EECE 284: Electronics Laboratory

Project Report for Fast Orange—an Electromagnetic Track Rover

John Harvey, 29240116 (25%)

Liam Hodgson, 29154119 (25%)

Scott Lawson, 34359117 (25%)

Rowan Walsh, 30817118 (25%)

March 31st, 2014

Course taught by Dr. Jesús Calviño-Fraga, P. Eng

TO-DO:

* Electrical
  + Magnitude of gains
  + Circuit diagram for MOSFET controllers
* Mechanical
* Software
  + Write section
  + Block diagram

Contents

[1.0—Introduction 3](#_Toc384568054)

[1.1—Mechanical 3](#_Toc384568055)

[1.2—Electrical 4](#_Toc384568056)

[1.2.1—Power supply 4](#_Toc384568057)

[1.2.2—Motor control 4](#_Toc384568058)

[1.2.3—Sensor array 5](#_Toc384568059)

[1.2.4—Microcontroller 5](#_Toc384568060)

[1.2.5—LCD shield 5](#_Toc384568061)

[1.3—Software 6](#_Toc384568062)

[2.0—Project analysis 7](#_Toc384568063)

[2.1—Testing 7](#_Toc384568064)

[2.2—Design analysis 7](#_Toc384568065)

[3.0—Conclusions 8](#_Toc384568066)

[References 9](#_Toc384568067)

[Bibliography 10](#_Toc384568068)

[Appendices 11](#_Toc384568069)

# 1.0—Introduction

Our project for EECE 284 was an Arduino-based rover, designed to follow a sinusoidal, 15.175 kHz electromagnetic signal. The format of the course was a time trial: each group’s robot completed a circuit of a racetrack. The runs were timed, and after each group’s entry attempted the course, the robot with the fastest time was awarded the highest mark.

The signal the robot was to follow was generated by running a 20V peak-to-peak 15.175 kHz sinusoidal voltage through a wire. The wire was embedded in the competition surface, creating the path the robots were to follow. In addition to straight track, signals were placed along the track, indicating the beginning and end of the path, as well as the direction to turn at the intersection of wires. Bridges were another feature of the track, which added an extra challenge, as robots could not vary their path too greatly while on the bridge, for risk of falling off, and as the depth of the wire from the surface was greater on the bridges.

The robot followed the racecourse by detecting the electromagnetic field generated by the wire, using inductors mounted at the front of the robot. These signals were put through peak-detection and amplification circuits, and then fed into a PID algorithm. The outputs of the PID algorithm drove MOSFET motor drivers, allowing the robot to follow the wire and racecourse with a high degree of accuracy.

## 1.1—Mechanical

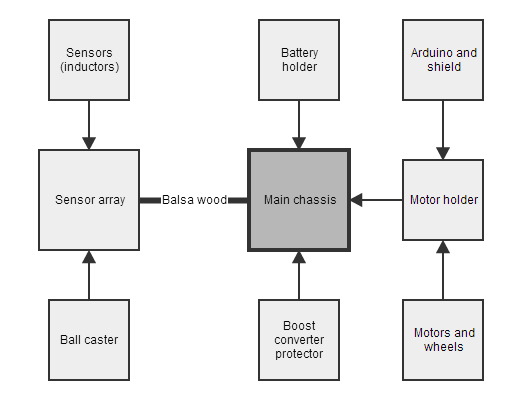
The chassis, the main mechanical component of the robot, was designed to minimize mass and structurally incorporate as many essential components (microcontroller, motors, battery) as possible. The robot consisted of three sections: the main chassis, sensor array, and a piece of balsa wood which joined the two. The front chassis and sensor array were 3D printed in bright orange poly-lactic acid (PLA), to minimize mass, and in homage to the industrial-strength, orange-scented hand cleaner, Fast Orange.

