Spring | 12

EE 396

Micromouse

Final Report

робот

Professor Tep Dobry (Advisor)

Darcy Bibb (Team Lead – Software/Hardware Engineer)

Brylian Foronda (Software Engineer)

Ryan Zukeran (Software Engineer)

Ethan K. Hottendorf Jr. (Hardware Engineer)

University of Hawai‘i at Mānoa

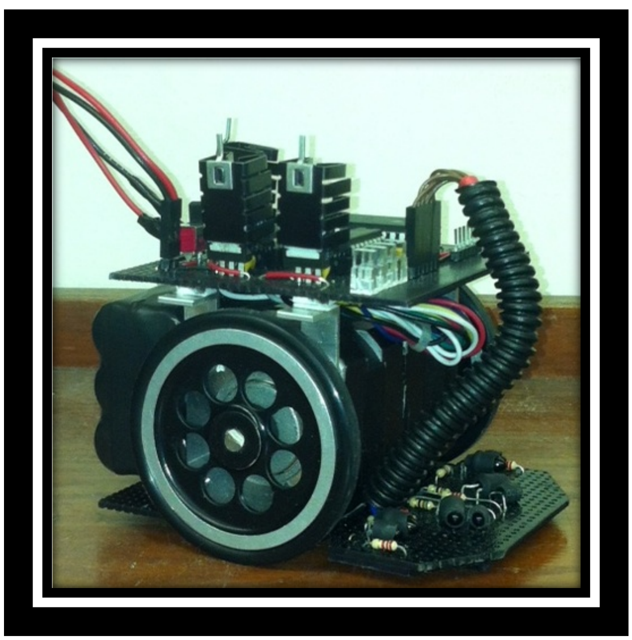
05 May,2012

Table of Contents

Abstract

A Micromouse is an autonomous robotic mouse that is designed to navigate and negotiate a path to the center of a maze within an optimal amount of time. The basic design of the mouse consists of a microcontroller, motors, power supply, and sensors, which act as the brain, legs, and eyes of the mouse. We have constructed a Micromouse utilizing a dsPIC30F microcontroller, unipolar stepper motors, and infrared side sensors. Using this platform, we were able to realize a proportional control system for tracking, successfully map a maze, and finally solve a maze with a modified maze flood fill solving algorithm. This report provides the design and interface of the different modules that make up the Micromouse, as well as the data that is related to the performance of the mouse.

робот



Overview

The goal of our Micromouse project is to design a self-powered autonomous robotic mouse that will navigate and negotiate its way to the center of a maze. The mouse will need to be built in accordance with “IEEE Micromouse Contest Rules ”.

Objectives

In compliance to the rules set by IEEE, the following objectives have been developed in order for our Micromouse project to succeed and to meet the expectations of an EE 396 project:

- Autonomously find the center of a 16x16 maze from a predetermined starting point and find its way back to the original starting point.
- Complete the maze in less than 10 minutes.
- The body of the mouse must fit inside a single 16cm x 16cm square
- Side mounted analog sensors must be used.

Within these rules, the objective of the project is to navigate to the center of the maze in the minimum amount of time.

Approach

In order to accomplish these goals, the Micromouse was divided into four main modules. These four modules are the chassis and motors, the electrical or power components, the electronic and signal components, and the software or programming of the microcontroller. The micromouse chassis will be constructed to house the motors and the electrical and electronic system, including the independent and mobile power supply. The electrical and power system will provide the individual components the necessary power (i.e. volts and current) that is needed in order to operate properly. The sensors will be used to detect walls and openings in the walls as it traverses around within the maze. The microcontroller will keep track of its movements and what cells it has been in. By this method, the microcontroller will be used to solve the maze by finding the center of the maze and calculate the quickest time to the center of the maze. The microcontroller will also be used to control the tracking and navigation of the mouse.

User Manual

To operate the робот Micromouse, a fully charged battery cell must be used at a voltage level of approximately 7.2 volts. This voltage level is necessary in order to provide sufficient power to the electronic circuitry, stepper motors, and IR sensors. With the battery cell securely located at the rear of the mouse, connect the battery to the робот mouse power connectors located at the right rear corner of the mouse adjacent to the ON/OFF switch. When performing this connection, be sure that the correct polarity (+ positive/ - negative) is made. The correct polarity is with the positive voltage (red) on the outer most terminal. Once this connection has been made and verified to be correctly installed, place the робот Micromouse in the center of the starting square if solving the maze is desired. The performance of the mouse will be observed if the робот Micromouse is placed in a random square. Note that this placement, however, will not solve the maze. Once it is situated in the middle of the square and facing straight ahead, slide the switch to the ON position, or to the right side of the switch. At this point, the mouse should be operating. Should it crash and stop moving at any point during the operation of the mouse, slide the mouse to the OFF position (left side of the switch) as soon as possible to prevent damage to the motors. The mouse can then be placed back at the original square and turned back ON. When the use of the робот Micromouse is complete, be sure to slide the switch to the OFF position and disconnect the battery pack from the mouse.

Chassis and wheels and tires

In order to keep the design of the mouse simple and easy to work with, we initially decided to use aluminum sheet metal to construct the chassis of the mouse. This material was cheap and easy to cut and shape, making it a decent candidate for this portion. The first iteration of our mouse design did show some flaws with this material, as it was not rigid enough for abuse. Additionally, the metal amplified the vibrations of the motors to the rest of the mouse, possibly contributing to noise in sensor readings. In constructing our second mouse, we decided to using something much more rigid, but also cheap and easy to work with. Perforated prototyping boards were stiff enough for our foundation, and easy to cut to any shape we needed, so our final design incorporated this material.

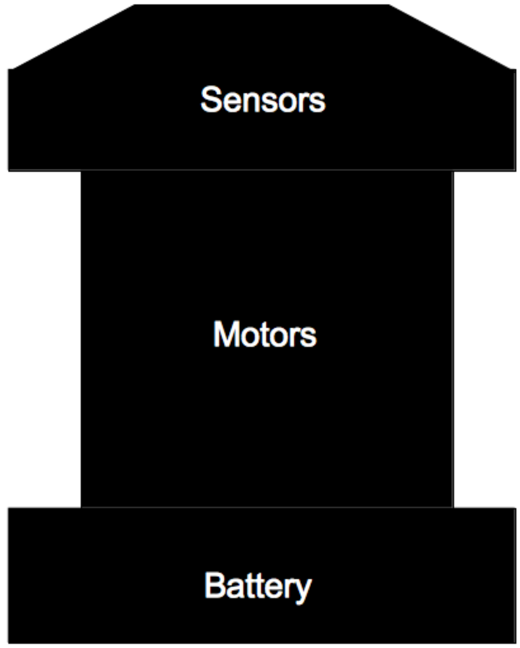


Figure 1: Chassis Layout

The final design of the chassis itself is illustrated in Figure 1 above. The center area of the chassis is reserved for mounting the motors. Cutouts in this portion allow room for the wheels of the mouse so that the chassis board does not interfere with their rotation. The front shaped portion of the chassis allows ample area for the front and side facing infrared sensors to be mounted side-by-side. We also needed to make sure that the required resistors and sensor circuit would fit in this area. Finally, the rear section of the Micromouse chassis is provided as a space for a battery to be fixed.

With the use of standard NEMA 17 stepper motors, we did not have a wide selection of wheels from which to choose. We did require wheels that fit on a 5mm shaft, and a large enough diameter to provide enough ground clearance when mounted. Machined aluminum wheels meeting these specifications were provided to us for use on our mouse.

By using the wheels provided, there are few other choices for tires other than rubber O-rings. We experimented with several different sizes of O-rings, but not so much with different rubbers or compounds. From very thin to comparably thicker O-rings, we experienced problems with traction in most cases. In the end, we found it best to use a very thick O-ring, and flatten the outer surface of the ring to create a tire with as much surface area as possible. This creates a much larger contact patch, and made a big difference in the performance of our mouse.

Motors

Most micromice are based on either stepper motors or DC motors. With the stepped nature of stepper motors making them possible to be controlled with an open-loop system, we chose to use unipolar stepper motors for ease of implementation. The exact stepper motors we utilized are KYSAN 4V motors, which we drive at 5V. These motors have a step angle of 1.8°, which means that exactly 200 steps will rotate the wheel one 360° revolution. By measuring the outer diameter of the tire surface, we know how far the mouse will travel in one wheel rotation, and thus the distance travelled per step. This makes working with stepper motors fairly simple, for the most part, without receiving any feedback from the motors.

