University of Florida

EEL5666

Intelligent Machine Design Lab

Dr. A. Arroyo

Equilibrium

The Autonomous Bi-wheel Robot

Christopher A. Taylor

7934-5340

I. Table of Contents

Abstract……....3

Executive Summary……..……4

Introduction……..……5

Integrated System……6

Mobile Platform……...7

Actuation……..8

Sensors……...……..9

Behaviors……...10

Results……11

Conclusion……12

Documentation……...13

Appendix……14

II. Abstract

Equilibrium (EQ) encounters problems similar to those of the inverted pendulum. Equilibrium attempts to stand vertically and keep its balance while using only two wheels. The main obstacle is attempting to fight gravity, as this always wants to pull him down. This is accomplished with the use of an accelerometer and an RC gyroscope. Equilibrium is completely autonomous, with all required systems onboard.

III. Executive Summary

Equilibrium is an autonomous bi-wheel robot that balances in a similar fashion to the Segway®. It will continue to do so until it is turned off or runs into something. Obstacle avoidance was not implemented into the system, as time did not allow.

The "brains" of the whole operation are an Atmel ATMega128 microcontroller, using the M128B development board from BDMicro. Port A is used for the LCD output, Port B is used for PWM outputs, Ports C, D, &E are not used, and port F is used for the A/D conversion of the accelerometer.

The LCD screen is used to give feedback as to the condition of the robot. It displays real-time information on PWM outputs, and current deviation from center-point. This information can be used for tweaking purposes.

IV. Introduction

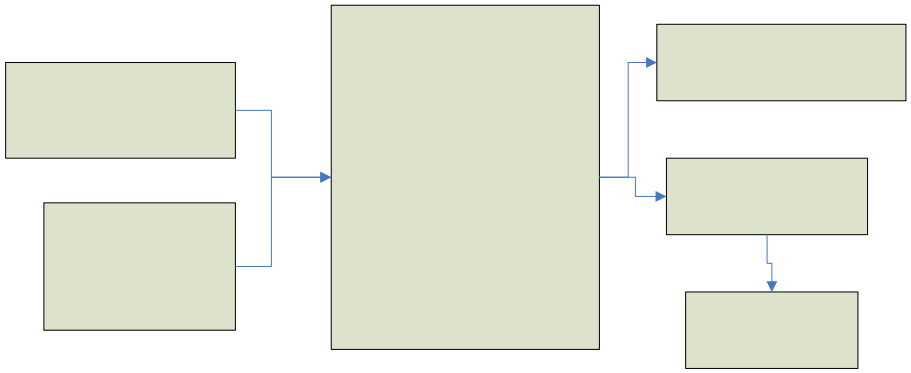
The inverted pendulum has been a topic of interest since about 1900 when it was discovered by E. Wiechert of Göttingen that an inverted pendulum system could be used as a seismograph system (USGS.gov). Mastering the inverted pendulum is an issue that is still being tackled today by engineers.

Equilibrium demonstrates the simplest form of the inverted pendulum, as only one axis needs to be adjusted to compensate for tilt. Using an accelerometer and a gyroscope, EQ will have the capability to traverse on almost any terrain. This is because the accelerometer is an absolute tilt device, which means it detects its angle compared to the earth.

In a robot such as this, software control is a large part of the workings. Without proper controls, the robot is practically useless. If proper controls are implemented, however, terrain and other harsh conditions should not interfere with proper operation.

V. Integrated System

Equilibrium's entire system consists of an Atmel AVR Board, and LCD for output, motor drivers, DC Motors, and an accelerometer and gyroscope as inputs. A chart is shown below.



The AVR Board is a BDMicro M128B board with an Atmel ATMega128 microprocessor onboard. The table below shows some of it's features:

- 128k Flash memory (program space)
- 4k EEPROM
- 8 Channel A/D
- Dual UARTs
- I^2C Interface
- 48 IO pins
- ISP
- 2-8 bit timers, 1-16 bit timer
- 6 PWM Channels
- 16 MHz

Accelerome

This board was definitely overkill for the project and most others. However, it makes a great board for prototyping, and for many other projects, as it contains everything you could possibly need.

