Bluetooth Racer

Erik Sargent
Weston Jensen
David Christensen

December 2015

Contents

List of Figures

List of Tables

1 Introduction

This document describes the design and process of retrofitting a remote control car into a Bluetooth controlled racer. A simple App was developed so that the movement of a smartphone controls the acceleration and steering of the car. The object is to enable the car to turn left and right, and move forward and backward, all controlled by the Bluetooth connection to the phone. Included in this document, the software and hardware design are discussed and portrayed with diagrams. In order to prove that the Bluetooth controlled car works the way it is supposed to, test documentation is included, as well as a video link of the car working. The challenges associated with this lab, include: working with new hardware i.e. Bluetooth Module, Motor Driver, Schmidt Trigger and an RC car with no accompanying data sheet for the motors.

2 Scope

This document discusses, in detail, the electrical and software design for the Bluetooth Racer. It includes the requirements, dependencies and theory of operation. Schematics and code segments are used to give a more thorough explanation of the design. Testing procedures and results of each requirement are included. The complete code is located in Appendix A.

3 Design Overview

3.1 Requirements

The following are the given requirements for Bluetooth Racer:

- The system shall use the Tiva C Series TM4C123GH6PM microcontroller.
- The system shall use two external batteries. A 5 volt battery with USB connection to the microcontroller and a 9 volt DC supply for the drive and steering.
- The system shall use a Bluetooth 4.0 Module.
- The system shall use a Dual TB6612FNG Motor Driver (H-Bridge).
- The Racer shall have PWM output to controller variable motor speed.
- Working LEDs should be used for the front and tail lights, as well as an inside light.
- Upon powering up, the inside light should be turned on.
- Upon connecting to Bluetooth, the front and tail lights should turn on.
- The microcontroller will transmit and receive data from the Bluetooth module through UART.
- The user shall command the Racer with a smartphone over Bluetooth connection. The movement of the phone will control the direction in which the car will move.
- The Racer shall be able to turn and drive forward and backward on command of the user.

3.2 Materials

The following is a list of materials used for the Bluetooth Racer:

Table 1: Material List

Item	Price
An RC car with working motors for both steering and driving	Free
9 volt DC power supply	$2.20
5 volt DC power supply with USB connection	Free
Dual TB6612FNG Motor Driver	$8.95
Tiva C Series TM4C123GH6PM microcontroller	$14.00
Bluetooth 4.0 Module	$13.95
Schmidt Trigger	$0.40
5 LEDs	$0.20 ea
Spray Paint	$1.00
NPN Transistor	$0.25
Assorted resistors and capacitors	$1.00

4 Bluetooth Racer Design

4.1 Software Design

The Software design is very simple, using an iOS smart-phone app a one byte message will be transmitted via Bluetooth to the Bluetooth module located on the RC car. The micro-controller's UART module is then connected to the Bluetooth module and the message is received. This message is then parsed into two bit sections that in turn enable/disable the two motors on the car and set the variable speed of the two motors.

We begin with the UART's initialization, the Bluetooth module by default transmits at 9600 baud, therefore we configure the UART to also receive messages at 9600 baud in order to ensure messages are received correctly. Interrupts are enabled to trigger on the UART when the fifo buffer is 1/8 full, in our configuration we only use the lower 8 bits, so the upper 8 bits are discarded. The UART Handler function will be discussed in greater detail below. Next, we configure a systick timer in order to control the the motors for Pulse Width Modulation(PWM), the systick timer was set to trigger an interrupt every 10 milliseconds, the systick handler function will be discussed below. When the car is turning, feedback is sent back to the microcontroller to stop the motor as it has already traveled its full distance. This feed back is inputted to Port A, Port A is configured to trigger an interrupt on both edges and is set as a pull up resistor. The RC car has tail lights and head lights, in order to turn these lights we use the GPIO port E, the port direction is set to output and as a pull up resistor. Finally we use GPTM Timer 1A to act as a watchdog timer, The timer is set to trigger every .5 second. If the timer triggers we turn off all the lights and set the PWM speed to 0, to disable the car. This ensures that once the app is disengaged that the car will not continue driving away.

We will now discuss the interrupt handler functions with their associated logic. When the UART buffer is 1/8 full the UART interrupt is triggered. Inside of the UART handler GPTM timer 1A is reloaded with a .5 second delay, this is done so the only time the watchdog timer expires is when we have not received data from the smartphone for .5 second. This is followed by reading in the data received from the Bluetooth module. As mentioned before we actually receive two bytes of data but only need the lower 8 bits so the upper 8 are discarded. The byte of data is then parsed into 2 bit parts. The first 2 bits are to enable H-bridge1 which is followed by the 2 bits to determine the PWM frequency. The next 4 bits follow the

same format but are for H-bridge2 and its associated PWM frequency. GPIO port A is responsible for receiving feedback from the steering module, if an interrupt is triggered on this port it is determined if we have steered all the way left or right. Once determined we set either the variable "turningRight"=0 or "turningLeft" =0. After parsing the data in the UART handler continues by checking the variables "turningRight" and "turningLeft" and if they are equal to 1 and if H-bridge1 is enabled we send a command to turn the desired direction. Finally we write the desired PWM value to the variable PWM0Duty and PWM1Duty which are used in the systick handler. When the systick handler interrupt is triggered we begin by incrementing two count values that represent PWM0 and PWM1, if the count is greater than 100 we reset the value back to 0. Next the PWM duty cycles are updated and we exit the function. This process is continued until the user terminates transmission and the car then shuts down.