Power Supply/Battery

Due to the fact that the PIC microcontroller and motors run at 5 V, a power source that would sufficiently provide power through 5V voltage regulators is required. The regulators used to supply the motors have a voltage drop-out of about 2 V. This means a voltage supply of at least 7 V is needed. It is also determined that the робот mouse could potentially draw a worst-case peak of about 2.3A. For a mouse to be able to run for at least 10 minutes, this means a source needs to ideally provide 7V at around 400mAh. Since batteries are not ideal, a 7.2V 2000mAh NiMH battery is chosen, which is on the conservative side of the requirements and is also inexpensive.

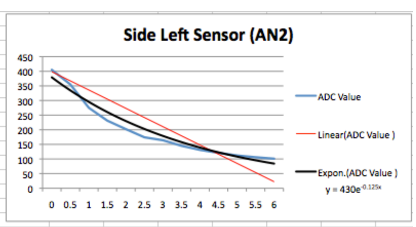
Electrical Components

In an attempt to isolate the motor power circuit from the PIC and logic power, we chose to run each stepper motor on their own voltage regulator, and use a 3rd regulator for the PIC and sensor circuit. We believe that this would minimize noise and glitches due to the motors drawing current and loading the source. Decoupling capacitors are also used throughout for this purpose. The voltage regulator used for the PIC and sensor circuit is a low drop-out 5V 1A regulator. The PIC microcontroller, ideally, will never draw more than 300mA at any time, and the sensor circuit draws a maximum of 300mA, so this regulator is sufficient in this case. We also use two separate 2V drop-out 5V 1.5A voltage regulator for each stepper motor. This was necessary, as running our motors at 5V can draw up to 1.2A, exceeding the 1A rating of our other regulators.

Sensors

When researching different options for measuring the presence and distance of walls, we found most side-sensing mice choose to use Sharp GPD120 distance sensors. We also found that these sensors have some inherent annoyances, as well as a fairly high cost. Mostly to keep costs down, we decided to use separate infrared LEDs and infrared phototransistors, which operate at the same wavelength. These matched pairs can then be used as a way to measure the distance of an object. The amount of reflected light depends on the distance of the reflecting object. The amount of reflected light can then be measured as a voltage using a phototransistor. By converting this analog voltage to digital data in the microprocessor, we now have a means to not only detect the presence of an object, but also the distance of the object. However, because the amount of light reflected from a surface depends on the cosine of the angle of reflection, and also by the square of the illuminated area, the measurement from the phototransistor will not at all be linear with distance. This is an important factor when using a linear control system, such as a proportional controller for tracking. The following figure illustrates the non-linearity of measured values with a linearly changing distance. An attempt to linearize this data is also shown.

Measured From Side Bumper	ADC Value	Distance (cm)	Linearized Function
Left Side (AN2)	405	0	x = 0.02548874671·(430 - y)
	355	0.5	
	275	1	
	231	1.5	
	202	2	
	174	2.5	
	164	3	
	145	3.5	
	131	4	
	121	4.5	
	112	5	
	106	5.5	
	101	6	



Measured From Side Bumper	ADC Value	Distance (cm)	Linearized Function
Right Side (AN3)	448	0	x = 0.02356101123·(511 - y)
	379	0.5	
	320	1	
	257	1.5	
	216	2	Wall There
	185	2.5	
	161	3	
	144	3.5	
	130	4	
	116	4.5	
	108	5	
	101	5.5	
	96	6	

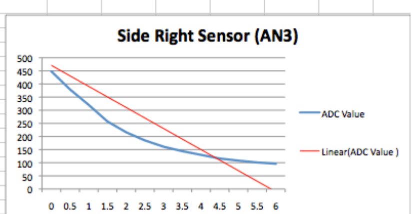


Figure 2: Linearization of Sensors

Electronic Components

Before deciding on a specific microcontroller to use for our Micromouse, we generated some requirements given the design decisions on the rest of the mouse. By using two forward and two side facing analog sensors, we would need at a minimum of four analog-to-digital converters. We also needed an ample amount of output interfaces in order to drive our stepper motors. Because we are generating the step sequence manually in the software, we would need an individual output port to each motor winding. It was desirable to have a microcontroller available in both through-hole and surface mount packages in the event that we had enough time for a tighter design. With these specifications in mind, we determined that a Microchip dsPIC30F4011 would be the best option having all the features we required, as well as other features that may make certain implementation aspects simpler. This is a 5V device with 30MIPS, fully capable of everything we want to accomplish with it.

While the outputs of the microcontroller can provide 5V, the device is not meant to provide exceptional amounts of current. As such, we chose to use a quad 1.5A Darlington array as an interface from the microcontroller to each winding of the stepper motors. This requires one array for each motor, and each Darlington switch in the array is used to switch a motor winding on or off from the microcontroller.

Software

The software portion of the project integrates all hardware components of the mouse into one. It is programmed in embedded C using Microchip's software for PIC microcontrollers, MPLAB X. The current Micromouse software consists of four main sections: sensor controls, motor controls, tracking, and the maze solving algorithm. The sensor controls are responsible for retrieving and processing the sensor readings. The motor controls handle the basic movement of the mouse such as going forward or turning. The tracking portion integrates the sensor and motor controls, specifically altering individual left and right motor speeds to ensure the mouse stays in the center of a cell. Finally, the maze solving algorithm takes care of the necessary movements needed to reach the destination goal.

Sensor Controls

The basic function of the sensor control is to handle the data received from sensor readings. Since the values received from the phototransistors are analog, it must be converted to a digital signal to be processed by the microcontroller. Utilizing the Analog-To-Digital (ADC) control register of the dsPIC30F4011f microcontroller can do this. The ADC control register uses a 16-bit register for handling conversions and stores the value as a 10-bit integer value. The typical digital to analog conversion can be calculated as shown in the equation below.

Equation 1

*Av = (Vref * Dv)/(1023)*

Av = Analog Voltage

Dv = Digtal Voltage

Vref = Reference voltage (Configured in PIC usually 5V)

The dsPIC30F4011f ADC control register port is capable of sampling and converting manually or automatically, and either sequentially or simultaneously based on the control register configuration. Knowing the capability of the microcontroller, it is configured to automatically and sequentially sample four channels, triggering an interrupt after the conversions for simplification. Rather than directly reading from the buffer for the sensor values, they are first stored to an array data structure to avoid complications. The logic diagram below illustrates the basic process of how ADC values processed, stored and retrieved.

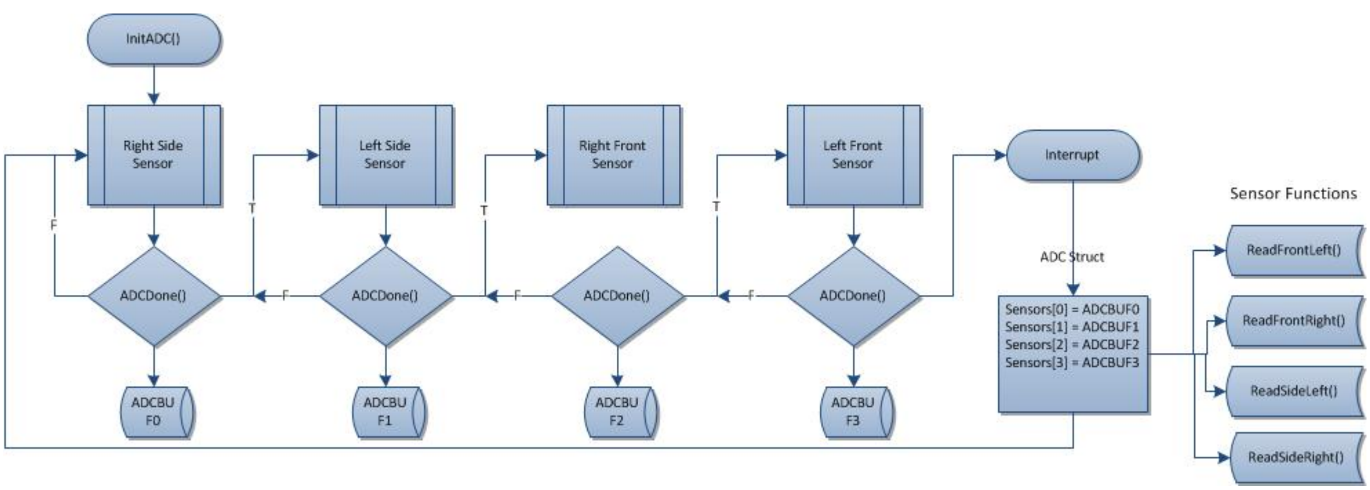


Figure 3: Analog to Digital Conversion Logic Diagram

Motor Controls

The motor control is responsible for basic movements of the mouse. For basic turning controls, the built in timer delay function was utilized since no other logic is required. Simply pulsing the motors in opposite directions does this. For moving forward on other hand, stepping the motors are handled separately by using two 16-bit timers, with the internal clock oscillator as the counter. By loading a period to these timer registers, the built in microcontroller comparator can be used to analyze the counter and period values. The greater the period values, the greater the delay between the motor pulses. The equation below can be used to obtain the period needed to pulse the motors at a desired time, with the microcontroller configured to operate at its maximum speed of approximately 30 million instructions per second (MIPS).

