Inverting Pendulum Robot

Design Documentation

Gregory Daniels
Cameron Madsen

May 2013

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the design of an inverted pendulum robot. The major purpose of the device is to stay upright and balanced. The robot has two wheels; each wheel is directly connected to its own DC motor. The device is controlled by a Cortex M3 microcontroller on an Open 103R development board. To stay balanced the microcontroller communicates with a gyroscope/accelerometer, depending on the angular position of the robot, the microcontroller controls the speed and direction of the motors in an effort to keep the robot balanced.

2 Scope

This document discusses the electrical and software design of the inverted pendulum robot in detail. The mechanical design is discussed but not in great detail. Included in this document are: requirements, dependencies and the theory of operation. Testing procedures and results are also included.

3 Design Overview

3.1 Requirements

The following are the requirements for the inverted pendulum robot.

1. The robots' motors shall run on an 11.1 volt Lithium Polymer 3 cell battery.
2. The microcontroller shall be mounted to the robot itself.
3. The control loop of the robot shall update every 2ms.
4. The robot shall be capable of balancing by itself after being initially set upright.
5. The robot's only contact with the ground shall be its two wheels.

3.2 Dependencies

The following are dependencies for the inverted pendulum robot:

- 5V power supply for the microcontroller
- 8 MHz High Speed External Oscillator
- 11.1V Lithium Polymer 3 cell Battery
- SEEED SLD01102P Motor Shield ESC (Electronic Speed Controller)
- Waveshare Open103R Development board with Cortex M-3 microcontroller
- Invensense MPU-6050 IMU
- (2) Pololu 90x10 mm Wheels
- (2) 29:1 12V 265 rpm MetalGear DC Motor

3.3 Theory of Operation

Prior to power up or reset, the robot is placed and held in a roughly vertical position. When power is turned on, support is removed and the robot remains upright and stable. It does this by constantly measuring its angle with respect to the ground and making adjustments to motor speed and direction.

3.4 Design Alternatives

3.4.1 Mechanical Design Alternatives

The initial prototype was built from scrap parts. The robot body was made from an old Erector Set. The wheels were from an old RC car. The motors were purchased at Radio Shack. The motors drove the axel with a pulley system from the Erector Set. The battery was placed on top of the robot. The IMU and speed controller were mounted to the side. This design was not balanced very well. It quickly became clear that it would be difficult if not impossible to control the robot in this configuration. The initial mechanical design was abandoned.

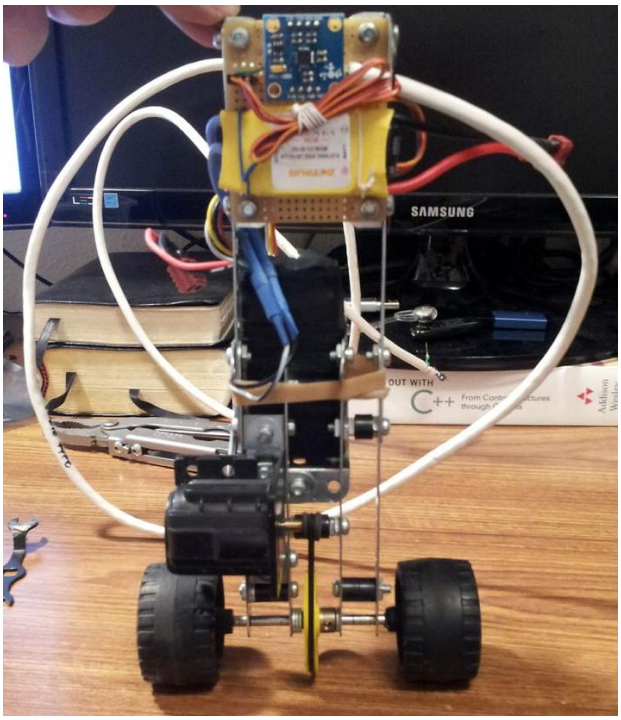

Figure 1: Initial design of inverted pendulum robot.

3.4.2 Control Algorithm Alternatives

In the initial stages of the project Fuzzy Logic was used in an attempt to balance the robot. It was soon discovered that Fuzzy Logic was more of a brute force approach to the problem. In this application the Fuzzy Logic approach would equate to a lot of “if else” statements. If the angle is x, motor speed=y. Fuzzy Logic would have been very tedious to implement and tune properly. For this reason Fuzzy Logic was abandoned and a PID loop was implemented to control the robot.

3.4.3 Alternative Digital Filters

In order to get accurate measurements from the IMU it became clear that some kind of filter was necessary. The IMU has two main components, a gyroscope and an accelerometer. The gyroscope measures angular velocity and is subject to drifting. The accelerometer measures

angular position but vibrations have a huge impact on the readings; because of this to get an accurate angle measurement both must be taken into consideration. A simple low pass filter of the accelerometer was considered, this would have involved taking the average of several accelerometer samples. The time required to gather and average enough samples to get a good measurement would have been detrimental to the system, so a simple low pass filter was rejected. A Kalman Filter was also considered, but due to the complexity of the math involved, it would have been difficult to code and would have taken too much processor time to do the computations. It was determined that a complementary filter would work the best for the inverted pendulum robot.

4 Design Details

4.1 Hardware Design

4.1.1 Mechanical Design

The design of the robot was loosely based on a design by TJKelectronics [2]. Two 12 V motors with 29:1 gear ratios were used to control the robot. Two 90 mm Pololu wheels were mounted directly to the gearboxes. The battery was attached to the top with Velcro. The Speed Controller, IMU, and microcontroller were mounted to the frame of the robot. The dimensions of the robot are 3.5x7.5x9 inches. The main supports are 3/8 inch threaded stock, and the platforms are ½ inch pine. A piece of heavy iron was attached to the top of the robot with Velcro, this weight at the top helps the robot stay balanced.