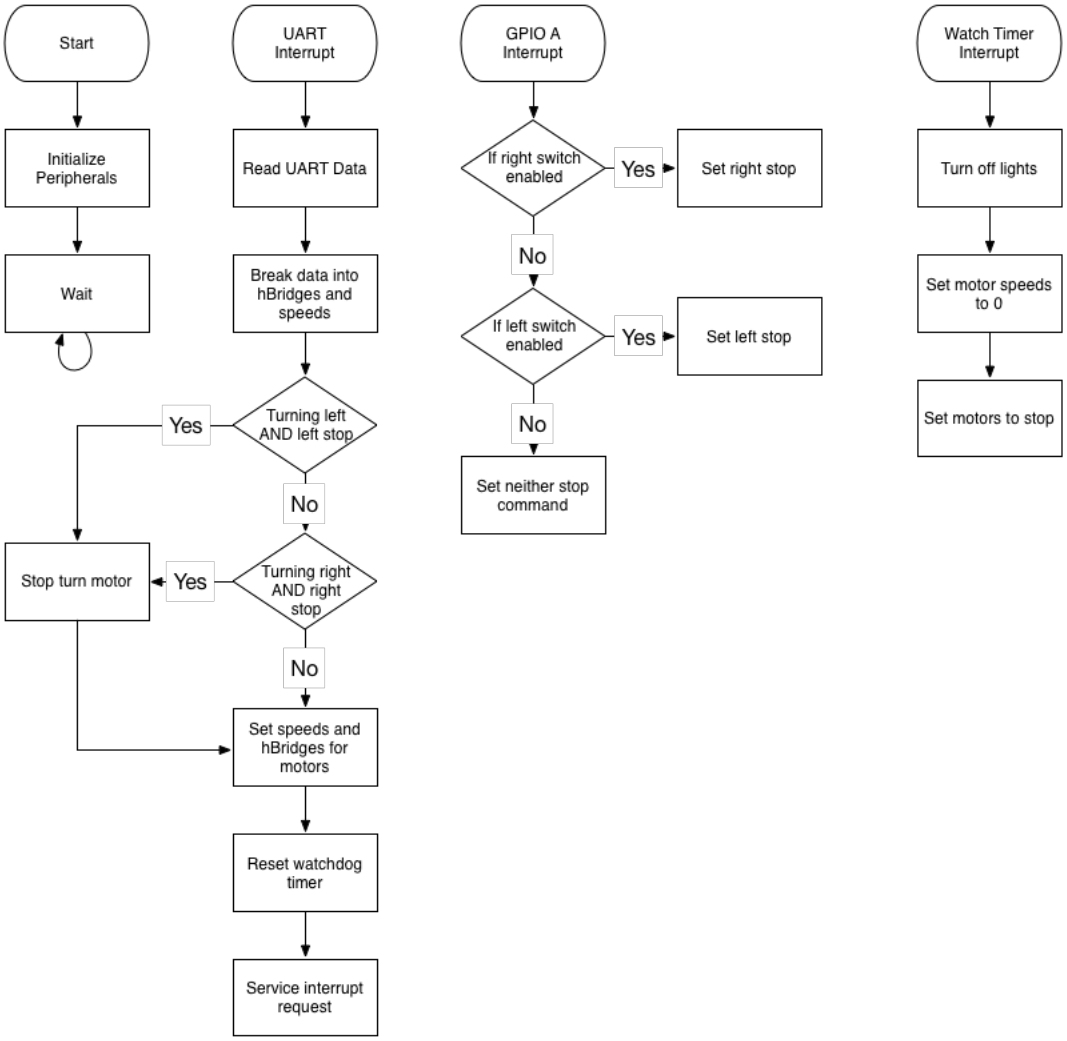

Figure 1: Software Flowchart

4.2 Hardware Design

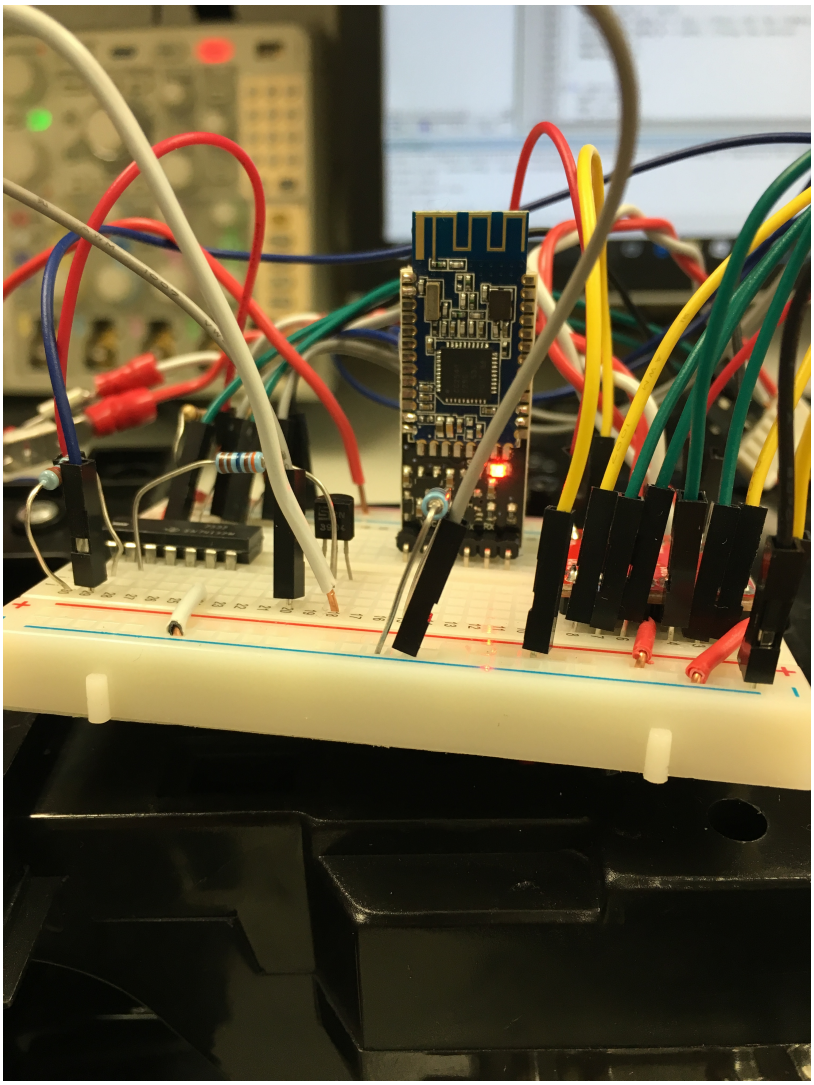

Figure 2: Bluetooth Module

The heart of the hardware design is the bluetooth module. The bluetooth module is connected to 5V from the microcontroller and ground. The TX and RX from the bluetooth module are connected to the microcontroller's UART RX and TX inputs. The bluetooth module can been seen in Figure 2. Along side the bluetooth module is a dual H-Bridge motor driver from Sparkfun. The motor driver can be configured to drive the motors forward, backwards or stop. Each driver also has a PWM input to change the voltage that is output to the motor, thereby changing the speed. The microcontroller has two GPIO ports connected to each h-bridge line for 2-bit configuration of each, and an additional line per driver connected to the PWM input on the motor driver. The motors were each connected to their respective connections on the other side of the motor driver.

