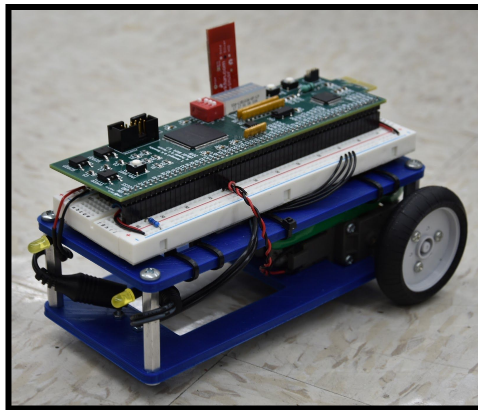
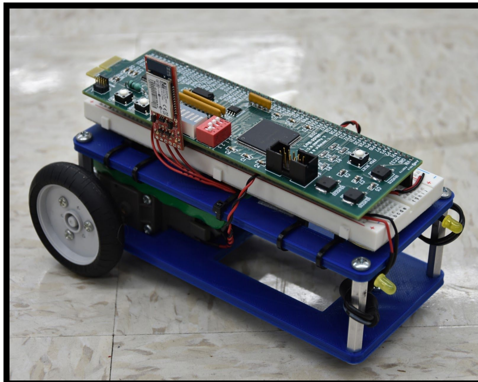
B.R.O.C.
Bluetooth Remotely-Operated Car

Final Project Report
December 13th, 2019

Omar Aleman and Leonardo Vilchez





Abstract:

The goal of this project was to make a remote control car using a bluetooth chip, ATSAM microcontroller, FPGA, LEDs, gearbox, motors, wheels, and a chassis to hold it all together. The wireless connection between the bluetooth chip and the PC with bluetooth support allows a keyboard to be used as the controller that sends commands to the motors on the RC car. The μMudd board with the FPGA and microcontroller, the bluetooth chip, and all the external circuitry are all contained on one breadboard so that it is compact enough to fit on the RC car. The functionality of the LEDs is based on the desired movement of the RC car. Thus, the user presses specific keys on the keyboard to control the movement of the RC car equipped with headlights and taillights in an intuitive manner.

I. Introduction:

Remote Controlled (RC) cars are battery-powered model cars that can be controlled from a distance using a remote control. RC cars are exciting and fun for all ages, including Mudd professors, which is why our final project is implementing a Bluetooth interface between a PC and a bluetooth chip to interpret human input for controlling a small robot car.

The top-level block diagram of the system is shown in Figure 1.1. The python script (see Appendix A) establishes a serial port connection between the bluetooth chip and the PC. It wirelessly sends a character corresponding to a user input command on the keyboard to the bluetooth chip (see Table 1.1). Once the connection is established, the bluetooth chip acts as a transparent data gateway between the PC and the microcontroller. It is connected to the microcontroller via UART, thus the microcontroller receives the character from the PC via UART. The microcontroller interprets the data, powers the desired LEDs, and sends an encoded data signal to the FPGA via SPI. The FPGA then decodes the data and finally sends two PWM signals to the motors.

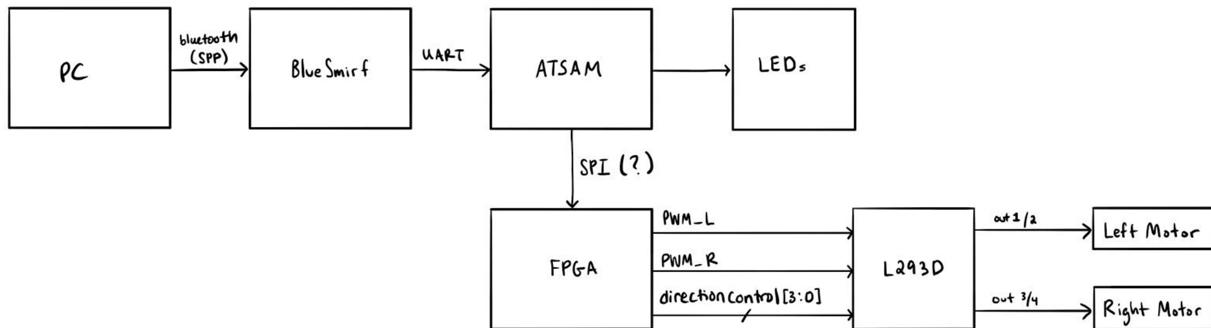


Figure 1.1: Top-Level Block Diagram

Keyboard Input	UART character	Desired Function
‘w’	‘w’	straight forward
‘s’	‘s’	straight backward
‘a’	‘a’	spin left
‘d’	‘d’	spin right
‘w’ + ‘a’	‘q’	forward left
‘w’ + ‘d’	‘e’	forward right
‘s’ + ‘a’	‘z’	backward left
‘s’ + ‘d’	‘x’	backward right
else	‘p’	pause

Table 1.1: Keyboard Input and corresponding UART character sent over bluetooth

II. New Hardware

The new hardware we worked with was the BlueSMiRF Silver Bluetooth Modem from SparkFun. It established a wireless connection (serial port connection) between the PC and the RC car.

