“Buzzed” the Balancing Robot

Formal Report

Department of Electrical and Computer Engineering
EEL5666
Intelligent Machine Design Laboratory

Prepared by: Jairo Andres Vargas
Instructors: A. Antonio Arroyo PhD
Eric M. Schwartz PhD
TAs: Mike Pridgen
Adam Barnett
Sara Keen

Table of Contents

Opening

Abstract

"Buzzed" the balancing robot is a robot that stands itself up and balances autonomously. The robot incorporates three wheels for the standing sequence and then balances on the two main wheels. The robot has a large robust platform that has good inertial properties in order for good balancing characteristics.

The accelerometer and the gyroscope are inertial sensors used to relay the robot its position in balancing. A microprocessor will filter and process the data for balancing and relate movement to motor controllers to correct the error.

Executive Summary

"Buzzed" balancing robot is an automous robot capable of balancing from its main drive wheels. The robot will start lying down and roll to a wall, roll the top wheel up the wall to stand itself up and then push off a small amount and balance.

"Buzzed" has the ability to run from two user defined start points to either start from the lying own position or from a standing position to balance for tuning the balancing constants. The use of a PD loop as well as filtering will be used to maintain the robot in a balancing state.

Introduction

Humans have the ability to stand upright in a constant and steady act of balancing. It is this easy that causes the physical act of balancing to be overlooked as something simple and unimportant. My background as a gymnastics coach has also cause me to take balancing for granted due to the fact that I expect that to be part of the self control needed to do some of the many skills in the sport. Watching younger children still battle with the control over their balancing made me wonder what is necessary to make a robot to balance.

Anatomically humans have sensors in the inner ear that are extremely sensitive to movement in all directions. Heavy fluid sits inside of the cochlea, vestibule and three inner ear canals. The vestibular system, including the posterior, superior, and horizontal canals, has tiny little hairs that are subject to motion when the fluids move around them due to a change in movement. This motion of the hairs is turned into electrical signals that relate motion to the brain. (inner ear, Wikipedia)

The act of balancing has been tackled by several different robots in unique ways. The segway is one of the popular and commercial versions of the balancing robot that people use to move around in. Fully autonomous robots however have come in all shapes and sizes.

The problem is tackled using the inverted pendulum problem, where the center of mass is outstretched and high up from the base. The pendulum can be either attached to the ground or it can be on a moving platform that stays underneath the mass. Dynamics and kinematics are used to solve for a mathematical representation of the motion of the robot. My controls TA Josue Munoz gave me a lecture on the problem of an inverted pendulum as shown in the appendix.

The simple balancing robot is tall and removed from the bottom drive assembly. It looks much like an upside down broom with wheels on the bottom. In this case however the act of balancing is easier to recreate because of the isolation of the mass up high from the robot where the inertial position is calculated using a tilt sensor and a gyroscope. When the motors correct the position, they just change the orientation of the robot rather then pushing the mass out of the way. This robot however is just a robot that does the upright balancing

Other robots out there like Botka the balancing robot and Robo-erectus are actually robot platforms that once they are placed upright they stay in those positions even while running into things and pushing into objects. These scenarios are different because the center of mass is much closer to the drive wheels and thus causing the movement of the motors to be much more effective on moving the robot rather than correcting for position. The use of both accelerometers and gyroscopes in each application with strong filtering to compensate for noise and misc. impacts are used to process inertial position. The application of the upright platform is much more useful as a robot that can do other tasks as demonstrated by Botka and Robo-erectus. (Hackaday.com, Cognitoresearch.com)

This paper will outline the system and the components that went together in creating "Buzzed" the balancing robot.

Main Body

Integrated System

"Buzzed" has several different components working together to accomplish the task of balancing. The robot takes in data form the IMU, processes it and tells the motor controllers how much to move to maintain upright position. The heart of the system is the Atmega128 that's incorporated on the Mavric II-B. The flow chart in Figure 1 expresses the physical components of "Buzzed."

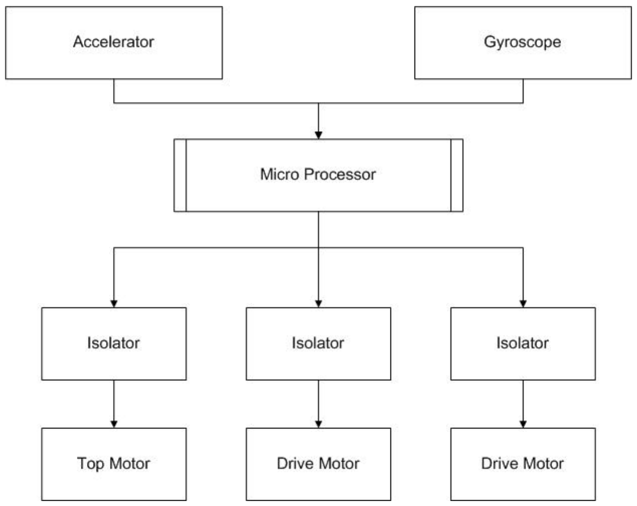

Figure 1 Balancing Robot Physical Implementation

Buzzed will read MEMS analog sensor outputs on an interrupt basis to filter the data from any motion not related to inertial position and run a PD control loop that controls the motors to keep the robot upright. This setup is capable of maintaining an upright position and with the aid of the top wheel, getting itself upright.

