Navigation Lab Write Up

Haley Chow and Dana Fesjian

March 31, 2016

1 Introduction

In part two of Car Lab, we were tasked with updating our car so that it could follow the black tape of the track. In order to do this, we were given a camera to attach to a mast that we designed, angled down so it could see the track. We then focused on processing the information outputted from the camera into a form that would allow us to know whether to adjust the wheels and choose which direction the wheels should be adjusted.

2 Hardware

2.1 New Voltage Regulator Circuit

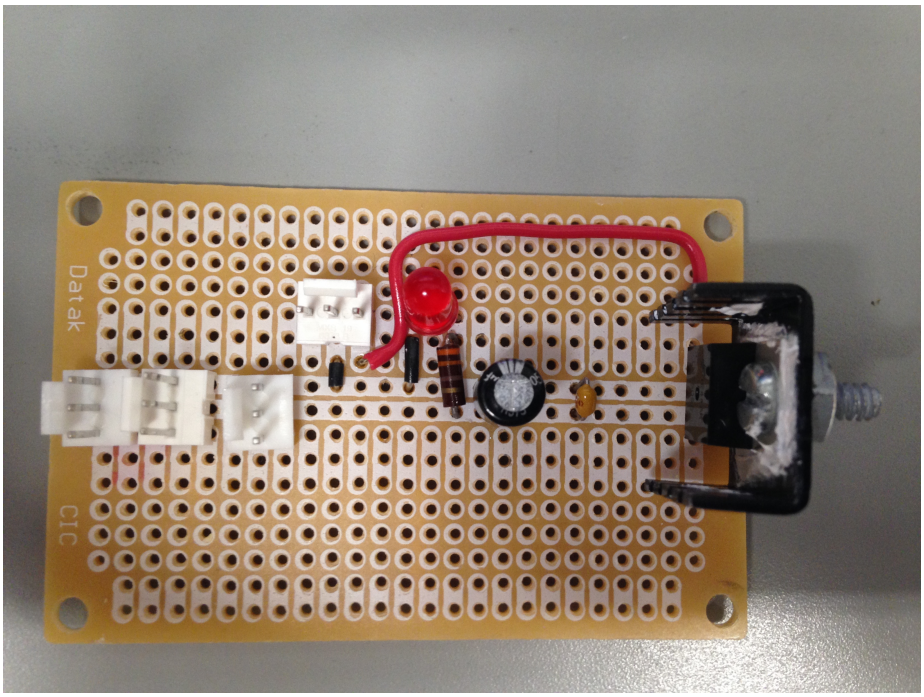

Figure 1: Voltage Regulator Circuit

Our reasoning for creating a new Voltage Regulator Circuit was that it would be easily accessed by any future boards that we would create in the front of the car. When we built the Voltage Regulator circuit, we were not aware of how much we would value the placement in the future. Our two other circuits that we built for part two relied on getting power from this new circuit.

Instead of simply copying the original voltage regulator circuit from part one, we built the circuit that was shown in the camera's data sheet. This circuit used different values of capacitors, but the chip that we used was the same and it had very similar properties to the first circuit. The circuit was primarily built to power the camera, so we wanted to make sure that we followed the circuit schematic.

2.2 Camera Circuit

The camera circuit relied on the LM1881 chip that would split the output that the camera generated into parts that we could use. The camera itself outputted a low amplitude high frequency signal of about 60 frames per second that included information about each of the pixels that it saw on every line of every frame. We weren't interested in every single bit of information from the camera so we needed to break it up. Using the LM1881 Video Sync chip, we were able to create a circuit that would output the composite syncs and the vertical syncs separately.

This board also contained many other miscellaneous additions. We added a location for the Servo J-Connector on this board so we could power it and control the input from the PSOC. We also added a 75 Ohm resistor to the camera signal to match impedances. At first we took the signal before the capacitor that was necessary for the LM1881 circuit and used that as our camera signal, but we found that after the capacitor, it added a DC value to the camera output that was more helpful to us when we made a comparator circuit because it shifted up the output signal from the camera, making the values easier to compare.