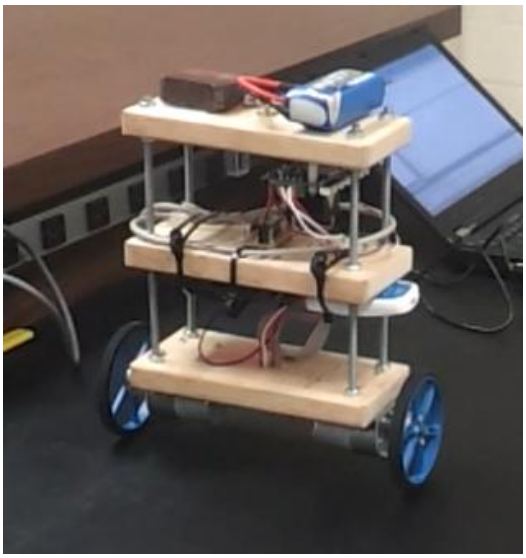

Figure 2: Mechanical design of inverted pendulum robot.

4.1.2 Electrical Design

The inverted pendulum robot is implemented using a Cortex M3 microcontroller on an Open103R development board. The microcontroller interfaces directly with the Invensense MPU-6050 IMU via the I2C protocol. The microcontroller also interfaces directly with the SEEED SLD01102P Motor Shield ESC (Electronic Speed Controller) via PWM (Pulse Width Modulation). The Motor Shield controls the speed of the DC motors based on the PWM signal it receives from the microcontroller and the direction based on two lines the microcontroller drives (This is discussed in more detail in section 4.2.3). The Motor Shield connects directly to an 11.1V Lithium Polymer 3 cell Battery. The motor shield has a 5V output port that is used to power the microcontroller. A detailed electrical schematic is shown in Figure 1.

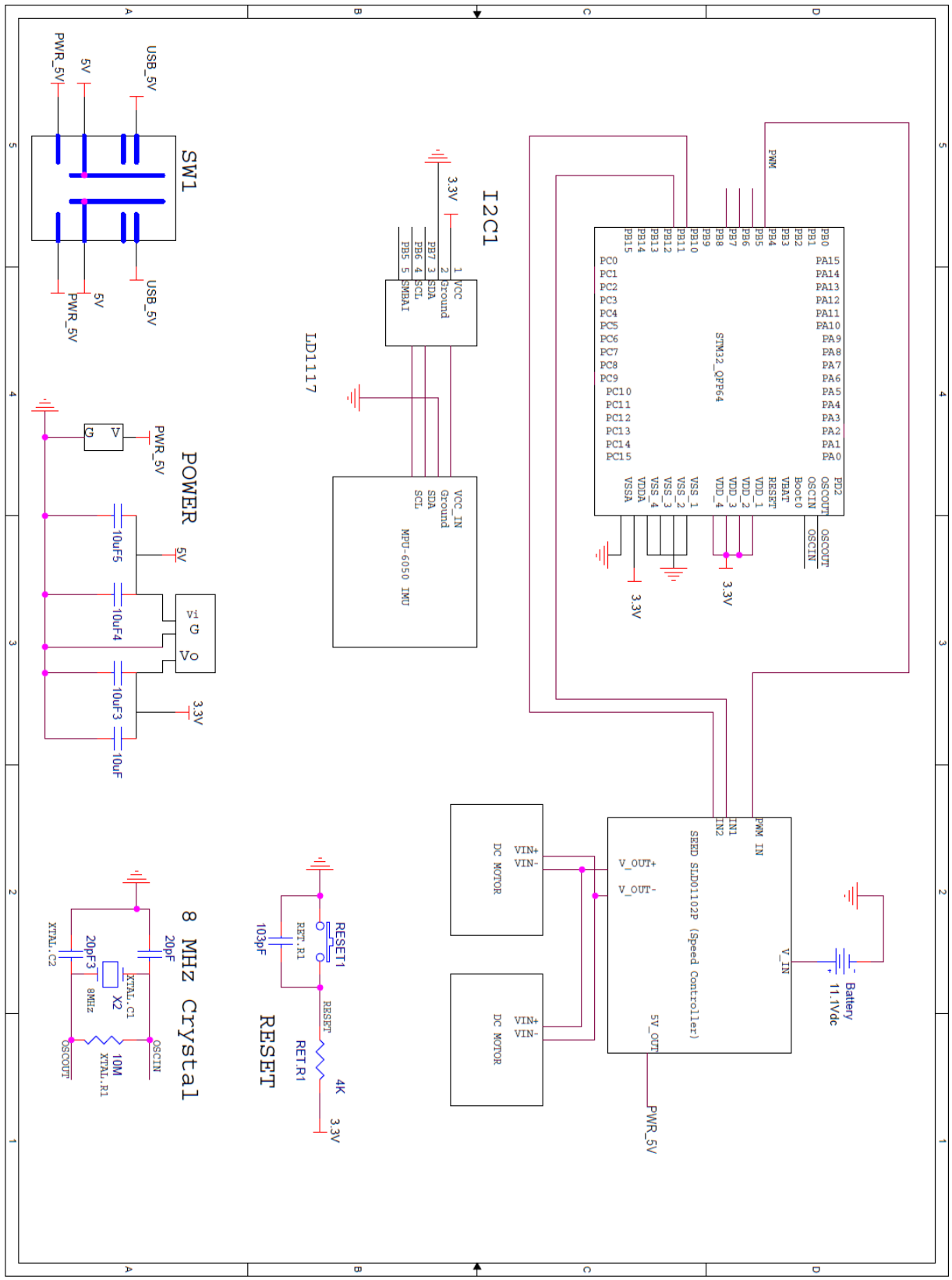

Figure 3: Schematic of electrical components. I2C uses PB 6 and PB 7 PWM uses PB4, the motor direction is controlled by PB10 and PB11.