Mobile Platform

The main objective of Buzzed is to get balance and get himself upright from a 90^o wall. Other lesser design specifications were carried out to meet the main objectives and are described as follows:

- Mounting for:
  - Main drive transmission and wheel assembly
  - Top wheel assembly
  - Mavric IIb board
  - Accelerometer in the center of gravity of the robot
  - Gyroscope on the axis of rotation of the main drive wheels
  - LCD screen
  - Motor controllers near the motor they control
  - Battery for main drive motors, top motors and Mavric board.

  All mounted with relative easy of access.
- Everything must be protected from the impact cause from falling over by being inside of the robot.
- High center of gravity for better physical control of upright orientation.
- Ease of access to JTAG programming terminal without disassembly
- Rigid and balanced longitudinally.

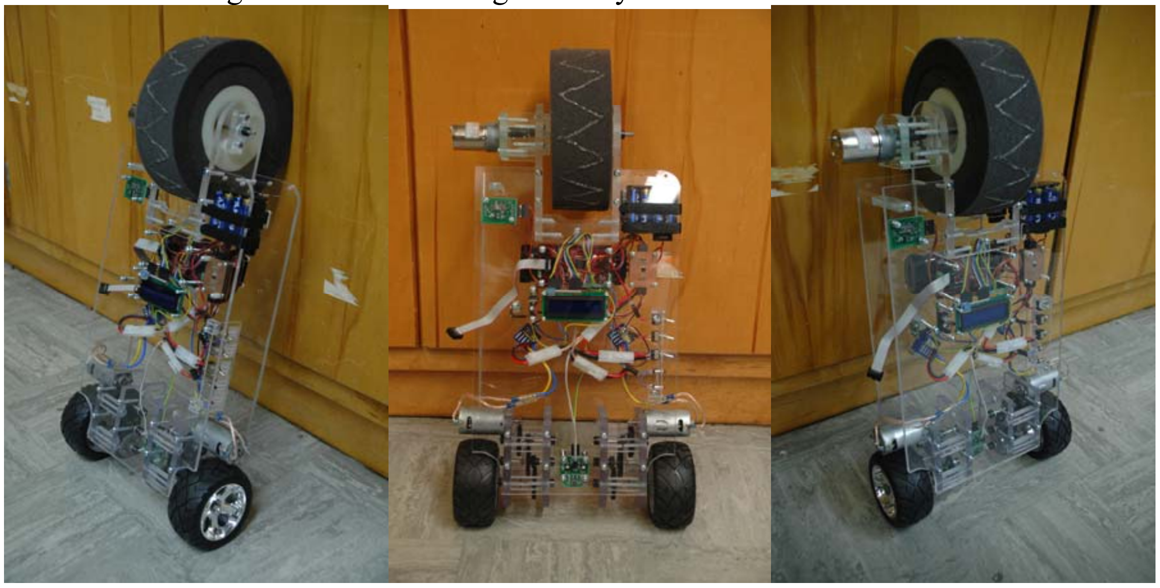
Figure 2 Different views of "Buzzed"

The robot was designed with Acrylic and plexi-glass for aesthetics and so the internals can be seen. In figure 2, most of the internal components are located in the upper part of the robot to maintain the higher center of gravity. In the center photograph the two main drive assemblies can be seen with the gyroscope chip placed along the drive axis. The top wheel drive motor is securely fastened to the top wheel assembly and the offset weight is balanced out by the battery placement. The platform structure is comprised of the two outer plexi-glass layers that are 3/16" in thickness for rigidity. The original design was with on plexi-glass frame but it cracked along the drilled hole lines. Also the single sheet left the internals exposed at all times, so a second sheet was added.

There were many lessons learned from the design and assembly of the platform. The main thing was that designing something is always a work in progress. You may have an initial idea, but as its being prototyped a better solution presents itself and resulting alterations are made. Also acrylic and plexi-glass as a material are different than any other material I have worked with due to its brittle characteristics and its inability to conduct heat. It is easy to work with in the sense that you have to feed slowly with tooling because it melts locally along the edges but allows for plenty of time to think. This constant changing of ideas causes for lengthened time frames for prototyping the platform and the one resource we don't have an abundance of is time. The manufacturing of the prototyping should have been started sooner to allow more work in less experienced and new areas for me in robotics.

Actuation

Buzzed was actuated through the used of three driven wheels. The function of the top wheel is to help it get to and climb a wall to stand upright. The function of the drive wheels is to drive Buzzed to a wall, push it up and maintain it in a balanced upright position. The two driven wheels are run with RC 25 T 550 case motors with LRP Runner Plus Reverse RC speed controller. A transmission was designed using 6 RC car gears per transmission to attain a ratio of 38:1 for good speed and torque modulation. The high torque of a 25 turn motor coupled with the reversibility proportionality of the RC controller fit my applications balancing demands. The top wheel of the robot is run by a Tamiya Planetary Gear System set to a ratio of 80:1 to give the low torque motor enough strength to stand up.

The tires of an RC car were used due to the high friction rubber for grip. On the top wheel, RC truck foam inserts were stacked on in layers on a rim and glued to create a large diameter squishy impact wheel, if in last resort the robot falls its not subjected to high impact forces.

Sensors

Scope

The balancing robot will have two separate MEMS sensors: a gyroscope and an accelerometer. The sensors will compose the inertial measurement unit that will orient the robot in the desired position.