2.3 Comparator Circuit

We experienced many difficulties with our comparator circuit, but it eventually worked exactly how we wanted it to after creating 3 different boards. We tested the comparator circuit many times on our oscilloscope whenever we thought it wasn't working. The main goal for the comparator circuit was to take the busy output from the composite camera signal and turn it into a signal that would solely indicate whether there was a black line in the camera frame. The output of the camera dipped low when a black line was in the frame due to the fact that the color black is represented by 0. Using the comparator circuit, we were able to make those dips more pronounced and get rid of information we did not need.

We built the comparator circuit using an LM311 chip and flipped the output. We added a pull-up resistor to the output and then set our V_{EE} value to Ground, which meant that if our input voltage was higher than the DC voltage we were going to compare it to, we would see the always high output drop low. To get the value of DC voltage, we used a 10kOhm potentiometer which allowed us to easily adjust the output of the comparator.

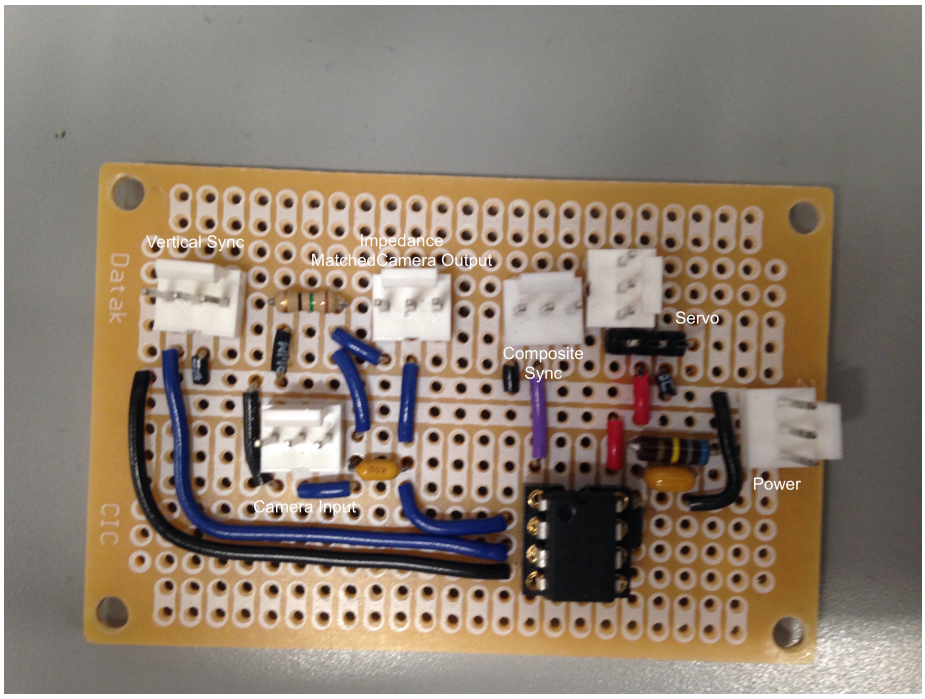

Figure 2: Camera Circuit

The downside of this circuit was that it also dipped low for each composite sync and stayed low for the burst signal which would always come right after a composite sync to start the frame. We had to account for these dips in our code so that the PSOC didn't treat each drop in the comparator output as a separate black line.

3 Software

3.1 Method

Initially when we started to focus on software and how we were going to program our car to go around the track, we started thinking about shifting angles. We thought the best way to go about trying to follow the black line was for the car to calculate its angle compared to the black tape and then make a small adjustment that would get the angle closer to zero. However, we eventually realized there was a much better way for the car to calculate its direction in relation to the tape by just measuring how far the black line in the camera frame was from being straight.

We had some trouble envisioning how to use our output from the comparator and translate that into determining whether the car was straight or angled compared to the black tape. Our first attempt involved using many different interrupts that slowed down the speed of our program, and we found it difficult to configure the all of the interrupts in a way that allowed the timing of each one to happen properly. In the end, we ended up with a schematic that only