4.2 Software Design

The software of the inverted pendulum robot is divided into four main parts: IIC and IMU integration, a complementary digital filter, PWM (Pulse Width Modulation), and PID control. All of these parts are used in order to interface with the necessary components to properly control the robot. After initialization and setup, the program enters an infinite while loop. In the while loop a bit is polled waiting for a timer to expire; the timer is set to expire every 2ms. Upon timer expiration, the current angle and angular velocity are requested and received from the IMU. Using these values and data from previous calculations in a PID (Proportional-Integral-Derivative) control algorithm, a PWM signal is generated and output to the ESC (electronic speed controller) to set the speed and direction of the motors. Data is then stored for use the next time the timer expires.

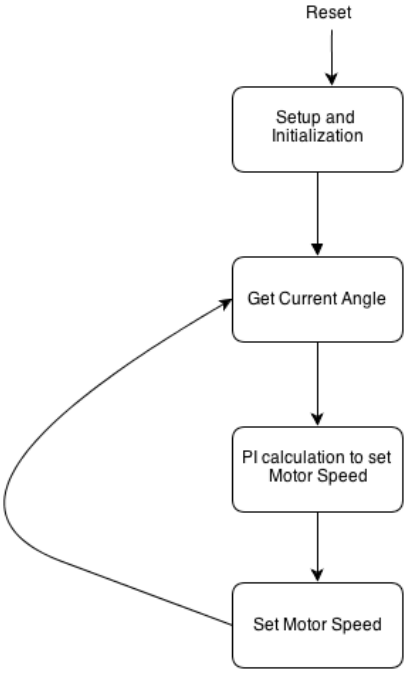

Figure 4: Flow chart of program. The loop is repeated every 2ms indefinitely.

4.2.1 IIC and IMU Integration

The Invensense IMU (Inertial Movement Unit) combines a 3 axis accelerometer and a 3 axis gyroscope for a total of 6 DOF (degrees of freedom). Only three of these were used to obtain the orientation of the robot (see section 4.2.2 for further explanation). The Invensense IMU communicates with the Open103R development board via IIC (Inter Integrated Circuit) to provide 16 bit data representing the orientation.

IIC drivers were developed for the microcontroller to communicate with the IMU. The IMU has the capability to run at 400 KHz but due to the length of the cable used to connect the IMU to the microcontroller, the communication failed at speeds greater than 250 KHz. 100 KHz was chosen to add a safety factor to the communication.

The IIC drivers are based on polling. This approach was chosen because of its simplicity over Interrupt driven communication. This has little to no detriment to the performance of the software because the software must wait until it has the orientation data from the IMU before it can continue. There are several IIC events that the microcontroller must do and acknowledge to communicate properly.

The initial version of the drivers worked great until any sort of error occurred on the IIC bus. On the occurrence of an error, such as a failed ACK, the microcontroller would be stuck in an infinite loop. To remedy this, code was added to detect when a bit was being polled for too long. When this occurred the IIC module on the microcontroller was reset and communication resumed before the error occurred. See Figure 4 below.

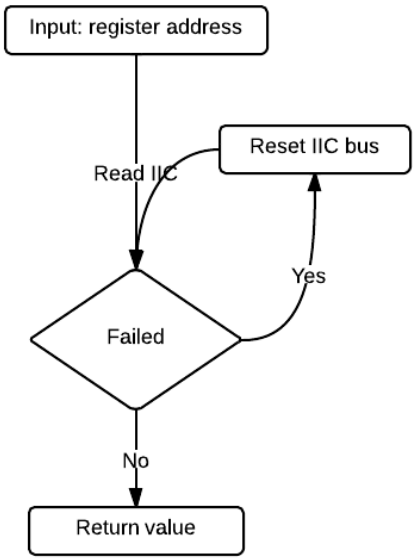

Figure 5: Error recovery of IIC.

4.2.2 Complementary Filter

To properly calculate the orientation from the sensor data a complementary filter was used. The design of this filter is based on a presentation provided by MIT [1]. The filter works by taking the previous angle, adding to it how far the IMU traveled (from the gyroscope) multiplied by a constant less than 1 and adding to that the angle calculated from the accelerometer multiplied by another constant, see Figure 5. Both constants must add to 1.

$$angle = a * (previousangle + gyro * dt) + (1 - a) * (xacc)$$

Figure 6: Complimentary filter equation

The code (Figure 6) uses this equation and returns the angle.

```
Angle=A*(Angle + *XAngularVelocity*LOOP_TIME_CONSTANT) + (1-A)*(accelAngle);
```

Figure 7: Code for complimentary filter

Angle is calculated by taking the arctangent of the x and y readings off of the accelerometer. The Angular velocity is calculated by reading the x axis off of the gyroscope.

The variable “A” has the value of .98 giving more weight to the gyroscope readings while allowing the accelerometer to compensate for any drift that might otherwise occur. gyro*dt is the angular velocity multiplied by the time since the last calculation. This yields how many degrees the imu has traveled. The angle calculated from the accelerometer readings is then multiplied by 0.02 and added onto the value calculated by the gyro. This allows the filter to be fairly stable.

4.2.3 PWM

A PWM (Pulse Width Modulation) signal and two direction bits output by the microcontroller are used to interact with the SEEED SLD01102P Motor Shield ESC (Electronic Speed Controller). The following table shows how the microcontroller controls the ESC.

