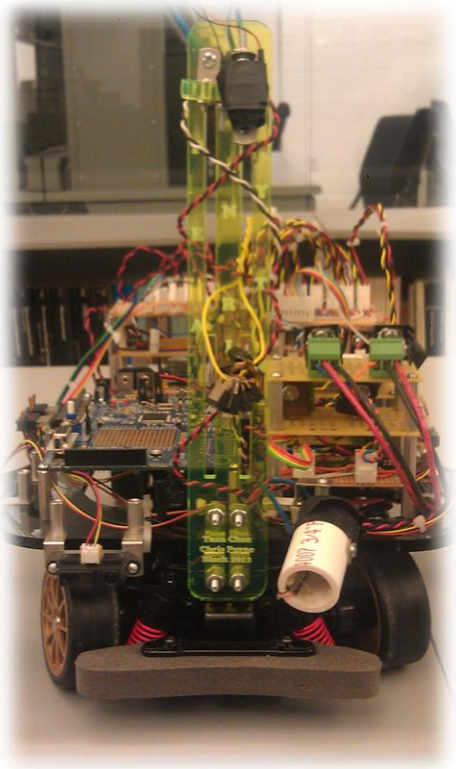
Chris Payne
Yuan Chen
ELE 302 Stage 3, Independent Project Report
Team: **BLACK**
5.13.12



Final Independent Project Goal

We set out to accomplish two main goals. The first was to allow our car to operate within a lighted ring in a manner that safely confined the car's movements to the interior of the ring and therefore removing the need for a static "track" for the car to follow. The second accompanying goal was for our car to automatically maneuver away from objects within the light ring, whether these objects were stationary or dynamic such as another car attempting to bump it. From these two straightforward goals we encountered ample complications such as reading data from our 10 analog sensors and learning to implement code in using our new PSOC 3 microprocessor.

Final Hardware Design

The hardware component of the design prominently featured four photoresistors, six infrared (IR) sensors, each group of sensors accompanying preprocessing circuit board, an H Bridge and its Float Protection Board and a PSoC 3 microprocessor. Additionally, we used a new mounting board provided by Radd.

Photoresistor Sensors & Preprocessing Board

We used four Cadium Sulfide photosensors in order to detect when our car was near the boundary of its operation, that being the lightrope. The sensor worked by changing its resistance in response to how much light was incident on it. Treating the photosensor as a simple resistor, we placed it in series with a constant 43 kΩ resistor on a preprocessing board creating a voltage divider. We then transmitted the analog voltage value present over the 43 kΩ resistor to our microprocessor.

The first challenge we encountered was how to make the resistance indicating the photosensor was right next to the lightrope (R^{Light}) and the resistance indicating the photosensor was far away from the lightrope (R^{Dark}) as distinct as possible. Initially, we found that the ambient light kept R^{Dark} at a rather low value such that it barley differed from R^{Light}. We ended up placing the photosensor inside a ¾ inch PVC pipe. This caused the R^{Dark} value to be much larger because ambient light was blocked, yet since one side of the tube that pointed at the light rope was left open, the R^{Light} value was distinctly low enough. Compare the experimental values in Table 1.

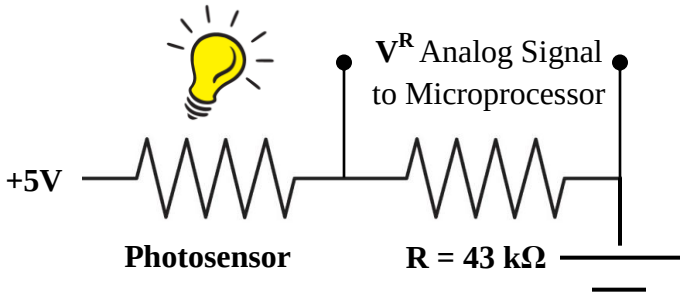
Table 1	R^{Dark} (kΩ)	R^{Light} (kΩ)
Without PVC Pipe Cover	50	7
With PVC Pipe Cover	200	10

Now that we had a wider range of resistance values, we needed to choose a constant resistance (R) that would optimize the difference between the voltage over R when the photosensor was dark and when it was light, in other words, optimize Equation1:

$$diff(R) = \frac{R}{R + \mathrm{R^{Light}}} - \frac{R}{R + \mathrm{R^{Dark}}} \quad (Equation\ 1)$$

By taking the derivative of *diff(R)* we found that R = 43 kΩ resulted in the largest voltage difference over R between when the photosensor was in the light and when it was in the dark. This voltage over R was then sent to the microprocessor as an analog signal that represented how much light was incident on the photosensor (See Figure One).