Figure 1: Mechanical block diagram

The main chassis consisted of three components: a battery holder, boost-converter protector, and motor holder. The battery used fit neatly under the robot, as it was flat and thin. Four short wires extended from the battery holder to the boost-converter protector, mounted at the front of the main chassis. Wires carrying 5 and 10 volts left the boost converter, carrying power to the Arduino Uno and motors, respectively. A shield incorporating an LCD monitor, potentiometers, several screw-terminal ports, and six buttons sat atop the Arduino Uno, which was mounted on the motor holder. Unsurprisingly, the motor holder held the motors. The other components (boost-converter protector, battery holder, balsa wood) were attached to the motor holder.

The sensor array held a Tamiya 70144 ball caster and three inductors—two perpendicular to the direction of motion of the robot, one parallel, and all mounted close to the ground. The balsa wood was firmly inserted in the sensor array, holding it in front of the main chassis.

Various nuts, bolts, plastic spacers, and some hot glue held the components together.

## 1.2—Electrical

### 1.2.1—Power supply

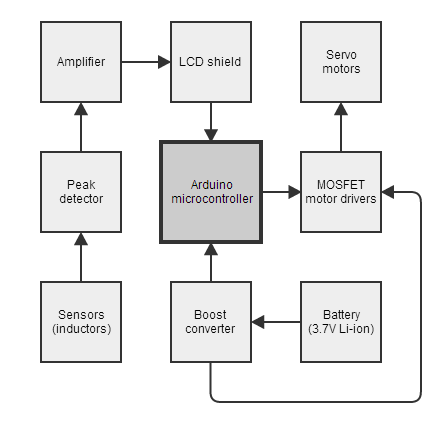
The choice of power supply for our robot was important, as a higher voltage would allow us to run the motors at a higher RPM, and a larger capacity would allow for longer testing time. The battery chosen, a rechargeable lithium ion battery with a voltage of 3.7 volts and a capacity of 950 mAh, was taken from an old cellphone. Quick to recharge and lightweight, the battery fell well below the limit set for this competition. To provide an appropriate voltage for the Arduino and motors, a DC-DC boost converter was used, allowing us to supply 10V to the motors, and 5V to the Arduino, without surpassing the maximum battery voltage of 6V. An appropriate choice of battery to work with the boost converter was important: a battery with an extremely low internal resistance had to be used to supply sufficient current to the converter, ruling out traditional alkaline batteries. Additionally, by using a boost converter, we were able to ensure that the motor voltage did not fluctuate as the battery drained, allowing for more repeatability in testing.

Figure 2: Electrical block diagram

The converter used was a VPack PBC, based on an LM2623MM DC/DC boost converter, modified to supply ten and five volts. Said modification consisted of de-soldering and replacing several surface-mount components. The battery came from an LG Rumour 2.

### 1.2.2—Motor control

The two Solarbotics GM4 servos provided the driving force for the robot, and were controlled via MOSFET drivers. The gate of the MOSFET was connected to the PWM output of the Arduino, and the source and drain were connected to the 10V supplied by the boost converter and the motor, respectively.

[MOSFET circuit?]

### 1.2.3—Sensor array

The robot was to follow a path defined by a buried wire carrying a 20V peak-to-peak signal at 15.175 kHz. This magnetic field produced by the wire was detected by two 100mH inductors. By placing appropriately-sized capacitors () in parallel with the inductors, the sensors could be tuned to a resonant frequency equal to that of the desired field. 1nF capacitors were selected, producing a resonant frequency of 15.9kHz, which was sufficiently close to the desired frequency to be effective.

Two inductors were mounted perpendicular to the direction of motion, to provide relative signal strength, indicating whether the robot was to the left or right of the path. One inductor was mounted parallel to the direction of motion, to detect wire perpendicular to the track, indicating the start, end, and turn warnings on the track. These inductors were mounted closer to the bulk of the robot than the other sensors—the further away the perpendicular sensors were from the main chassis, the more sensitive they would be to small changes in the angle of the robot. By placing the parallel sensor close to the main chassis, turns could be detected later, reducing the delay between detection of a turn and the turn itself.