Microcontroller Pin:	PB4	PB11	PB10	
Speed Controller Pin:	**Enable (PWM)**	**IN1**	**IN2**	**Motion of Motor**
	0	x	x	Stop
	1	0	0	Stop
	1	0	1	Clockwise
	1	1	0	Counter Clockwise
	1	1	1	Stop

Table 1: Microcontroller and Speed Controller interaction

The speed controller interprets the PWM signal to control the speed of the motors. A duty cycle of 0% will turn the motors off while a higher duty cycle PWM input will drive the motors in the direction indicated by the direction bits. A duty cycle of 100% means the motors are going as

fast as they can go. PWM is configured on the Open103R development board by using a timer. A function, setDC(unsigned short duty_cycle, unsigned char direction), was written to set the desired duty cycle of the PWM signal. The variable passed into the function (duty_cycle) is expected to be a number from 0 to 1000. This number is multiplied by the reload value of the timer and then is divided by 1000 to give the desired duty cycle for the signal. It was done this way in an attempt to avoid integer division problems. The direction variable expects either a 0 or a 1. A value of 1 moves the motors clockwise, a value of 0 moves the motors counter clockwise. For example, to output a 50% duty cycle wave (this turns the motors halfway on) and move the motors clockwise, the user would make the following function call:

```
setDC(500,1);
```

Figure 8: Example of SetDC() function call

This would output a PWM signal with a 50% duty cycle. The following table explains what duty cycle values were used to control the motors. The timer controlling the PWM signal was configured to output at around 1000 Hz.

4.2.4 PID Control

To control the inverted pendulum robot a PID control algorithm was implemented. The implementation of the PID loop is shown by the following flow chart:

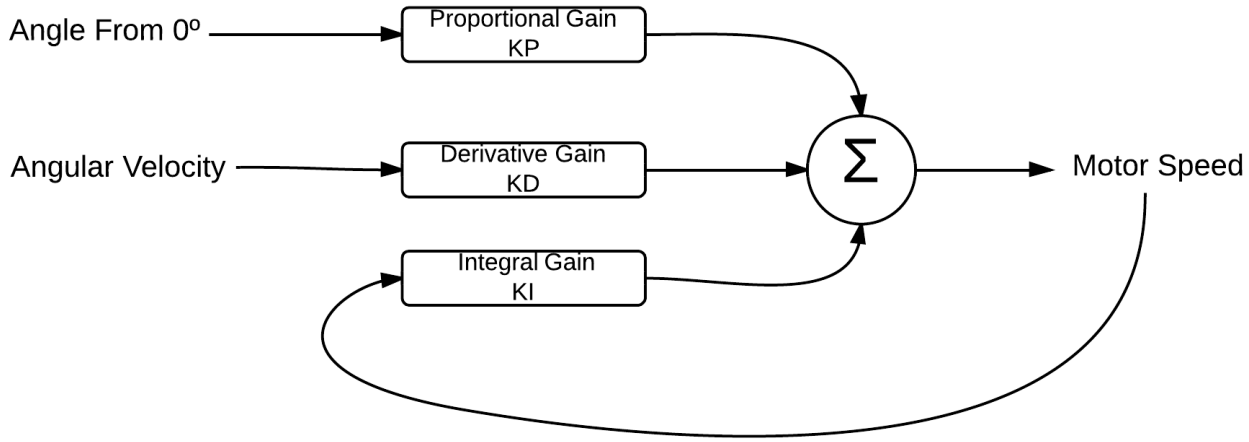

Figure 9: PID Loop implementation flow chart.

The angle from 0° represents the current error. When there is no error in the system, the angle is 0° and the robot is perfectly balanced. KP is the proportional gain, KI is the integral gain, and KD is the derivative gain. These values are constants and their proper values were calculated by trial and error (see section 5.3 for details). KP is multiplied by the current angle; this term accounts for the current error in the system. KI is multiplied by the last speed of the motor, this term accounts for past errors in the system. KD is multiplied by the current angular velocity, this

term accounts for the likelihood of future errors in the system [3]. All three terms are added together and set as the duty cycle of the motor. If the speed value is negative, the motors are set to rotate clockwise. If the speed value is positive the motors are set to rotate counter clockwise. The following code snippet demonstrates how this is accomplished:

```
speed = KP * angle + KD*angluar_vel + KI*speed; //pid calculation

if(speed>0)
{
   if (speed>MOTORRANGE)
       {
          speed=MOTORRANGE;
       }
          setDC(DEADZONE + speed, 0); //Set the motor counter clockwise
}

  else
  {
    tempspeed=-speed;

        if(tempspeed>MOTORRANGE)
        {
            speed = -MOTORRANGE;
            tempspeed=MOTORRANGE;
        }
           setDC(DEADZONE+ tempspeed, 1); //Set the motor clockwise
   }
```

Figure 10: Code for implementing PID control algorithm and setting the motor speed. The MOTORRANGE value is a predefined constant that clamps the speed of the motors. It is set to 500, so the motors are never on more than 50%. DEADZONE is a constant that offsets the speed. It was found that the motors did not start to spin the wheels until a 20% duty cycle PWM was applied, so the constant DEADZONE was set to 200.

5 Testing

5.1 IIC, IMU, and Filter Testing

The IIC and IMU were tested at the same time because there is no effective way to simulate the way the IMU communicates with the microcontroller. The IMU has a temperature sensor that provides 16 bit values in the same manner as the accelerometers and gyroscopes. The values returned were monitored using the Keil debugger, and verified to be correct. The filter was then implemented and the angle was monitored and determined to be correct by reading 90 degrees when flat on the table and 0 degrees when upright.

5.2 PWM and Motor Testing

To test the PWM, an oscilloscope was connected to the PWM output pin; the set_DC function (discussed in 4.2.3) was then called with various values. All of the values tested set the duty cycle correctly. The motors were then hooked up and tested. It was found that the motors did not start spinning until a 20% duty cycle PWM signal was sent to the ESC.

The function call set_DC(500,0) produced the following output. Notice the wave is output at exactly a 50% duty cycle.