The steering motor has two hardware switches to provide feedback when the motor is turned all the way to one side. This switch can be seen in Figure 5 as the green board connected to the end of the motor shaft. These switches were originally connected directly to the microcontroller, but it was found to have a lot of noise as the motor turned which would trigger unnecessary interrupts in the software. To remmedy this, a hardware RC filter was added to smooth out the noise, and a schmitt trigger was added in line to add hysteresis to the signal. This filtered signal is then connected to the microcontroller, and provides much

cleaner signal. The motor controller (red) and the schmitt trigger with filter are shown in Figure 3.

The seats were removed from the top of the car to make room for the electronics to fit in the car. The breadboard with electronics, microcontroller, and batterys were strapped to the top of the frame, then the shell of the car was placed on top. The top was left easy to remove to facilitate easy maintenance. The electronics on the car are shown in Figure 4. A series of LEDs were connected to a transistor to supply power that was switched by a GPIO port on the microcontroller. These LEDs were used to display when the car was connected to the phone and were glued into the head and break lights.

The full hardware schematic is shown below in Figure 6.

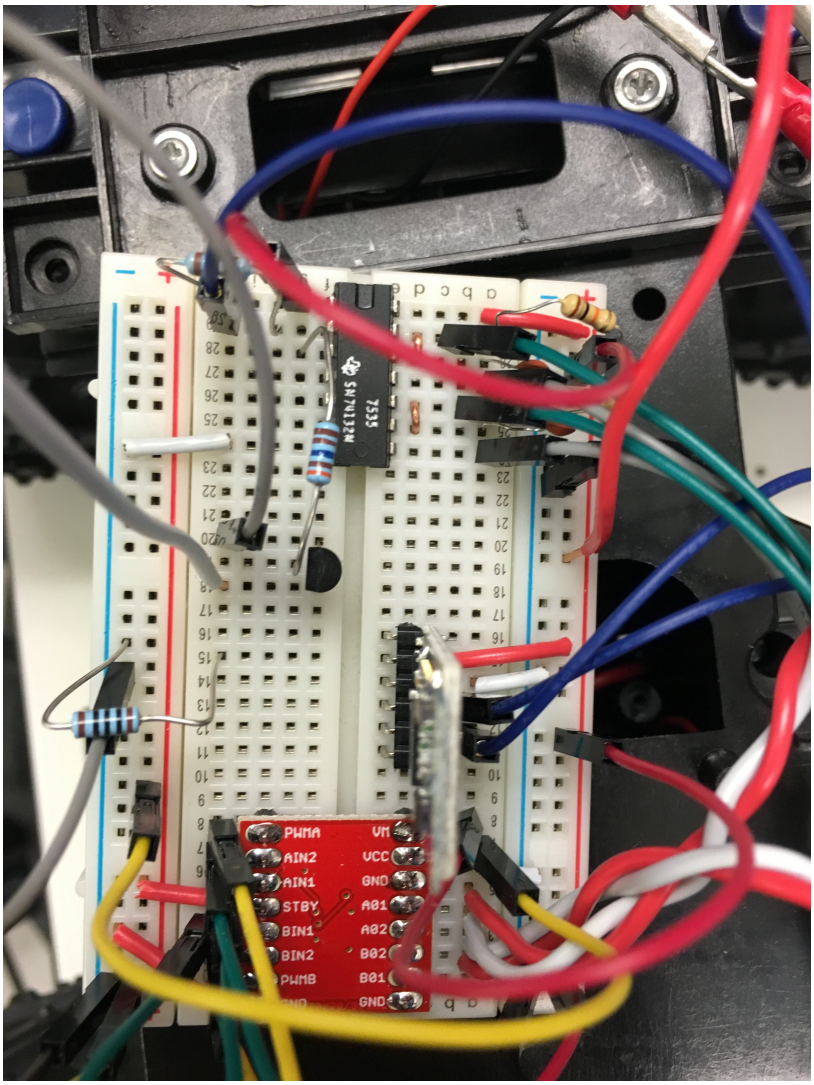

Figure 3: Breadboard

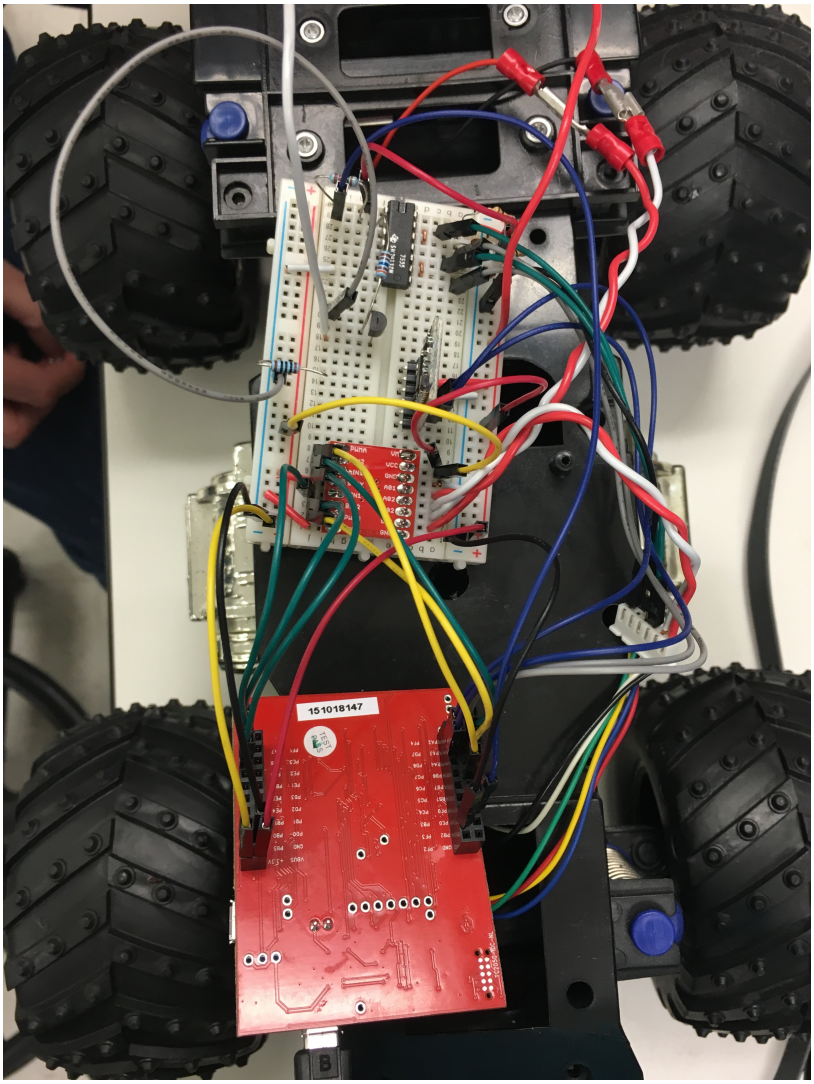

Figure 4: Under the Hood

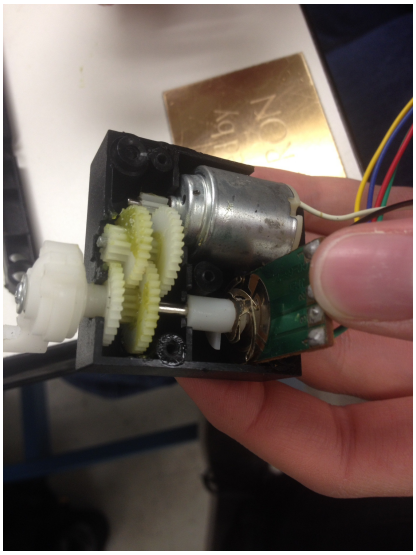

Figure 5: Steering Controller

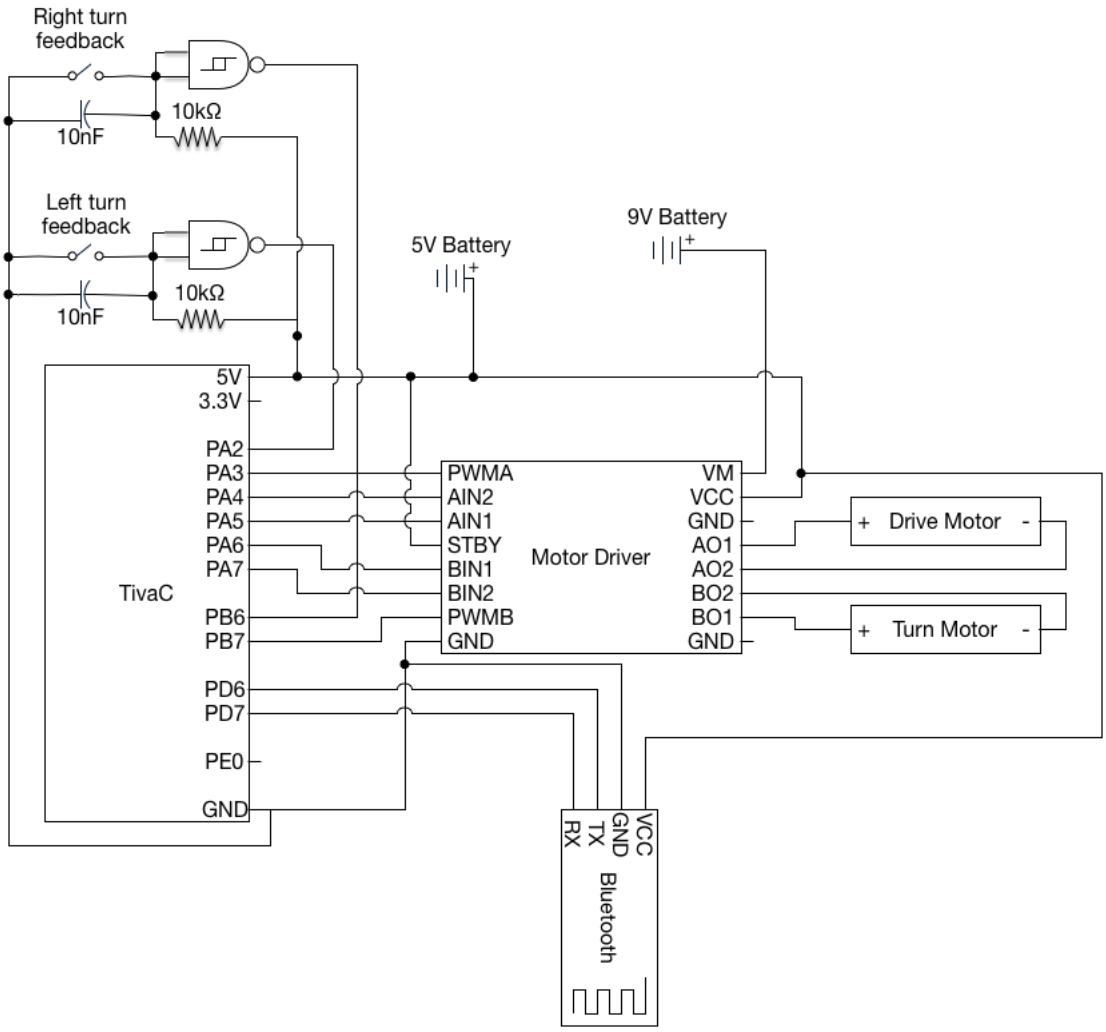

Figure 6: Hardware Schematic

5 Testing

In order to verify our iOS app was sending the correct signal and the RC car was receiving the proper command 3 tests were performed. The first test shows the message received on the micro-controller via the Bluetooth module. The second and third screen-shots demonstrate pulse width modulation(PWM) acting on the motor to drive the car at different speeds.