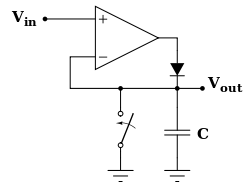
The AC signal induced in the inductor was then converted to DC via a peak detector, and amplified. (GAIN GAIN GAIN—was it variable?) The structure of the peak detector was a BJT (acting as a switch) connected to a digital output port, wired in parallel with a capacitor. A diode rectified the input voltage, which charged the capacitor to the peak voltage of the signal. This voltage was read by one of the digital inputs on the Arduino. After reading, the gate of the BJT shorts to ground, discharging the capacitor and restarting the process. A non-inverting amplifier circuit, consisting of an op-amp and a potentiometer, was used to provide variable gain.

Figure 3: Peak-detector circuit diagram

### 1.2.4—Microcontroller

An Arduino UNO microcontroller was chosen for the controller of our robot, as team members were more familiar with it than the provided microcontroller. Sufficiently powerful, reasonably inexpensive, and with a wealth of libraries, the Arduino read the sensors, controlled the motors voltage, displayed information on the LCD, and ran the PID algorithm on the sensor inputs.

### 1.2.5—LCD shield

An LCD shield incorporating several buttons and knobs (DO THE KNOBS ACTUALLY EXIST) was used to provide an easy interface for debugging and tuning the robot.

[LCD shield photo]

## 1.3—Software

Some words about the code that makes the robot work

# 2.0—Project analysis

Some words about the things we’re going to analyze

## 2.1—Design analysis

While our robot performed well, several adjustments would be made if the project was undertaken again. Most importantly, the inductors would be more securely fastened. While it was easy to make fine adjustments to their positions, by bending the wires from which they hung, this created problems. The inductors were bumped around during transport of the robot, and in the occasional collision, which had dramatic effects on the robot’s ability to sense the line. Seemingly inconsequential adjustments proved to be the difference between successful and unsuccessful runs, and it was extremely difficult to quantify the adjustments made.

Using a cell phone battery was one of the better decisions made: in addition to allowing us to use a higher motor voltage, the size, ability to recharge, and weight proved advantageous. Teams, on average, went through 6-8 AA batteries. We used one battery, from an old cell phone, which was easier on the environment and pocketbook. We would likely reduce the motor voltage if the project was undertaken again: the robot ran at approximately a third of its top speed, as controlling the robot became extremely difficult after that point. A reduction in maximum speed would allow for more precise control over motor velocity.

Wire management became a concern later on in the project: last-minute additions lead to a large number of unruly wires, which came close to challenging the height requirements. Incorporating wire management into the chassis, or perhaps using piping in place of the balsa wood, and routing the wires through them, would be possibilities to explore.

One of the more successful debugging features incorporated were the ‘turn signals’. Two LEDs mounted near the front of the robot would blink before turns were initiated, allowing us to observe what the robot was sensing without having to observe whether or not it turned, and, on several occasions, allowing us to catch the robot before it turned off the table.

# 3.0—Conclusions

The robot ran very successfully—it completed the track without any human intervention with the fastest recorded time of any team: 1:09. By using a lighter battery and a 3D printed chassis, mass was significantly reduced, allowing for greater speed. Wiring became unruly as the project progressed, and the inductors were extremely vulnerable to changes in position—a small bump would result in unquantifiable motion of the inductors, changing the behaviour of the robot.

The planning, design, coding, manufacture, and testing of the robot took approximately 100-120 hours in total. If the robot was to be constructed again, it is estimated that it would take no more than 20 hours—printing the parts, manufacturing the circuits, and adjusting the inductors would take the lion’s share of the time.

# References

If we referred to it in the body of the report (a book, website, source of information) we need to have the full citation here. The in-text citation should just be a number in square brackets, [n], and the references should look something like this:

[1] Smith, J, and F. Jones, “Designing a universal logic circuit”, Journal of Impossibly Wonderful Electronic Circuits, v.3, n.1, pp. 21-35, March, 1910.

[2] Jones, F and J. Smith, “Why universal logic circuits are impractical” , …

# Bibliography

If we didn’t refer to it in the report but it’s good background reading, or some literature we used but didn’t mention in the report, it goes here.