The user can enter two modes: command and data mode. Command mode is used to configure the bluetooth module. Characteristics such as device name, baud rate, PIN code, and data rate can be adjusted in command mode. We chose to work with the default settings (115200 baud rate, 8 bits of data, 1 stop bit, no parity) for convenience. We also had to configure the UART settings of the microcontroller to accept the data from the chip. The driver settings of the PC must also be updated under Device Manager to match the bluetooth settings [1]. In data mode, the bluetooth module acts as a transparent data gateway. Any data received over the bluetooth connection is routed to the chip's TX pin. Any data sent to the chips RX pin is sent over the bluetooth connection.

The bluetooth chip has two LEDs ("Stat" and "Connect") that indicate the status of the module. To connect the PC and the bluetooth chip, the chip paired to the PC as a device. We created a python script to establish a serial interface to communicate with the bluetooth chip. The provided python script also has a function to enter command mode and configure the bluetooth chip settings, but we weren't able to get it working correctly.

We also worked with the L293D motor driver that acts as a current amplifier for the two motors on the RC car. The PWM signals from the FPGA are routed to the ENABLE pins and the motor wires are connected to the OUTPUT pins of the L293D chip.

III. Schematics

The schematics in Figure 3.2 shows the connections between the FPGA and microcontroller on the μMudd board, the LEDs, the BlueSMiRF chip, the motor driver and the motors. Figure 3.1 shows how we fit circuits in the schematic on the breadboard.

The RX and TX pins on the BlueSMiRF chip are connected to the TX and RX pins on the microcontroller so that they can communicate via UART. The BlueSMiRF is powered with 5 volts with a voltage regulator from an external battery. According to the Bluetooth User Guide for the RN-42 chip, CTS needs to be grounded when interfacing with a microprocessor via UART [2].

The L293D motor driver is used to drive our two DC motors, where each side of the motor driver is used for one of the motors. Each side has an enable pin, two input pins, and two output pins. When the enable pin is HIGH, the corresponding side of the motor driver will activate and the motor will receive power. With this function we can control the speed of the motor by connecting the enable pin to the PWM signal from the FPGA. When an input pin is set HIGH, the current flows through the corresponding output pin, so by connecting the motor leads to the output pins, we can control the direction that the motors spin by setting one input pin HIGH and keeping the other LOW. Table 3.1 shows how the inputs control the direction of the motors. Vcc is the internal voltage supply, which we connected to 5 volts, and Vss is the motor voltage supply, which we connected to the battery voltage, which is about 8 volts, so that the motors would be driven with the most power.

The headlight and taillight LEDs are connected to four microcontroller GPIO pins with current limiting resistors. The taillights use 75Ω resistors and the headlights use 45Ω resistors because the yellow LEDs were less bright than the red LEDs when they used the same resistors so we wanted to make them brighter.

The microcontroller MISO, MOSI and SPCK pins are connected to the FPGA sdo, sdi and sck pins for SPI communication.

There is also a pushbutton with a 10k pull-down resistor connecting 5V to the reset for the PWM module in the FPGA. It wasn't necessary after testing and debugging for the PWM module, but there was no harm in keeping it.