Objectives

The gyroscope will have the task of reading the rate of tilt along the axis of the drive wheels. This will help stabilize the robot telling the microcontroller how fasts its falling. The accelerometer will be used as a tilt sensor due to its sensitivity to gravity and it will also read data is the robot is pushed.

Accelerometer

The accelerometer used in this robot is the ADXL203CE from Analog Devices and it was purchased from Digikey Electronics with an evaluation board.

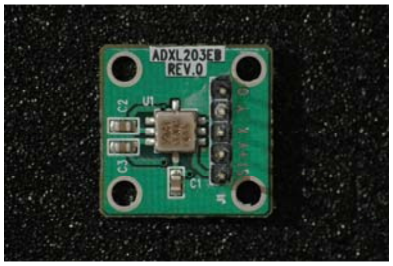

Figure 3 ADXL203CE Accelerometer

This is a Dual Axis Accelerometer that measures accelerations on a full scale range of ±1.7g on a single IC chip with a sensitivity of 1000mV/g. Its purpose is to help measure tilt and moving acceleration to help maintain the robot upright during stationary standing, movement, and under a physical push. The accelerometer will be the main sensor to detect angle of tilt in a forward or reverse direction of the robot when balancing and will aid in measuring dynamic motion of the robot when balancing.

This accelerometer is an analog MEMS sensor which means that the output voltages are proportional to the accelerations experienced by the IC chip. The accelerometer work by having a polysilicon surface-micromachined sensor and signal conditioning circuitry to implement an open-loop acceleration measurement architecture. The main mass structure is suspended in place by a polysilicon acting as a spring. Deflection of the structure is measured using a differential capacitor with one set of plates attached to the moving mass and the other to the silicon wafer (ADXL203CE datasheet). This means that a capacitance is created and the changes initiated within it are a direct result of accelerative forces. Figure 4 shows a block diagram of the functionality of the accelerometer. The applications vary from small handheld devices to the automotive industry and from NASA to military research and operations (ezinearticles.com).

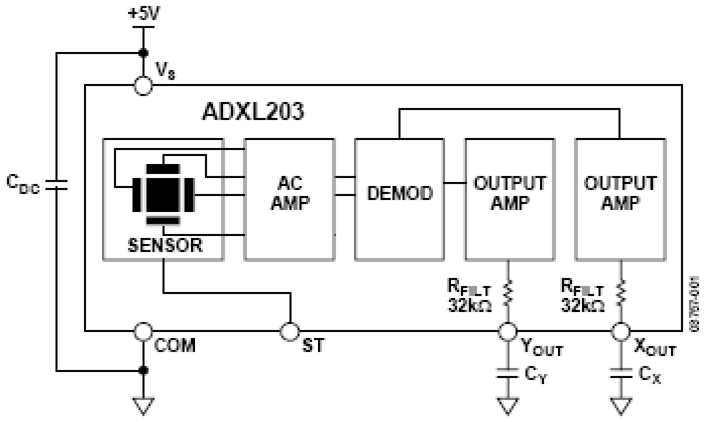

Figure 4 Accelerometer Functional Block Diagram. Courtesy of ADXL203CE Data Sheet.

The circuit board that was supplied with the evaluation board has a power supply and common ground as the input and has two analog outputs for each axis. The circuit board is also represented in Figure 1 as everything that sits outside of the actual ADXL203CE IC chip.

Gyroscope

The gyroscope used in this robot is the IDG300 from Invensense and it was purchased from Sparkfun Electronics.

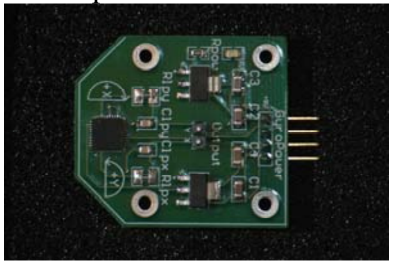
Figure 5 IDG300 Gyroscope

This is a Dual Axis Gyroscope that measures rotation on a full scale range of 500°/sec with a sensitivity 2 mV/(°/sec). Its purpose is to measure its rotational position and tilt during movement and stationary balancing. The gyroscope will be the main sensor that will measure rate of tilt in a forwards or reverse direction and measure the counter force.

The gyroscope is also a MEMS IC chip that works by relating voltages that are proportional to the angular rates experienced by the chip. The gyroscope does this through vertically driven, vibrating silicon masses. The vibrating dual-mass bulk silicon configurations sense the rate of rotation about the X- and Y-axis resulting in an integrated dual-axis gyro with vibration rejection and high cross-axis isolation.

A circuit board for the gyroscope was designed on Protel with the aid of Mike Pridgen. The layout was created based of the block diagram of the gyroscope from the data sheet as shown in figure 6 with onboard low-pass filtering to dampen the signal from large spikes. An LED was also placed on the board to show that power is supplied.