Equation 2

*P = t/(2*Prescaler*(1/FCY))*

P = Period To Load The Timer Register

t = Desired Time Delay Between Pulse

Prescaler = Scaling Factor For Handling Slower Time Delays

FCY = Instruction Clock Rate (29.4 Mips)

In the event where the timer registers matches the counter value, a flag is generated known as the Interrupt Service Routine (ISR). In the ISR, the motor control logic is performed, specifically pulsing the two motors a step forward and modifying the period registers based on tracking decisions if necessary. Figure 4 below, provides a basic explanation of the motor control process of moving forward.

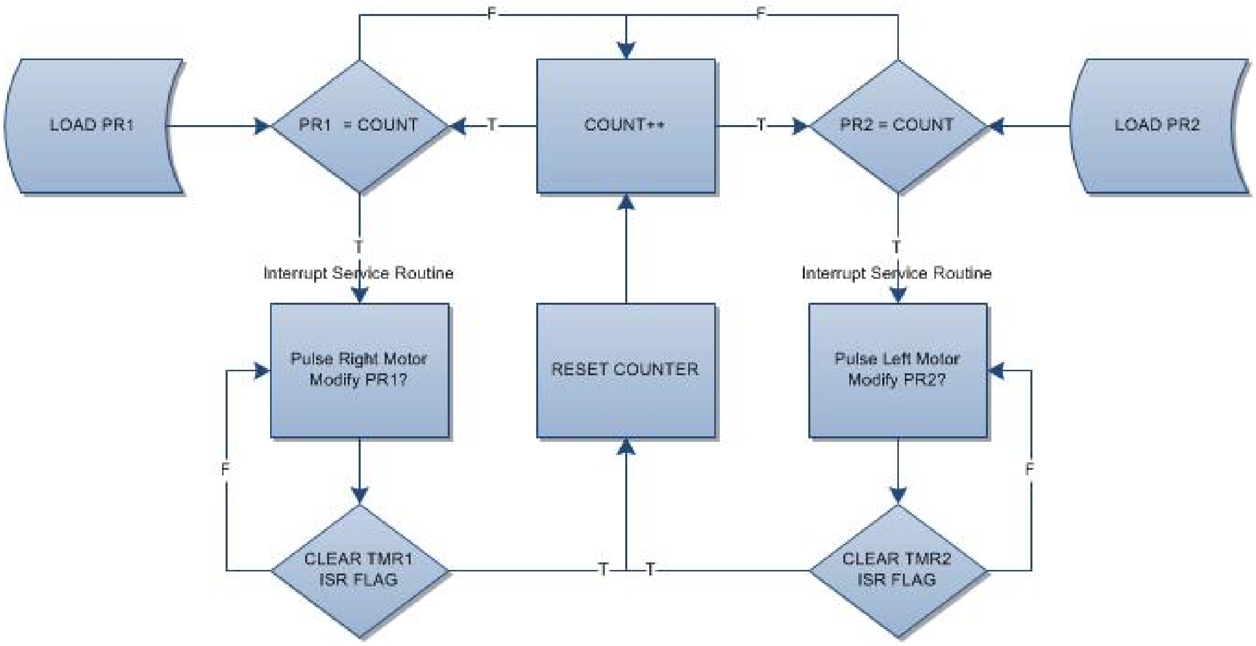


Figure 4: Motor Control Logic Diagram

Tracking

Tracking is required to ensure that the mouse stays within the cell center to avoid crashing, getting lost, and incorrectly mapping walls. This is taken care of using a Proportional-Derivative (PD) controller to improve the overall mouse stability. The proportional term P is the difference between what is considered center of the cell from the current sensor readings. This works efficiently well, since the higher the error, the higher the correction factor. The derivative term D on the other hand is the difference between the current P error and the previous P error. With this term, the step response due to the P error can be dampened to reduce the oscillations between corrections. Figure 5 below shows the software implementation used for tracking.

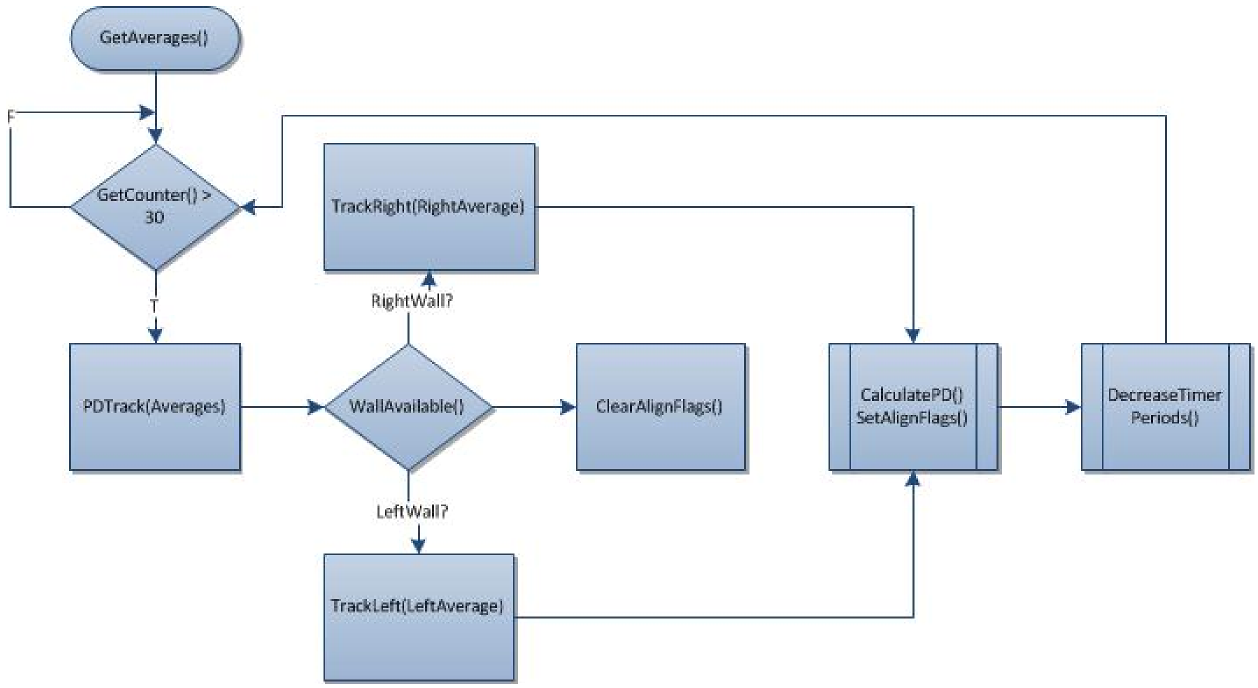


Figure 5: Tracking Logic Diagram

Individual proportional and derivative gains were also incorporated in the controller for finer tuning. With these gains K_p, and K_d, the rate at which the mouse corrects can be controlled for faster corrections times. Figure 6 below illustrates a simple effect of adjusting the proportional gain K_p.

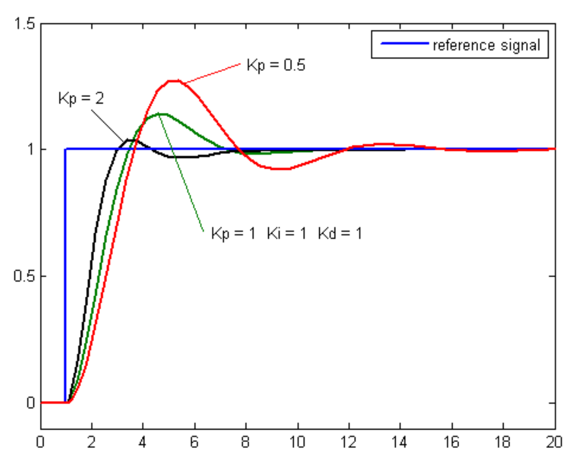


Figure 6: Effects of Proportional Gain Kp

Maze Solving Algorithm

The mouse incorporates a modified floodfill algorithm as the maze solving logic. This approach is similar to a normal floodfill algorithm but instead of re-flooding the entire maze after each movement, only the required cells have their values updated. Due to this implementation, a more efficient maze solving algorithm was accomplished.