# Appendices

Any ‘extensive theoretical analyses’, part drawings, and source code should go here. The code should be properly documented (whatever that means) and indented.

Don’t attach datasheets, compiler manuals, or anything which was given in references or the bibliography.

## Appendix 1: Code

#include <SimpleTimer.h> // Pre-built library

#include <LiquidCrystal.h> // Pre-built library

#include <EEPROM.h> // Pre-built library

LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

// Configuration

#define SUBTRACTIVE\_MOTOR\_SPEED // Comment for additive and subtractive motor speed

#define CALCULATE\_DERIVATIVE\_ERROR // Comment to remove derivative gain from error calculations

#define CALCULATE\_INTEGRAL\_ERROR // Comment to remove integral gain from error calculations

#define CALCULATE\_ANALOG\_PID // Comment to calculate PID error using boolean logic instead

// of analog values

#define USE\_SMART\_INTERSECTIONS // Comment to use the intersection map instead of sensing intersection signals

//#define DEBUG\_MODE // Comment to display voltage and time instead of debug

// values

// Pin definitions

#define LEFT\_SENSOR 1 // ANALOG

#define RIGHT\_SENSOR 2 // ANALOG

#define CENTER\_SENSOR 3 // ANALOG

#define BATTERY\_SENSOR 4 // ANALOG

#define BUTTON\_INPUT 0 // ANALOG

#define LEFT\_MOTOR 11 // PWM

#define RIGHT\_MOTOR 3 // PWM

#define SENSOR\_RESET 2 // DIGITAL

#define LEFT\_TURN\_LED 12 // DIGITAL

#define RIGHT\_TURN\_LED 13 // DIGITAL

// Keypad button definitions

#define RIGHT 0

#define UP 1

#define DOWN 2

#define LEFT 3

#define SELECT 4

#define NONE 5

// LCD cursor definitions

#define TOP 0

#define BOTTOM 1

// Wire States

#define TOO\_LEFT -1

#define TOO\_RIGHT 1

#define OFF\_WIRE 5

// Turn indicator LEDs

SimpleTimer turnSignalTimer;

bool turnSignalState = false;

// "Clock" speed

unsigned long loopPeriod;

unsigned long lastLoop;

enum state

{

menu,

moveStraight,

turnLeft,

turnRight

};

struct Parameter

{

String Name;

byte Value;

int Index;

};

// Prevents LCD flicker

short lcdRefreshCount = 0;

short lcdRefreshPeriod = 200;

// Prevents excess battery readings

short batteryCount = 0;

short batteryPeriod = 10000;

volatile float batteryVoltage = 0.0;

// Sensors

volatile short left = 0;

volatile short right = 0;

volatile short leftDetected = false;

volatile short rightDetected = false;

// Smart intersection detection

#define INTERSECTION\_COUNT\_LEFT 2

#define INTERSECTION\_COUNT\_RIGHT 3

#define INTERSECTION\_COUNT\_ORIGIN 4

volatile short center = 0;

volatile short centerOldHigh = false;

volatile short intersectionSignalRecent = 0;

volatile short centerTimeoutCount = 0;

volatile short intersectionTurnFlag = 0;

// Mapped intersection detection

enum intersectionCommand

{

straight,

leftTurn,

rightTurn,

stop

};

int intersections[] =

{

straight,

straight,

straight,

straight,

straight,

leftTurn,

straight,

straight,

rightTurn,

straight,

straight,

};

#define INTERSECTION\_COUNT (sizeof(intersections)/sizeof(int)) // MUST EQUAL NUMBER OF PARAMETERS

int intersectionIndex = -1;

volatile int centerNew = false;

volatile int centerRisingEdge = false;

volatile int centerHighDetected = false;

volatile int centerLowDetected = false;

// Control

float proportional = 0.0;

float integral = 0.0;

float derivative = 0.0;

short derivativeCounter = 0;

short integralOffsetPeriod = 50;

short integralOffsetCounter = integralOffsetPeriod;

float error = 0.0;

float previousError = 0.0;

