Portland State University

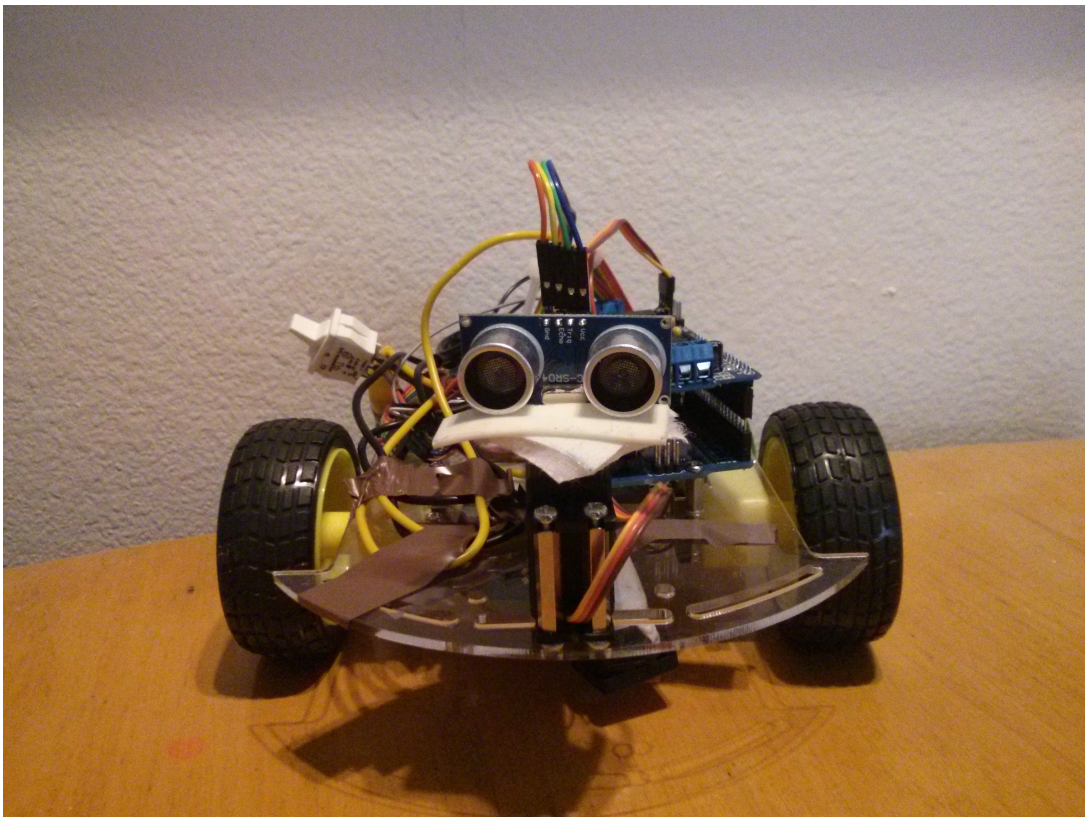
Embedded Systems on FPGAs

ECE544

# Autonomous Robot Car

Erik Rhodes Caren Zgheib

June 8, 2014



# Contents

# 1 Introduction

We created an autonomous robot car (SaMMY) that can successfully navigate its way through a given area without hitting any objects . It uses the popular **Arduino** platform and libraries along with a ultrasonic sensor for object detection.

# 2 Hardware Components

There are many different components needed to successfully and robustly create an autonomous robot. The bulk of our design used a robot kit we ordered online (ref). We also added various parts to make the production time more convenient, including switches, additional batteries, and various cables. The important components can be seen in Figure 1 and are described in Table 1.