Two one-dimensional arrays are used to keep track of the distance values and map the walls of the maze. To ensure proper operation, north is the current cell plus 16, south is the current cell minus 16, west is the current cell minus one, and east is the current cell plus one. Prior to the first movement of the mouse, distance values are initialized into the distance array, assuming no walls are present. This provides an initial movement path for the mouse to take during its exploration of the maze. After each arrival into a newly visited cell, the walls are updated according to the following bit masked configuration:

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
X	X	X	Visited	West	South	East	North

Table 1: Wall Map Array, Bit Mask Configuration

The current location cell of the mouse is pushed onto a stack. The following algorithm is performed until the stack is empty. A cell is popped off the stack. The distance value is then compared to each of its open neighbors'. If necessary, the cell's distance value is changed to one unit higher than the minimum value of its open neighbor. If the distance value of the current cell needed to be updated to satisfy the "lowest+1" condition, then each of the current cell's open neighbors are pushed onto the stack to have their distance values checked in the same manner.

After this is completed, the mouse is instructed to move to the neighbor with the lowest distance value. Following each turn, the direction of the mouse is updated. This ensures that the walls are updated in relation to the maze. By repeatedly performing this algorithm, the mouse will find the center of the maze.

Problems and Shortcomings

Currently the mouse is able to find the center of the maze. Issues that need to be worked on include the consistency of the tracking to ensure that the center is found every time and without crashing. It was observed that the sensor values fluctuate at times, causing tracking issues. However, with consistent lighting, these problems become not as big of an issue. Tire slippage will at times cause the step count to be inconsistent. This leads to crashing when taking turns. To correct this, the mouse aligns itself to each front wall and resets the step count. Issues with the step count become more pronounced after long straights because the mouse has no front wall to align to.

The next step towards implementing speed runs is to be able to flood back to the starting square. Currently an acceleration function can be implemented for speed runs. Advance movements such as s-turns and 45's would help lower the solving time however; this was a low priority group.

Learning Outcomes

Through the course of the semester, many opportunities for learning were presented. Hardware design is an important component of any electrical design. Through this project, learning about hardware implementation was achieved. Considerations such as power consumption, size, and design features need to be taken into account. Also, the compatibility of the components is important when creating an effective Micromouse. Both the hardware and software teams need to work together to ensure the proper interface of the two modules. Reading and understanding the datasheets of the electrical components ensure the correct implementation to achieve the objectives of the project.

The embedded C language used by Microchip was used to program the microcontroller. No team member possessed any significant experience so this was a great learning opportunity. Header files specific to the type of microcontroller provided functions that can be incorporated to the design. Software interrupts were an important aspect of the programming of the Micromouse as well.

Many intangible learning benefits were gained through the process of this project. Time management was an important consideration when planning the intermediate goals to be incrementally accomplished. Setbacks to the plan had to be dealt with because through the course of a project, problems will arise, creating delays to the timeline. Communication and teamwork among members contributed to the success of the project. Complications prevented the achievement of the high level of success originally sought at the beginning of the project.

Appendix

Datasheets

1. Microchip dsPIC30F4011
   ww1.microchip.com/downloads/en/devicedoc/70135c.pdf

2. ST Quad Darlington Switch ULN2066B
   http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00000177.pdf

3. Osram Infrared Emitters SFH4545
   http://catalog.osram-os.com/catalogue/catalogue.do;jsessionid=F699112EB60200AB4CF6A70237AA2A27?act=downloadFile&favOid=020000010001461a000200b6

4. Vishay Infrared Phototransistor TEFT4300
   www.vishay.com/docs/81549/teft4300.pdf

5. ST 1.5A Voltage Regulator L7805C
   http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00000444.pdf

6. ST 1.0A Voltage Regulator L4941
   http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00000443.pdf

7. Kysan 42BYGH068 Stepper Motors
   http://www.kysanelectronics.com/Products/datasheet_display.php?recordID=2861

Source Code

I. Main

```
#define FCY 29491200UL

#include "p30fxxxx.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

_FOSC(
    FRC_PLL16 &      // Primary Oscillator Mode (FRC w/ PLL 16x)

    PRI &            // Oscillator Source (Primary Oscillator)

    CSW_FSCM_OFF // Clock Switching and Monitor (Sw Disabled, Mon Disabled)
);

_FWDT(
    WDTPSB_16 &    // WDT Prescaler B (1:16)

    WDTPSA_512 &   // WDT Prescaler A (1:512)

    WDT_OFF        // Watchdog Timer (Disabled)
);

_FBORPOR(
    PWRT_64 &      // POR Timer Value (64ms)

    BORV27 &       // Brown Out Voltage (Reserved)

    PBOR_OFF &     // PBOR Enable (Enabled)

    PWMxL_ACT_HI & // Low-side PWM Output Polarity (Active High)

    PWMxH_ACT_HI & // High-side PWM Output Polarity (Active High)

    RST_IOPIN &    // PWM Output Pin Reset (Control with PORT/TRIS regs)

    MCLR_EN        // Master Clear Enable (Enabled)
);

_FGS(
    GWRP_OFF &     // General Code Segment Write Protect (Disabled)

    CODE_PROT_OFF  // General Segment Code Protection (Disabled)
);

/*
_FICD(
    ICS_PGD        // Comm Channel Select (Use PGC/EMUC and PGD/EMUD)
);
*/
#include "init.h"
#include "map.h"
#include "motor.h"
#include "interrupts.h"
#include "sensors.h"


void DebugReadings(void);
void Read_Front(void);
void Startup(void);
```

```
// Averages of sensor values, used for tracking
int SideRight = 0;
int SideLeft = 0;
int RightAverage = 0;
int LeftAverage = 0;
int FrontAverage = 0;
int FrontRight = 0;
int FrontLeft = 0;
/*
int LeftFlag = 0;
int RightFlag = 0;
int FrontFlag = 0;
*/
unsigned char WALLS = 0;

int cellcount = 0;

int DELAY = 4;           // NOTE: also must change in map.c Turning Delay
int i = 0;

bool floodBack = false;
bool back = false;

int main(void)
{
    InitPorts();          // Initialize Ports to use
    InitADC();            // Initialize ADC Conversion
    InitPD();
    InitFloodFill();

    //DebugReadings();

    //Startup();

    __delay_ms(500);
    RightAverage = readSideRight();
    __delay_ms(500);
    LeftAverage = readSideLeft();
    __delay_ms(500);

//  ResetCounter();
    InitRMotorTimer();     // Initialize Right Motor Interrupt
    InitLMotorTimer();     // Initialize Left Motor Interrupt

    //Flood Fill implementation in progress
    while (!floodBack)
    {
        if ((LocationEqual(GetCurrLocation(), GetDestination(0))))
        {
            StopMotors();

            PORTE = BLED1 | BLED2 | RLED1 | RLED2;
            __delay_ms(2000);
            floodBack = true;
        }
        StartMotors();

        if (GetCounter() >= 25 && GetCounter() < 195)
        {
            PDTrack(RightAverage, LeftAverage);
        }

        if (GetCounter() > 105 && GetCounter() <= 120)
        {
            MapWalls();
        }
        if (GetCounter() == 195)
```

```
        {
            StopMotors();

            if (FrontWall())
            {
                AlignToFront();
                __delay_ms(250);
            }

            // Flood if necessary, and Move mouse based on lowest flood value
            FloodFill(GetCurrLocation(), GetDestination(0));

            PORTE = OFFLED;
            ResetCounter();
            SetAccel();
        }
    }
    InitMazeBack();
    while(!back)
    {
        if ((LocationEqual(GetCurrLocation(), GetDestination(1))))
        {
            StopMotors();
            PORTE = BLED1 | BLED2 | RLED1 | RLED2;
            TurnAround(3);
            __delay_ms(2000);
            back = true;
        }
        StartMotors();

        if (GetCounter() >= 25 && GetCounter() < 195)
        {
            PDTrack(RightAverage, LeftAverage);
        }

        if (GetCounter() > 105 && GetCounter() <= 120)
        {
            MapWalls();
        }
        if (GetCounter() == 195)
        {
            StopMotors();

            if (FrontWall())
            {
                AlignToFront();
                __delay_ms(250);
            }

            // Flood if necessary, and Move mouse based on lowest flood value
            FloodFill(GetCurrLocation(), GetDestination(1));

            PORTE = OFFLED;
            ResetCounter();
            SetAccel();
        }
    }
}
```