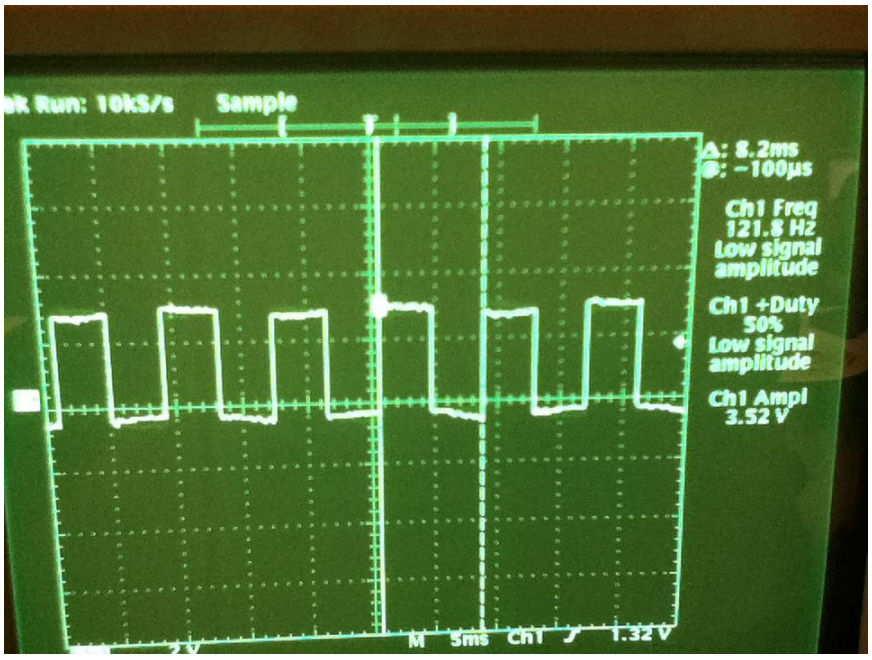

Figure 11: 50% duty cycle wave.

The function call set_DC(100,0) produced the following output. Notice the wave is output at a 9.95% duty cycle

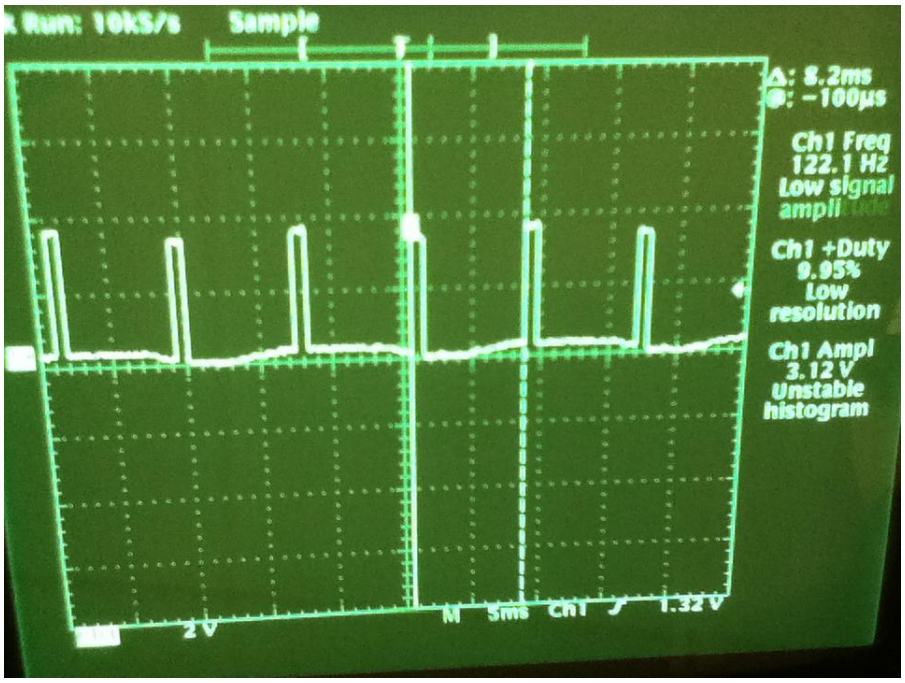

Figure 12: 10% duty cycle wave.

The function call, set_DC(183,1), produced the following output. Notice the wave is output at a 18.3% duty cycle.

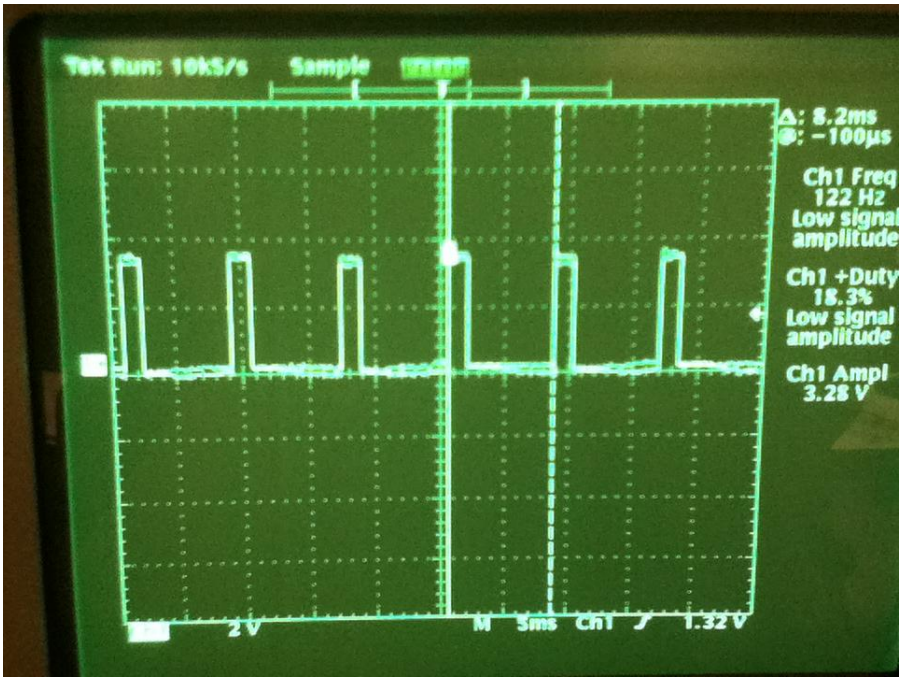

Figure 13: 18.3% duty cycle wave.