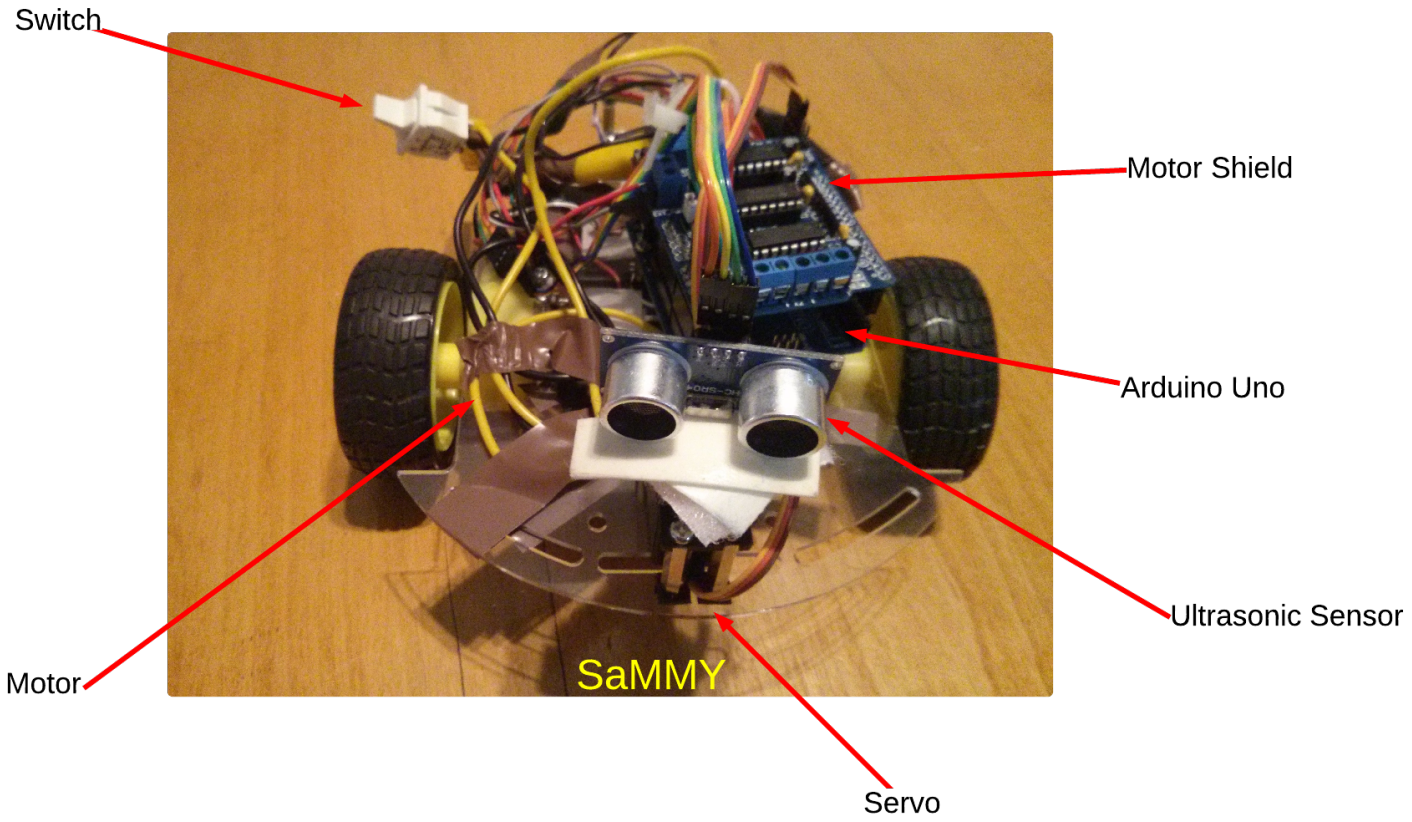



*Figure 1: Block Diagram of Control System*

Blah Blah blah fill this section lines
lines
lines

| Part Name | Function | Details |
|---|---|---|
| 1. | Motor | insert link |
| 2. | 180°rotation using PWM | insert link |
| 3. | Sonar sensor for object detection | insert link |
| 4. Arduino Uno | Microcontroller Board | insert link |
| 5. | Motor Shield | insert link |
| 6. Switch | On/Off power button | N/A |

*Table 1: Hardware Components List*

# 3 Robot Control

## 3.1 Functions

The software portion was written in the **Arduino** IDE, and we modified some of the existing libraries. The main functions for our project are listed in Table 2.

| Component | Function | Details |
|---|---|---|
| Motor | Drive Forward | Both wheels moving forward at predefined speed |
| | Drive Backward | Both wheels moving backward at predefined speed |
| | Turn Left/Right | Both wheels spin for "in-place" turn of 90° |
| | Veer Left/Right | Only outside wheel turns and vehicle does not stop |
| | U-Turn | Backs up then turns 180° |
| | Victory Dance | After certain time, bot does various spins |
| Sonar + Servo | Look Forward | Scans a 40°range in front while moving forward |
| | Look Left/Right | Scans 90°left/right to determine next turn |

*Table 2: Robot Functions*

After initializing our components, the software ran an infinite loop which scans for objects and makes decisions on which direction to go. The main control flow is seen in listing 1 below.