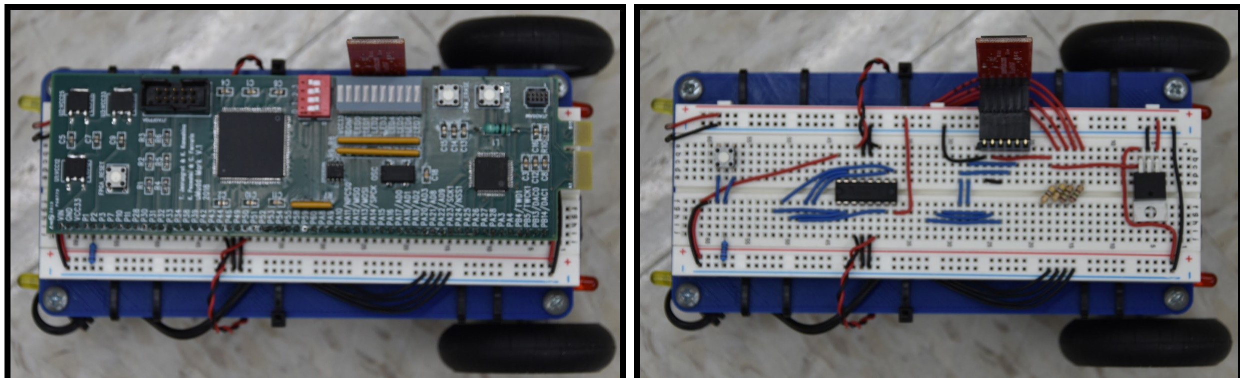


Figure 3.1: Breadboard Circuit

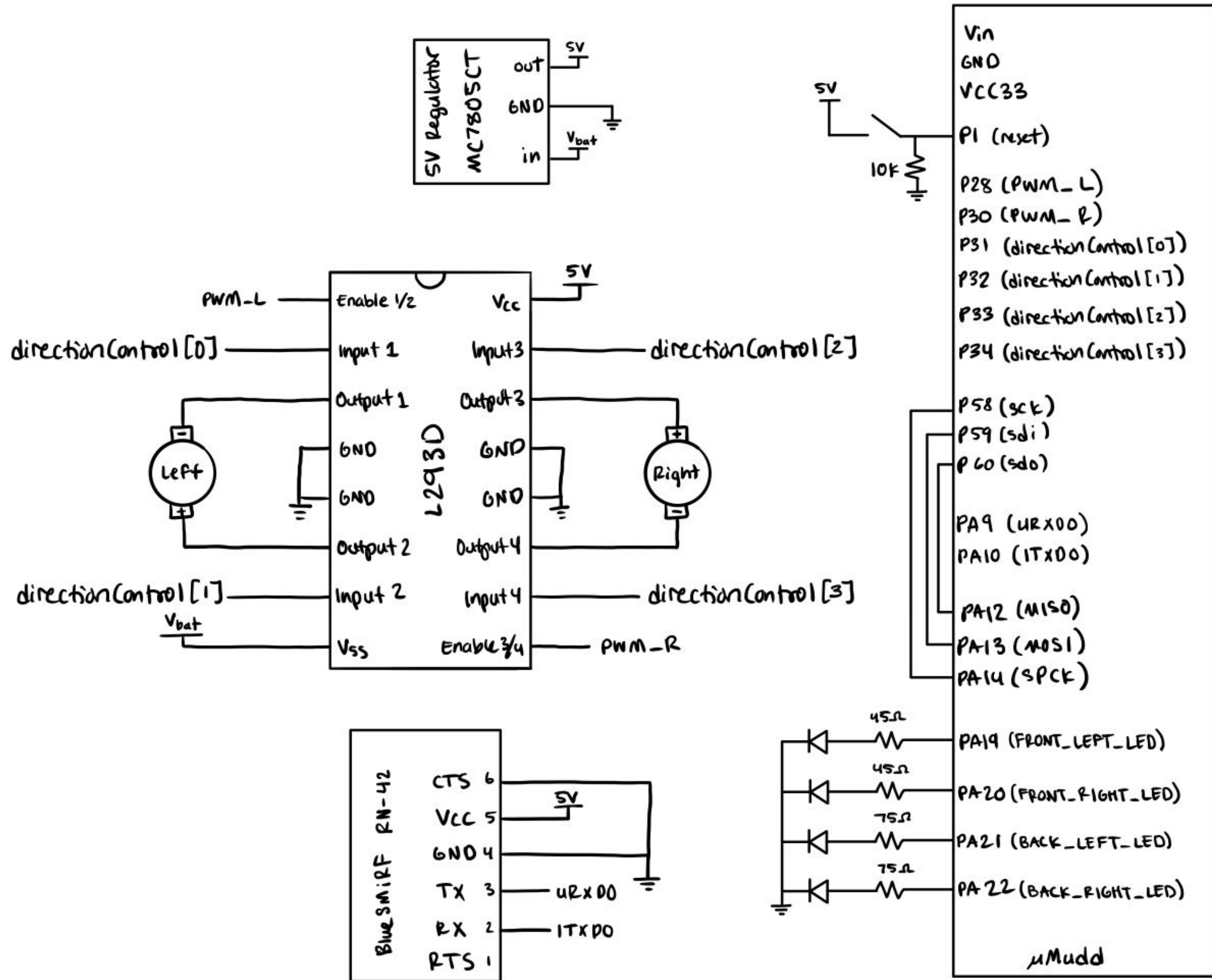


Figure 3.2: Breadboard Schematic

Motor	**Motor Driver pins**	**FPGA pins**	**Forward**	**Backward**
Left	Input 1	`directionControl[0]`	Low	High
	Input 2	`directionControl[1]`	High	Low
Right	Input 3	`directionControl[2]`	High	Low
	Input 4	`directionControl[3]`	Low	High

Table 3.1: Direction Control of Motors

IV. Microcontroller Design

In the main function of our code, an infinite while loop is used to update the motor control and LED control signals. The microcontroller receives an 8-bit command character from the bluetooth chip via UART which represents the desired function of the RC car (forward, backward, left, right, etc). This command character is then given to a function that encodes the desired function of the motors in an 8-bit control signal. This function encodes the desired functionality for each motor in two 4-bit parts: 1 bit for direction and 3 bits for PWM power level based on the desired direction of the car.