II. Init

```
#include "init.h"

// Port Initialzations
void InitPorts()
{
    TRISD = 0x00;      // Sets PORT D to Drive Right Motors
    TRISE = 0x00;      // Sets PORT E to Drive Debug LED's
    TRISF = 0x00;      // Sets PORT F to Drive Left Motors
};
```

III. Interrupts

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#include "interrupts.h"
#include "init.h"

#define STARTING_DELAY 0x076C
char R_ALIGN = 0;                    // Flags to set Alignment
char L_ALIGN = 0;

unsigned int STARTING_SPEED = 2815;
unsigned int MAX_SPEED = 1900;      // Maximum velocity 500

unsigned int RPREVIOUS_DELAY = 0;
unsigned int LPREVIOUS_DELAY = 0;

int COUNTER = 0;         // Counts steps

char RSTATE = 1;   // Sets to first right motor step (Blue)
char LSTATE = 1;   // Sets to first left motor step (Blue)

/*****************************************************************/
/***************** START OF MOTOR INTERRUPTS *****************/
/*****************************************************************/

// Initializes Right Motor Timer
void InitRMotorTimer()
{
    T1CONbits.TON=0;       /*Stops Timer1 */
    T1CONbits.TCS=0;       /*Select internal clock cycle*/
    T1CONbits.TGATE=0;     /*Disable Gated timer mode*/
    T1CONbits.TCKPS=0b10;   /*Prescaler 1:256, 1x iteration/cycle*/
                               /*1, 8, 64, 256 Prescalers*/
    TMR1=0x00;            /*Clear Timer1 Register*/
    PR1=STARTING_DELAY;     /*Sets max period of 65535 */

    IPC0bits.T1IP=7;     /*Sets highest priority 7*/
    IFS0bits.T1IF=0;       /*Clear Timer1 Interrupt Flag*/
    IEC0bits.T1IE=1;       /*Enable Timer1 Interrupt*/
    T1CONbits.TON=1;       /*Starts Timer1*/
}

// Initializes Left Motor Timer
void InitLMotorTimer()
{
    T2CONbits.TON=0;       /*Stops Timer2 */
    T2CONbits.TCS=0;       /*Select internal clock cycle*/
    T2CONbits.TGATE=0;     /*Disable Gated timer mode*/
    T2CONbits.TCKPS=0b10;   /*Prescaler 1:1, 1x iteration/cycle*/
```

```
    TMR2=0x00;             /*Clear Timer2 Register*/
    PR2=STARTING_DELAY;    /*Sets max period of 65535 */

    IPC1bits.T2IP= 7;   /*Sets highest priority 7*/
    IFS0bits.T2IF=0;      /*Clear Timer2 Interrupt Flag*/
    IEC0bits.T2IE=1;      /*Enable Timer2 Interrupt*/
    T2CONbits.TON=1;      /*Starts Timer2*/
}

// Right Motor Interrupt
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
{
    COUNTER++;
    AccelerateRight();
    RPREVIOUS_DELAY = PR1;
    if(R_ALIGN == 1)
    {
        if(ValidPDError('R'))
            PR1 = PR1 - GetPDError();      // Speed Up Right Motor
        else
            PR1 = 1200;

        if(PR1 < 1200)
        {
            PR1 = 1200;
        }
    }
    else
    {
        PR1 = RPREVIOUS_DELAY;
    }
    RightForward(RSTATE);
    IFS0bits.T1IF = 0;    //Clear the INT1 interrupt flag or else
                          //the CPU will keep vectoring back to the ISR
}

// Left Motor Interrupt
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void)
{
    AccelerateLeft();
    LPREVIOUS_DELAY = PR2;

    if(L_ALIGN == 1)
    {
        if(ValidPDError('L'))
            PR2 = PR2 - GetPDError();      //Speed Up Left Motor
        else
            PR2 = 1200;
        if(PR2 < 1200)
        {
            PR2 = 1200;
        }
    }
    else
    {
        PR2 = LPREVIOUS_DELAY;
    }

    LeftForward(LSTATE);
    IFS0bits.T2IF = 0;    //Clear the INT1 interrupt flag or else
                          //the CPU will keep vectoring back to the ISR
}

// Functions to start and stop motors
void StartMotors(void)
{
    IEC0bits.T1IE=1;
```

```
    IEC0bits.T2IE=1;

}

void StopMotors(void)
{
    IEC0bits.T1IE=0;
    IEC0bits.T2IE=0;
}

// Acceleration Function
void AccelerateRight()
{
    if(PR1 > MAX_SPEED)
    {
        PR1 = PR1-25;
    }
    else
    {
        PR1 = PR1+25;
    }
    if(PR1 < MAX_SPEED)
        PR1 = MAX_SPEED;
    if(PR1 > STARTING_SPEED)
        PR1 = STARTING_DELAY;
}

void AccelerateLeft()
{
    if(PR2 > MAX_SPEED)
    {
        PR2 = PR2-25;
    }
    else
    {
        PR2 = PR2+25;
    }
    if(PR2 < MAX_SPEED)
        PR2 = MAX_SPEED;
    if(PR2 > STARTING_DELAY)
        PR2 = STARTING_DELAY;
}

//
char ValidPDError(char side)
{
    if(side == 'R')
    {
        if((PR1 - GetPDError() > STARTING_DELAY))
            return 0;
        else
            return 1;
    }
    else if(side == 'L')
    {
        if((PR2 - GetPDError() > STARTING_DELAY))
            return 0;
        else
            return 1;
    }
}

// Functions for keeping track of step counts
int GetCounter(void)
{   return COUNTER; }

void ResetCounter(void)
{  COUNTER = 0; }
```

```
// Functions to keep track of motor stepping
char getRSTATE(void)
{   return RSTATE;      }

char getLSTATE(void)
{   return LSTATE;      }

void setRSTATE(char state)
{   RSTATE=state; }

void setLSTATE(char state)
{   LSTATE=state; }


// Flags to trigger mouse alignment
void RClearAlignFlag()
{   R_ALIGN = 0;

}

void LClearAlignFlag()
{   L_ALIGN = 0;    }

void RSetAlignFlag()
{   R_ALIGN = 1;    }

void LSetAlignFlag()
{   L_ALIGN = 1;    }

/*****************************************************************/
/*************** START OF ACCELERATION FUNCTION ***************/
/*****************************************************************/
void SetAccel()
{
   PR1 = STARTING_DELAY;
   PR2 = STARTING_DELAY;
   LClearAlignFlag();
   RClearAlignFlag();
}

void InitAccelTable()
{
   int i=0;
   float temp = 7350;
   float desiredTime = 0;

   for(i=1;i<100;i++)
   {
      desiredTime = temp - ((2*temp)/(4*i+1));
      temp = desiredTime;
      //desiredTime = (float)(4*i-1)/(4*i+1);
      //desiredTime =  sqrt((float)(2*(i+1))/1052) - sqrt((float)(2*(i))/1052);
      //ACCEL_TABLE[i-1] = desiredTime;//(float)(desiredTime/(4*PRESCALER*TCY));
   }
}

#endif
```

IV. Motors

```
#include "init.h"
#include "motor.h"

unsigned char TURNSTATE = 0;
unsigned char STurnFlag = 0;

/******************************************************************/
/***************** START OF MOTOR FUNCTIONS ******************/
/******************************************************************/

// Step Right Motor Forward
void RightForward(char STATE)
{
    switch(STATE) {
        case 1:
            RSTEPPER = RRED;
            STATE=2;
            break;
        case 2:
            RSTEPPER = RYELLOW;
            STATE=3;
            break;
        case 3:
            RSTEPPER = RGREEN;
            STATE=4;
            break;
        case 4:
            RSTEPPER = RBLUE;
            STATE=1;
            break;
        default:
            STATE=1;
    }
    setRSTATE(STATE);
}