*Listing 1: Control Loop*

```
void loop() {
  if (scanClear())  // If clear to go forward
  {
    drive_forward();
  }
  else              // if path is blocked
  {
    freewheel();    // stop
    lookLeft();
    int distanceLeft = getAverageDistance();

    lookRight();
    int distanceRight = getAverageDistance();

    // re-center the ultrasonic
```

```
16      servo_position(CENTER);
17
18      // go the least obstructed way
19      if (distanceLeft > distanceRight && distanceLeft > dangerThreshold)
20      {
21        drive_backward();
22        delay(400);
23        rotate_left();
24      }
25      else if (distanceRight > distanceLeft && distanceRight > dangerThreshold)
26      {
27        drive_backward();
28        delay(400);
29        rotate_right();
30      }
31      else    // equally blocked or less than danger threshold left or right
32      {
33        freewheel();
34        delay(20);
35        drive_backward();
36        delay(500);
37        u_turn();
38      }
39    }
40 }
41
```

### 3.1.1 PID

The more prevalent method, PID, involves making multiple calculations to predict the accurately control the behavior of the output. The proportional (P) method involves using the previous error margin to calculate the next appropriate one. The integral (I) method calculates how the system behaves over time, and the derivative (D) calculates how fast the output is changing at that point in time. Our implementation is seen in listing 2.
Using a specific arrangement of these parameters will yield a large increase in accuracy, so tuning the application is desirable. When tuning, we first attempted to follow the *Ziegler-Nichols* method. After understanding how each method changes the output, we modified the parameters as we saw fit to acquire the most accurate configuration for our control system.

*Listing 2: PID Algorithm*

```
1     for (smpl_idx = 1; smpl_idx < NUM_FRQ_SAMPLES; smpl_idx++)
2     {
3         delay_msecs(100);
4
5         // get count from light sensor and convert to voltage
6         sample[smpl_idx] = LIGHTSENSOR_Capture(LIGHTSENSOR_BASEADDR, slope, offset,
              is_scaled, freq_min_cnt);
7         volt_out = (-3.3 / 4095.0) * (sample[smpl_idx]) + 3.3;
8
9         // calculate derivative;
10        error = setpoint - volt_out;
11        deriv = error - prev_error;
12
13        // calculate integral
```

```
14          if (error < setpoint/10) integral += error;
15          else integral = 0;
16
17          // Control offset is gotten from characterization
18          volt_out = offset + (error * prop_gain) + (deriv * deriv_gain) + (integral
                * integral_gain);
19          duty_out = (volt_out)* (MAX_DUTY+1)/VOLT_MAX;
20
21          // establish bounds
22          if (duty_out < 1) duty_out = 1;
23          if (duty_out > 99)duty_out = 99;
24
25          // activate PWM
26          Status = PWM_SetParams(&PWMTimerInst, pwm_freq, duty_out);
27          if (Status == XST_SUCCESS)  PWM_Start(&PWMTimerInst);
28      }
```

### 3.2 Peripheral Interface

The **TSL237** light-to-frequency converter outputs a period that directly corresponds to the intensity of the light emitted from the LED. The PWM detection module receives this information and converts it to a "count", which is then scaled to fit within the parameters (between 0 and 4095) needed for our control calculations. These functions are handled in the light sensor driver. The driver initially converted the scaled counts to a voltage, but we later decided to move that functionality to the program application. The peripheral has the user-visible registers described in table 3.

| Register | Number | Format | Description |
|---|---|---|---|
| Control | slv_reg0 | RESERVED[30:0], EN | Enable of the peripheral |
| Status | slv_reg1 | RESERVED[30:0], EN | Current settings of the peripheral |
| HighTime | slv_reg2 | HighTime[31:0] | Detected high time count |
| Period | slv_reg3 | Period[31:0] | Detected period count |
| SpareReg1 | slv_reg4 | RESERVED[31:0] | No specific purpose |
| SpareReg2 | slv_reg5 | RESERVED[31:0] | No specific purpose |

*Table 3: Lightsensor peripheral registers*

### 3.3 Frequency detection

The measurements output by the light sensor were received by a hardware module that interpreted the data. This Verilog program closely resembles the PWM detection module done in project 1. It increments the count on each clock edge, depending on whether the input is high or low. Once a full period is detected, the high count and period are sent to registers which are able to be read by the light sensor driver.

