD25 has a register to provide this. It works by using a value into the acceleration register and incrementing the power by that value. Changing between the current speed of the motors and the new speed (from speed 1 and 2 registers). So if the motors were traveling at full speed in the forward direction (255) and were instructed to move at full speed in reverse (0), there would be 255 steps with an acceleration register value of 1, but 128 for a value of 2. The default acceleration value is 5, meaning the speed is changed from full forward to full reverse in 1.25 seconds. The register will accept values of 1 up to 10 which equates to a period of only 0.65 seconds to travel from full speed in one direction to full speed in the opposite direction.

So to calculate the time (in seconds) for the acceleration to complete :

if new speed > current speed
steps = (new speed - current speed) / acceleration register

if new speed < current speed
steps = (current speed - new speed) / acceleration register

time = steps * 25ms

For example :

Acceleration register	Time/step	Current speed	New speed	Steps	Acceleration time
1	25ms	0	255	255	6.375s
2	25ms	127	255	64	1.6s
3	25ms	80	0	27	0.675s
5 (default)	25ms	0	255	51	1.275s
10	25ms	255	0	26	0.65s

Mode Register

The mode register selects which mode of operation and I2C data input type the user requires. The options being:

0, (Default Setting) If a value of 0 is written to the mode register then the meaning of the speed registers is literal speeds in the range of 0 (Full Reverse) 128 (Stop) 255 (Full Forward).

1, Mode 1 is similar to Mode 0, except that the speed registers are interpreted as signed values. The meaning of the speed registers is literal speeds in the range of -128 (Full Reverse) 0 (Stop) 127 (Full Forward).

2, Writing a value of 2 to the mode register will make speed1 control both motors speed, and speed2 becomes the turn value.
Data is in the range of 0 (Full Reverse) 128 (Stop) 255 (Full Forward).

3, Mode 3 is similar to Mode 2, except that the speed registers are interpreted as signed values.
Data is in the range of -128 (Full Reverse) 0 (Stop) 127 (Full Forward)

Command register

Command		Action
Dec	Hex	
32	20	Resets the encoder registers to zero
48	30	Disables automatic speed regulation
49	31	Enables automatic speed regulation (default)
50	32	Disables 2 second timeout of motors (Version 2 onwards only)
51	33	Enables 2 second timeout of motors when no I2C comms (default) (Version 2 onwards only)
160	A0	1st in sequence to change I2C address
170	AA	2nd in sequence to change I2C address
165	A5	3rd in sequence to change I2C address

Changing the I2C Bus Address

To change the I2C address of the MD25 by writing a new address you must have only one module on the bus. Write the 3 sequence commands in the correct order followed by the address. Example; to change the address of an MD25 currently at 0xB0 (the default shipped address) to 0xB4, write the following to address 0xB0; (0xA0, 0xAA, 0xA5, 0xB4). These commands must be sent in the correct sequence to change the I2C address, additionally, no other command may be issued in the middle of the sequence. The sequence must be sent to the command register at location 16, which means 4 separate write transactions on the I2C bus. Because of the way the MD25 works internally, there MUST be a delay of at least 5mS between the writing of each of these 4 transactions. When done, you should label the MD25 with its address, however if you do forget, just power it up without sending any commands. The MD25 will flash its address out on the green communication LED. One long flash followed by a number of shorter flashes indicating its address. Any command sent to the MD25 during this period will still be received and writing new speeds or a write to the command register will terminate the flashing.

Address		Long Flash	Short Flashes
Decimal	Hex		
176	B0	1	0
178	B2	1	1
180	B4	1	2
182	B6	1	3
184	B8	1	4
186	BA	1	5
188	BC	1	6
190	BE	1	7

Take care not to set more than one MD25 to the same address, there will be a bus collision and very unpredictable results.