Figure One



(Between R^{Dark} and R^{Light})

With the conclusion of this discussion we now had a viable analog signal (V^R) that would indicate to our microprocessor which side of the car was nearing the light rope – we will discuss in the software portion of the report how this signal was processed and interpreted by the microprocessor.

IR Sensors & Preprocessing Board

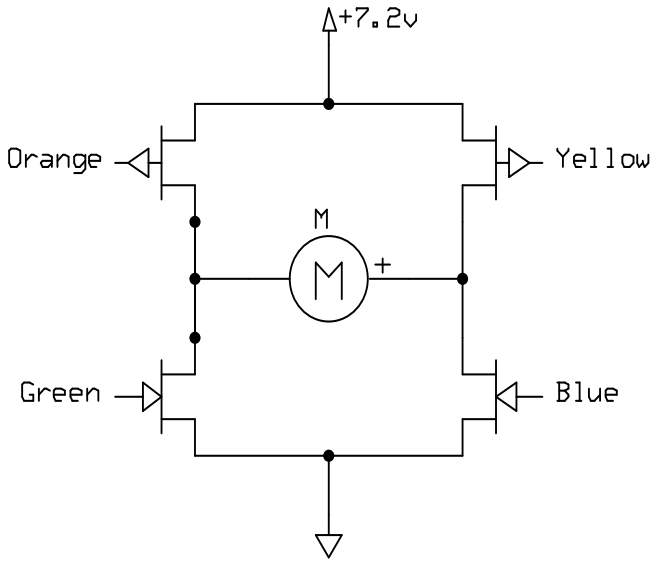
In order to detect objects near the car, we used 6 IR sensors, with one placed on both the front and back of the car and two placed on each side. Each sensor was powered by a regulated 5 volt source and returned approximately 900 mV when no object obstructed its view. As an object moved closer to the sensor, the output signal increased non-linearly to approximately 3000 mV. The fact that the IR sensors were built to relay a voltage signal made our Preprocessing Board very simple as all the board had to do was power each sensor and relay the sensor's analog output to our microprocessor. We encountered minimal difficulties in setting up the IR sensors and recommend them to future groups, although it should be noted that their usable range is rather short (not much more than a foot) and that their readings have a fair amount of noise

present. We will discuss processing and interpreting the analog signal in the software section of this report.

H Bridge & Float Protection Board

In order to allow our car to reverse away from an obstacle we needed to build an H Bridge. Our H-Bridge was designed as shown in Figure 2, using MTP52N06VL (52A, 60V) N type MOSFETs and IRF5305BF(31A, 55V) P type MOSFETs.

Figure 2



The bright control signals, orange and yellow, correspond to the P type MOSFETS while the darker control signals, green and blue, correspond to the N type MOSFETS. From observing Figure 2, it's clear that only three configurations of the control signals should be allowed in order to cause the car to drive forward and backward. These configurations are summarized in Table 2:

Table 2

Configuration	Orange (Reverse P)	Yellow (Forward P)	Blue (Reverse N)	Green (Forward N)
Neutral (No movement)	Off	Off	Off	Off
Forward	Off	On	Off	On
Reverse	On	Off	On	Off

It is important to keep in mind though that since some of the MOSFETS are P type and some are N type, different control signals correspond to a particular MOSFET being “On” or “Off”. Table 3 summarizes the electrical values of the various control signal states and how they correspond to their respective MOSFET:

Table 3

Configuration (See Table 2)	**Orange Ctrl Signal [V]**	**Yellow Ctrl Signal [V]**	**Blue Ctrl Signal [V]**	**Green Ctrl Signal [V]**
“Off”	7.2	7.2	0	0
“On”	0	0	3.2	3.2

You will note that the Orange and Yellow signals that correspond to turning the P MOSFET “Off” are 7.2 Volts, higher than the maximal 3.2 volts that the microprocessor outputs to indicate a “high” signal. In order to reach 7.2 Volts, we used a comparator tied directly to the 7.2 Volt battery. When the Orange or Yellow microprocessor signal exceeded a 2 volt threshold, the comparator boosted the Orange or Yellow signal to the P MOSFETS to 7.2 volts. This higher signal was needed because the P MOSFETS had a Gate Threshold close to 3.2 volts and we did not want fluctuations in the output of the microprocessor causing the P MOSFETS to erroneously turn on and off. In the software section we will go into more detail as to what the various states of the control signal were and how the microprocessor controlled these states.