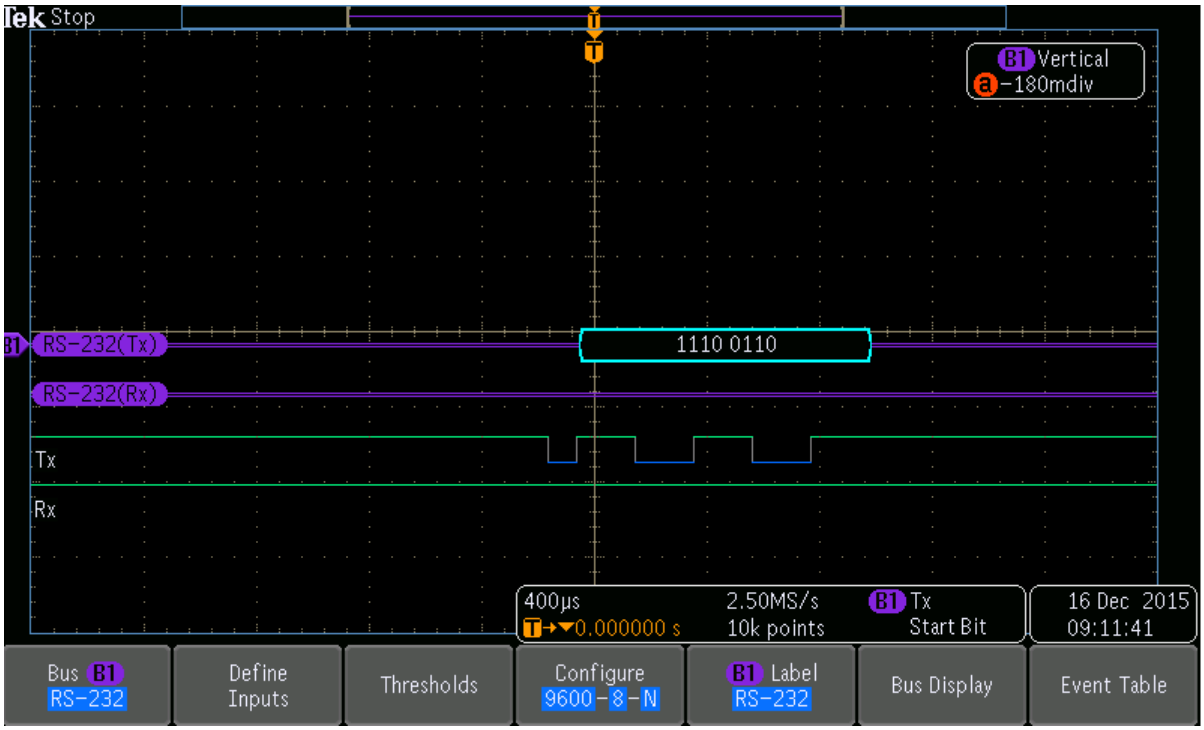

Figure 7: UART Data Message

As specified before, the car is controlled via Bluetooth. The iOS app sends a 1 byte command that the UART module on the micro-controller receives. This byte of data contains the information necessary to enable the two H-bridges and the data to set the variable speed for PWM. The byte of data is represented in the following format: two bits for H-bridge drive followed by two bits to set the variable speed, these four bits are followed by two bits for H-bridge turn followed by two bits to set the variable speed. Above is a print out of the message received by the Bluetooth module

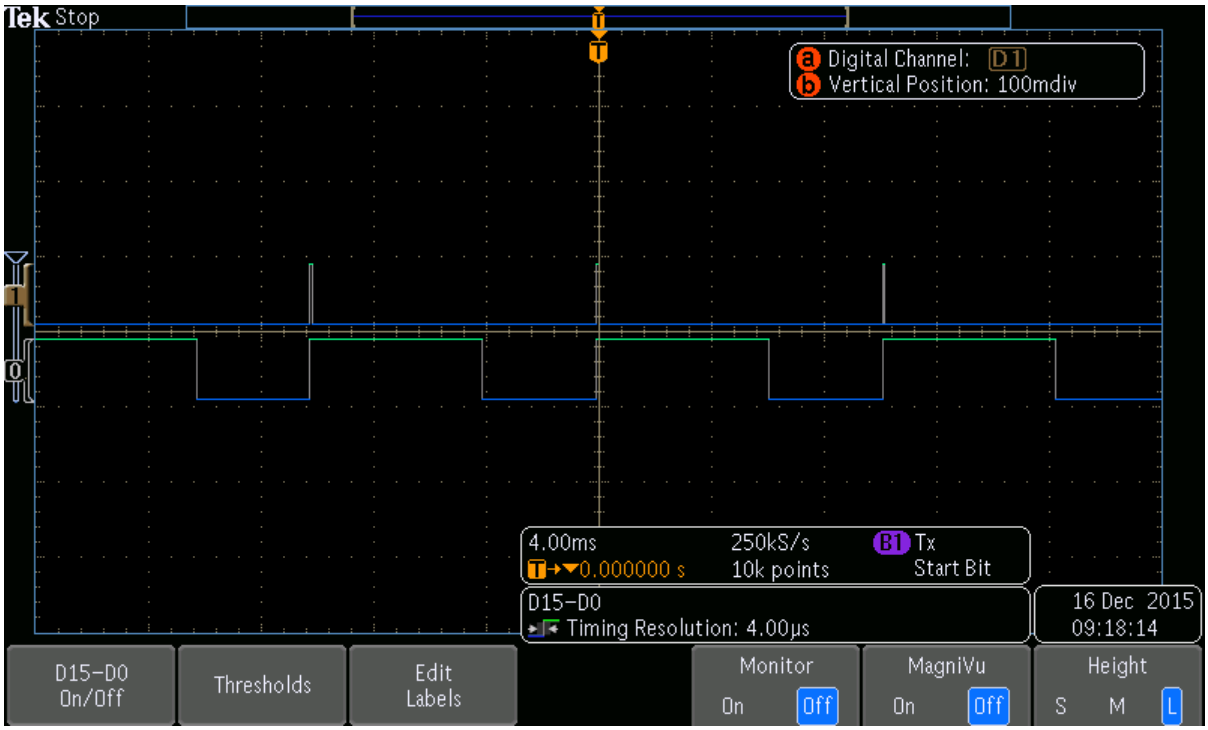

Figure 8: Half Power PWM

The RC car is designed to drive and different speeds, using PWM we are able to enable the motor for specified amounts of time. The longer the motor is enabled the faster the car will travel. In this screen shot from the logic analyzer, the wave form is high for little more than half of the entire wave, this means the car is driving slightly faster than half speed.