The microcontroller then sends this control signal to the FPGA via SPI with a hardcoded CS signal called LOAD. First, the load pin is turned high. Then, the control signal is sent. Finally, the load pin is turned low. Rinse and repeat.

To control the LEDs, an LED control function takes in the command character from the bluetooth chip and an external counter variable initialized outside the while loop. The command character dictates what the 2 headlight and 2 tail light LEDs do. The LEDs either blink to act as turn signals when the car is turning left or right, the tail light LEDs blink when the car is going backwards, or all the LEDs blink when the car is spinning. This function also outputs the counter variable so that it maintains its value for when the LED control function is called in the next iteration of the while loop.

To blink the LEDs, a separate blink function is called in the LED control function which takes in two LED pins and the external counter variable. This function checks the value of the counter and either drives the LED pins high or low if the counter is less than 15 or between 15 and 30, respectively, followed by a 5 ms delay and increments the counter or resets the counter if it is equal to 30, then it returns the counter variable (the delay function is given in the header file for the Timer Counter peripheral). This way, the counter will increase for each iteration of the while loop and the LED control function only has a delay of around 5 ms, stacking the delays so that the LEDs will stay on or off for the desired amount of time without creating a large delay that will slow down the rate at which the control signal is sent to the FPGA to change the function of the motors, which would in turn slow down the response of the RC car to the keyboard input.

V. FPGA Design

The top level module of our FPGA takes in 5 inputs: *sck*, *sdi*, *load*, *clk*, and *reset*, and outputs the PWM signals for the two motors, *PWM_L* and *PWM_R*, along with a 4-bit signal for the four input pins on the L293D, *directionControl[3:0]* and an *sdo* signal. The *sck, sdi* and *load* inputs and the *sdo* output are for SPI communication with the microcontroller, the *clk* input is from the external 40MHz crystal oscillator on the µMudd board, and the *reset* input is from a push button on the breadboard. Within the top level modules there are 4 other modules, an SPI module, a decoder module, and two PWM modules. The block diagram of this top-level module is shown in Figure 5.1.

The inputs of the SPI module are *sck*, *sdi*, *load* and *clk*, and it outputs the 8-bit control signal that contains the encoded signals for the desired behavior of the motors, *controlSignal[7:0]*, along with the *sdo* signal. In this module there is also an internal signal that holds in the incoming data while it's waiting for all 8 bits of the control signal to be transferred, *loadingSignal[7:0]*. There is a shift register with an enable that, on the rising edge of *sck*, when the input *load* is asserted, shifts the 7 least significant bits of *loadingSignal[7:0]* over once to the 7 most significant bits, and shifts in the *sdi* input to become the new least significant bit. After 8 *sck* cycles, the entire 8-bit control signal is loaded and *load* is driven low by the microcontroller. Then, there is an 8-bit register with an enable that, on the rising edge of *clk*, when *load* is deasserted, gives the value of the completed loading signal to the control signal output *controlSignal[7:0]*. The *sdo* output is set to 0 since the FPGA is not talking to the microcontroller.

The *controlSignal[7:0]* output from the SPI module is held as internal logic in the top level module and then sent to the decoder module, which outputs the PWM percent for the left and right motors, *percent_L* and *percent_R*, along with the *directionControl[3:0]* output. The decoder has four case statements that decide the direction bits and the PWM percent of each motor. The direction of the left and right motor is encoded in *controlSignal[7]* and *controlSignal[3]*, respectively, while the power level of the left and right motors are encoded in *controlSignal[6:4]* and *controlSignal[2:0]*, respectively. The direction bit of each motor dictates the two bits of the direction control signal that correspond to each motor, *directionControl[1:0]* for the left motor and *directionControl[3:2]* for the right motor. Tables 5.1 and 5.2 show the logic of the direction control case statements for the left and right motors. The three power level bits give the power level between 0 and 7 for each motor, and the PWM percent output of each motor is determined by the power level of that motor. Table 5.3 shows the corresponding PWM percent for each power level.

controlSignal[7]	**directionControl[1]**	**directionControl[0]**
1 (forward)	1	0
0 (backward)	0	1

Table 5.1: Left motor direction control logic

controlSignal[3]	**directionControl[3]**	**directionControl[2]**
1 (forward)	1	0
0 (backward)	0	1

Table 5.2: Right motor direction control logic

Power Level	**PWM Percent**
0	0%
1	40%
2	50%
3	60%
4	70%
5	80%
6	90%
7	100%

Table 5.3: Corresponding PWM Percent for Motor Power Level

The *percent_L* and *percent_R* outputs from the decoder module are held in internal logic in the top level module and then sent to the two PWM modules for each motor, which then output *PWM_L* and *PWM_R*. The PWM module takes in a *clk* and *reset* and an 8-bit percent input, *percent[7:0]*, and outputs the *PWM* signal. This module also has an 8-bit internal logic signal *count[7:0]* which holds the current value of the counter, and a single internal logic bit *restart* which resets the counter. The *restart* bit is assigned to be high when the *count[7:0]* signal reaches 100 in binary. The counter increments *count* on the rising edge of *clk* and has an asynchronous reset with the *reset* input and a synchronous reset with the *restart* bit. Thus, the counter will continuously count up to 100 and reset back to 0. The *PWM* output is assigned to be high when the *count* signal is less than or equal to the *percent* input and low otherwise, creating a signal that is high for the desired percentage of the time.