Lastly, we saw from other groups that issues with floating control signals could cause the MOSFETS to be turned on in a manner that could cause a short, thus we needed to ensure that when the control signals from the microprocessor were floating (such as when the microprocessor was not connected to a power source) the control signals would be pulled up or pulled down in a manner that would turn all four MOSFETs into the “Off”. This was done by attaching individual pull-up resistors (2.2 kΩ) to the P MOSFETS control signals(Orange and Yellow) while attaching individual pull-down resistors(2.2 kΩ) to the N MOSFET control signals(Blue and Green). This ensured that the default (and float state) of the control signals would cause all the MOSFETs to be in an “Off” state.

After we had one of our MOSFETs burn out, we added a 20 amp fuse between the 7.2 Volt battery’s positive terminal and the components of the H-Bridge. In the event of a short, this

fuse would blow, discontinuing the flow of electricity and therefore preventing higher currents from damaging the circuitry.

PSoC3 Microprocessor

We decided to use this new processor in order to take advantage of the many more analog input pins it offered compared to the old processor. Additionally, we enjoyed the challenge of learning an additional IDE(PSoC Creator). In terms of hardware, it was easy to mount the PSoC microprocessor to the new mounting board Radd provided and we were able to hook it up to the same 5V power source the previous microprocessor used.

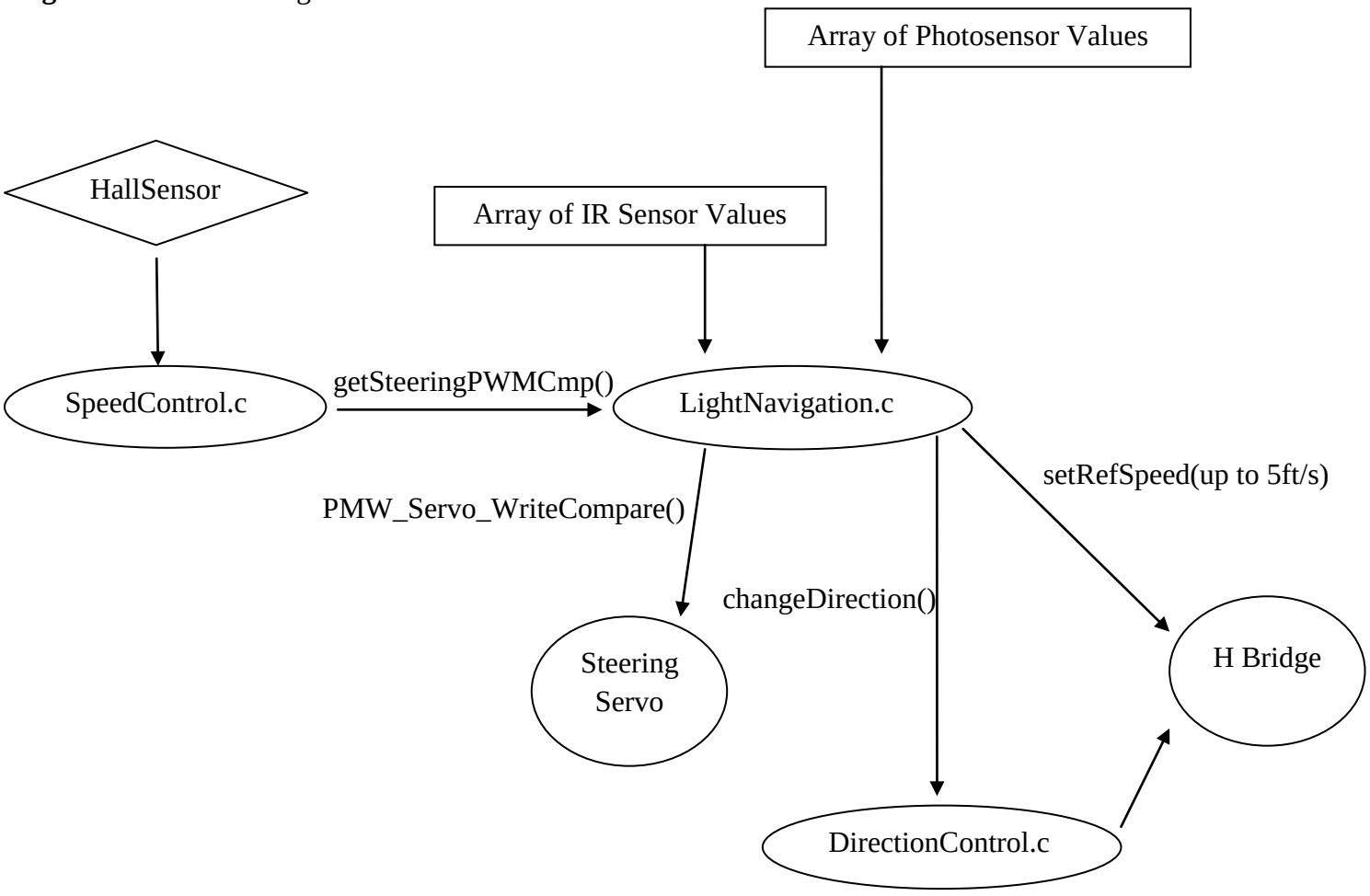
New Fiberglass Mounting Board

This black fiberglass board provided by Radd was outfitted to support the larger footprint of our new PSoC 3 processor. Additionally, it conveniently included mounting holes around the perimeter of the board that allowed for easy attachment of our IR and photosensors.