5.3 PID Testing

The PID testing was mostly PID tuning. To do this, the KP, KI, and KD variables were changed by trial and error until the robot remained upright and balanced. The microcontroller was connected to a computer and the real-time debugger was used to change the KP, KI, and KD terms. The KP term was tuned first; the KI and KD terms were then tuned to fine tune the control of the robot. There were too many unknowns in the system to be able to find an elegant mathematical solution to the control algorithm, so the guess and check method was used. After much trial and error the values KP=100, KI=30, and KD=50 gave the best results. With these values the robot was able to stay upright and balanced.

6 Conclusion

The robot is able to stay upright and balanced for as long as the battery has power. It is also fairly good at recovering from disturbances. The performance and functionality of the design were adequate for the specified requirements. There are a few things that could be improved in this design. The PID loop that was used is not self-calibrating. As such it will be subject to component changes over time. A self-calibrating PID loop, or possibly a different control algorithm would be more stable. As it is now the user will have to manually tune the PID loop if the characteristics of components change. Sometimes the robot moves around some, encoders could be attached to the wheels and incorporated into the design to control the robot to have it stay in the same position. Overall this design is functional.

Sources

[1] S. Colton. “The Balance Filter.” Internet: http://web.mit.edu/scolton/www/filter.pdf , June 25, 2007 [April 23, 2012].

[2] TJK Electronics. “Balanduino-Balancing Robot Kit” Internet: http://www.balanduino.net , [April 24, 2012].

[3] G. F. Franklin, J. D. Powell, and M. L. Workman, *Digital Control of Dynamic Systems*, 3rd ed. Boston, MA: Addison-Wesley,1997.

Appendix

Software Code:

Main.c:

```
#include <stm32f10x.h> //This header file has all of the address for all the registers of the
                                  //board, we utilize this in configuring and writing out to
registers.
#include "IIC.h"
#include "IMU.h"
#include "pwm.h"
#include "math.h" //This is for arctan

void SystemClock_init()
{
   unsigned int temp;

   RCC->CR=0x00030080;    //Enable the HSE (High Speed External Oscillator)

    temp=RCC->CFGR;        //Read
   temp=temp&0xFFFFFFFC;  //Modify
   temp=temp|0x1;         //Modify
   RCC->CFGR=temp;     //Write; Set the uC to use the 8MHz external oscillator as the system
clock
}

void SystemInit()
{
   SystemClock_init();
   delay(80);
   IMU_Init(); //initialze the IMU
   PWMInit(8000); //Set the PWM to run at 1KHz
   tim2init(16000);//set the timer to expire ever 2 ms
}

float MOTORRANGE = 500; //Value for clamping the motor range so too much current is never drawn.
#define DEADZONE  200 //The motors don't start until they see a 20% DC wave

float KI = 30;    //PID KI TERM
float KP = 100; //PID KP TERM
float KD = 50; //PID KD TERM

float ANGLE_OFFSET = -10; //OFFSET, this is changed to prevent the robot from favoring one side
float angluar_vel;
float speed = 0;
float tempspeed;
float angle = 0;
float dt = .002;

int main()
{
       /*//This is a function for testing the motors.
   int test =0;
   int dir = 0;
   while(1)
   {
      setDC(test, dir);
   }
   */
```

```
   while(1)
   {
      while((TIM2->SR&0x1)==0);//wait for the timer to count down
      TIM2->SR=0; //reset the status register

      angle = -getCurAngle(&angluar_vel) - ANGLE_OFFSET; //get the angle

      speed = KP * angle + KD*angluar_vel + KI*speed; //pid calculation

      if(speed>0)
      {
         if (speed>MOTORRANGE)
         {
            speed=MOTORRANGE; //Clamp the speed so the motor doesn't draw too much current
         }

         setDC(DEADZONE +  speed, 0);
       }

      else
      {
         tempspeed=-speed; //use tempspeed, so the speed variable is not messed
                           //with. The PID requires the last speed to work.

         if(tempspeed>MOTORRANGE)
         {
            speed = -MOTORRANGE;
            tempspeed=MOTORRANGE; //Clamp the speed so the motor doesn't draw too much current
         }
             setDC(DEADZONE+ tempspeed, 1);

       }

     }
}
```

IIC.h:

```
#include <stm32f10x.h>

#ifndef IIC_H_
#define IIC_H_

#define MPU6050_PWR  0x6B   // R/W
#define MPU6050_I2C_ADDRESS 0x68
#define READ 1
#define WRITE 0

#define START 0x100
#define STOP 0x200
#define I2C I2C1

#define XACCEL 0x3B
#define YACCEL 0x3D
#define ZACCEL 0x3F

#define XGYRO 0x43
```

```
#define YGYRO 0x45
#define ZGYRO 0x47

#define TIMEOUT 500 //at 8 mhz 500 loop cycles ~ .34 ms

void I2C_Sub_Init(void);
void I2C_Init(void); //done

unsigned char SendSlaveAddress(unsigned char addr, unsigned char type, unsigned char data);
void GenStartStop(unsigned int comm);
void setAck(unsigned char t);
void I2C_delay(unsigned short cnt);
unsigned char I2C_write(unsigned char i2c_address, unsigned char Reg_addr, unsigned char data);
unsigned char I2C_read(unsigned char i2c_address, unsigned char Reg_addr, unsigned char* data);
void ClearStop(void);
short GetValue(unsigned char );
void  resetIMU(void);
void setError(void);
void resetError(void);
void delay(unsigned short ms);
void setForward(void);
void setBackward(void);

#endif
```