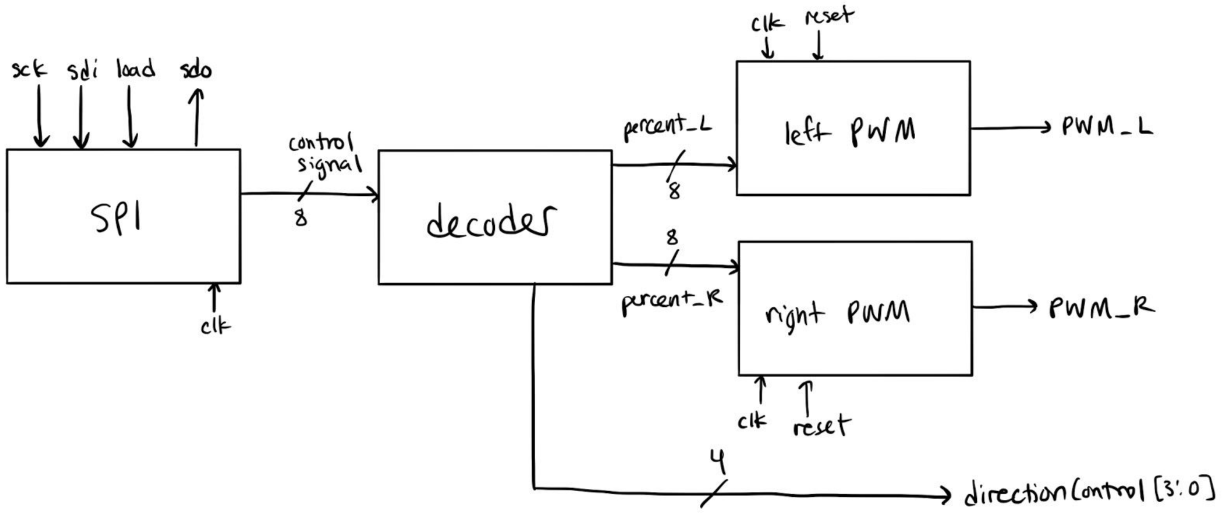


Figure 5.1: Top-Level FPGA Module Block Diagram

VI. Results

We successfully sent PWM data wirelessly via Bluetooth, avoided significant delays between the motor direction and LED signal updates, and executed the navigation of the RC car perfectly.

The team experienced difficulty exiting command mode when configuring the bluetooth chip settings. Entering command mode was successful (as indicated by the blinking of the Stat LED on the chip); however, exiting command mode proved to be a failure. The instructions on the datasheet were followed but did not work in practice.

In the mechanical design of the robot, the µMudd board would sometimes disconnect from the breadboard since the board was lifted up by female header pins, which is supporting the entire PCB on one side, making it easy to fall to the side and disconnect when bumped. In the future, we would like to add support for the other side of the µMudd board and zip-tie it down to make it more secure.

There were no differences between our initial proposal and final results. We fully executed the desired functionality of our RC car.

References

[1] https://learn.sparkfun.com/tutorials/using-the-bluesmirf/all
[2] https://cdn.sparkfun.com/assets/1/e/e/5/d/5217b297757b7fd3748b4567.pdf

Parts List

Part	Source	Vendor Part #	Price
Blue SMiRF	https://www.sparkfun.com/products/12577	WRL-12577	27.95
Breadboard	https://www.amazon.com/gp/product/B01EV6LJ7G/ref=ppx_yo_dt_b_asin_title_o00_s00?ie=UTF8&psc=1	None	8.99
Gearbox	https://www.pololu.com/product/114	Polulu 114	N/A
L293D	https://www.digikey.com/product-detail/en/stmicroelectronics/L293D/497-2936-5-ND/634700	497-2936-5-ND	N/A
DC Motors	https://www.pololu.com/file/0J11/fa_130ra.pdf	FA-130RA	N/A