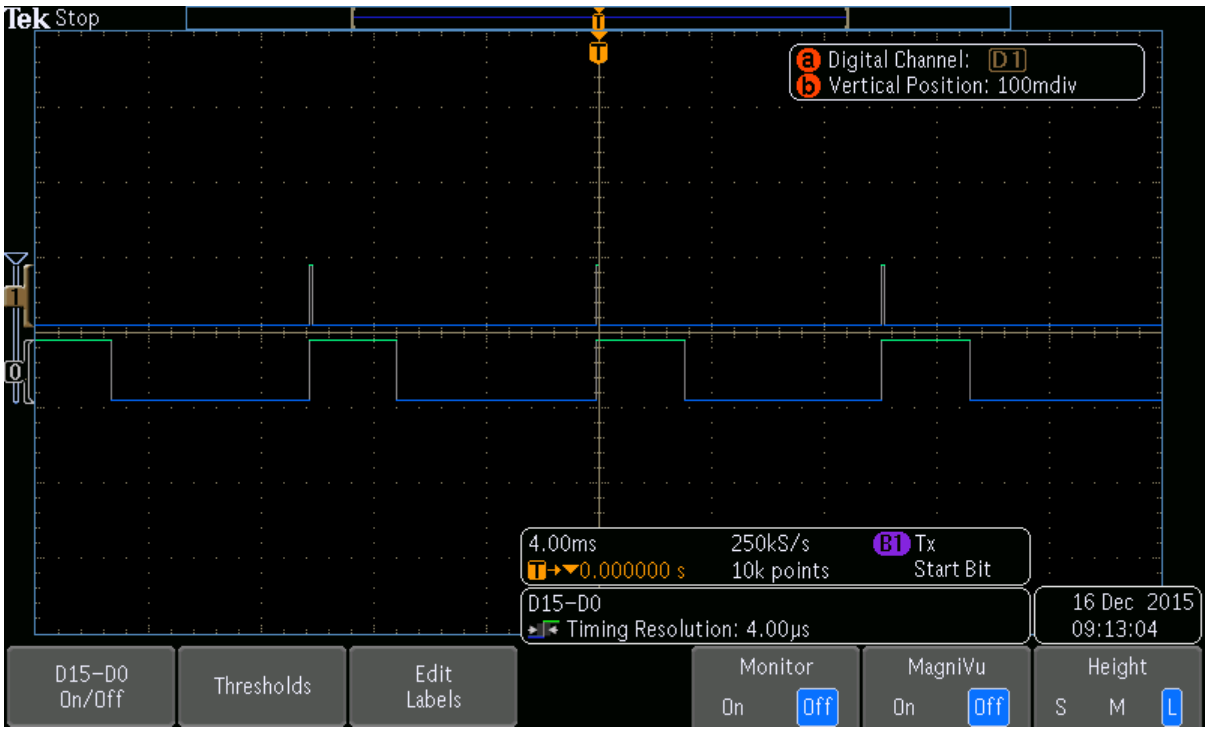

Figure 9: Quarter Power PWM

The RC car can also travel at slower speeds as when it is first starting to accelerate. Above is a screen shot from the logic analyzer, the wave form is high for close to quarter of the entire wave, this means the car is driving half as fast compared to the half-power wave form.

6 Conclusion

In conclusion our Bluetooth racing car met all of our specified requirements of being able to drive forward/backward and be able to steer left and right all the time being controlled by a smartphone app. As we progressed on the project we had numerous ideas that we would of been able to implement if we had a larger time frame to do so. The biggest restriction we confronted was associated with getting enough power to run our car at 100% capacity. The RC car we used to implemented the Bluetooth Racer was a used car that had seen better days, and unfortunately the 9V batter that came with the car was depleted and our efforts to recharge it were in vain. It was exciting to see all the skills we learned in lecture and past labs come together to build the Bluetooth Racer, these skills include but are not limited to: systick timer, GPTM timer, GPIO port interrupts, PWM, UART and c logic. This project has given us the skills and confidence to find everyday devices that with the help of a micro-controller can be put to new use with exciting results.

7 Apendix A

All code projects can be found on Github here: `https://github.com/eriksargent/ECE3710FinalProject`