IIC.c:

```
#include "IIC.h"

#include <stm32f10x.h>
volatile int PCP9 __attribute__((at(0x422201a4)));     //Bit band address for port c, pin 1
volatile int PCP10 __attribute__((at(0x422201a8)));      ////Bit band address for port c, pin 2
volatile int PCP11 __attribute__((at(0x422201ac)));      //Bit band address for port c, pin 3

short GetValue(unsigned char address)
{
    unsigned char  high, low;
    short toReturn;

   while(I2C_read(MPU6050_I2C_ADDRESS,address, &high) ==1
         && I2C_read(MPU6050_I2C_ADDRESS, address+1, &low)==1
         && I2C_write(MPU6050_I2C_ADDRESS, 0x6A, 0x1) ==1)
   {
      resetIMU();
   }
    toReturn = high;
    toReturn = toReturn<<8;
    toReturn |= low;

    return toReturn;
}


void I2C_Init(void)
{
   unsigned int temp;

   RCC->APB1ENR |= 0x200000; //enable the clock for iic1
   RCC->APB2ENR |= 0x8; //enable clock on GPIOB this may be problamatic for touch screen

   //need to enable afio for iic
   //AFIO->
```

```
   AFIO ->MAPR |= 0x2;

   temp = GPIOB->CRL; //= 0xFF444444;  //enable for alternate output open drain
   temp &= 0x00FFFFFF;
   temp |= 0xFF000000;
   GPIOB->CRL = temp;
   I2C_Sub_Init();

   RCC->APB2ENR |= 0x10;
   GPIOC->CRH =0x44433334;//ENABLE OUTPUT ON PORT C
   PCP9 = 0; PCP10 = 1;PCP11 = 0;//turn the LEDs OFF initially


}

void I2C_Sub_Init()
{
   I2C->CR2 = 0x008;
   I2C->CCR |= 0x02f;
   I2C->OAR1 = 0x4030;
   I2C->TRISE = 0x25;
   I2C->CR1 = 0x0401;
}

void I2C_delay(unsigned short cnt)
{
   int i;
   for(i =0; i<cnt; ++i);
}

//sets the ack bit of cr1 to the value sent;
void setAck(unsigned char t)
{
   if(t==0)
   {
      I2C1->CR1 &=0xFBFF;
   }
   else
   {
      I2C1->CR1 |=0x400;
   }

}

//use this with START or STOP to generate stop or start events
void GenStartStop(unsigned int comm)
{
   I2C1->CR1 |= comm;
}

void ClearStop()
{
   I2C->CR1 &= 0xFDFF;
}


//sends a 7 bit address and sets the dirrection of transmission.

/*
send start sb bit is set  sr1 bit 0
read sr1 and write slave address to DR
```

```
send address addr is set sr1 bit 1
read SR1 and SR2

see page 729 of the data sheet
*/
unsigned char SendSlaveAddress(unsigned char addr, unsigned char dir, unsigned char data)
{
   unsigned short timeout =0;

   if(dir == READ)
   {
      addr <<=1;
      addr |= READ;
   }
   else
   {
      addr<<=1;
      addr|=WRITE;
   }

      GenStartStop(START);
   //wait for sb to be set the read that gets us past should clear it

   //EV5
   while((I2C->SR1 & 0x1) == 0x0)
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }
   I2C->DR = addr; //write the address


   //wait for addr bit to go high meaning we got an ACK
   //EV6
   timeout = 0;
   while((I2C->SR1 & 0x2)==0)
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }

   I2C->SR1;
   I2C->SR2;
   return 0;
}
/* procedure for clearing a stuck buss
1) Master tries to assert a Logic 1 on the SDA line
2) Master still sees a Logic 0 and then generates a clock
pulse on SCL (1-0-1 transition)
3) Master examines SDA. If SDA = 0, go to Step 2; if
SDA = 1, go to Step 4
4) Generate a STOP condition
*/


void resetIMU()
{
   int IICSWRESET = 0x8000;
```

```
   unsigned int temp;

   setError(); //turn on the light

   I2C->CR1 |= IICSWRESET; //reset the IIC

   //setup the pins so that we can change them
   temp = GPIOB->CRL;
   temp = temp & 0x00FFFFFF;
   temp = temp | 0x33000000;
   GPIOB->CRL = temp;

   //reset the gpio pins for the IIC
   GPIOB->ODR &=0x3F;
   temp = GPIOB->CRL;
   temp = temp & 0x00FFFFFF;
   temp = temp | 0xFF000000;
   GPIOB->CRL = temp;

   //reinitialize the iic
   I2C->CR1 = 0 ;//clear swrst and set pe
   I2C_Sub_Init();
   I2C_delay(1667);


   resetError();
}

unsigned char I2C_write(unsigned char i2c_address, unsigned char Reg_addr, unsigned char data)
{
   unsigned short timeout =0;
   if(SendSlaveAddress(i2c_address, WRITE, data))
   {
         return 1;
   }

   //I2C->SR2; //dummy read to clear addr bit and start transmission

   //ev8_1
   while((I2C->SR1& 0x80) ==0x0)
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }
   I2C->DR = Reg_addr;

   //ev8 if this doesnt work change to check the btf not txe
   timeout = 0;
   while((I2C->SR1& 0x80) !=0x80)
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }
   I2C->DR = data;

   //EV8_2
```

```
   timeout = 0;
   while((I2C->SR1 & 0x84) !=0x84) //wait for txe and BTF bits to be set
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }
   GenStartStop(STOP);
   return 0;
}


unsigned char I2C_read(unsigned char i2c_address, unsigned char Reg_addr, unsigned char* data)
{
   unsigned short timeout=0;
   if(SendSlaveAddress(i2c_address, WRITE, *data))
   {
      return 1;

   }

   //ev8_1
   while((I2C->SR1& 0x80) ==0x0)
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }
   I2C->DR = Reg_addr;

   //ev8 if this doesnt work change to check the btf not txe
         timeout =0;
      while((I2C->SR1& 0x84) !=0x80)
     {
         if(timeout++>TIMEOUT)
         {
            return 1;
         }
      }


      if(SendSlaveAddress(i2c_address, READ, *data))
      {
         return 1;
      }

   //EV6_1
      setAck(0);
      I2C->SR1;
      I2C->SR2;
      GenStartStop(STOP);

   while((I2C->SR1 & 0x40)==0)//wait for RXNE to go high
   {
      if(timeout++>TIMEOUT)
      {
         return 1;
      }
   }
```

```
      *data = I2C->DR;
   return 0;
}

void setForward() //update LEDS
{
   PCP9=1;
   PCP11 = 0;
}

void setBackward() //update LEDS
{
   PCP9 = 0;
   PCP11 = 1;
}

void delay(unsigned short ms) //Delay function
{
   int t;
   for(t = 0; t<1333*ms; ++t);
}
```