Appendix A - python script

```
"""
@author: Leonardo Vilchez - lvilchez@g.hmc.edu
@author: Omar Aleman - oaleman@g.hmc.edu
date: 12/11/2019
Bluetooth Connection with BlueSmirf
"""
import serial
import time
import keyboard

### Establish COM port connection with Bluetooth Chip ###
ser = serial.Serial(port='COM4',baudrate=115200, parity = serial.PARITY_NONE,stopbits=serial.STOPBI

def commandMode(): #failure
    start = "$"
    end = '-'
    brate = 'SU,96' # changes baud rate of bluetooth chip to 9600

    time.sleep(0.5) # delay for 500 ms
    # enter command mode
    ser.write(start.encode())
    ser.write(start.encode())
    ser.write(start.encode())

    time.sleep(0.1) # short delay, wait for BlueSmirf to send back CMD
    ser.write(brate.encode())

    # exit command mode
    ser.write(end.encode())
    ser.write(end.encode())
    ser.write(end.encode())

def checkPort():
    if ser.isOpen(): # if the serial port is open
            command = 'A'
            ser.write(command.encode())
            print("write")
    else:
        print ("Cannot open serial port.")

while True:
    try:
        if keyboard.is_pressed('a') and keyboard.is_pressed('w'): # forward left
            command = 'q'
            ser.write(command.encode())
            print('forward left')
        elif keyboard.is_pressed('d') and keyboard.is_pressed('w'): # forward right
            command = 'e'
            ser.write(command.encode())
            print('forward right')
        elif keyboard.is_pressed('a') and keyboard.is_pressed('s'): # backward left
            command = 'z'
            ser.write(command.encode())
            print('backward left')
        elif keyboard.is_pressed('d') and keyboard.is_pressed('s'): # backward right
            command = 'x'
            ser.write(command.encode())
```

```
            print('backward right')
        elif keyboard.is_pressed('w'): # forward
            command = 'w'
            ser.write(command.encode())
            print("forwards")
        elif keyboard.is_pressed('s'): # backwards
            command = 's'
            ser.write(command.encode())
            print('backwards')
        elif keyboard.is_pressed('a'): # left
            command = 'a'
            ser.write(command.encode())
            print('left')
        elif keyboard.is_pressed('d'): # right
            command = 'd'
            ser.write(command.encode())
            print('right')
        elif keyboard.is_pressed('x'): #exit program
            break
        else: #pause
            command = 'p'
            ser.write(command.encode())
            print('pause')
    except:
        break

ser.close() # closes COM port
```