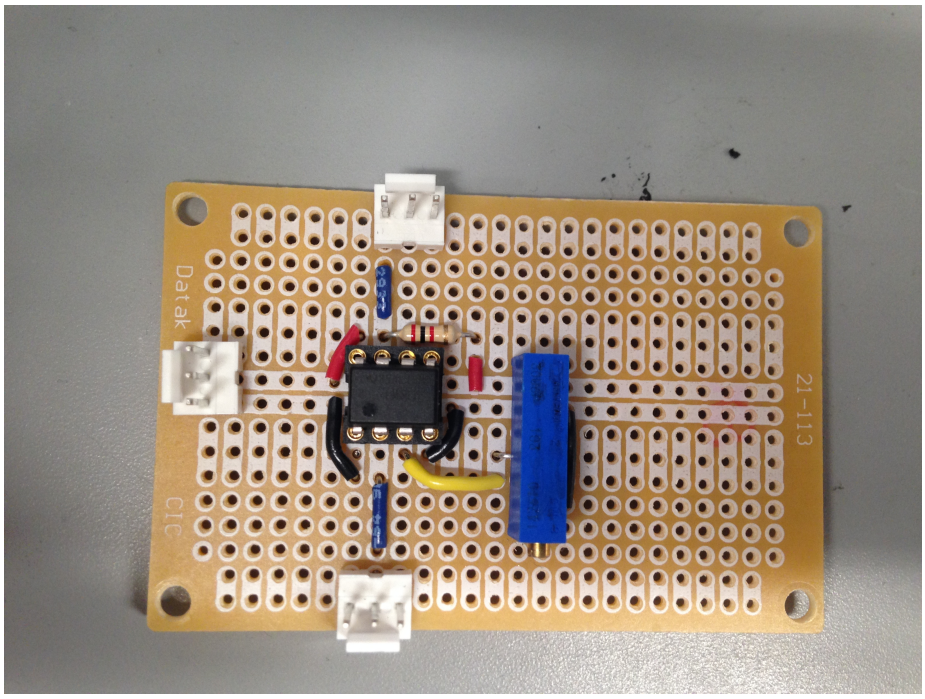

Figure 3: Comparator Circuit

included one interrupt, which was much more efficient.

3.2 Reading in the Camera Signal

Our goal was to take the several outputs we had from the various hardware circuits that we wired and use our PSOC to process that information. We did not use the actual camera's signal, but instead used the output of the comparator instead.

3.3 Vertical Syncs

The vertical syncs indicated the start of a new frame outputted by the camera. Our goal was to calculate the location of the black line in each frame, so this was a natural breaking point for us to start our calculation.

3.4 Composite Syncs

The composite sync indicated how many lines were in each frame. The composite syncs also caused our comparator to act similarly to how it would act in the case of a black line. Since they indicated each line in the frame, counting composite syncs meant that we could look at the same line in the frame every time. We used the vertical syncs to reset a counter that counted 100 composite syncs before it would look at the comparator input.

3.5 Comparator Signal

We used the comparator signal to indicate how far the line in the frame was from being straight. In order to do this, we calculated the number of microseconds after the last composite sync was measured until the next drop in the comparator circuit. We then lined the track up so that the car was perfectly straight and printed off the number of microseconds for the interrupt to happen after the last composite sync is counted. When we did this, the timer outputted between 28 and 29 microseconds. Therefore, we knew the time of when the car was straight and were able to compare the angle the car was actually at to that reference point.

3.6 Proportional Control

Once we were able to get the car to line up with the track, we moved on to perfecting its control. The first thing we did was add a kp value that would multiply the error by some value to allow the car to adjust itself. When we tested just the kp value we found that it wasn't necessary to have other integral or derivative control in order to get our car around the track.

One of the most difficult aspects for us to wrap our heads around was that our duty cycle needed to adjust the time of the pulse from the PWM in the firmware, rather than the actual duty cycle of our Servo PWM. Once we had configured the PWM to have a fast enough clock to give us adequate control over our servo, we also needed to shift the center of our duty cycle calculation within our dutyCycle function. Since 1.52 miliseconds was straight for the car servo, 152 became the center of our error calculation.

One final feature that we needed to add was making sure there was a MAX and MIN duty cycle in our Servo PWM. In this way, we knew that we wouldn't be forcing the servo pin to output a signal that would break the servo. In addition, we wrote code that would account for if the car did not see a track and set the PWM output to 0 if that were the case.

4 Results

We tested our car on the track a few times before we demoed because we had to accurately set our kp value. In the end we needed to increase our kp and have the car move at 3.2 feet per second in order for it to follow the track perfectly. We added a small calculation in our code to sense when the car had gone around the track 2 times by calculating how many times the hall effect sensor saw the magnets. We calculated it should see a magnet 1500 times, so we set it to move for that long. When we demoed, we went around the track two times in about 1 minute and 3 seconds.