7.1 Main.c

```
#include "GPIO.h"

volatile int UART2_NVIC_R __attribute__((at(0xE000E104)));

int PWM0Count = 0, PWM1Count = 0, PWM0Duty = 0, PWM1Duty = 0;
char turningRight = 0, turningLeft = 0;
char inData, hBridge1, hBridge2, drive1, drive2;


void PWM_init() {
        SYSCTL_RCGC2_R |= 0x2;
        SYSCTL_RCGCPWM_R = 0x1;

        //enable PWM clock in RCGC0 register
        __nop();
        __nop();
        __nop();

        //wait for clock to stabilize
        while((SYSCTL_PRGPIO_R&SYSCTL_PRGPIO_R1) == 0){};

        //Setup GPIO port B
        GPIO_PORTB_LOCK_R = 0x4C4F434B;
        GPIO_PORTB_PUR_R = 0xC0;
        GPIO_PORTB_DIR_R = 0xC0;
        GPIO_PORTB_DEN_R = 0xC0;
}

void UART2_INIT() {
        //UART2 D6,D7
        SYSCTL_RCGCUART_R = 0x4;
        SYSCTL_RCGCGPIO_R |= 0x8;

        //clock needs some time to initialize
        __nop();
        __nop();
        __nop();

        GPIO_PORTD_LOCK_R = 0x4C4F434B; //unlock port c
        GPIO_PORTD_CR_R |= 0xC0;                //enable control register,i dont think we
            need this
        GPIO_PORTD_AFSEL_R |= 0xC0;             //alternative functionality enabled
        GPIO_PORTD_DEN_R |= 0xC0;               //digital enable enabled
        GPIO_PORTD_PCTL_R = 0x11000000; //page 686 PD6 (1), PD7 (1)
        UART2_CTL_R = 0x0;                              //disable uart
        //(16e6)/(9600*16) = 104.167
        UART2_IBRD_R = 0x68;
        //.167 * 64 + .5 = 8.5 rounddown to 11
        UART2_FBRD_R = 0xB;
        UART2_LCRH_R |= 0x72;                   //Set serial parameters: 8-bit word, start
            /stop/parity bits
        UART2_CTL_R = 0x301;                    // Enable rx, tx on uart
        UART2_IFLS_R = 0x0;                     //set interrupts rx 1/8 full queue
```

```
        UART2_RIS_R = 0x10;                     //page 925
        UART2_IM_R  = 0x10;                     //interrupt mask, Rxim page 921
        UART2_NVIC_R = 0x2;
}

//Trigger when queue is 1/8 full
void UART2_Handler() {
        TIMER1_TAILR_R = 0x7A1200;        // Reset watchdog timer value
        GPIO_PORTE_DATA_R = 0x0;          // Turn on headlights
        if((UART2_FR_R & 0x10) == 0x0) { // RxFE bit Read 1/8 full
                inData = UART2_DR_R;
        }

        //parse incoming data
        hBridge1 = inData & 0x3;
        drive1 = (inData & 0xC) >> 2;
        hBridge2 = (inData & 0x30) >> 4;
        drive2 = (inData & 0xC0) >> 6;

        //Only turn the drive motor if it is not on the stop
        if (hBridge1 == 0x1 && turningRight == 1) {
                hBridge1 = 0x11;
        }
        else if (hBridge1 == 0x10 && turningLeft == 1) {
                hBridge1 = 0x11;
        }

        // Write new command to gpio port
        GPIO_PORTA_DATA_R = (((hBridge1 << 2) | (hBridge2))<<4);
        PWM0Duty = drive1 * 30;
        PWM1Duty = drive2 * 30;

        //Awknowledge Interrupt
        UART2_ICR_R = 0x1;
}

//Systick initialization. Used for PWM output
void SystickConfig() {
        NVIC_ST_CTRL_R = 0;
        NVIC_ST_RELOAD_R = 0x640;
        NVIC_ST_CTRL_R = 0x7;
}

//Output PWM when systick expires
void SysTick_Handler(){
        //Increment pwm clock
        PWM0Count++;
        PWM1Count++;

        //Loop around if it has reached 100
        if (PWM0Count > 100) {
                PWM0Count = 0;
        }

        if (PWM1Count > 100) {
                PWM1Count = 0;
        }

        //Turn on for the first half of the duty cycle, then turn off
        if (PWM0Count > PWM0Duty) {
```

```
                GPIO_PORTB_DATA_R &= ~0x40;
        }
        else {
                GPIO_PORTB_DATA_R |= 0x40;
        }

        //Turn on for the first half of the duty cycle, then turn off
        if (PWM1Count > PWM1Duty) {
                GPIO_PORTB_DATA_R &= ~0x80;
        }
        else {
                GPIO_PORTB_DATA_R |= 0x80;
        }

        //Restart Systick
        NVIC_ST_CTRL_R = 0x7;
}

//Initialize port A, used for controlling the motors
void GPIOA_INIT() {
        //A2, A3, A4, A5, A6, A7
        SYSCTL_RCGC2_R |= 0x1;    //Enable clock for PortA
        GPIO_PORTA_DIR_R = 0xF0; //Set direction to output
        GPIO_PORTA_PUR_R = 0xC;  //Pull up resistor
        GPIO_PORTA_DEN_R = 0xFC; //Digitial enable
        GPIO_PORTA_IS_R = 0x1;    //Edge triggering
        GPIO_PORTA_IBE_R = 0x1;  //Trigger both edges
        GPIO_PORTA_IM_R = 0xC;    //Pin interrupt
        NVIC_EN0_R = 0x1;         //NVIC
}

//One of the interrupts from the turning feedback was triggered
void GPIOA_Handler() {
        char PORTA_DATA = GPIO_PORTA_DATA_R;
        //Check and see if it is turned to the right
        if ((PORTA_DATA & 0x8) != 0) {
                turningRight = 0;
                turningLeft = 1;
        }
        //Check and see if it is turned to the left
        else if ((PORTA_DATA & 0x4) != 0) {
                turningRight = 1;
                turningLeft = 0;
        }
        //Otherwise, somewhere in the middle
        else {
                turningRight = 0;
                turningLeft = 0;
        }

        GPIO_PORTA_ICR_R = 0xFF; //Service the interrupt
}

//Setup the GPIO port for the LED outptu (head and break lights)
void setupLED(){
        SYSCTL_RCGC2_R |= 0x10;  //Enable clock for PortE
        GPIO_PORTE_DIR_R = 0x1;  //set direction to output
        GPIO_PORTE_PUR_R = 0x1;  //Pull up resistor
        GPIO_PORTE_DEN_R = 0x1;  //digitial enable
```

```
        GPIO_PORTE_DATA_R = 0x1; //turn off LED lights
}

//Initialize timer 1, used as a watchdog
void Timer1A_init() {
        SYSCTL_RCGCTIMER_R = 0x2;
        TIMER1_CTL_R = 0x0;          //Stop timer
        TIMER1_CFG_R = 0x0;          //Select 32 bit mode
        TIMER1_TAMR_R = 0x2;         //Periodic mode
        TIMER1_TAILR_R = 0x7A1200; //Timer set to expire after 1 second
        TIMER1_IMR_R = 0x1;          //Enable interrupt on port
        NVIC_EN0_R |= (1 << 21);     //Enable in NVIC
        TIMER1_CTL_R = 0x1;          //Start timer
}

//Respond to the watchdog call
void TIMER1A_Handler() {
        TIMER1_ICR_R = 0x1F;
        GPIO_PORTE_DATA_R = 0x1;  //Turn off the lights
        GPIO_PORTA_DATA_R = 0xF0; //Stop the motors

        //Put the motors in stop mode
        PWM0Duty = 0;
        PWM1Duty = 0;
}

int main(void) {
        //Setup everything
        UART2_INIT();
        PWM_init();
        SystickConfig();
        GPIOA_INIT();
        setupLED();
        Timer1A_init();

        //Wait for bluetooth command
        while(1);
}
```

7.2 GPIO.h

This is a file provided by Valvano, please see link for GPIO.h file.
http : *//users.ece.utexas.edu/ valvano/arm/*