An LCD is mounted at the bottom of Equilibrium. The purpose is mostly for debugging purposes. The LCD can display data from any of the sensors installed. It is a great addition to any robot. For me, it was useful in displaying current motor speeds, and its current angle. This turned out to be immensely useful.

VI. Mobile Platform

As with an inverted pendulum, a higher center of gravity makes it far easier to balance. Think of balancing a baseball bat, which side would you try to balance? So obviously, a tall platform will be helpful in controlling Equilibrium. The platform should also be able to withstand an impact from a fall in case of a control failure. During the testing process, this is inevitable.

Since the wood is free, the platform will be cut from balsa wood using the T-Tec machine. I would expect that a polycarbonate body would perform better, however in the interest of time and money, this is not necessary. The body was designed in AutoCAD and exported to the T-Tec and then printed and assembled using lots of wood glue, hot glue, tape and wire ties. A sturdier assembly is definitely recommended.

A large platform means other things have to be considered. Such as motors torque and speed, and battery power. A heavier platform puts more strain on motors and therefore on batteries.

VII. Actuation

To effectively battle gravity, the motors will need to have high torque, in order to switch directions abruptly. The motors I will be using are made by ITT Automotive, and at 12V DC are rated at 180mA with 75 oz/in torque @ 115RPM. The motor is rated up to 24V DC. Although the motors are large, they will get the job done properly. As I have found out the hard way, good motors are ESSENTIAL!

The 2 motors will each be driven from a 55V, 3A H-bridge motor driver from National Semiconductor. The driver chips are available for free from National Semiconductor. Although the circuit board for them has to be designed, it was simple and worth saving a little money. The circuit board ultimately ended up being done professionally, thanks to Max.

Finally, for locomotion, large 7" wheels were attached using hubs to the gear head motors. Large wheels were useful for a few reasons. First, then altered the center of gravity in my favor. Also, they gave me enough ground clearance, as the motors are a little bit large.

VIII. Sensors

Equilibrium will input 3 variables:

1. Current angle with relation to the ground
2. If he has tipped over
3. Rate of fall

Equilibrium will maintain balance primarily by the use of an accelerometer. The accelerometer measures tilt with relation to gravity. It is an absolute tilt device. More information on specifics can be found in my sensor report on the analog devices accelerometer. Limits are set in software to determine a level which the robot has no chance of recovery. These limits are set to about 30º of tilt, which is very excessive actually. This just allows me to transport it while it is on.

The rate of fall is determined by an RC Helicopter gyroscope. With this particular gyroscope however, I have no control over the signal. It is designed to automatically modify a servo pulse to correct the rate. It does not work so well for this application.

IX. Behaviors

Equilibrium will demonstrate minimal behaviors. The primary behavior is balancing on two wheels. If efficient controls were developed, further behaviors could easily be added. Some behaviors to add in the future include: random movements, obstacle avoidance, wall following, or edge detection, among other things. Once again, these cannot be implemented until the primary function of balancing itself is fluent.

X. Conclusion

Unfortunately, Equilibrium's outcome was not so good. After having uncountable problems, things started to work out in the end. Until the software time came. This is when the motors started being used. I went through 2 pairs of motors. The gears continually stripped out. This cost me lots of valuable time, probably over a week in total, and an extra $120. And in the end, I ended up using what I first purchased. I would definitely recommend good motors to begin with. Speed is not greatly important; however I would recommend a substantial amount of torque. Metal gears are also a necessity.

Another recommendation would be to definitely get the robot platform complete early. That way, much more time could be spent on coding. Software is the main part of the project, and probably the most tedious.

All in all, despite the failure, a plethora of knowledge was gained from this experience. I feel that I learned more during this semester than all semesters combined. For future projects though, I will definitely do thorough research before hand, to help me get an idea of what problems I may incur, and what to do to remedy them.

XI. Documentation

Credits

Dr. Arroyo, Dr. Schwartz, Luis, William, and Max for a great experience and immense help in lab.

Greg Beckham for adapting LCD code by Max Billingsly, and various other issues encountered throughout the making of Equilibrium.

Max for making the motor driver circuit.

Chris Staymates in a last minute attempt to save my robot by helping me greatly with interrupts and input capture.