// Step Left Motor Forward
void LeftForward(char STATE)
{
    switch(STATE) {
        case 1:
            LSTEPPER = LBLUE;
            STATE=2;

            break;
        case 2:
            LSTEPPER = LGREEN;
            STATE=3;

            break;
        case 3:
            LSTEPPER = LYELLOW;
            STATE=4;

            break;
        case 4:
            LSTEPPER = LRED;
            STATE=1;

            break;
        default:
            STATE=1;
```

```
    }
    setLSTATE(STATE);
}

// Step Right Motor Back
void RightBack(char STATE)
{
        switch(STATE) {
        case 1:
            RSTEPPER = RBLUE;
            STATE = 2;
            break;
        case 2:
            RSTEPPER = RGREEN;

            STATE = 3;
            break;
        case 3:
            RSTEPPER = RYELLOW;

            STATE = 4;
            break;
        case 4:
            RSTEPPER = RRED;
            STATE = 1;
            break;
        default:
            STATE=1;
    }
        setRSTATE(STATE);
}

// Step Left Motor Back
void LeftBack(char STATE)
{
    switch(STATE) {
        case 1:
            LSTEPPER = LRED;
            STATE=2;
            break;
        case 2:
            LSTEPPER = LYELLOW;
            STATE=3;
            break;
        case 3:
            LSTEPPER = LGREEN;
            STATE=4;
            break;
        case 4:
            LSTEPPER = LBLUE;
            STATE=1;
            break;
        default:
            STATE=1;
    }
    setLSTATE(STATE);

}

// Turn 90 Degrees Counter-Clockwise
void TurnRight(int delay)
{
    int count = 0;

    for(count=1; count < 78; count++)
    {
        RightBack(getRSTATE());
        LeftForward(getLSTATE());
```

```
        __delay_ms(delay);
    }
}

// Turn 90 Degrees Clockwise
void TurnLeft(int delay)
{
    int count = 0;

    for(count=1; count < 78; count++)
    {
        LeftBack(getLSTATE());
        RightForward(getRSTATE());
        __delay_ms(delay);
    }
}

// Turn 180 Degrees Clockwise
void TurnAround(int delay)
{
    int count;

    for(count=1; count < 155; count++)
    {
        RightBack(getRSTATE());
        LeftForward(getLSTATE());
        __delay_ms(delay);
    }
}
```

V. Sensors

```
#include <p30F4011.h>
#include "sensors.h"
#include "interrupts.h"

struct ADC Sensors = {0, 0, 0, 0};

int EMITTER_TIME = 6;
int NUMBER_SAMPLES = 10;

int CENTER_AVERAGE = 0;
int RSIDE_AVERAGE = 0;
int LSIDE_AVERAGE = 0;
int FRONT_AVERAGE = 0;

/***************************************************************/
/***************** START OF SENSOR FUNCTIONS *****************/
/***************************************************************/

int readSideRight()
{   return Sensors.SideRight;   }

int readSideLeft()
{   return Sensors.SideLeft;    }

int readFrontRight()
{   return Sensors.FrontRight;  }

int readFrontLeft()
{   return Sensors.FrontLeft;   }

int getRSide_Average(void)
{   return ADCBUF0; }

```

```c
int getLSide_Average(void)
{   return ADCBUF1; }

int getFront_Average(void)
{   return (readFrontRight() + readFrontLeft())/2; }

/*****************************************************************/
/**************** START OF ADC INTERRUPTS *******************/
/*****************************************************************/

void InitADC()
{
    //ADCON1 Register
    ADCON1bits.FORM = 0b00;         // Read in values in integer format
    ADCON1bits.SSRC = 7;            // Internal counter starts/stop sampling
    ADCON1bits.ASAM = 1;            // Sets up A/D for Automatic Sampling
    ADCON1bits.SIMSAM = 0;          // Samples channels sequentially

    //ADCON2 Register
    ADCON2bits.SMPI = 4;            // A/D interrupting after 4 samples gets
                                    // filled in the buffer

    ADCON2bits.BUFM = 0;            // Single 16-bit Buffer
    ADCON2bits.ALTS = 0;            // Use MUX A Input select
    //ADCON2bits.VCFG = 0;          // Use Avdd Reference
    ADCON2bits.CSCNA = 1;           // Scan CH0+ for Inputs

    //ADCON3 Register
    //We would like to set up a sampling rate of 1 MSPS
    //Total Conversion Time= 1/Sampling Rate = 125 microseconds
    //At 29.4 MIPS, Tcy = 33.9 ns = Instruction Cycle Time
    //The A/D converter will take 12*Tad periods to convert each sample
    //So for ~1 MSPS we need to have Tad close to 83.3ns
    //Using equaion in the Family Reference Manual we have
    //ADCS = 2*Tad/Tcy - 1
    ADCON3bits.SAMC = 0;
    ADCON3bits.ADCS = 4;        // Conversion Time 1/Sampling Rate = 125 us

    //ADCSSL Register
    ADCSSL = 0x000F;            // Scan AN0..AN3

    //ADPCFG Register
    ADPCFG = 0xFFF0;            // Sets AN0..AN3 to Analog
    //ADPCFGbits.PCFG3 = 0;

    IFS0bits.ADIF = 0;          //Clear the A/D interrupt flag bit
    IEC0bits.ADIE = 1;          //Set the A/D interrupt enable bit

    InitADCValues();            // Initiazlize ADC Values

    //Turn on the A/D converter
    ADCON1bits.ADON = 1;
}

void __attribute__((interrupt, no_auto_psv)) _ADCInterrupt(void)
{
    Sensors.SideRight = ADCBUF0;
    Sensors.SideLeft = ADCBUF1;
    Sensors.FrontRight = ADCBUF2;
    Sensors.FrontLeft = ADCBUF3;

    //Clear the A/D Interrupt flag bit or else the CPU will
    //keep vectoring back to the ISR
    IFS0bits.ADIF = 0;
}

void InitADCValues(void)
{
```

```
   char i = 0;

   Sensors.SideRight = 0;
   Sensors.SideLeft = 0;
   Sensors.FrontRight = 0;
   Sensors.FrontLeft = 0;
}
```

VI. Controller

```
#include "init.h"
#include "controller.h"

// Initializes Controller Struct
struct Controller PID;

// Controller Gain Constants
#define Kp 8 //5
#define Kd 1 //1

int RTRACKTHRESHOLD = 5;
int LTRACKTHRESHOLD = 3;
int FRONTCENTER = 200;

/*****************************************************************/
/***************** START OF PD CONTROLLER *********************/
/*****************************************************************/

void PDTrack(int RightAverage, int LeftAverage)
{
   StopMotors();
   if(RightTrack())          // Right Wall Avail For Tracking
   {
      PDTrackRight(RightAverage);
   }
   else if(LeftTrack())       // Left Wall Available for Tracking
   {
      PDTrackLeft(LeftAverage);
   }
   else                       // No Walls to track...
   {
      ClearPDError();                  // Reset PD Errors
      RClearAlignFlag();
      LClearAlignFlag();
   }
   StartMotors();
}

/*****************************************************************/
/************ START OF PD CONTROLLER FUNCTIONS ***************/
/*****************************************************************/

// Returns PD value for SR sensors based on errors
// If Proportional => pos, pulling right, speed up right motor
// If Proportional => neg, pulling left, speed up left motor
// RTRACKTHRESHOLD accounts for acceptable errors, Error < -5 || Error > 5
// Proportional Term Still Needs To Be Linearize using slope equation
void PDTrackRight(int RightAverage)
{
   //PORTE = BLED1;

   CalculatePD(RightAverage, 'R');

   if(GetP()-RTRACKTHRESHOLD > 0)         // Pulling Right, Speed up R Motor
```

```
  {
    LSetAlignFlag();
    RClearAlignFlag();              // Clear Opposite motor flag to return to proper speed
  }
  else
  if(GetP()+RTRACKTHRESHOLD < 0)    // Pulling Left, Speed up L motor
  {
    RSetAlignFlag();
    LClearAlignFlag();
  }
  else                              // No Error, Go Straight
  {
    ClearPDError();                 // Be Sure to clear  PD Error
    RClearAlignFlag();
    LClearAlignFlag();
  }
  //PORTE = OFFLED;
}

void PDTrackLeft(int LeftAverage)
{
  //PORTE = BLED2;
  CalculatePD(LeftAverage, 'L');

  if(GetP()-LTRACKTHRESHOLD > 0)              // Pulling Left, Speed up L Motor
  {
    RSetAlignFlag();
    LClearAlignFlag();                    // Clear Opposite motor flag to return to proper speed
  }
  else if(GetP()+LTRACKTHRESHOLD < 0)         // Pulling Right, Speed up R motor
  {
    LSetAlignFlag();
    RClearAlignFlag();
  }
  else                                  // No Error, Go Straight
  {
    ClearPDError();
    RClearAlignFlag();
    LClearAlignFlag();
  }
  //PORTE = OFFLED;
}