5 Code

```
#include <device.h>
#include <stdio.h>

double ref = 3.2;
```

```
double angleRef = 0;
double inchPerPing = 1.5707963268;
double ticks = 0;
int clockPeriod = 0;
int index = 0;
double avgError = 0.0;
double lastError = 0;
double integral = 0.0;

int numCompositeSyncs = 0;
int numFrames = 0;
double deltaT = 0;
double elapsedTime = 0;


//calculate speed
double carSpeed(int cycles)
{
    //distance traveled converting from pings
    double speedInches = inchPerPing/((double)cycles/10000.0);
    double speedFeet = speedInches/12.0;
    return speedFeet;
}

//print to LCD function
void printDouble(double d, double e) {
    char strbuff1[9];
    char strbuff[9];
    sprintf(strbuff, "%f", d);
    LCD_Position(0,0);
    LCD_PrintString(strbuff);
    sprintf(strbuff1, "%f", e);
    LCD_Position(1,1);
    LCD_PrintString(strbuff1);

}

//calculate time of each cycle
double getSeconds(int cycles) {
    return ((double) cycles)/10000.0;
}



//do PID control based on interrupt
CY_ISR(inter)
{
    int captureTime = Speed_Timer_ReadCounter();
    int elapsedCycles = (clockPeriod-1) - captureTime;

    double error = 0;
    double duty = 0;
    int MAXDUTY = 70;
    int MINDUTY = 0;
```

```
    double derivative = 0;

    //control coefficients
    double kp = 5;
    double ki = 7;
    double kd = 0;

    //calculate input to the PWM
    double speed = carSpeed(elapsedCycles);
    error = ref - speed;

    ticks++;
    printDouble(ticks,0.0);


    //reset counter
    Speed_Timer_WriteCounter(clockPeriod-1);
    //derivative term
    if (lastError != 0){
        derivative = ((error - lastError)/getSeconds(elapsedCycles));
    }
    lastError = error;

    //integral term
    integral += error*getSeconds(elapsedCycles);


    duty = (kp*error) + (ki*integral) + (kd*derivative);

    if (ticks >= 1500)
        duty = MINDUTY;

    if (duty > MAXDUTY)
        duty = MAXDUTY;
    else if (duty < MINDUTY)
        duty = MINDUTY;
    Speed_PWM_WriteCompare((int)duty);

    //calculate avg error
    index++;
    avgError += error;
    if (index > 100){
        //printDouble(avgError/index, speed);
        avgError = 0.0;
        index = 0;
    }

}


//interrupt from timer after getting input from counter
//counter gets composite syncs and vertical syncs as inputs
```

```
CY_ISR(inter2){

    //get time
    double referenceTime = 28.5; //need to check if it hits next
        composite sync
    uint32 captureTime = Servo_Timer_1_ReadCapture();
    uint32 elapsedCycles = (Servo_Timer_1_ReadPeriod()-1) - captureTime;

    double dutyCycle = 0;
    double kp = 2.5;
    int MAXDUTY = 200;
    int MINDUTY = 100; //100

    elapsedTime = (double)elapsedCycles; //time in microseconds
    //time - ref time and then get pos or neg error and then decide if
        move right or left
    deltaT = elapsedTime - referenceTime;

    dutyCycle = kp*deltaT + 152;

    if (dutyCycle > MAXDUTY)
        dutyCycle = MAXDUTY;
    else if (dutyCycle <= MINDUTY)
        dutyCycle = 0;


    Servo_PWM_WriteCompare((uint16)dutyCycle);


    Servo_Timer_1_ReadStatusRegister();




}

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

    CyGlobalIntEnable;

    //call hall effect interrupt
    hall_inter_Start();
    hall_inter_SetVector(inter);

    //start timer and call timer interrupt
    Speed_Timer_Start();
    clockPeriod = Speed_Timer_ReadPeriod();

    Speed_PWM_Start();
    Speed_PWM_WriteCompare(40);
```

```
    New_Sync_Counter_Start();


    steer_inter_Start();
    steer_inter_SetVector(inter2);


    //start servo timer
    Servo_Timer_1_Start();

    Servo_PWM_Start();
    //Servo_PWM_WriteCompare(30); //chose this number arbitrarily, tried
        to use this to check if servo was working

    LCD_Start();

    for(;;)
    {
        /* Place your application code here. */
    }
}

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