// State tracking

state currentState = menu;

short menuIndex = 0;

// Clock

volatile unsigned int clockTime = 0;

volatile int clockRun = false;

volatile unsigned int clockStart = 0;

// EEPROM values

Parameter proportionalGain = {"P-gain", 0, 1};

Parameter integralGain = {"I-gain", 0, 2};

Parameter derivativeGain = {"D-gain", 0, 3};

Parameter speed = {"Speed", 0, 4};

Parameter thresholdLeft = {"L-Thresh", 0, 5};

Parameter thresholdRight = {"R-Thresh", 0, 6};

Parameter centerSchmittHigh = {"C-Schmitt H", 0, 7};

Parameter centerSchmittLow = {"C-Schmitt L", 0, 8};

Parameter integralCap = {"I-Cap", 0, 9};

Parameter centerTimeoutLimit = {"Timeout", 0, 10};

// Make sure any EEPROM value is added to the array

Parameter \*parameters[] =

{

&proportionalGain, &integralGain, &derivativeGain,

&speed, &thresholdLeft, &thresholdRight, &centerSchmittHigh,

&centerSchmittLow, &integralCap, &centerTimeoutLimit

};

#define PARAMETER\_COUNT (sizeof(parameters)/sizeof(Parameter\*)) // MUST EQUAL NUMBER OF

// PARAMETERS

inline void TurnSignal()

{

if (intersectionTurnFlag == INTERSECTION\_COUNT\_LEFT)

{

digitalWrite(LEFT\_TURN\_LED, turnSignalState);

digitalWrite(RIGHT\_TURN\_LED, 0);

}

else if (intersectionTurnFlag == INTERSECTION\_COUNT\_RIGHT)

{

digitalWrite(LEFT\_TURN\_LED, 0);

digitalWrite(RIGHT\_TURN\_LED, turnSignalState);

}

else

{

digitalWrite(LEFT\_TURN\_LED, 0);

digitalWrite(RIGHT\_TURN\_LED, 0);

}

turnSignalState = !turnSignalState;

}

// Clears the LCD screen

inline void Clear()

{

lcd.clear();

}

// Prints a string to the LCD display, with an optional integer value beside it

inline void Print(String text, int value = -1)

{

lcd.print(text);

if (value != -1) lcd.print(value);

}

// Changes the LCD cursor location

inline void Cursor(int row, int column)

{

lcd.setCursor(column, row);

}

// SETUP

void setup()

{

// For algorithm speed checking

loopPeriod = 0;

lastLoop = micros();

// Initialize LCD

lcd.begin(16, 2);

lcd.setCursor(0,0);

pinMode(SENSOR\_RESET, OUTPUT);

// Turn signals

pinMode(LEFT\_TURN\_LED, OUTPUT);

pinMode(RIGHT\_TURN\_LED, OUTPUT);

digitalWrite(LEFT\_TURN\_LED, 0);

digitalWrite(RIGHT\_TURN\_LED, 0);

turnSignalTimer.setInterval(100, TurnSignal);

// Force motors off by default

MotorSpeed(LEFT\_MOTOR, 0);

MotorSpeed(RIGHT\_MOTOR, 0);

// Parameters need to be loaded from EEPROM

LoadFromEEPROM();

// Intro text

Clear();

Cursor(TOP, 0);

Print("FAST ORANGE");

Cursor(BOTTOM, 0);

#ifdef DEBUG\_MODE

Print("Debug Mode");

#else

Print("Race Mode");

#endif

delay(1000);

currentState = menu;

ClockReset();

}

// LOOP

void loop()

{

switch(currentState)

{

case menu:

MotorSpeed(LEFT\_MOTOR, 0);

MotorSpeed(RIGHT\_MOTOR, 0);

ShowMenu();

break;

case moveStraight:

Update();

#ifdef CALCULATE\_ANALOG\_PID

ProcessMovementAnalog();

#else

ProcessMovementDigital();

#endif

break;

case turnLeft:

case turnRight:

Turn();

break;

}

turnSignalTimer.run();

loopPeriod = micros() - lastLoop;

lastLoop = micros();

}

// Drains the capacitors on the peak detector so that can be read again

inline void SensorReset(int microseconds = 15)

{

digitalWrite(SENSOR\_RESET, HIGH);

delayMicroseconds(microseconds);

digitalWrite(SENSOR\_RESET, LOW);

}

// Returns the current button being pressed.