Appendix B - Keil µVision code

```
1   // E155 Final Project
2   // RC_Car.c - interfaces with bluetooth chip via UART, controls LEDs and sends motor control signal to
    FPGA via SPI
3   // Omar Aleman - oaleman@g.hmc.edu
4   // Leonardo Vilchez - lvilchez@g.hmc.edu
5   // 12/11/19
6
7   #include "SAM4S4B/SAM4S4B.h"
8   #include <string.h>
9   #include <stdlib.h>
10  #include <stdio.h>
11
12  // LED pins
13  #define FRONT_LEFT_LED 19
14  #define FRONT_RIGHT_LED 20
15  #define BACK_LEFT_LED 21
16  #define BACK_RIGHT_LED 22
17
18  // SPI load pin
19  #define LOAD_PIN 30
20
21  ////////////////////////////////////////////////////////////////////////////////////////////////////
22  // FPGA Communication Functions
23  ////////////////////////////////////////////////////////////////////////////////////////////////////
24
25  /* Returns the control signal corresponding to the command from the bluetooth chip
26   *    -- Input: command signal from bluetooth chip via UART
27   *    -- Output: corresponding control signal based on the FPGA decoder */
28  char getControl(char command){
29      if (command == 'w'){ // forward
30        return 0xFF;
31      }
32      else if (command == 's'){ // backward
33        return 0x77;
34      }
35      else if (command == 'a'){ // left
36        return 0x7F;
37      }
38      else if (command == 'd'){ // right
39        return 0xF7;
40      }
41      else if (command == 'q'){ // forward left
42        return 0xCF;
43      }
44      else if (command == 'e'){ // forward right
45        return 0xFC;
46      }
47      else if (command == 'z'){ // backward left
48        return 0x47;
49      }
50      else if (command == 'x'){ // backward right
51        return 0x74;
52      }
53      else if (command == 'p'){ // pause
54        return 0x00;
55      }
56      else return 0x00; // pause (default)
57  }
58
59  /* Sends control signal to FPGA via SPI with LOAD_PIN
60   *    -- Input: control signal to send to FPGA */
61  void sendControlSPI(char control){
62      pioDigitalWrite(LOAD_PIN,1);
63      spiSendReceive(control);
64      pioDigitalWrite(LOAD_PIN,0);
65  }
66
67  ////////////////////////////////////////////////////////////////////////////////////////////////////
68  // LED Functions
69  ////////////////////////////////////////////////////////////////////////////////////////////////////
70
71  /* Blinks two LEDs while avoiding large delays in the main while loop
```

```
72   *    -- Input: two pins and the counter variable
73   *    -- Output: the counter variable */
74  int blink(int pin0, int pin1, int counter){
75      if (counter < 15){
76        pioDigitalWrite(pin0,1);
77        pioDigitalWrite(pin1,1);
78        tcDelayMillis(5); // 5 ms delay
79        counter++;
80      }
81      else if (counter < 30){
82        pioDigitalWrite(pin0,0);
83        pioDigitalWrite(pin1,0);
84        tcDelayMillis(5); // 5 ms delay
85        counter++;
86      }
87      else counter = 0;
88      return counter;
89  }
90
91  /* Controls the function of the LEDs
92   *    -- Input: command signal from bluetooth chip via UART and the counter variable
93   *    -- Output: the counter variable */
94  int LEDcontrol(char command, int counter){
95      if (command == 'q'){ // forward left
96        counter = blink(FRONT_LEFT_LED,BACK_LEFT_LED,counter);
97        pioDigitalWrite(FRONT_RIGHT_LED,1);
98        pioDigitalWrite(BACK_RIGHT_LED,1);
99      }
100     else if (command == 'e'){ // forward right
101       counter = blink(FRONT_RIGHT_LED,BACK_RIGHT_LED,counter);
102       pioDigitalWrite(FRONT_LEFT_LED,1);
103       pioDigitalWrite(BACK_LEFT_LED,1);
104     }
105     else if (command == 's' || command == 'z' || command == 'x') { // backward
106       counter = blink(BACK_LEFT_LED,BACK_RIGHT_LED,counter);
107       pioDigitalWrite(FRONT_LEFT_LED,1);
108       pioDigitalWrite(FRONT_RIGHT_LED,1);
109     }
110     else if (command == 'a' || command == 'd') { // spinning
111       blink(FRONT_LEFT_LED,BACK_RIGHT_LED,counter);
112       counter = blink(FRONT_RIGHT_LED,BACK_LEFT_LED,counter);
113     }
114     else if (command == 'p') { // pause
115       pioDigitalWrite(FRONT_RIGHT_LED,0);
116       pioDigitalWrite(BACK_RIGHT_LED,0);
117       pioDigitalWrite(FRONT_LEFT_LED,0);
118       pioDigitalWrite(BACK_LEFT_LED,0);
119     }
120     else{
121       pioDigitalWrite(FRONT_RIGHT_LED,1);
122       pioDigitalWrite(BACK_RIGHT_LED,1);
123       pioDigitalWrite(FRONT_LEFT_LED,1);
124       pioDigitalWrite(BACK_LEFT_LED,1);
125     }
126     return counter;
127 }
128
129 ////////////////////////////////////////////////////////////////////////////////////////////////////
130 // Main Function
131 ////////////////////////////////////////////////////////////////////////////////////////////////////
132
133 int main(void) {
134   samInit(); // sets up external 40MHz clk for uC and disables watchdog timer
135   pioInit(); // sets up pio peripheral
136   uartInit(UART_MR_PAR_NO,22); // baudrate = 113636.36 (as close to 115200 as we could get, 1.35% error)
137   spiInit(MCK_FREQ/244000,0,1);
138   // "clock divide" = master clock frequency / desired baud rate
139   // the phase for the SPI clock is 1 and the polarity is 0
140   tcDelayInit();// TC channel 0, MCK/2, counter increases then resets low when an RC match occurs
141
142   // Sets LED and LOAD pins as PIO output pins
143   pioPinMode(FRONT_LEFT_LED,PIO_OUTPUT);
```

```
144     pioPinMode(FRONT_RIGHT_LED,PIO_OUTPUT);
145     pioPinMode(BACK_LEFT_LED,PIO_OUTPUT);
146     pioPinMode(BACK_RIGHT_LED,PIO_OUTPUT);
147     pioPinMode(LOAD_PIN,PIO_OUTPUT);
148
149     int counter = 0; // counter for LED control
150
151     while(1) {
152       // Wait for BlueSMiRF to send character from PC
153       while(!uartRxReady());
154
155       // Receive char from BlueSMiRF
156       char command = uartRx();
157
158       // Get control char to send to FPGA
159       char control = getControl(command);
160
161       // send command to FPGA via SPI
162       sendControlSPI(control);
163
164       // controls LED based on command from PC
165       counter = LEDcontrol(command, counter);
166
167     }
168   }
169
```