IMU.h:

```
#ifndef IMU_H_
#define IMU_H_

typedef struct
{
float CurAngle;
float loopTime;
int Tim2Reload;
float tau;
} Cont;

float getCurAngle(float*);
float getAccelAngle(void);
float getXAngularVelocity(void);
void tim2init(int count);

void IMU_Init(void);
#define LOOP_TIME_CONSTANT .002
#define TAU 0.075
#define A TAU/(TAU+LOOP_TIME_CONSTANT)
#endif
```

IMU.c:

```
#include "IMU.h"
#include "IIC.h"
#include "math.h"

float CurAngle=0;
//float loopTime = 2;//1
int Tim2Reload = 7999;
//float tau=0.075;
 volatile int PCP12 __attribute__((at(0x422201b0)));     //Bit band address for port c, pin 4
```

```
//light controlls for iic errors
void setError()
{
   PCP12 = 1;
}

void resetError()
{
   PCP12 = 0;
}


void IMU_Init()
{
   I2C_Init();
   I2C_write(MPU6050_I2C_ADDRESS,MPU6050_PWR,0x0); // wake up the microcontrollerdisable temp and
use pll with y gyro as clock
}

//returns the angle using arctangent of z and y
float getAccelAngle()
{
   short y,z;
   z = GetValue(XACCEL);
   y = GetValue(ZACCEL);

   return (atan2(z,y)*57.2957795);
}

//reads the x gyroscope

float getXAngularVelocity()
{
   short rawX;
   float toReturn;

   rawX = GetValue(YGYRO);
   toReturn = rawX/16.4;
   return toReturn;
}

/****************************************
complementary filter based on information
from http://robottini.altervista.org/kalman-filter-vs-complementary-filter/filters1
****************************************/
float getCurAngle(float * XAngularVelocity)
{
   float accelAngle = getAccelAngle();
   *XAngularVelocity = getXAngularVelocity();


   CurAngle= A* (CurAngle + *XAngularVelocity * LOOP_TIME_CONSTANT) + (1-A) * (accelAngle);

   return CurAngle;
}



//initialize tim2 used to run the loop that constantly updates position
void tim2init(int count)
{
```

```
   RCC->APB1ENR |= 1; //enable clock on the timer
   //NVIC_EnableIRQ(28); //enable the interrupt for tim2 at the core.
   TIM2->ARR = count;   //set the auto reset register
   TIM2->CNT = count;   //set the count of the timer.
   //TIM2->DIER |=1;       //enable interupts for the timer.
   TIM2->CR1 |= 0x15;   //start the timer, count down.
}
```

PWM.h:

```
/***********************************
uses timer 3 to output pwm on pin 5 of port b
***********************************/

#ifndef PWM_H_
#define PWM_H_
   void PWMInit(int count);
   void setDC(unsigned short DC, unsigned int);
   void setMotorOff(void);
#endif
```

PWM.c:

```
#include "pwm.h"
#include <stm32f10x.h>

volatile int back __attribute__((at(0x422181a8))); //back uses PB10
volatile int forward __attribute__((at(0x422181ac)));//forward uses PB11

//initialize tim3 for PWM
void PWMInit(int count)
{
   unsigned int temp; //configuration variable

   RCC->APB1ENR |= 2;   //enable clock on the timer

   RCC->APB2ENR |= 0xD; //Enable GPIO port A and Port B

   AFIO->MAPR |=0x800; //REMAP the TIMER So PWM uses PB5
   temp=GPIOB->CRL;
   temp=temp&0xFF0FFFFF;
   temp=temp|0x00B00000;
   GPIOB->CRL=temp;

   temp=GPIOB->CRH;
   temp=temp&0xFFFF00FF; //SETUP PB10 and 11 for the direction bits.
   temp=temp|0x00003300;
   GPIOB->CRH=temp;

   TIM3->ARR = count;   //set the auto reset register
   TIM3->CNT = count;   //set the count of the timer.
   TIM3->CCR2=1830;  //PWM duty cycle
   TIM3->CCER=0x10; //OC1 signal is output on the corresponding output pin
   TIM3->CCMR1=0x6000;
   TIM3->CR1 |= 0x95;   //start the timer, count down.

}

int last = 0; //keep track of the last dirrection to save time.
void setDC(unsigned short DC, unsigned int dirrection)
```

```
{
   unsigned int tim3ARR;
   if (last != dirrection) //swap dirrection if it has changed
   {
      if(dirrection)
      {
         back = 0;
         forward = 1; //set the direction to forward
      }
      else
      {

         forward = 0;
         back = 1;  //set the direction to backward
      }
      last = dirrection; //save the previous dirrection
   }

   tim3ARR=TIM3->ARR;               //Read the auto load register value
   TIM3->CCR2=tim3ARR*DC/1000;    //Set the DC based on TIM3's ARR

}
```