Final Software Design

In order to meet our project goal, we re-implemented our speed control module mirroring the implementation discussed in our "ELE 302 Stage 1: Speed Control" report. We added a primary source code file, LightNavigation.c and a secondary source code file, DirectionControl.c in order to process our new sensor inputs and control our new H Bridge. Figure 3 shows a broad overview of our software design. On each Hall sensor interrupt, the LightNavigation module examines the various values the photosensors and IR sensors are transmitting to the microprocessor. Based off these values, the module might assign a new PWM value to the steering servo in order to initiate a turn. Additionally, the LightNavigation module might determine that the car should change direction, which would cause the DirectionControl.c module to change the state of the control signals to the H bridge. The LightNavigation module could also increase the speed of the car by directly altering the PWM signal sent to the H Bridge. In the next section, we will individually explore each of these components to understand how they behave both individually and as part of the overall system.

Figure 3 Software Design Outline



Steps for processing Photosensor and IR sensor values

1. A Hall sensor interrupt causes the normal speed calculations to be completed as found previously within our SpeedControl source code. With our independent project however, we now trigger a call to the LightNavigation module

2. Within LightNavigation we initialize a variable *delta* which will ultimately be the new PWM sent to the steering servo to determine the direction of the car. This value is initialized as shown in Equation 2.

$$Offset_{Photosensor}^{Current} = Photosensor_{Left} - Photosensor_{Right}$$

$$delta = K_p^{Photo}(Offset_{Photosensor}^{Current}) + K_d^{Photo}\left(Offset_{Photosensor}^{Current} - Offset_{Photosensor}^{Previous}\right)$$

(Equation 2)

We can see that the code allows proportional feedback (K_p^{Photo}) and derivative feedback (K_d^{Photo}) by comparing the last two photoresistor values.

3. With the initial *delta* value calculated, we now check to see if the car is headed directly towards the lightrope which would necessitate reversing the car's direction. Experimentally, we determined that if the leading photosensor[1] of the car was above a value of 600[2] and the absolute value of our *delta* was below 45, we would not be able to turn away from the lightrope and thus we would have to reverse the car's direction. If the leading photosensor of the car was above 700, we reversed the direction of the car because no matter what value the *delta* was, the car would not be able to turn away from the lightrope without hitting it because it was too close.

4. After examining the photosensor values, we check to see if the leading IR sensor is above the experimentally determined value of 500 and if the trailing light sensor value is below 500. If so, the car can safely reverse direction away from the object in front of it without running into any lightrope.

[1] The leading photosensor depended on what direction the car was traveling in, a one bit parameter maintained by the DirectionControl.c module
[2] See the upcoming discussion regarding AC to DC conversion

5. If no action was taken in Step 4, the module then checked whether the trailing IR sensor had a value above 500. If so, this indicated that "something" was chasing the car. To respond to this, the car's speed was increased depending on the magnitude of the IR sensor value (how close the object was to the car) up to 5 ft/s.

6. The side IR sensor values were now analyzed in order to create an overall *delta* value that accounted for both the photosensors and the IR sensor readings. The following adjustment shown in Equation 3 was made to the *delta* value found in Equation 2:

$$Offset_{IR} = IR_{LeadingLeft} + IR_{TrailingLeft} - IR_{LeadingRight} - IR_{TrailingRight}$$

$$delta = delta^{Equation\ 1} + K_p^{IR}(Offset_{IR})$$

(Equation 3)

The *delta* value was now updated in a manner that accounted for the readings from the photosensor and IR sensors such that a negative *delta* indicated that the car should turn left while a positive value indicated it should turn right.[3] A near zero value indicated the car did not need to turn. Notice that the last line of code in this section scales the *delta* value by the inverse of velocity of the car squared. This relationship, discussed in lecture, reflects the fact that as the car moves faster the wheels need to be turned less to attain a constant change in lateral position.