// Move mouse forward until in center, using Front sensors for guide
void AlignToFront()
{
   while(((readFrontRight() < 312) || (readFrontRight() > 322 ))
        || ((readFrontLeft() < 311) || (readFrontLeft() > 321)))
   {
      __delay_us(3200);
      if(readFrontRight() < 312) RightForward(getRSTATE());
      else if(readFrontRight() > 322) RightBack(getRSTATE());
      if(readFrontLeft() < 311) LeftForward(getLSTATE());
      else if(readFrontLeft() > 321) LeftBack(getLSTATE());
      __delay_us(3200);
   }
   LeftBack(getLSTATE());
   __delay_ms(5);
   LeftBack(getLSTATE());
   __delay_ms(5);
}

// Controller Calculation Functions
void CalculatePD(int Average, char Side)
{
   SetPrevError(GetP());            // Sets Previous Error For Nex Iteration
   Proportional(Average, Side);       // Calculates Proportiona Error
   Derivative();                    // Calculate Derivative Error
```

```
    SetPDError(GetP() + GetD());
}

void Proportional(int Average, char Side)
{
    if(Side == 'R')
        SetP(Kp*(Average - readSideRight()));
    if(Side == 'L')
        SetP(Kp*(Average - readSideLeft()));
}

void Derivative()
{   SetD((abs(GetP()) - GetPrevError())/Kd);    }

// Initializers

void InitPD()
{
    PID.Derivative = 0;
    PID.PDError = 0;
    PID.ePrev = 0;
    PID.Proportional = 0;
}

void ClearPDError()
{
    SetPrevError(0);
    SetPDError(0);
    SetP(0);
    SetD(0);
}

// Controller Accessor Functions
int GetPDError()
{   return PID.PDError; };

int GetPrevError()
{   return PID.ePrev;   }

int GetP()
{   return PID.Proportional;   }

int GetD()
{   return PID.Derivative;  }

// Controller Mutator Functions
void SetPrevError(int e)
{   PID.ePrev = e;  }

void SetPDError(int e)
{   PID.PDError = e;    }

void SetP(int P)
{   PID.Proportional = P; }

void SetD(int D)
{  PID.Derivative = D;  }
```

VII. Map

```
#include "init.h"
#include "map.h"
#include "delay.h"
#include "motor.h"
#include "interrupts.h"
#include "sensors.h"
#include <stdio.h>
#include "map.h"
#include "stack.h"

Stack cellStack;

struct Flood f;

void FloodFill(Location currLoc, Location destination)
{
   // Map walls mouse sees at current cell locations
   if (GetDirection() == NORTH)
   {
      SetLocation(GetRow(currLoc), GetCol(currLoc)+1);    //j+1
   }
   if (GetDirection() == EAST)
   {
     SetLocation(GetRow(currLoc)+1, GetCol(currLoc));    //i+1
   }
   if (GetDirection() == SOUTH)
   {
      SetLocation(GetRow(currLoc), GetCol(currLoc)-1);    //j-1
   }
   if (GetDirection() == WEST)
   {
      SetLocation(GetRow(currLoc)-1, GetCol(currLoc));    // i-1
   }

   init(&cellStack);

   // Push current cell location into stack
   if (empty(&cellStack))
   {
      push(&cellStack, currLoc);
   }

   // Cell locations to check
   Location check;
   unsigned char lowestVal = 255;

   while (!empty(&cellStack))
   {
      lowestVal = 255;

      check = pop(&cellStack);
      // Gets the lowest value of neighbors of cell to check
      if (!((GetCell(check) & NORTH) == NORTH))
      {
         if (GetDistance(GetRow(check), GetCol(check) + 1) < lowestVal)
         {
            lowestVal = GetDistance(GetRow(check), GetCol(check) + 1);
         }
      }
      if (!((GetCell(check) & EAST) == EAST))
      {
         if (GetDistance(GetRow(check) + 1, GetCol(check)) < lowestVal)
         {
```

```
                lowestVal = GetDistance(GetRow(check) + 1, GetCol(check));
            }
        }
        if (!((GetCell(check) & SOUTH) == SOUTH))
        {
            if (GetDistance(GetRow(check), GetCol(check) - 1) < lowestVal)
            {
                lowestVal = GetDistance(GetRow(check), GetCol(check) - 1);
            }
        }
        if (!((GetCell(check) & WEST) == WEST))
        {
            if (GetDistance(GetRow(check) - 1, GetCol(check)) < lowestVal)
            {
                lowestVal = GetDistance(GetRow(check) - 1, GetCol(check));
            }
        }
        // Checks if
        if (!(GetDistance(GetRow(check), GetCol(check)) == (lowestVal + 1)))
        {
            // Not Destination Cell
            if (!LocationEqual(check, destination))
            {
                SetDistance(lowestVal + 1, GetRow(check), GetCol(check));
                //distanceMap[check] = lowestVal+1;
            }

            // Finds all neighbor open to flood values
            // Push North Neighbor to Stack
            if (!((GetCell(check) & NORTH) == NORTH))
            {
                push(&cellStack, GetLocation(GetRow(check), GetCol(check) + 1));
            }
            // Push East Neighbor to Stack
            if (!((GetCell(check) & EAST) == EAST))
            {
                push(&cellStack, GetLocation(GetRow(check) + 1, GetCol(check)));
            }
            // Push South Neighbor to Stack
            if (!((GetCell(check) & SOUTH) == SOUTH))
            {
                push(&cellStack, GetLocation(GetRow(check), GetCol(check) - 1));
            }
            // Push West Neighbor to Stack
            if (!((GetCell(check) & WEST) == WEST))
            {
                push(&cellStack, GetLocation(GetRow(check) - 1, GetCol(check)));
            }
        }
    }
    MakeMove(GetLowestDir());
}

void MapWalls()
{
    Location loc;
    loc = GetCurrLocation();

    if (GetDirection() == NORTH)
    {
        if (LeftWall())
        {
            UpdateWall(GetLocation(GetRow(loc) + 1, GetCol(loc)), WEST);
            PORTE = RLED2;
            __delay_ms(100);
        }
```

```
    if (RightWall())
    {
      UpdateWall(GetLocation(GetRow(loc) + 1, GetCol(loc)), EAST);
      PORTE = RLED1;
      _delay_ms(100);
    }
    if (FrontWall())
    {
      UpdateWall(GetLocation(GetRow(loc) + 1, GetCol(loc)), NORTH);
      PORTE = BLED1;
      _delay_ms(100);
    }
  }
  if (GetDirection() == EAST)
  {
    if (LeftWall())
    {
      UpdateWall(GetLocation(GetRow(loc), GetCol(loc) + 1), NORTH);
      PORTE = RLED2;
      _delay_ms(100);
    }
    if (RightWall())
    {
      UpdateWall(GetLocation(GetRow(loc), GetCol(loc) + 1), SOUTH);
      PORTE = RLED1;
      _delay_ms(100);
    }
    if (FrontWall())
    {
      UpdateWall(GetLocation(GetRow(loc), GetCol(loc) + 1), EAST);
      PORTE = BLED1;
      _delay_ms(100);
    }
  }
  if (GetDirection() == SOUTH)
  {
    if (LeftWall())
    {
      UpdateWall(GetLocation(GetRow(loc), GetCol(loc) - 1), EAST);
      PORTE = RLED2;
      _delay_ms(100);
    }
    if (RightWall())
    {
      UpdateWall(GetLocation(GetRow(loc), GetCol(loc) - 1), WEST);
      PORTE = RLED1;
      _delay_ms(100);
    }
    if (FrontWall())
    {
      UpdateWall(GetLocation(GetRow(loc), GetCol(loc) - 1), SOUTH);
      PORTE = BLED1;
      _delay_ms(100);
    }
  }
  if (GetDirection() == WEST)
  {
    if (LeftWall())
    {
      UpdateWall(GetLocation(GetRow(loc) - 1, GetCol(loc)), SOUTH);
    }
    if (RightWall())
    {
      UpdateWall(GetLocation(GetRow(loc) - 1, GetCol(loc)), NORTH);
      PORTE = RLED1;
      _delay_ms(100);
```

```
    }
    if (FrontWall())
    {
        UpdateWall(GetLocation(GetRow(loc) - 1, GetCol(loc)), WEST);
        PORTE = BLED1;
        __delay_ms(100);
    }
  }
}

void UpdateWall(Location loc, unsigned char direction)
{
  f.wallMap[GetRow(loc)][GetCol(loc)] = (f.wallMap[GetRow(loc)][GetCol(loc)] | direction);