// Can detect one button being pressed at a time.

inline int ReadButton()

{

int value = analogRead(BUTTON\_INPUT);

if (value > 1000) return NONE;

if (value < 50) return RIGHT;

if (value < 250) return UP;

if (value < 450) return DOWN;

if (value < 650) return LEFT;

if (value < 850) return SELECT;

return NONE;

}

// Variables used to modify rate of change of values in menu

int previousButton = UP;

int holdCounter = 0;

// Display the menu on screen

void ShowMenu()

{

// Show menu item on top row

Clear();

Cursor(TOP, 0);

Print(parameters[menuIndex]->Name);

Print(" ", parameters[menuIndex]->Value);

// Show sensor info on bottom row. Useful for threshold calibration

Cursor(BOTTOM, 0);

Print(" ", analogRead(LEFT\_SENSOR));

Print(" ", analogRead(RIGHT\_SENSOR));

Print(" ", analogRead(CENTER\_SENSOR));

SensorReset();

switch(ReadButton())

{

case UP: // Increase item value

holdCounter = (previousButton == UP) ? (holdCounter + 1) : 0;

previousButton = UP;

parameters[menuIndex]->Value += 1 + (holdCounter / 20);

break;

case DOWN: // Lower item value

holdCounter = (previousButton == DOWN) ? (holdCounter + 1) : 0;

previousButton = DOWN;

parameters[menuIndex]->Value -= (1 + (holdCounter / 20));

break;

case LEFT: // Next menu item

menuIndex = (menuIndex > 0) ? (menuIndex - 1) : (PARAMETER\_COUNT - 1);

break;

case RIGHT: // Previous menu item

menuIndex = (menuIndex < PARAMETER\_COUNT - 1) ? (menuIndex + 1) : 0;

break;

case SELECT:// Exit menu

delay(500);

if (ReadButton() == SELECT)

{

Clear();

Cursor(TOP, 0);

Print("Exiting menu");

SaveToEEPROM(); // Save values to EEPROM before exiting

delay(750);

currentState = moveStraight;

intersectionTurnFlag = 0;

//ClockToggle();

Clear();

return;

}

break;

}

delay(150);

}

// Loads all values from the EEPROM

void LoadFromEEPROM()

{

for(int i = 0; i < PARAMETER\_COUNT; i++)

parameters[i]->Value = EEPROM.read(parameters[i]->Index);

}

// Saves all values to the EEPROM

void SaveToEEPROM()

{

for(int i = 0; i < PARAMETER\_COUNT; i++)

EEPROM.write(parameters[i]->Index, parameters[i]->Value);

}

inline void DisplaySensorValues()

{

if(lcdRefreshCount == 0) // Mitigates screen flicker and time consuming operations

{

#ifdef DEBUG\_MODE

// Sensors on top line

Cursor(BOTTOM, 0);

#ifdef USE\_SMART\_INTERSECTIONS

Print("Signal: ", intersectionTurnFlag);

#else

Print("Intersection: ", intersectionIndex + 1);

#endif

#else

Cursor(TOP, 0); Print(" "); Cursor(TOP, 0);

Print("Time: "); lcd.print(Clock()/1000.0); Print("s");

#endif

}

#ifndef DEBUG\_MODE

if(batteryCount == 0)

{

Cursor(BOTTOM, 0);

batteryVoltage = float(analogRead(BATTERY\_SENSOR)) / 1024.0 \* 5.0;

Print("Battery: "); lcd.print(batteryVoltage); Print("V");

}

#endif

}

inline void CenterMapUpdate()