### 3.4 User Controls

The program starts by characterizing the system. This allows the initial scaling to be performed, which varies by the amount of light detected by the light sensor. After this is done, the default

control parameter values are displayed and the user is given a chance to change them. The type of test and starting input voltage can also be selected. Once these have been configured, the user starts the test with the rotary button. When it has finished, long pressing on the rotary button sends the data to the computer connected serially via the UART port. If the user wants to change the test, they can modify the switch values and update the LCD by pressing on the rotary button. While the user interface remained close to the recommended specifications, it also included a few features that made it amazingly incredible. The control assignments are seen in table 4.

| **Name** | **Value** | **Function** |
|---|---|---|
| Switch[1:0] | 00 | Bang Bang |
| Switch[1:0] | 01 | PID |
| Switch[1:0] | 10 | Unused |
| Switch[1:0] | 11 | Characterization |
| Switch[2] | 0 | Vin Low |
| Switch[2] | 1 | Vin High |
| Pushbutton | North | Move Cursor |
| Pushbutton | East | Increase Value |
| Pushbutton | West | Decrease Value |
| LCD Display | Row 1 | PID Values |
| LCD Display | Row 2 | Setpoint and Offset |
| Rotary Encoder | Clockwise | Increase Setpoint |
| Rotary Encoder | Counter-Clockwise | Decrease Setpoint |
| Rotary Encoder | Press Button | Next section |
| Rotary Encoder | Long Press Button | Initiate test |

*Table 4: Nexys3 Controls*

# 4 Conclusion

`Servo and Protecto` was quite fun for us to build and was certainly the biggest "crowd pleaser" out of all the projects we've made. The assembly perhaps took the least amount of time, although we did face problems with a few of the parts not being built correctly (mainly the battery holder and motor plate mount). Fixing the power problems was the most frustrating part of the project, but once we finished everything, the final product was quite rewarding. We both spent equal times on all the different aspects of this project. Listed below are some of the challenges we faced.

## 4.1 Challenges

- **Power Issues**: After the initial assembly, we were not able to properly move the servo and control the motors. Our initial thought was inadequate power, but the correspondent we communicated with ensured us that it could run perfectly fine on 6V of AA batteries. Even after upping the voltage we saw problems, which were due to the power demands of our servo. After increasing the power to 9V of AA batteries the problem diminished.

- **Motor control**: The motors we received are not precise enough to deliver equal power on

both sides. This caused inevitable veering of the robot. We were able to use our own veering function to correct a lot of this, but were not able to control the bot perfectly.

- **CMUcam**: We were able to calibrate and test the CMUcam, which was intended for color detection. However, communicating the frames from the camera to the Arduino proved difficult. Even the example code would not work on our device. We were not able to solve the communication error before the project was due.
- **Erroneous Sonar Readings**: Occasionally the ultrasonic sensor picked up erroneous values that would usually be below 10cm (possibly due to ghost echoes). This cause the robot to stop immediately. We adjusted this by filtering out the first ping that it received below 10cm.

### 4.2 Future Additions

The Arduino platform and components in this project provide a large amount of functionality. Because of this, the amount of improvements is virtually limitless. Our goals in the future could involve adding:

- **Battery Pack**: Buying a powerful, rechargeable battery pack would save money, improve performance and decrease hassle.
- **External Apparatus**: We initially wanted to attach a simple device that could be triggered based on certain events. While launching or shooting an object would have been interesting, in most cases this would require PID algorithms, which would involve sufficiently more work and tuning.
- **Color Detection**: One of our original ideas (and part of the reason for the name **Servo and Protecto**) was to use color detection to differentiate between "friends" and "enemies". There would be certain objects that the robot would purposely avoid, and others that it would just run over.
- **Infrared**: Infrared sensors can be used to detect edges, and they could also be used as an external controller/remote.
- **Object Following**: The sonar and/or color detector could be used to follow an object instead of avoid them. This would be useful when if we attached a physical arm or apparatus of some sort.

## 5 References

Include oddwires stuff?