  // Update South Wall of Cell Above
  if (direction == NORTH)
  {
    f.wallMap[GetRow(loc)][GetCol(loc) + 1] = (f.wallMap[GetRow(loc)][GetCol(loc) + 1] | SOUTH);
  }
  // Update West Wall of Cell to the Right
  if (direction == EAST)
  {
    f.wallMap[GetRow(loc) + 1][GetCol(loc)] = (f.wallMap[GetRow(loc) + 1][GetCol(loc)] | WEST);
  }
  // Update North Wall of Cell Below
  if (direction == SOUTH)
  {
    f.wallMap[GetRow(loc)][GetCol(loc) - 1] = (f.wallMap[GetRow(loc)][GetCol(loc) - 1] | NORTH);
  }
  // Update East Wall of Cell to the Left
  if (direction == WEST)
  {
    f.wallMap[GetRow(loc) - 1][GetCol(loc)] = (f.wallMap[GetRow(loc) - 1][GetCol(loc)] | EAST);
  }
}

void MakeMove(unsigned char lowestDir)
{
  unsigned char direction;

  if (GetDirection() == NORTH)
  {
    if (lowestDir == EAST)
    {
      TurnRight(3);
      __delay_ms(250);
      direction = EAST;
    }
    if (lowestDir == SOUTH)
    {
      TurnAround(3);
      __delay_ms(250);
      direction = SOUTH;
    }
    if (lowestDir == WEST)
    {
      TurnLeft(3);
      __delay_ms(250);
      direction = WEST;
    }
  }
  if (GetDirection() == EAST)
  {
    if (lowestDir == NORTH)
    {
      TurnLeft(3);
```

```
        _delay_ms(250);
        direction = NORTH;
    }
    if (lowestDir == SOUTH)
    {
        TurnRight(3);
        _delay_ms(250);
        direction = SOUTH;
    }
    if (lowestDir == WEST)
    {
        TurnAround(3);
        _delay_ms(250);
        direction = WEST;
    }
  }
  if (GetDirection() == SOUTH)
  {
    if (lowestDir == NORTH)
    {
        TurnAround(3);
        _delay_ms(250);
        direction = NORTH;
    }
    if (lowestDir == EAST)
    {
        TurnLeft(3);
        _delay_ms(250);
        direction = EAST;
    }
    if (lowestDir == WEST)
    {
        TurnRight(3);
        _delay_ms(250);
        direction = WEST;
    }
  }
  if (GetDirection() == WEST)
  {
    if (lowestDir == NORTH)
    {
        TurnRight(3);
        _delay_ms(250);
        direction = NORTH;
    }
    if (lowestDir == EAST)
    {
        TurnAround(3);
        _delay_ms(250);
        direction = EAST;
    }
    if (lowestDir == SOUTH)
    {
        TurnLeft(3);
        _delay_ms(250);
        direction = SOUTH;
    }
  }
  // Updates mouse new location direction
  SetDirection(direction);
}

unsigned char GetLowestDir()
{
  unsigned char lowest = 255;
  unsigned char lowestDir = 0;
```

```
    
    Location currentCell = GetCurrLocation();
    
    // No North Neighbor?
    if (!((GetCell(currentCell) & NORTH) == NORTH))
    {
        // Get Flood Distance Value
        if (GetDistance(GetRow(currentCell), GetCol(currentCell) + 1) < lowest)
        {
            // Set Lowest to North Neighbor
            lowest = GetDistance(GetRow(currentCell), GetCol(currentCell) + 1);
            lowestDir = NORTH;
        }
    }
    // No East Neighbor? No -> Check if Lowest flood val
    if (!((GetCell(currentCell) & EAST) == EAST))
    {
        if (GetDistance(GetRow(currentCell) + 1, GetCol(currentCell)) < lowest)
        {
            lowest = GetDistance(GetRow(currentCell) + 1, GetCol(currentCell));
            lowestDir = EAST;
        }
    }
    // No South Neighbor? No -> Check if Lowest flood val
    if (!((GetCell(currentCell) & SOUTH) == SOUTH))
    {
        if (GetDistance(GetRow(currentCell), GetCol(currentCell) - 1) < lowest)
        {
            lowest = GetDistance(GetRow(currentCell), GetCol(currentCell) - 1);
            lowestDir = SOUTH;
        }
    }
    // No West Neighbor? No -> Check if Lowest flood val
    if (!((GetCell(currentCell) & WEST) == WEST))
    {
        if (GetDistance(GetRow(currentCell) - 1, GetCol(currentCell)) < lowest)
        {
            lowest = GetDistance(GetRow(currentCell) - 1, GetCol(currentCell));
            lowestDir = WEST;
        }
    }
    return lowestDir;
}

void InitFloodFill()
{
    f.direction = NORTH;
    f.currentCell.col = 0;
    f.currentCell.row = 0;
    InitWalls();
    InitMaze();
    InitDestination();
}

void InitWalls()
{
    char i = 0;
    char j = 0;
    Location loc;
    loc = GetCurrLocation();
    
    for (i = 0; i < 16; i++)
    {
        for (j = 0; j < 16; j++)
        {
            f.wallMap[i][j] = 0;
```

```
        }
    }
    UpdateWall(loc, EAST);
    UpdateWall(loc, SOUTH);
    UpdateWall(loc, WEST);
}

void InitMazeBack()
{
    char i = 0;
    char j = 0;

    for(i=0; i < 16; i++)
    {
        for(j = 0; j < 16; j++)
        {
            f.wallFlood[i][j] = j + i;
        }
    }
}

void InitMaze()
{
    char i = 0;
    char j = 0;

    for (i = 0; i < 16; i++)
    {
        for (j = 0; j < 16; j++)
        {
            if (j <= 7)
            {
                if (i <= 7) // Fills i <= 7, j<=7 quadrant
                    f.wallFlood[i][j] = 14 - i - j;
                if (i > 7) // Fills i > 7, j <=7 quadrant
                    f.wallFlood[i][j] = i - j - 1;
            }
            if (j > 7)
            {
                if (i <= 7) // Fills i <=7 , j > 7 quadrant
                    f.wallFlood[i][j] = j - i - 1;
                if (i > 7) // Fills i > 7, j > 7 quadrant
                    f.wallFlood[i][j] = (i - 8) + (j - 8);
            }
        }
    }
}

void InitDestination()
{
    SetDestination(0, 7, 7);      // Center destination
    SetDestination(1, 0, 0);      // Flooding back destination
}

void DisplayMaze()
{
    char i = 0;
    char j = 0;
    for (i = 0; i < 16; i++)
    {
        for (j = 0; j < 16; j++)
        {
            printf("%d ", f.wallFlood[i][j]);
        }
        printf("\n");
    }
```

```
}

Location GetCurrLocation()
{
    return f.currentCell;
}

Location GetLocation(char row, char col)
{
    Location loc;
    loc.row = row;
    loc.col = col;
    return loc;
}

char GetRow(Location loc)
{
    return loc.row;
}

char GetCol(Location loc)
{
    return loc.col;
}

unsigned char GetDirection()
{
    return f.direction;
}

unsigned char GetDistance(char row, char col)
{
    return f.wallFlood[row][col];
}

Location GetDestination(char i)
{
    return f.destination[i];
}

void SetDistance(unsigned char distance, char row, char col)
{
    f.wallFlood[row][col] = distance;
}

unsigned char GetCell(Location loc)
{
    return f.wallMap[loc.row][loc.col];
}

void SetLocation(char row, char col)
{
    f.currentCell.row = row;
    f.currentCell.col = col;
}

void SetDirection(unsigned char direction)
{
    f.direction = direction;
}

void SetDestination(char index,char row, char col)
{
    f.destination[index].row = row;
    f.destination[index].col = col;
}
```

```
bool LocationEqual(Location loc, Location other)
{
    if ((loc.row == other.row) && (loc.col == other.col))
        return true;
    else
        return false;
}
```

VII. Stack

```
#include "stack.h"
#include <stdbool.h>

void push(Stack *S, Location val)
{
    S->s[S->top] = val;
    (S->top)++;
}

Location pop(Stack *S)
{
    (S->top)--;
    return (S->s[S->top]);
}

void init(Stack *S)
{
    S->top = 0;
}

bool empty(Stack *S)
{
    return (S->top == 0);
}
```