A few websites with similar projects:
The nBot: http://geology.heroy.smu.edu/~dpa-www/robo/nbot/
Ted Larson's robot: http://www.tedlarson.com/robots/balancingbot.htm
These projects helped guide me.

Parts

Part	Item	Cost
Processor	ATMEL ATMega128	Came w/ board
Board	BDMicro M128 Rev. B	Unassembled: $99
Batteries	2x7.2V 3000mAh & 20x1.2V 2250mAh, with chargers	$80
Programmer	AVRISP programmer from Digikey. Part #ATAVRISP-ND	$29.00
Accelerometer	Analog Devices ADXL203 Accelerometer.	FREE!
Motors	6x Hsiang Neng Gearhead Motors. 138 RPM, 156.6 oz-in torque. Available at Lynxmotion. Part #GHM-05 NOT RECOMMENDED 2xITT Automotive 12V DC, 180mA, 115 RPM, torque=75 oz/in. From Herbach	6x$16.50 2x$25.00
Motor Drivers	LMD18200 from National Semiconductor. Using my own circuit.	FREE!
Motor Driver	Driver board by Max	$15

Board		
Wheels	7" Plastic wheels from Herbach	$6/pair
Hardware	Misc. hinges, screws, etc from Lowes.	Est.$20.00
Shipping	Estimated shipping costs	$50
Time	1,000,000 hours of free time	Priceless
Screwups	Lots of screwups	$50 + hours
Total	Lots of Stuff	$498 + a semester of my life

XIII. Appendix

LCD Code

Lcd.h

```
/*
* lcd.h
*
* Author: Max Billingsley
* Adapted by: Greg Beckham
*/
#include <avr/io.h>
#include <avr/signal.h>
#include <inttypes.h>


#define LCD_PORT PORTA
#define LCD_DDR DDRA
#define ENABLE 0x08
/* function prototypes */
void lcd_set_ddr(void);
void lcd_init(void);
void lcd_delay(void);
void lcd_send_str(char *s);
void lcd_send_byte(uint8_t val);
void lcd_send_command(uint8_t val);
```

lcd.c

```
/*
* lcd.c
*
* Author: Max Billingsley
* Adapted by: Greg Beckham
*/
/*
* LCD_PORT1  = RS
* LCD_PORT2  = R/W
* LCD_PORT3  = EN
* LCD_PORT4  = DB4
* LCD_PORT5  = DB5
* LCD_PORT6 = DB6
* LCD_PORT7  = DB7
*
* RS: Register Select:
*
* 0 - Command Register
```

```
* 1 - Data Register
*
*/
#include "lcd.h"
/* entry point */

void lcd_init(void)
{
lcd_send_command(0x83);
lcd_send_command(0x83);
lcd_send_command(0x83);
lcd_send_command(0x82);
lcd_send_command(0x82);
lcd_send_command(0x8c);
lcd_send_command(0x80);
lcd_send_command(0x0f);//0f
lcd_send_command(0x00);
lcd_send_command(0x01);
}

void lcd_set_ddr(void)
{
LCD_DDR = 0xff;
}

void lcd_delay(void)
{
uint16_t i;
for(i = 0;i<2000;i++){}
}

void lcd_send_str(char *s)
{
while (*s) lcd_send_byte(*s++);
}

void lcd_send_byte(uint8_t val)
{
uint8_t temp = val;
val &= 0xf0;
val |= 0x02;
LCD_PORT = val;
lcd_delay();
LCD_PORT |= ENABLE;
LCD_PORT &= ~ENABLE;
temp <<= 4;
```

```
temp |= 0x02;
LCD_PORT = temp;
lcd_delay();
LCD_PORT |= ENABLE;
LCD_PORT &= ~ENABLE;
}

void lcd_send_command(uint8_t val)
{
uint8_t temp = val;
val &= 0xf0;
LCD_PORT = val;
lcd_delay();
LCD_PORT |= ENABLE;
LCD_PORT &= ~ENABLE;
temp <<= 4;
LCD_PORT = temp;
lcd_delay();
LCD_PORT |= ENABLE;
LCD_PORT &= ~ENABLE;
lcd_delay();
}
```