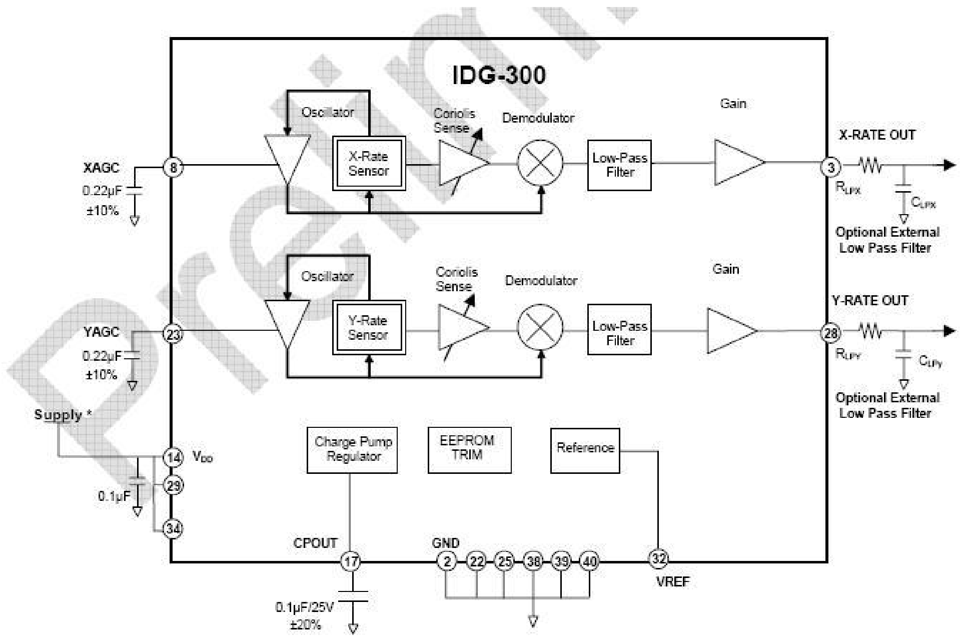

Figure 6 Gyroscope Functional Block Diagram. Courtesy of IDG300 Data Sheet.

Implementation

Each sensor had a voltage readout level that corresponds to a physical motion increment. Using the given numbers it becomes easy to calculate the physical motion of the device but since the robot doesn't need to physically display the values no calculation is necessary. Just plugging the values into the equations of motion and tuning the constants will take care of analyzing any data spit out by the sensors. The tilt sensor was really accurate in expressing what position it was in relation to horizontal. Table 1 represents the sensor output value to the angle from horizontal of the physical platform based on its placement on it. As for the gyroscope, the data it outputs is more difficult to quantify other than applying a simple conversion value expressed to the data sheet.

Table 1 Sensor Output on ADC Related to the Platforms Angle to Horizontal

Sensor Angle Data From ADC	Angle Increments from Horizontal	Degree Increment
718	0	2.311111
706	5	
695	10	
683	15	
672	20	
660	25	
649	30	
637	35	
626	40	
614	45	
602	50	
591	55	
579	60	
568	65	
556	70	
545	75	
533	80	
522	85	
510	90	
498	95	
487	100	
475	105	
464	110	
452	115	
441	120	
429	125	
418	130	
406	135	
394	140	
383	145	
371	150	
360	155	
348	160	
337	165	
325	170	
314	175	
302	180	

Behaviors

Buzzed has a menu of two different choices when initialized. If the first button is pushed the robot goes into a forward motion and moves towards a wall and stands up. It then jumps into its second function which pushes the robot off the wall and goes into the PID balancing loop. Figure 7 shows the functional block diagram of the code of the robot.

