Carisma

This toy car will have everyone in the room charmed for hours.

Mai Tai Beach Babies Incorporated

Revision	Description
1A	Ethan Miller – Sections 1,2, Definition and description of Carisma device as of 1/27/2019
1B	Evan Mason – Section 4, Hardware text and diagrams
1C	Lauren Crawford – Section 6, Testing Procedures
1D	Hunter Morrisroe – Section 3 – Block Diagram Overview and descriptions
2A	Hunter Morrisroe – Updated hardware schematics and text
2B	Ethan Miller – Included mechanical framework information, added abbreviations
2C	Lauren Crawford – Included additonal testing procedure, added abbreviations
2D	Evan Mason – Added mechanical and electrical documentation in section 4
3A	Ethan Miller – Rewrote different test processes, input previous revision changes and took photos
3B	Lauren Crawford – Included flowcharts, description and code for main.c and ports.c
3C	Evan Mason – Addition of information relating to IR usage, the ADC, and the ADC interrupt
4A	Ethan Miller – Creation of additional flowcharts, updating vehicle capabilities, Table of Contents, code
4B	Lauren Crawford – Addition of info in section 7, supplying section 9 code, flowchart creation
4C	Evan Mason – Updated abbreviations table, Serial info in sections 7, 8, 9
5A	Ethan Miller – Updated TOC & Figure Table, added pictures, revised various sections of text
5B	Lauren Crawford – Included more test processes, updated flowchart, and added necessary labels.
5C	Evan Mason – Included IOT information in sections 3 and 4. Updated section 5 and 8 contents.

Team Members
Ethan Miller
Evan Mason
Lauren Crawford

Originator: ECE 306 Team 24

Checked: Ethan Miller | Released: 4/27/20

Filename: Proj_Writeup_Team24.docx

Title: **Carisma**
This toy car will have everyone in the room charmed for hours.

Date:	Document Number:	Rev:	Sheet:
04/27/2020	**0-0000-420-0000-05**	**5C**	**1 of 39**

Table of Contents

Figure Index

1. Scope

This document contains the device information and technical specifications of Carisma, the most entertaining toy around. Carisma is a small battery powered vehicle designed to autonomously find and follow a black line of any shape on the floor. The product can trace out various shapes on the floor, and even communicate with other devices and controllers using its WiFi antenna! Carisma is a fascinating toy built to be enjoyed by anyone ages 5-99+. Continue reading for an overview and in-depth technical description of Carisma.

2. Abbreviations

Abbreviation	Definition
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
MSP	Mixed-Signal-Processor
FET	Field Effect Transistor
RAM	Random Access Memory
FRAM	Ferroelectric Random Access Memory
I/O	Input/Output
DC	Direct Current
PCB	Printed Circuit Board
IR	Infrared
ADC	Analog-to-Digital Converter
UART	Universal Asynchronous Receiver-Transmitter
Rx	Serial Receiving of Data
Tx	Serial Transmission of Data
IOT	Internet-Of-Things
TCP	Transmission Control Protocol
IP	Internet Protocol

3. Block Diagram Overview

The Carisma design can be broken down into 7 main parts as seen in the blocks below. See subsequent sections for block-by-block descriptions of each system.

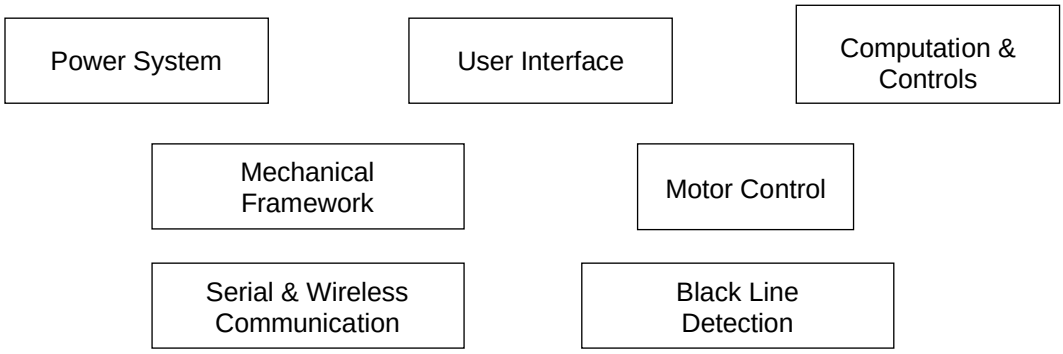


Figure 1 - Block Diagram Overview

3.1. Block 1 – Power Delivery

Carisma's power system consists of a battery source, and electricity delivery circuitry for proper voltage control as well as fail safe operation.

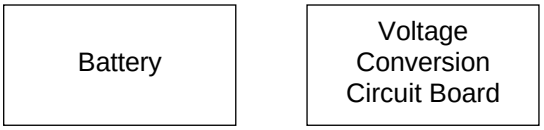


Figure 2 – Power System Overview

3.2. Block 2 – User Interface

Carisma's user interface consists of 2 control switches, 3 LED's, a Thumb Wheel, and an LCD display. The user can read device feedback on the display, read device signals via the LED's, and give various inputs to the device using the switches and thumb wheel.