8. The *delta* value, after its magnitude is capped at 50 due to the constraints of the servo is then set as the PWM signal for the steering servo using the following call: PWM_Servo_WriteCompare(*delta*). Keep in mind that the *delta* value is appropriately scaled to account for the 4ms period of the servo in addition to the fact that ~1ms corresponds to a full left turn whereas ~2ms corresponds to a full right turn. The angle of the wheels and the acceleration of the car has now been set according to the photo and IR sensors. The process outlined in steps 1 – 8 is then repeated as long as the car is running.

<u>Process for updating IR and Photosensor values</u>

As mentioned in the previous section, we must be able to query the value of a particular IR or Photosensor in order for the car to operate properly. In order to keep track of these values, a

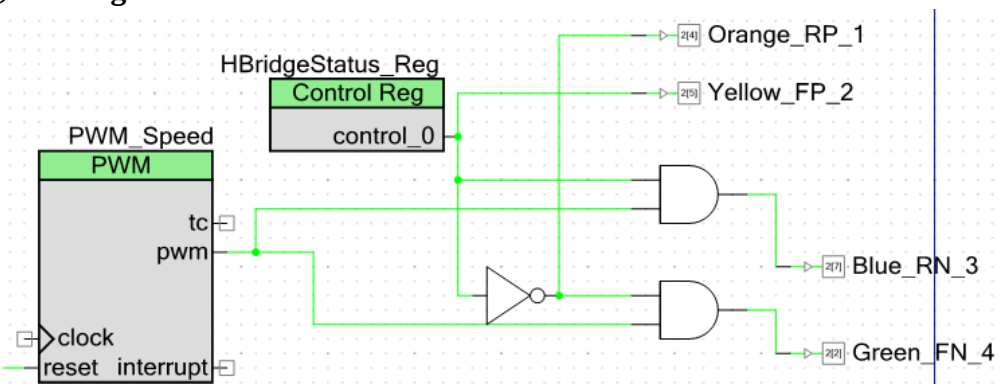
[3] The turn necessitated by either the lightrope, an object or a combination of both obstacles.

separate 1 kHz clock called msClock was tied to an interrupt that corresponded to the TimerIsr vector. Every ms, when a TimerIsr interrupt occurred, one of the ten sensor values would be updated by either using the lightNavSetValue()[4] or setIRValue() methods. An index, *lightSensorIndex*, between 0 and 9 was maintained with each index corresponding to a specific analog sensor input. The *lightSensorIndex* was used in conjunction with a MUX in order to select the analog sensor signal that would be recorded using our 16 bit (Admittedly an overzealous resolution) ADC converter. Therefore, after a period of 10 ms, all values of the sensors would be refreshed for the LightNavigation.c module to utilize.

H Bridge Control Signal Handling in DirectionControl.c

The final topic to discuss is how the H Bridge control signals mentioned in the hardware section were handled in manner that allowed safe control over the cars speed and direction. Figure 4 is an outline of the control logic that was implemented in our software:

Figure 4, H Bridge Control



From Figure 4, one can note that we had a one bit control register (HBridgeStatus_Reg) which was responsible for determining the state of the MOSFETS. When the control bit was zero, the MOSFETS were in a "Forward" state as detailed in Table 2. In this scenario, the Yellow and Blue output pins were low while the Orange pin was high. The Green pin then carried a PWM signal to the Forward N MOSFET. On the other hand, when the control bit was one, the MOSFETS were in a "Reverse" state as detailed in Table 2. In this scenario, the Orange and Green pins were held low while the Yellow pin was high. The Blue pin then carried a PWM to the Reverse N MOSFET. Note that the N MOSFETS received the PWM signal because they had

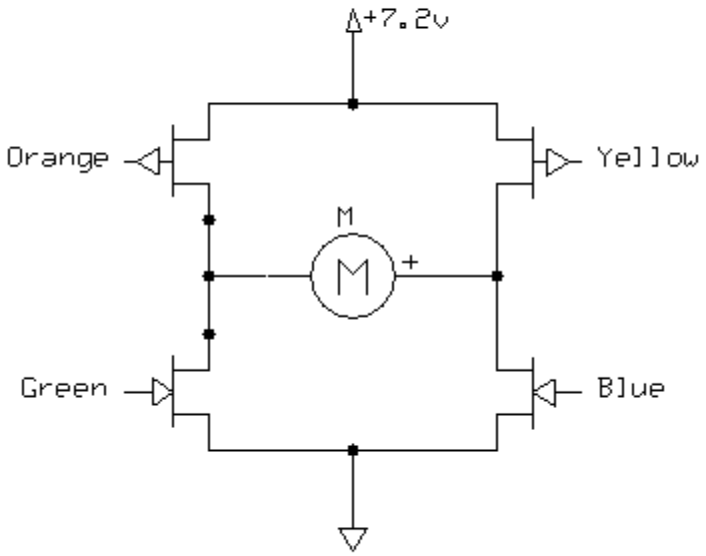
[4] This method also stored the previous photosensor value so that a temporal derivative could be calculated.

a quicker switching time than the P MOSFETS . In order to attain the “Neutral” state noted in Table 2, the control bit could be either high or low, the PWM signal would just be consistently zero.