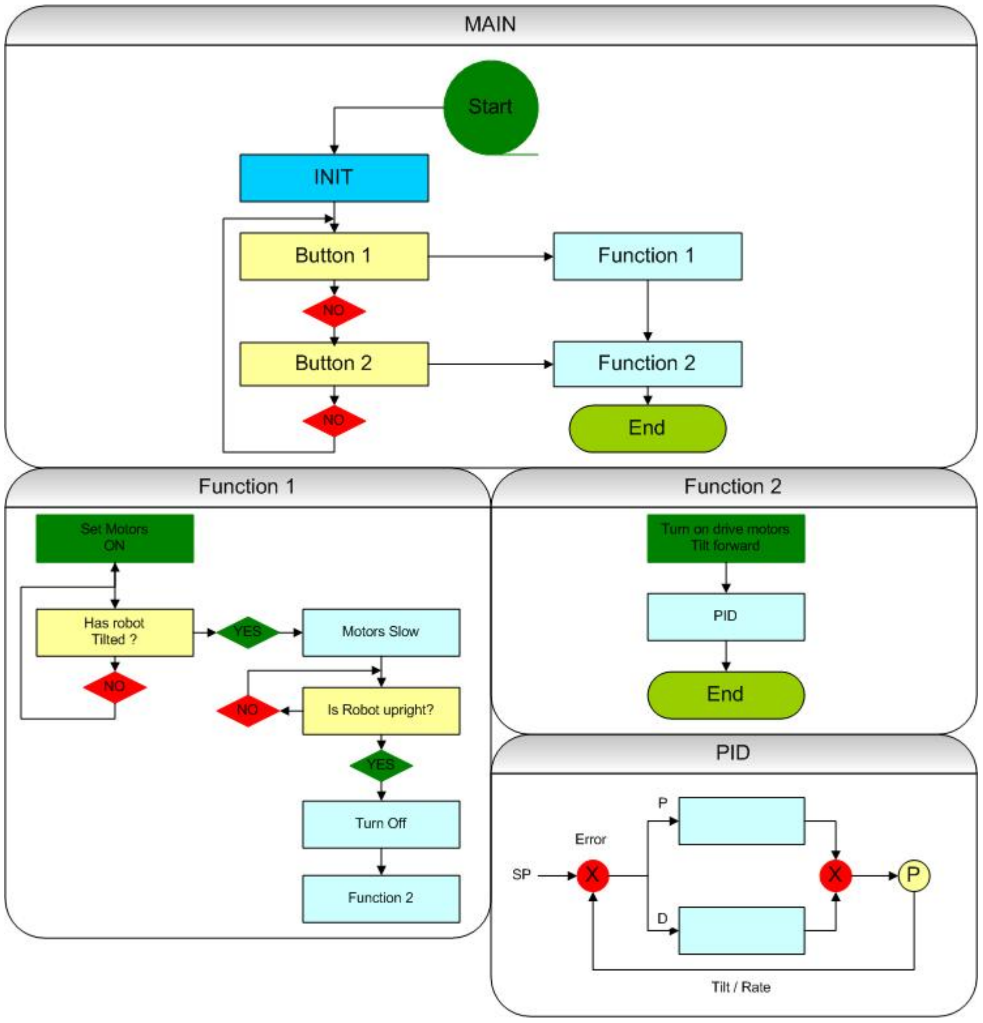

Figure 7 Code Organization

The two separate functions were created so a choice could be made between the two behaviors. The reason this was utilized was for tuning purposes. This allowed for immediate access to the balancing behavior to test the constants that were set.

Closing

Conclusion

A balancing robot is a difficult project in general. It requires good control, quality sensors, good filtering of data and a rigid and robust platform in order to achieve and repeat the goal. I learned an enormous amount of information on how to program and how to create working electrical circuits. There is way more to robotics then I had imagined about it worked and how to implement. I feel like the wealth of knowledge gained working on this robot is indispensable due to the fact that it ties different disciplines together to create a finished product.

There were several set backs to the creation of the robot but this was just part of the learning process. Mechanically, the last version of the robot was effective once the cracking issue was fixed. Electronically, the robot ran into several obstacles because the use of to many RC components. The control through the RC components is not effective due to low proportionality and delay in the motor controller as well as the high current drawn by the motor. Not having any electrical experience I feel accomplished in the final design but I had to go through more than my fair share of trial and error with the electronics.

The control in the robot is the hardest part to build due to the fact that it requires large amounts of tuning after the code has been implemented. This in itself is stressful for both the designer and the robot because the act of balancing can be catastrophic to the platform setting the project behind. The tuning phase of the design of the robot is the most time consuming because there is always room for improvement.

The limitations to Buzzed are that he cannot currently compensate for a force or stay balanced for more than a minute. In the semester much was accomplished but in all reality there wasn't enough time for proper trouble shooting and then tuning. In the area of tuning alone, it can be optimized because the oscillations were still occurring. The other balancing robots I researched had much better balancing behaviors then that of Buzzed. Also Buzzed doesn't have the option of moving around because he doesn't have necessary hardware. Encoders are necessary due to the fact that they express motion in position and velocity. This allows for the ability of moving around and motion other then the balancing sequence.

Future plans for buzzed include getting a very highly tuned balancing behavior. Also the ability to compensate for pushes on the platform as being able to push back on an object or wall. With the aid of encoders I would like to be able to move through a preprogrammed formation or path. This platform has also the ability of letting the robot have sensors for line and wall following and obstacle avoidance once it's moving around with the aid of the encoders.

This project, although difficult in undertaking in the time frame of a semester, is a very interesting and encompassing learning experience that teaches about the integration of different disciplines of engineering into one autonomous robot.