{

center = analogRead(CENTER\_SENSOR);

centerHighDetected = center > centerSchmittHigh.Value;

centerLowDetected = center > centerSchmittLow.Value;

if(!centerLowDetected)

{

centerNew = true;

}

if(centerNew && centerHighDetected)

{

delay(50);

if (analogRead(CENTER\_SENSOR) > centerSchmittHigh.Value)

{

centerRisingEdge = true;

delay(50);

centerNew = false;

intersectionIndex++;

if (intersectionIndex == INTERSECTION\_COUNT)

intersectionIndex = 0;

switch(intersections[intersectionIndex])

{

case straight:

currentState = moveStraight;

break;

case leftTurn:

currentState = turnLeft;

break;

case rightTurn:

currentState = turnRight;

break;

}

}

}

}

inline void CenterSmartUpdate()

{

center = analogRead(CENTER\_SENSOR);

if (centerTimeoutCount)

centerTimeoutCount += 1;

if (!centerOldHigh && (center > centerSchmittHigh.Value)) // On switch-to-high

{

centerOldHigh = true; // Switch current center state to high

if((intersectionSignalRecent + 1) == INTERSECTION\_COUNT\_ORIGIN)

// If the start/stop signal is encountered, toggle clock.

// This is done on when it switches high (instead of waiting for timeout) to save a bit // of clock-time.

{

ClockToggle();

}

switch(intersectionTurnFlag)

{

case INTERSECTION\_COUNT\_LEFT: // First intersection after left signal

currentState = turnLeft;

break;

case INTERSECTION\_COUNT\_RIGHT: // First intersection after right signal

currentState = turnRight;

break;

default: // Else

currentState = moveStraight;

intersectionSignalRecent += 1;

break;

}

centerTimeoutCount = 0; // Stop timeout counter

intersectionTurnFlag = 0; // Reset flag

}

else if (centerOldHigh && (center < centerSchmittLow.Value)) // On switch-to-low

{

centerOldHigh = false; // Swtich current center state to low

centerTimeoutCount = 1; // Start timeout counter

}

if (centerTimeoutCount >= (centerTimeoutLimit.Value \* 2)) // Timeout

{

if (intersectionTurnFlag < intersectionSignalRecent)

// Do not let false signals effect the flag

intersectionTurnFlag = intersectionSignalRecent; // Set flag

intersectionSignalRecent = 0; // Reset signal counter

centerTimeoutCount = 0; // Stop timeout counter

}

}

// Updates the sensors and machine state

void Update()

{

if ((lcdRefreshCount == 0) && (currentState != menu) && (ReadButton() == SELECT))

{

delay(750);

if(ReadButton() == SELECT) // debounce MENU button

{

// Stop motors before entering menu

MotorSpeed(LEFT\_MOTOR, 0);

MotorSpeed(RIGHT\_MOTOR, 0);

Clear();

Cursor(TOP, 0);

Print("Entering menu");

currentState = menu;

delay(1000);

ClockReset();

}

}

// Read raw sensor values

left = analogRead(LEFT\_SENSOR);

right = analogRead(RIGHT\_SENSOR);

leftDetected = left > thresholdLeft.Value;

rightDetected = right > thresholdRight.Value;

if (leftDetected || rightDetected)

{

#ifdef USE\_SMART\_INTERSECTIONS

CenterSmartUpdate();

#else

CenterMapUpdate();

#endif

}

SensorReset(); // Drain capacitor in preparation for next sensor reading

// Display values on LCD

lcdRefreshCount = (lcdRefreshCount <= 0) ? lcdRefreshPeriod : (lcdRefreshCount - 1);

batteryCount = (batteryCount <= 0) ? batteryPeriod : (batteryCount - 1);

DisplaySensorValues();

}

// Calculates PID values for a single iteration of movement

void ProcessMovementDigital()