Adc.h

```
/*
 * $Id: adc.h,v 1.1 2003/12/11 01:35:00 bsd Exp $
 */
#include <avr/io.h>
#include <stdio.h>

#ifndef __adc_h__
#define __adc_h__

void    adc_init(void);

void    adc_chsel(uint8_t channel);

void    adc_wait(void);

void    adc_start(void);

uint16_t adc_read(void);

uint16_t adc_readn(uint8_t channel, uint8_t n);

#endif
```

Adc.c

```
/*
 * $Id: adc.c,v 1.2 2003/12/11 02:15:39 bsd Exp $
 */

/*
 * ATmega128 A/D Converter utility routines
 */

#include "adc.h"


/*
 * adc_init() - initialize A/D converter
 *
 * Initialize A/D converter to free running, start conversion, use
 * internal 5.0V reference, pre-scale ADC clock to 125 kHz (assuming
 * 16 MHz MCU clock)
 */
void adc_init(void)
{
  /* configure ADC port (PORTF) as input */
  DDRF  = 0x00;
  PORTF = 0x00;

  ADMUX = BV(REFS0);
  ADCSR = BV(ADEN)|BV(ADSC)|BV(ADFR) |
BV(ADPS2)|BV(ADPS1)|BV(ADPS0);
}


/*
 * adc_chsel() - A/D Channel Select
 *
 * Select the specified A/D channel for the next conversion
 */
void adc_chsel(uint8_t channel)
{
  /* select channel */
  ADMUX = (ADMUX & 0xe0) | (channel & 0x07);
}


/*
 * adc_wait() - A/D Wait for conversion
 *
```

```
 * Wait for conversion complete.
 */
void adc_wait(void)
{
  /* wait for last conversion to complete */
  while ((ADCSR & BV(ADIF)) == 0)
    ;
}


/*
 * adc_start() - A/D start conversion
 *
 * Start an A/D conversion on the selected channel
 */
void adc_start(void)
{
  /* clear conversion, start another conversion */
  ADCSR |= BV(ADIF);
}


/*
 * adc_read() - A/D Converter - read channel
 *
 * Read the currently selected A/D Converter channel.
 */
uint16_t adc_read(void)
{
  return ADC;
}


/*
 * adc_readn() - A/D Converter, read multiple times and average
 *
 * Read the specified A/D channel 'n' times and return the average of
 * the samples
 */
uint16_t adc_readn(uint8_t channel, uint8_t n)
{
  uint16_t t;
  uint8_t i;

  adc_chsel(channel);
  adc_start();
```

```
  adc_wait();

  adc_start();

  /* sample selected channel n times, take the average */
  t = 0;
  for (i=0; i<n; i++) {
    adc_wait();
    t += adc_read();
    adc_start();
  }

  /* return the average of n samples */
  return t / n;
}
```

Timer.h

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#include <inttypes.h>

volatile uint16_t ms_count;

void ms_sleep(uint16_t ms);
void init_timer(void);
```

Timer.c

```
#include "timer.h"
/*
 * ms_sleep() - delay for specified number of milliseconds
 */
void ms_sleep(uint16_t ms)
{
  TCNT0  = 0;
  ms_count = 0;
  while (ms_count != ms)
    ;
}


/*
 * millisecond counter interrupt vector
 */
```

```
SIGNAL(SIG_OUTPUT_COMPARE0)
{
 ms_count++;
}


/*
 * initialize timer 0 to use the real time clock crystal connected to
 * TOSC1 and TOSC2 to generate a near 1 ms interrupt source
 */
void init_timer(void)
{
 /*
  * Initialize timer0 to use the 32.768 kHz real-time clock crystal
  * attached to TOSC1 & 2.  Enable output compare interrupt and set
  * the output compare register to 32 which will cause an interrupt
  * to be generated every 0.9765625 milliseconds - close enough to a
  * millisecond.
  */
 TIFR  |= BV(OCIE0)|BV(TOIE0);
 TIMSK |= BV(OCIE0);    /* enable output compare interrupt */
 TIMSK &= ~BV(TOIE0);   /* disable overflow interrupt */
 ASSR  |= BV(AS0);      /* use asynchronous clock source */
 TCNT0  = 0;
 OCR0   = 32;           /* match in 0.9765625 ms */
 TCCR0  = BV(WGM01) | BV(CS00); /* CTC, no prescale */
 while (ASSR & 0x07)
  ;
 TIFR  |= BV(OCIE0)|BV(TOIE0);
}
```

Main.c