Appendices
Program Code

```
/***************************************************************************
 *
                                                                       *
 * Title:            adc_position rate.c
                                                   *
 * Programmer:   Andres Vargas
         *
 * Date:             April 9, 2008
         *
                                                   *
 * Description:
                                                   *
 *          Verifies the position rate works with the wheels on a BDMICRO MAVRICII                    *
 *  Collects samples on analog channel 0
         *
 ***************************************************************************/

/* define CPU frequency in Mhz here if not defined in Makefile */
#ifndef F_CPU
#define F_CPU 14.7456E6
#endif

/*************************** Includes ********************************/
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

/*************************** Prototypes *******************************/
void lcd_delayRead();
void lcd_delay();       // short delay (50000 clocks)
void lcd_init();        // sets lcd in 4 bit mode, 2-line mode, with cursor on and set to blink
void lcd_cmd();         // use to send commands to lcd
void lcd_disp();        // use to display text on lcd
void lcd_clear();       // use to clear LCD and return cursor to home position
void line1();                                   // cursor to start of line 1
void line2();                                   // cursor to start of line 2
void config_adc(void);
int  analog(int);
void pwm_init(void);


volatile uint16_t mscount;


/******************** Global Variables *********************/
signed int motorRdrive;
signed int motorLdrive;
signed int motorTdrive;

/*
 * millisecond counter interrupt vector
 * monitor Timer0, will fire when it's value = OCR0
 *
 */
ISR(TIMER0_COMP_vect)     //interupt service routine
{
        TCNT0 = 0; // reset the interrupt counter
        // make the led blink
        if(++mscount == 125) {
                PORTB ^= 0x01;
                mscount = 0;
        }

}


void init_timer(void)
{
        /*
         * Initialize timer0 to generate an output compare interrupt, and
         * set the output compare register so that we get that interrupt
         * every millisecond.
         *
         * this is also needed for the gyro
         */
        TIFR  |= _BV(OCIE0) | _BV(OCIE2);
        TCCR0  = _BV(WGM01)|_BV(CS02)|_BV(CS00); /* CTC, prescale = 128 */
        TCNT0  = 0;
        TIMSK |= _BV(OCIE0);    /* enable output compare interrupt */
        //OCR0   = 125;          /* match in 1 ms */
        OCR0 = 128;  // ((14745600 Hz / 128) / 2^7) = 900 Hz
        mscount = 0;

}


void init(void)
{
        // Enable interrupts
        sei();

        // Let everything settle out before running code
        _delay_ms(200);

        // Initialize the timer for interrupts
        init_timer();

        // Initialize the A/D converter
        config_adc();

        // Initialize the PWM servo signal
        pwm_init();

        // Initialize lcd in 4 bit mode, 2-line mode, with cursor on and set to blink
        lcd_init();

        // set portB pin 1 to output (led on mavric)
        DDRB = 0x01;
```

```
}


/* IMPORTANT!

Before using this code make sure your LCD is wired up the same way mine is, or change the

code to
match the wiring of your own LCD.  My LCD is connected to PortC of the At-Mega128 in the

following manner:

PortC bit 7   :   LCD data bit 7 (MSB)
PortC bit 6   :   LCD data bit 6
PortC bit 5   :   LCD data bit 5
PortC bit 4   :   LCD data bit 4 (LSB)

PortC bit 3   :   (not connected)
PortC bit 2   :   LCD enable pin (clock)
PortC bit 1   :   LCD R/W (read / write) signal
PortC bit 0   :   LCD RS (register select) pin

Also remember you must connect a potentiometer (variable resistor) to the vcc, gnd, and

contrast pins on the LCD.
The output of the pot (middle pin) should be connected to the contrast pin.  The other two

can be on either pin.

*/


void lcd_delayRead()      // delay for 10000 clock cycles
{
            long int ms_count = 0;
            while (ms_count < 1000000)
            {
    ms_count = ms_count + 1;
            }
}

void lcd_delay()      // delay for 10000 clock cycles
{
            long int ms_count = 0;
            while (ms_count < 750)
            {
    ms_count = ms_count + 1;
            }
}

void lcd_cmd( unsigned int myData )
{

            /* READ THIS!!!
```

```
        The & and | functions are the BITWISE AND and BITWISE OR functions respectively.  DO NOT
        confuse these with the && and || functions (which are the LOGICAL AND and LOGICAL OR
functions).

        The logical functions will only return a single 1 or 0 value, thus they do not work in this scenario
        since we need the 8-bit value passed to this function to be preserved as 8-bits
        */


        unsigned int temp_data = 0;

        temp_data = ( myData | 0b00000100 );         // these two lines leave the upper nibble as-is, and set
        temp_data = ( temp_data & 0b11110100 );  // the appropriate control bits in the lower nibble
        PORTC = temp_data;
        lcd_delay();
        PORTC = (temp_data & 0b11110000);         // we have written upper nibble to the LCD

        temp_data = ( myData << 4 );         // here, we reload myData into our temp. variable and shift the
bits
                                             // to the left 4 times.  This puts the lower nibble into the upper
4 bits

        temp_data = (temp_data & 0b11110100);   // temp_data now contains the original
        temp_data = (temp_data | 0b00000100);   // lower nibble plus high clock signal

        PORTC = temp_data;                  // write the data to PortC
        lcd_delay();
        PORTC = (temp_data & 0b11110000);      // re-write the data to PortC with the clock signal low
(thus creating the falling edge)
        lcd_delay();

}

void lcd_disp(unsigned int disp)
{

        /*

        This function is identical to the lcd_cmd function with only one exception.  This least significant
bit of
        PortC is forced high so the LCD interprets the values written to is as data instead of a command.

        */

        unsigned int temp_data = 0;

        temp_data = ( disp & 0b11110000 );
        temp_data = ( temp_data | 0b00000101 );
        PORTC = temp_data;
        lcd_delay();
        PORTC = (temp_data & 0b11110001);
        lcd_delay();                // upper nibble

        temp_data = (disp << 4 );
        temp_data = ( temp_data & 0b11110000 );
```

```
		temp_data = ( temp_data | 0b00000101 );
		PORTC = temp_data;
		lcd_delay();
		PORTC = (temp_data & 0b11110001);
		lcd_delay();			// lower nibble

}


void lcd_init()
{
		DDRC = 0xFF;			// set portC to output  (could also use DDRC = 0b11111111)
		lcd_cmd(0x33);	// writing 0x33 followed by
		lcd_cmd(0x32);	// 0x32 puts the LCD in 4-bit mode
		lcd_cmd(0x28);	// writing 0x28 puts the LCD in 2-line mode
		lcd_cmd(0x0F);	// writing 0x0F turns the display on, curson on, and puts the cursor in blink
mode
		lcd_cmd(0x01);	// writing 0x01 clears the LCD and sets the cursor to the home (top left)
position
		//LCD is on... ready to write
		//DDRC = 0xFF;			// set portC to output  (could also use DDRC = 0b11111111)

}

void lcd_string(char *a)
{
		/*
		This function writes a string to the LCD.  LCDs can only print one character at a time so we need
to
		print each letter or number in the string one at a time.  This is accomplished by creating a pointer
to
		the beginning of the string (which logically points to the first character).  It is important to
understand
		that all strings in C end with the "null" character which is interpreted by the language as a 0.  So to
print
		an entire string to the LCD we point to the beginning of the string, print the first letter, then we
increment
		the pointer (thus making it point to the second letter), print that letter, and keep incrementing until
we reach
		the "null" character".  This can all be easily done by using a while loop that continuously prints a
letter and
		increments the pointer as long as a 0 is not what the pointer points to.
		*/