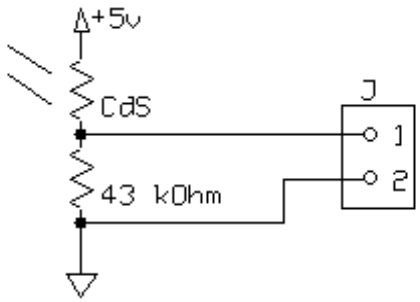
Thus in order to change the direction of the car, the control register bit was simply inverted and in order to control speed, the PWM was modified just as was done in our previous speed control module.

Schematics

H Bridge



Photosensor



Code

```
/* ========================================
 *
 * Main File, Chris Payne, Yuan Chen
 *
 * ========================================
*/
#include <device.h>
#include "SpeedControl.h"
#include "LightNavigation.h"
#include "NavControl.h"

void main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

      uint8 lightSensorIndex = 0;
      int i =0;

      //InitiateComPort();

      PWM_Speed_Start();
      PWM_Servo_Start();
      initSpeedControl();
      initNavControl();
      Video_Timer_Start();
      ADC_Light_Start();
      ADC_Light_StartConvert();
      LightMux_Start();
      LightMux_Select(lightSensorIndex);
      CYGlobalIntEnable; /* Uncomment this line to enable global interrupts.

      */
      for(;;)
    {
            /*
        int mv;
            int32 adval = ADC_Light_GetResult32();
            mv = ADC_Light_CountsTo_mVolts(adval);

            lightNavSetValue(lightSensorIndex, mv);

            lightSensorIndex = (lightSensorIndex + 1) % 4;
            LightMux_FastSelect(lightSensorIndex);*/
    }
}

/* [] END OF FILE */
```

```
// Yuan Chen
// Chris Payne
// ELE 302
//
// LightNavigation.c

#include "LightNavigation.h"
#include "SpeedControl.h"

int prevNavigationValues[] = {0, 0, 0, 0};
int navigationValues[] = {0, 0, 0, 0};

int irValues[] = {0, 0, 0, 0, 0, 0};

const int irThreshold = 1000;
const int navigationThresholds[] = {3000, 3050, 3200, 3000};
double navK = 1.7; // 0.7 kp, 1.0 kd works well
double navKd = 1.8;
double irKp = 3.0;

void setIrValue(int index, int value)
{
	if ((value - irThreshold) > 0)
	{
		irValues[index] = value - irThreshold;
	}
	else
	{
		irValues[index] = 0;
	}
}

int getIrValue(int index)
{
	return irValues[index];
}

int lightNavGetValue (int index)
{
	return navigationValues[index];
}

void lightNavSetValue(int index, int value)
{
	prevNavigationValues[index] = navigationValues[index];
	if ((value - navigationThresholds[index]) > 0)
	{
		navigationValues[index] = value - navigationThresholds[index];
	}
	else
	{
		navigationValues[index] = 0;
```

```
	}
}

int getSteeringPWMCmp()
{
	int result;
	double delta = 0.0;
	int hasIncreasedSpeed = 0;
	int netOffset = navigationValues[1] - navigationValues[2];
	int netDerivative = netOffset - (prevNavigationValues[1] -
prevNavigationValues[2]);
	int irOffset = irValues[1] + irValues[3] - irValues[2] - irValues[4];

	delta = (navK * (double)netOffset * .05);

	delta = delta + (navKd * (double)netDerivative * .05);



	// If the car is heading straight toward the light, stop it
	if (spdControl_getDirection() == 0 && getDriveState() == NORMAL_DRIVE)
	{
		if (navigationValues[0] > 450 && navigationValues[0] < 600)
		{
			delta = delta * ((double)navigationValues[0]/450.0);
		}
		if (navigationValues[0] > 600 && (delta < 45) && (delta > -45) ||
navigationValues[0] > 700)
		{
			spdControl_changeDirection();
		}
		if (irValues[0] > 500 && navigationValues[3] < 500)
		{
			spdControl_changeDirection();
		}
		else
		{
			if (irValues[5] > 500)
			{
				double newSpeed = getRefSpeed() + irValues[5]/1000.0;
				if (newSpeed > 5.0)
				{
					newSpeed = 5.0;
				}
				setRefSpeed(newSpeed);
				hasIncreasedSpeed = 1;
			}
		}
	}
	else if (spdControl_getDirection() == 1 && getDriveState() ==
NORMAL_DRIVE)
	{
		if (navigationValues[3] > 450 && navigationValues[3] < 600)