{

if (leftDetected && rightDetected) error = 0; // No error if both sensors see the wire

else if (!leftDetected && rightDetected) error = TOO\_LEFT;

else if (leftDetected && !rightDetected) error = TOO\_RIGHT;

else if (!leftDetected && !rightDetected) error = (previousError <= TOO\_LEFT) ? -OFF\_WIRE : OFF\_WIRE;

// Proportional

proportional = error \* float(proportionalGain.Value);

// Integral

#ifdef CALCULATE\_INTEGRAL\_ERROR

if (integralOffsetCounter < 0)

{

integral += (error \* integralGain.Value);

integralOffsetCounter = integralOffsetPeriod;

}

else

{

integralOffsetCounter--;

}

if (integral > integralCap.Value)

integral = integralCap.Value;

else if (integral < -integralCap.Value)

integral = -integralCap.Value;

#else

integral = 0;

#endif

// Derivative

#ifdef CALCULATE\_DERIVATIVE\_ERROR

derivative = (error - previousError) / float(derivativeCounter) \* float(derivativeGain.Value);

#else

derivative = 0;

#endif

int baseSpeed = (speed.Value \* 4.0);

int mLeft = baseSpeed + (proportional + derivative + integral);

int mRight = baseSpeed - (proportional + derivative + integral);

#ifdef SUBTRACTIVE\_MOTOR\_SPEED

if (mLeft > mRight)

mLeft = baseSpeed;

else

mRight = baseSpeed;

#endif

MotorSpeed(LEFT\_MOTOR, mLeft);

MotorSpeed(RIGHT\_MOTOR, mRight);

if(previousError != error)

{

previousError = error;

derivativeCounter = 1;

}

else derivativeCounter++;

}

void ProcessMovementAnalog()

{

if (!leftDetected && !rightDetected) error = (previousError <= 0) ? -500 : 500;

else error = float(left) - float(right);

// Proportional

proportional = error \* float(proportionalGain.Value);

derivative = (error - previousError) \* float(derivativeGain.Value) / 50.0;

integral += (error \* float(integralGain.Value)) / 50.0;

if (integral > integralCap.Value)

integral = integralCap.Value;

else if (integral < -integralCap.Value)

integral = -integralCap.Value;

#ifndef CALCULATE\_INTEGRAL\_ERROR

integral = 0;

#endif;

#ifndef CALCULATE\_DERIVATIVE\_ERROR

derivative = 0;

#endif;

int baseSpeed = (speed.Value \* 4.0);

int mLeft = baseSpeed + (proportional + derivative + integral);

int mRight = baseSpeed - (proportional + derivative + integral);

#ifdef SUBTRACTIVE\_MOTOR\_SPEED

if (mLeft > mRight)

mLeft = baseSpeed;

else

mRight = baseSpeed;

#endif

MotorSpeed(LEFT\_MOTOR, mLeft);

MotorSpeed(RIGHT\_MOTOR, mRight);

}

// Modifies the motor speed given a value between 0 and 100

inline void MotorSpeed(int motor, int speed)

{

if (speed > 1000) speed = 1000;

else if (speed < 0) speed = 0;

analogWrite(motor, speed/4.0);

}

void Turn()

{

int direction;

if (currentState == turnLeft) direction = 1;

else if (currentState == turnRight) direction = -1;

else return;

Clear();

Print("TURN");

MotorSpeed(LEFT\_MOTOR, 1000 \* direction);

MotorSpeed(RIGHT\_MOTOR, 1000 \* -direction);

delay(700);

previousError = direction;

currentState = moveStraight;

Clear();

}

inline unsigned int Clock() // Displays the clock's current value (in milliseconds)

{

if (clockRun)

{

unsigned int currentMillis = millis();

clockTime += currentMillis - clockStart;

clockStart = currentMillis;

}

return clockTime;

}

inline void ClockToggle() // Toggles the clock to run or pause

{

if (clockRun) // If clock is currently running

{

clockTime += millis() - clockStart;

clockRun = false;

}

else

{

clockRun = true;

clockStart = millis();

}

}

inline void ClockReset() // Sets the clock to 0 and pauses the clock

{

clockTime = 0;

clockRun = false;

}