		while (*a != 0)
		{
    lcd_disp((unsigned int) *a);   // display the character that our pointer (a) is pointing to
    a++;				// increment a
		}
		return;

}
void lcd_long (long value)
{
		/*
		This routine will take an integer and display it in the proper order on
```

```
        your LCD.  Thanks to Josh Hartman (IMDL Spring 2007) for writing this in lab
        */
        int temp_val;
        long x = 1000000000;        // since integers only go up to 32768, we only need to worry about
                        // numbers containing at most a ten-thousands place
        if (value < 0)
        {
                lcd_disp ('M');
                value *= -1;
   }

        while (value / x == 0)   // the purpose of this loop is to find out the largest position (in decimal)
        {                       // that our integer contains.  As soon as we get a non-zero value, we know
        x/=10;                  // how many positions there are int the int and x will be properly initialized to
the largest
        }                               // power of 10 that will return a non-zero value when our integer is
divided by x.

        if (value==0) lcd_disp(0x30);

        else while (x >= 1)           // this loop is where the printing to the LCD takes place.  First, we
divide
        {                       // our integer by x (properly initialized by the last loop) and store it in
        temp_val = value / x;      // a temporary variable so our original value is preserved.Next we
subtract the
        value -= temp_val * x;     // temp. variable times x from our original value.  This will "pull" off the
most
        lcd_disp(temp_val+ 0x30);  // significant digit from our original integer but leave all the remaining
digits alone.
                                // After this, we add a hex 30 to our temp. variable because ASCII values for
integers
        x /= 10;                // 0 through 9 correspond to hex numbers 30 through 39.  We then send this
value to the
        }                       // LCD (which understands ASCII).  Finally, we divide x by 10 and repeat the
process
                                // until we get a zero value (note: since our value is an integer, any decimal
value
        return;                   // less than 1 will be truncated to a 0)


}

void lcd_int(int value)
{
        /*
        This routine will take an integer and display it in the proper order on
        your LCD.  Thanks to Josh Hartman (IMDL Spring 2007) for writing this in lab
        */
        int temp_val;
        int x = 10000;          // since integers only go up to 32768, we only need to worry about
                // numbers containing at most a ten-thousands place
        while (value / x == 0)   // the purpose of this loop is to find out the largest position (in decimal)
        {                   // that our integer contains.  As soon as we get a non-zero value, we know
    x/=10;              // how many positions there are int the int and x will be properly initialized to the
largest
```

```
        }                              // power of 10 that will return a non-zero value when our integer is
divided by x.

        if (value==0) lcd_disp(0x30);

        else while (x >= 1)       // this loop is where the printing to the LCD takes place.  First, we divide
        {                         // our integer by x (properly initialized by the last loop) and store it in
   temp_val = value / x;     // a temporary variable so our original value is preserved.Next we subtract the
   value -= temp_val * x;    // temp. variable times x from our original value.  This will "pull" off the most
   lcd_disp(temp_val+ 0x30); // significant digit from our original integer but leave all the remaining
digits alone.
                              // After this, we add a hex 30 to our temp. variable because ASCII values for
integers
   x /= 10;                   // 0 through 9 correspond to hex numbers 30 through 39.  We then send this
value to the
        }                     // LCD (which understands ASCII).  Finally, we divide x by 10 and repeat the
process
                              // until we get a zero value (note: since our value is an integer, any decimal
value
        return;                // less than 1 will be truncated to a 0)

}

void lcd_clear()     // this function clears the LCD and sets the cursor to the home (upper left) position
{
        lcd_cmd(0x01);

        return;
}

void line1()
{
        lcd_cmd(0x80);
        return;
}

void line2()
{
        lcd_cmd(0xc0);
        return;
}


int main(void)
{
        long i, /*counter=0*/ angleX=700, /*angleY=0, rotationRateX=0, rotationRateY=0,*/ error=0,
speed=0; char s1[20]="Buzzed, the", s2[20]="Balancing Robot";
        init();
        line1();
        lcd_string(s1);    // if your LCD is wired up correctly, you will see this text
        line2();                       // on it when you power up your Micro-controller board.
        lcd_string(s2);
        lcd_delayRead();
        lcd_clear();
        lcd_string("Standing up");
        line2();
```

```
lcd_string("Behavior   ");
lcd_delayRead();
while (angleX > 590)
{
        OCR3A = 18444;
        OCR3B = 18444;
        OCR3C = 18444;
        angleX = analog(0x03);
        line1();
        lcd_string("A = ");
        lcd_long(angleX);
        lcd_string("     ");
}
while (angleX > 550)
{
        OCR3A = 13833;
        OCR3B = 18444;
        OCR3C = 18444;
        angleX = analog(0x03);
        line1();
        lcd_string("A = ");
        lcd_long(angleX);
        lcd_string("     ");
}
OCR3A = 9222;
lcd_clear;
lcd_string("Tilt Sensor");
line2();
lcd_string("Position Data");
lcd_delayRead();
lcd_clear();

while(1)
{
        angleX = analog(0x03);
        //angleY = analog(0x04);
        //rotationRateX = analog(0x01);
        //rotationRateY = analog(0x00);
        line1();
        lcd_string("A = ");
        lcd_long(angleX);
        lcd_string("     ");
        if (angleX < 510) error = 510-angleX;
        //else error = angleX-510;
        line2();
        lcd_string("Error = ");
        lcd_long(error);
        lcd_string("     ");
        speed = 9222+error*.2;      // P loop for simple balancing sequence;


        //lcd_string(" ");
        //lcd_long(angleY);
        //lcd_string("      ");
        //line2();
        //lcd_string("R=");
```

```
                //lcd_long(rotationRateX);
                //lcd_string(" ");
                //lcd_long(rotationRateY);
                for (i = 0; i < 10; i++)
                {
                        lcd_delay();  //delay to read LCD (humans reading)
                }
   //temp = PINB;                        // read portB, store value to temp
   //PORTB = !(temp);     // complement portB and write back

        }
        return 0;

}


void config_adc(void)
{
        DDRF = 0b00000000; // set port F to all input
          // Note: when JTAGEN fuse is set, F4 - F7 don't work
        PORTF = 0x00;      // make sure pull up resistor is not enabled

        ADMUX = 0b01000000; // 5V reference, select channel0 (pin F0)
        ADCSRA |= 0b10000111; // turn on ADC, don't start conversions
             // free funning
             // divide clock by 128

}


int analog(int analogch)
{
   int anval;
  ADMUX = 0b01000000|analogch;
  // Start AD conversion.
  ADCSRA |= (1 << ADSC);
  // Wait for ADC conversion to complete.
  while ( ADCSRA & (1 << ADSC) );
   anval = ADCL | (ADCH << 8); //place ACD value into one variable
  return anval;

        //analogLow = ADCL; // read ACD low register
        //analogHigh = ADCH; // read ACD high register


}


void pwm_init(void)
{// jump pin E3 E4 E5 to respective servo values

        DDRE = 0xFF; // set PWM pins as outputs, PE3 (OC3A) & PE4 (OC2B) & PE5 (OC2B)
        TCCR3A = 0b10101000; // ......00:P&FC PWM // 11......,..11....:OC1A
        TCCR3B = 0b00010010; // ...10...:P&FC PWM // .....001:bclk/1
        ICR3=18444;
        TCNT3=0x0000;
        OCR3A = 9222;  // 1/2 duty, ~2.5v dc level, motor 1 on PE3 (OC3A)
        OCR3B = 9222; // 1/2 duty, ~2.5v dc level, motor 1 on PE4 (OC3B)
        OCR3C = 9222; // 1/2 duty, ~2.5v dc level, motor 1 on PE5 (OC3C)
        return;
}
```