		{
			delta = delta * ((double)navigationValues[0]/450.0);
		}
```

```
            if (navigationValues[3] > 600 && (delta < 45) && (delta > -45) ||
navigationValues[3] > 700)
            {
                  spdControl_changeDirection();
            }
            if (irValues[5] > 500 && navigationValues[0] < 500)
            {
                  spdControl_changeDirection();
            }
            else
            {
                  if (irValues[0] > 500)
                  {
                        double newSpeed = getRefSpeed() + irValues[0]/1000.0;
                        if (newSpeed > 5.0)
                        {
                              newSpeed = 5.0;
                        }
                        setRefSpeed(newSpeed);
                        hasIncreasedSpeed = 1;
                  }
            }
      }

      // Object Avoidance from side sensors
      delta = delta + (irKp * (double)irOffset * .05);

      if ((irOffset > 500) || (irOffset < -500))
      {
            double newSpeed = getRefSpeed() + .2;
            if (newSpeed > 5.0)
            {
                  newSpeed = 5.0;
            }
            setRefSpeed(newSpeed);
            hasIncreasedSpeed = 1;
      }

      if (hasIncreasedSpeed == 0)
      {
            setRefSpeed(3.0);
      }

      delta = (1/(getRefSpeed() * getRefSpeed()) * delta)/9.0;

      if (delta > 50.0)
      {
            delta = 50.0;
      }
      if (delta < -50.0)
      {
            delta = -50.0;
      }

      result = 151 + (int)delta;
      return result;
      }
```

```c
/* ========================================
 *
 * Chris Payne, Yuan Chen
 * ELE 302, DirectionControl.c
 *
 * ========================================
*/

#include "DirectionControl.h"
#include <device.h>

// Global variable that tracks the car's current direction
// Initialize to 0, 0 is forward
int carDirection = 0;

// Changes the direction of the car to be opposite of its current direction,
// and sets the global variable carDirection to the correct value
void DirectionControl_ChangeDir()
{
	// Query the HBrigeStatus Register value
	carDirection = HBridgeStatus_Reg_Read();

	if (carDirection == 0)
	{
		carDirection = 1;
		HBridgeStatus_Reg_Write(1U);
	}
	else
	{
		carDirection = 0;
		HBridgeStatus_Reg_Write(0U);
	}

}

// Returns the current direction of the car
int carDirection_State()
{
	return carDirection;
}




	/* [] END OF FILE */
```

```
// Yuan Chen
// Chris Payne
// ELE 302
// SpeedControl.c

#include <device.h>
#include "SpeedControl.h"
#include "LightNavigation.h"
#include "DirectionControl.h"
#include "NavControl.h"

//Global Variables
double kp = 8.0; //original value is 9.7
double kd = 0.6;
double ki = .80; //original value is 1.2

double measuredSpeed = 0.0;
double error = 0.0;
double errorIntegral = 0.0;

double errorHistory[] = {0.0, 0.0, 0.0, 0.0, 0.0};
double errorDerivative = 0.0;

int timeDifferences[] = {0, 0, 0, 0, 0};
int magIndex = 0;
int noClickTime = 0;

int noDirChangeTime = 0;

double refSpeed = 3.0;

int transitionTime = 0;
int direction = 0;
int driveState = NORMAL_DRIVE;
int opStatus = MOTOR_ON;

// Variables for measuring and storing light sensor values
int lightSensorIndex = 0;

int getDriveState()
{
      return driveState;
}

int spdControl_getDirection()
{
      return direction;
}

void spdControl_changeDirection()
```

```
{
	transitionTime = 0;
	if (driveState == NORMAL_DRIVE)
	{
		driveState = TRANSITION;
		PWM_Speed_WriteCompare(0U);
	}
}

double getRefSpeed()
{
	return refSpeed;
}

void setRefSpeed(double s)