7.3 iOS Code

```
//
//  ViewController.swift
//  RCCar
//
//  Created by Erik Sargent on 11/26/15.
//  Copyright    2015 eriksargent. All rights reserved.
//

import UIKit
import CoreBluetooth
import CoreMotion
```

```
class ViewController: UIViewController {
    //MARK: - Properties
    var centralManager: CBCentralManager!
    var carPeripheral: CBPeripheral?

    var motionManager = CMMotionManager()

    var label: UILabel!

    //MARK: - View Lifecycle
    override func viewDidLoad() {
        super.viewDidLoad()

        //Setup the bluetooth manager
        centralManager = CBCentralManager(delegate: self, queue: nil)

        //Add the status label to the view
        label = UILabel(frame: CGRect(x: view.frame.width / 2 - 150, y: view.frame.height
            / 2 - 15, width: 300, height: 30))
        label.text = "Searching for car..."
        label.textAlignment = .Center
        label.font = UIFont.systemFontOfSize(20)

        view.addSubview(label)

        //Watch motion changes
        motionManager.deviceMotionUpdateInterval = 0.1
        motionManager.startDeviceMotionUpdatesToQueue(NSOperationQueue.mainQueue()) {
            motionData, error in
            guard let data = motionData where error == nil else {
                return
            }

            //Calculate the angles of device rotation
            let steer = atan2(data.gravity.x, data.gravity.y) - M_PI / 2
            let drive = atan2(data.gravity.x, data.gravity.z) - M_PI / 2
            self.label.transform = CGAffineTransformMakeRotation(CGFloat(steer))

            // | Turn Power | Turn H-Bridge | Drive Power | Drive H-Bridge |

            var transmitData: UInt8 = 0;

            let steerParam = 0.2
            let driveParam = 0.2

            //Add drive speed
            transmitData |= min(UInt8(abs(drive) * 3), 3)

            transmitData = transmitData << 2

            //Add drive h-bridge command
            if drive > -driveParam && drive < driveParam {
                transmitData |= 0x3
            } else if drive > driveParam {
                transmitData |= 0x2
            } else {
                transmitData |= 0x1
            }

            transmitData = transmitData << 2
```

```
            //Add turn speed
            transmitData |= min(UInt8(abs(steer) * 3), 3)

            transmitData = transmitData << 2

            //Add turn h-bridge command
            if steer > -steerParam && steer < steerParam {
                transmitData |= 0x3
            } else if steer > steerParam {
                transmitData |= 0x2
            } else {
                transmitData |= 0x1
            }

            //Generate byte to transmit
            let dataBytes = NSData(bytes: &transmitData, length: sizeof(UInt8))
            print(String(transmitData, radix: 2))

            //Transmit byte over bluetooth
            if let characteristic = self.carPeripheral?.services?.first?.characteristics?.
                first {
                self.carPeripheral?.writeValue(dataBytes, forCharacteristic:
                    characteristic, type: CBCharacteristicWriteType.WithoutResponse)
            }
        }
    }
}


//MARK: - CoreMotion
extension ViewController: CBCentralManagerDelegate, CBPeripheralDelegate {
    //Start searching for bluetooth devices
    func centralManagerDidUpdateState(central: CBCentralManager) {
        if central.state == CBCentralManagerState.PoweredOn {
            // Scan for peripherals if BLE is turned on
            central.scanForPeripheralsWithServices(nil, options: nil)
            print("Searching for device")
        }
        else {
            print("BLuetooth not initialized")
        }
    }

    //Found a device
    func centralManager(central: CBCentralManager, didDiscoverPeripheral peripheral:
        CBPeripheral, advertisementData: [String: AnyObject], RSSI: NSNumber) {
        let deviceName = "RCCAR1"
        let nameOfDeviceFound = (advertisementData as NSDictionary).objectForKey(
            CBAdvertisementDataLocalNameKey) as? NSString

        if nameOfDeviceFound == deviceName {
            // Update Status Label
            print("Found device")

            // Stop scanning
            centralManager.stopScan()
            // Set as the peripheral to use and establish connection
            carPeripheral = peripheral
            carPeripheral?.delegate = self
```

```
            centralManager.connectPeripheral(peripheral, options: nil)
        }
        else {
            print("Found \(nameOfDeviceFound) instead")
        }
    }

    //Connect to bluetooth device
    func centralManager(central: CBCentralManager, didConnectPeripheral peripheral:
        CBPeripheral) {
        print("I found it! Now discover services")
        peripheral.discoverServices(nil)
    }

    //Discover services on bluetooth device
    func peripheral(peripheral: CBPeripheral, didDiscoverServices error: NSError?) {
        guard let services = peripheral.services where error == nil else {
            return
        }

        print("Peripheral services: \(services.map{ $0.UUID })")

        for service in services {
            peripheral.discoverCharacteristics(nil, forService: service)
        }
    }

    //Discover characteristics for device
    func peripheral(peripheral: CBPeripheral, didDiscoverCharacteristicsForService service
        : CBService, error: NSError?) {
        print("enabling peripheral")

        guard let characteristics = service.characteristics where error == nil else {
            return
        }

        //Change status label
        NSOperationQueue.mainQueue().addOperationWithBlock {
            self.label.text = "Connected to car"
        }

        print("Characteristics: \(characteristics.map{ $0.UUID })")

        //Start data notifications for device
        for characteristic in characteristics {
            carPeripheral?.setNotifyValue(true, forCharacteristic: characteristic)
        }
    }

    //Device characteristics changed
    func peripheral(peripheral: CBPeripheral, didUpdateValueForCharacteristic
        characteristic: CBCharacteristic, error: NSError?) {
        guard let dataBytes = characteristic.value else {
            return
        }

        let dataLength = dataBytes.length
        var dataArray = [UInt8](count: dataLength, repeatedValue: 0)
        dataBytes.getBytes(&dataArray, length: dataLength * sizeof(UInt8))
```

```
        print(dataArray)
        print(String(bytes: dataArray, encoding: NSASCIIStringEncoding))
    }

    //Lost device
    func centralManager(central: CBCentralManager, didDisconnectPeripheral peripheral:
        CBPeripheral, error: NSError?) {
        label.text = "Searching for car..."
        print("Car disconnected")
        central.scanForPeripheralsWithServices(nil, options: nil)
    }
}
```