Transmission CAD

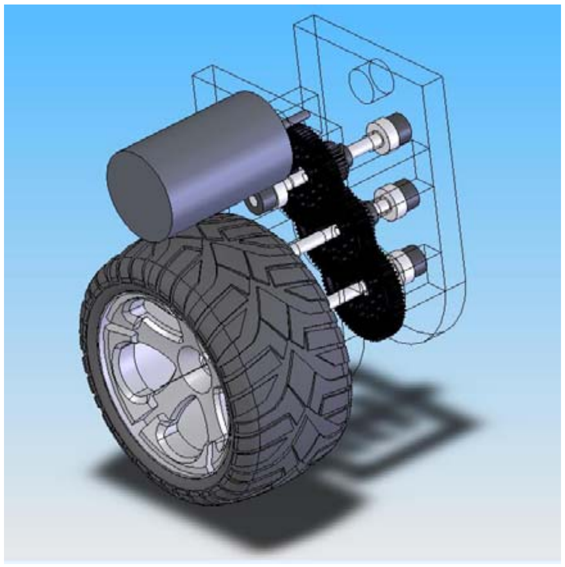

Figure 8 Isometric view

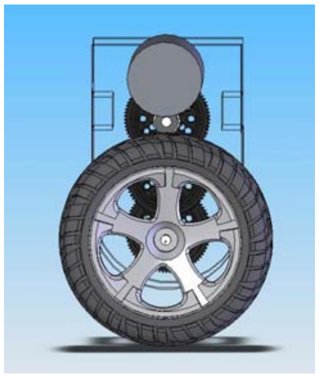

Figure 9 Front view

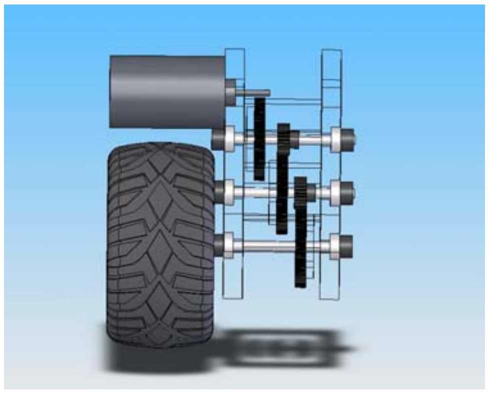

Figure 10 Left view