{
	refSpeed = s;
}

// An interrupt service routine for the speed PWM
CY_ISR(TimerIsr)
{
	int i;
	int mv;
	int32 adval = ADC_Light_GetResult32();

	mv = ADC_Light_CountsTo_mVolts(adval);



	if (lightSensorIndex < 4)
	{
		lightNavSetValue(lightSensorIndex, mv);
	}
	else
	{
		setIrValue(lightSensorIndex - 4, mv);
	}

	ADC_Light_StopConvert();
	lightSensorIndex = (lightSensorIndex + 1) % 10;
	LightMux_FastSelect(lightSensorIndex);
	ADC_Light_StartConvert();
	for (i = 0; i < 5; i++)
	{
		timeDifferences[i] = timeDifferences[i] + 1;
	}

	errorIntegral = errorIntegral + (refSpeed - measuredSpeed)/1000.0;
	noClickTime = noClickTime + 1;
	if (noClickTime > 100 && refSpeed > 0.5 && driveState == NORMAL_DRIVE)
	{

	//PWM_Speed_WriteCompare((uint8)(.3*(double)PWM_Speed_ReadPeriod()));
		PWM_Speed_WriteCompare(30U);
	}
```

```
	//Count the time since the last direction change, cap at 2000 to avoid
overflow

	if (noDirChangeTime < 2000)
	{
		noDirChangeTime++;
	}

	//Code for changing directions
	if (driveState == TRANSITION)
	{
		transitionTime++;
	}

	if (transitionTime > 5 && noDirChangeTime > 1000)
	{
		DirectionControl_ChangeDir();
		direction = carDirection_State();
		transitionTime = 0;
		driveState = NORMAL_DRIVE;
		refSpeed = 3.0;
		noDirChangeTime = 0;
	}

}

// An interrupt service routine for the hall sensor
CY_ISR(HallSensorIsr)
{
	int timeTicks;
	double speed;

	magIndex = magIndex % 5 + 1;
	timeTicks = timeDifferences[magIndex - 1];
	noClickTime = 0;

	if (timeTicks == 0)
	{
		measuredSpeed = 0.0;
		errorHistory[magIndex - 1] = refSpeed;
	}
	else
	{
		speed = (19.47/(double)timeTicks) * 1000.0/(2.54 * 12.0);
		measuredSpeed = speed;
		errorDerivative = ((refSpeed - measuredSpeed) -
errorHistory[magIndex - 1]) * 1000.0/(double)timeTicks;
		errorHistory[magIndex - 1] = refSpeed - speed;
	}

	timeDifferences[magIndex - 1] = 0;

	if (driveState == NORMAL_DRIVE)
	{
		PWM_Speed_WriteCompare((uint8)getNewSpeedSetting());
	}
	if (refSpeed == 0.0)
```

```
	{
		PWM_Speed_WriteCompare(0U);
	}
	PWM_Servo_WriteCompare((uint8)getSteeringPWMCmp());

	//PWM_Servo_WriteCompare((uint8)getNavServoPWMCmp());
}



// An interrupt servie routine for the switch
CY_ISR(KillSwitchIsr)
{
	int i;
	if (opStatus == MOTOR_ON)
	{
		opStatus = MOTOR_OFF;
		isr_msTimer_Stop();
		isr_HallEffect_Stop();

		PWM_Speed_WriteCompare(0U);

		for (i = 0; i < 5; i++)
		{
			timeDifferences[i] = 0;
			errorHistory[i] = 0.0;
		}

		measuredSpeed = 0.0;
		errorIntegral = 0.0;
		magIndex = 0;
		noClickTime = 0;
		PWM_Servo_WriteCompare(151);
	}
	else
	{
		isr_HallEffect_Start();
		isr_msTimer_Start();
		isr_HallEffect_SetVector(HallSensorIsr);
		isr_msTimer_SetVector(TimerIsr);
		opStatus = MOTOR_ON;
		refSpeed = 3.0;

	PWM_Speed_WriteCompare((uint8)(.3*(double)PWM_Speed_ReadPeriod()));
	}
}

int getNewSpeedSetting()
{
	double currentError;
	double dutyCycle = 0.0;

	currentError = refSpeed - measuredSpeed;

	dutyCycle = (currentError * kp/25.0) + (errorDerivative * kd/25.0) +
(errorIntegral * ki/25.0);
	if (dutyCycle > 1.0)
```

```
	{
		dutyCycle = 1.0;
	}
	if (dutyCycle < 0.0)
	{
		dutyCycle = 0.0;
	}
	return (int)(dutyCycle * (double)PWM_Speed_ReadPeriod());
}

// Code for initializing the speed controller
void initSpeedControl()
{
	/*isr_KillSwitch_StartEx(KillSwitchIsr);
	isr_HallEffect_StartEx(HallSensorIsr);
	isr_msTimer_StartEx(TimerIsr);*/



	isr_KillSwitch_Start();
	isr_HallEffect_Start();
	isr_msTimer_Start();
	isr_KillSwitch_SetVector(KillSwitchIsr);
	isr_HallEffect_SetVector(HallSensorIsr);
	isr_msTimer_SetVector(TimerIsr);
}
```