Appendix C - FPGA Quartus Verilog

```
1  // E155 Final Project
2  // RC_Car.sv - recieves and decodes motor control signal from microcontroller via SPI and
   drives motors accordingly
3  // Omar Aleman - oaleman@g.hmc.edu
4  // Leonardo Vilchez - lvilchez@g.hmc.edu
5  // 12/11/2019
6
7  ////////////////////////////////////////////////////////////////////////////////////
   ///////
8  // Top Level Module
9  ////////////////////////////////////////////////////////////////////////////////////
   ///////
10
11 module RC_Car(input  logic        clk, reset,
12               input  logic        sck,
13               input  logic        sdi, load,
14               output logic        sdo,
15               output logic        PWM_R, PWM_L,
16               output logic [3:0] directionControl);
17
18    logic [7:0] controlSignal; // control signal from MCU via SPI
19    logic [7:0] percent_R, percent_L; // PWM percent for left and right motors
20
21    // SPI module to recieve controlSignal from MCU
22    spi SPI(clk, sck, load, sdi, sdo, controlSignal);
23
24    // decoder module to interpret controlSignal
25    decoder DEC(controlSignal, percent_L, percent_R, directionControl);
26
27    // PWM modules to create PWM signals for motors
28    pwm PWMR(clk, reset, percent_R, PWM_R);
29    pwm PWML(clk, reset, percent_L, PWM_L);
30
31 endmodule
32
33 ////////////////////////////////////////////////////////////////////////////////////
   ///////
34 // PWM Module
35 ////////////////////////////////////////////////////////////////////////////////////
   ///////
36
37 module pwm(input  logic        clk, reset,
38            input  logic [7:0] percent,
39            output logic        PWM);
40
41    logic [7:0] count;
42    logic       restart; // restart counter when count gets to 100
43
44    // restart - high when count[7:0] = 100
45    assign restart = (~count[7])&count[6]&count[5]&(~count[4])&(~count[3])&count[2]&(~count[1
   ])&(~count[0]); // 100
46
47    // counter - use restart to restart counter on rising edge of clock when count reaches 10
48    always_ff@(posedge clk, posedge reset)
49       if      (reset)   count <= 0;
50       else if (restart) count <= 0;
51       else              count <= count+1;
52
53    // set PWM to be high whenever count is less than or equal to percent
54    assign PWM = (count <= percent);
55
56 endmodule
57
58 ////////////////////////////////////////////////////////////////////////////////////
   ///////
59 // Decoder Module
60 ////////////////////////////////////////////////////////////////////////////////////
   ///////
61
62 module decoder(input  logic [7:0] controlSignal,
63                output logic [7:0] percent_L, percent_R,
64                output logic [3:0] directionControl);
65
```

```
66      // directionControl decoder - left motor
67      always_comb
68         case(controlSignal[7])
69            1'b1: directionControl[1:0] = 2'b10; // forward
70            1'b0: directionControl[1:0] = 2'b01; // backward
71         endcase
72
73      // directionControl decoder - right motor
74      always_comb
75         case(controlSignal[3])
76            1'b1: directionControl[3:2] = 2'b01; // forward
77            1'b0: directionControl[3:2] = 2'b10; // backward
78         endcase
79
80      // PWM percent_L decoder
81      always_comb
82         case(controlSignal[6:4])
83            3'b000: percent_L = 0;   // powerlvl 0 (stop)
84            3'b001: percent_L = 40;  // powerlvl 1
85            3'b010: percent_L = 50;  // powerlvl 2
86            3'b011: percent_L = 60;  // powerlvl 3
87            3'b100: percent_L = 70;  // powerlvl 4
88            3'b101: percent_L = 80;  // powerlvl 5
89            3'b110: percent_L = 90;  // powerlvl 6
90            3'b111: percent_L = 100; // powerlvl 7
91         endcase
92
93      // PWM percent_R decoder
94      always_comb
95         case(controlSignal[2:0])
96            3'b000: percent_R = 0;   // powerlvl 0 (stop)
97            3'b001: percent_R = 40;  // powerlvl 1
98            3'b010: percent_R = 50;  // powerlvl 2
99            3'b011: percent_R = 60;  // powerlvl 3
100           3'b100: percent_R = 70;  // powerlvl 4
101           3'b101: percent_R = 80;  // powerlvl 5
102           3'b110: percent_R = 90;  // powerlvl 6
103           3'b111: percent_R = 100; // powerlvl 7
104        endcase
105
106  endmodule
107
108  ///////////////////////////////////////////////////////////////////////////////
     ///////
109  // SPI Module
110  ///////////////////////////////////////////////////////////////////////////////
     ///////
111
112  module spi(input  logic       clk,
113             input  logic       sck,
114             input  logic       load,
115             input  logic       sdi,
116             output logic       sdo,
117             output logic [7:0] controlSignal);
118
119     logic [7:0] loadingSignal; // 8-bit control signal while it's being loaded
120
121     // 8-bit shift register with enable
122     // load is asserted - apply 8 sclks to shift in 8-bit control signal
123     always_ff@(posedge sck)
124        if(load) loadingSignal <= {loadingSignal[6:0],sdi};
125
126     // 8-bit register with enable
127     // then load is deasserted - send complete 8-bit loadingSignal to controlSignal
128     always_ff@(posedge clk)
129        if(~load) controlSignal <= loadingSignal;
130
131     // sdo doesn't do anything
132     assign sdo = 0;
133
134  endmodule
135
```