```
#include "adc.h"
#include "timer.h"
#include "LCD.h"
#include <math.h>
#include <avr/pgmspace.h>
#include <stdio.h>
#include <inttypes.h>
#include <avr/delay.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <avr/io.h>


//Constants
```

```
static int STOP = 128;                  //Determines stop value
static int MAXREV = 0;                          //Full speed reverse //40 with gyro
static int MAXFWD = 255;                //Full speed forward //215 with gyro

int main(void)
{
PORTB=0x00;
DDRB=0xC0;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 62.500 kHz
// Mode: Fast PWM top=00FFh
// OC1A output: Non-Inv.
// OC1B output: Non-Inv.
// OC1C output: Non-Inv.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0xA1;                                    //0xA9 with gyro
TCCR1B=0x0A;                                    //0x0C with gyro
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
OCR1CH=0x00;
OCR1CL=0x00;

  uint16_t angle, rate, speed, old_speed;
  int i, center=0, centerR = 0 ,offset;
  int old_offset;
  float Kangle = 1.4, Krate = .06;
  init_timer();
  sei();
  adc_init();

  lcd_set_ddr();
  lcd_init();
  fdevopen(lcd_send_byte,NULL,0);

//This loop calibrates the accelerometer to a center value.  It takes 30 samples within
//3 seconds and takes the average.  This is the center point.
i=3;                                                    //how many seconds to center
for
while(i != 0)
```

```
 {
 lcd_send_command(00);
 lcd_send_command(01);
 printf("Calibrating: %d",i);
 ms_sleep(1000);
 center =  center + adc_readn(0,10);
 centerR = centerR + adc_readn(1,10);
 i--;
 }
center = center / 3;
centerR = centerR / 3;
old_speed=center;                                   //"old_angle" is needed for balancing
algorithm
old_offset = offset;

  while (1) {
        ms_sleep(20);                   //Samples the acceleromter 10 times per second
        angle = adc_readn(0, 1);                //sample channel 0 x times, take average
        offset = center - angle;                //calculate the offset of the robot from center

        rate = adc_readn(1,1);

//      if (offset < 15 && offset > -15) {
//              mult = 3.5;
//      }


        speed = (offset * mult) + 128;//Algorithm to convert angle into PWM value -
assumes an A/D differnce of no more than 50 values
                                                                    //For full range, use a
multiplier of 2.54
        speed = (.78 * speed) + (.22 * old_speed);

//offset*K(offset) + Rate*K(rate)
/*                                          This was anotehr algorithm that was tested.
Did not work either
        if (offset < 15 && offset > -15) {
                Kangle = 4;
        }

        if (offset > 0) {
                speed = ((offset * Kangle) + 128) + (rate  * Krate);
        }

        if (offset < 0) {
                speed = ((offset * Kangle) + 128) + (rate * Krate);
```

```
                if (speed > 128) {
                        speed = 0;
                }
        }
*/

        if (((offset > 0) && (old_offset > 0)) && (offset > old_offset)) {              //active
code to determine if the robot has caught up with itself
                speed += speed * .25;
                                        //if it hasn't, speed up or slow down accordingly
        }

        if (((offset < 0) && (old_offset < 0)) && (offset < old_offset)) {
                speed -= speed * .25;
        }


        if (speed < MAXREV) {                   //The value of diff should not get
below minimum, but just in case it does
                speed = MAXREV;
        }
        if (speed > MAXFWD) {                   //Just in case diff goes above the
maximum value.
                speed = MAXFWD;
        }

        lcd_send_command(00);
   lcd_send_command(01);
   printf("Speed: %3d | Offset: %3d", speed, offset);

        OCR1A = speed;
        OCR1B = speed;
        OCR1C = speed;

        old_speed = speed;
        old_offset = offset;


        }
}
```