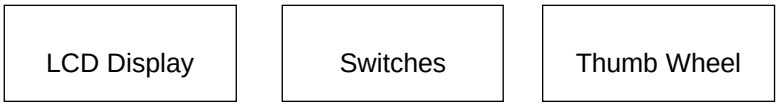


Figure 3 – User Interface Overview

3.3. Block 3 – Computation & Controls

Carisma's computation system is based off a Texas Instruments MSP430 Microcontroller. This board houses the processing and memory features of the Carisma. Other relevant hardware includes the Infrared emitters and detectors used to give Carisma sight.

Figure 4 – Computation Overview

3.4 Block 4 – Mechanical Framework

The physical structure of Carisma is composed of a chassis for the base of the car. Built upon the chassis are the left and right motors, the microprocessor, the power board, and the control board. Lego wheels are attached to the motors through a Lego adapter at the front of the car, with a caster wheel in the rear.

Figure 5 – Mechanical Overview

3.5 Block 5 – Motor Control

The vehicle's chassis houses two DC motors that are powered and programmed using Carisma's Power System and Computation System. To control the voltage delivered to these motors, Carisma boasts a 13-transistor full H-Bridge on its FET board. See section 4 for detailed information about the control board.

Figure 6 – Motor Control Overview

3.6 Block 6 – Black Line Detection

Carisma utilizes the power of Infrared technology to provide the car with an ability to follow any black line. The Left and Right Detectors are IR Phototransistors which receive IR light sent from the IR LED Emitter. Specifically, the IR light emitted bounces off of the ground and is received by the IR Phototransistors. Depending on the color of the area below the Emitter, varying amounts of IR light is received. The received signals are sent to the Texas Instruments MSP430 Microcontroller's ADC, where values are converted. See section 4 for detailed information about Black Line Detection.

Figure 7 – Black Line Detection Overview

3.6 Block 7 – Serial & Wireless Communication

In order to control Carisma wirelessly, we must have established bidirectional communication with an IOT device. In this application, the Telit GS2101MIP Wi-Fi module was used. This IOT module connects to J9 on Carisma's control board and allows data to be sent to/from the MSP430's UCA0 serial port. Opening a TCP Telnet Terminal and connecting to the module allows for the user to send commands that change certain functionalities of the vehicle. Commands sent from the TCP client must be formatted in a specific way to allow correct processing of user-sent commands. Command formatting for your device may vary, but follow a general format: [command specifier]+[passcode]+[command type]+[duration (if applicable)]. See section 4 for more information regarding wireless communication.

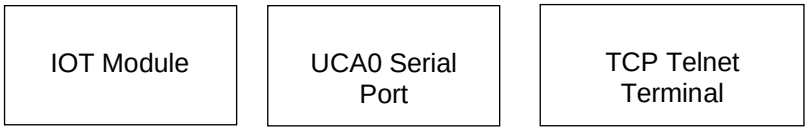


Figure 8 – Serial & Wireless Communication Overview

4. Technical Hardware Description

The electrical hardware utilized by Carisma consists of a battery-based power supply, DC motors, and multiple custom-built PCB's designed to interface with the motors and the processor. All these components were built around a Texas Instruments MSP-EXP430FR2355 LaunchPad™ Kit. Read the subsections below for detailed information on each component.

4.1 Power System

The power system for Carisma consists of two components – a 4x AA battery pack, and a voltage converter circuit board. The battery pack is connected to the voltage converter at J10, shown in the Power System schematic

below. The line RAW_BATTERY indicates a line connected directly to the battery pack. The battery pack provides roughly 6.4V. The power is then converted, through the schematic below, which contains a voltage divider, and buck boost (SEPIC) converters, to drop the power and provide a consistent supply of 3.3 V to the power board.

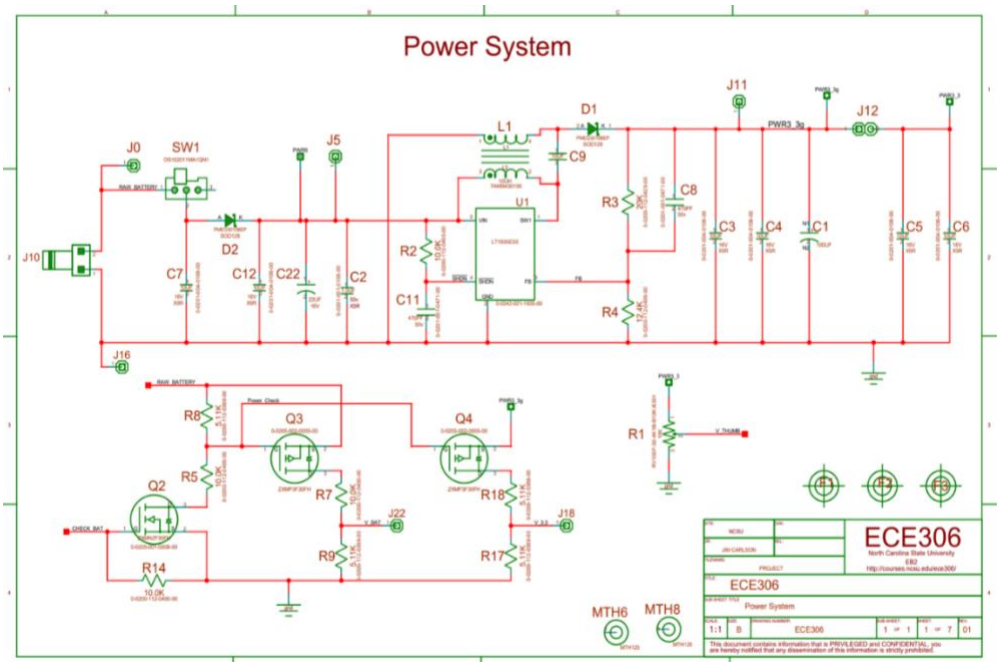


Figure 9 – Power System schematic

4.2 User Interface

The User Interface for this device consists of an LCD display, two control switches, and a thumb wheel. The LCD display shows lines of text to the user and can be switched to an alternate string of text when the switches are pressed. Likewise, the switches can be used to update the text in the display. The thumb wheel can be used to modify parameters in the processor.

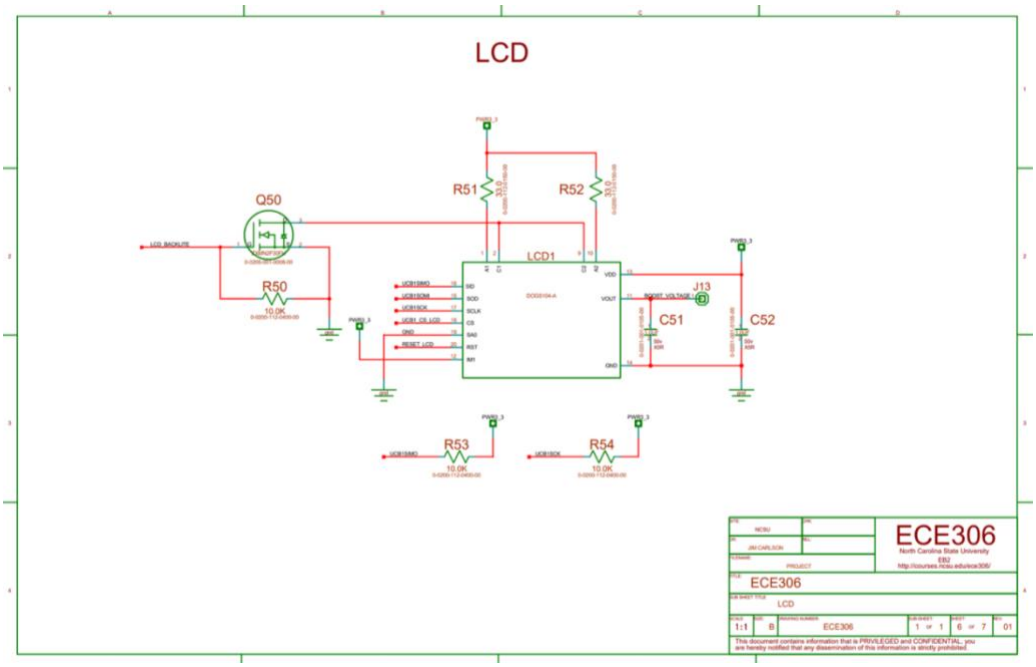


Figure 10 – LCD Display schematic

The schematic shown above shows how the LCD is assembled on to the device. Here there is power coming from the power system to power the display. There are various signals provided to the pins of the LCD display, to control its functionality. LCD_BACKLITE provides a signal to turn the backlight of the display on or off. Reset_LCD will reset the display, if low. These signals are defined in the software and can be controlled in the software to update, reset, or backlight the display.

4.3 Computation & Controls

At the heart of Carisma we are running a Texas Instruments MSP-EXP430FR2355. This microcontroller offers our product General Purpose Registers, RAM, FRAM, Digital Ports, Analog Ports, Timers, and Clocks. Peripherals on the vehicle such as the motors are controlled using the digital ports shown in the schematic below, using code to set pin voltages to high or low. The interconnect schematic shows the pin connections from the microcontroller to the power board, as well as the designated pin function. For our sensor network we are using 1 infrared emitting LED, and 2 infrared detecting LEDs. The emitter blasts the ground with IR light, and the detectors output analog voltages that our fed to the ADC inside of the MSP430 to give us digital values that Carisma can use to "see" the ground.

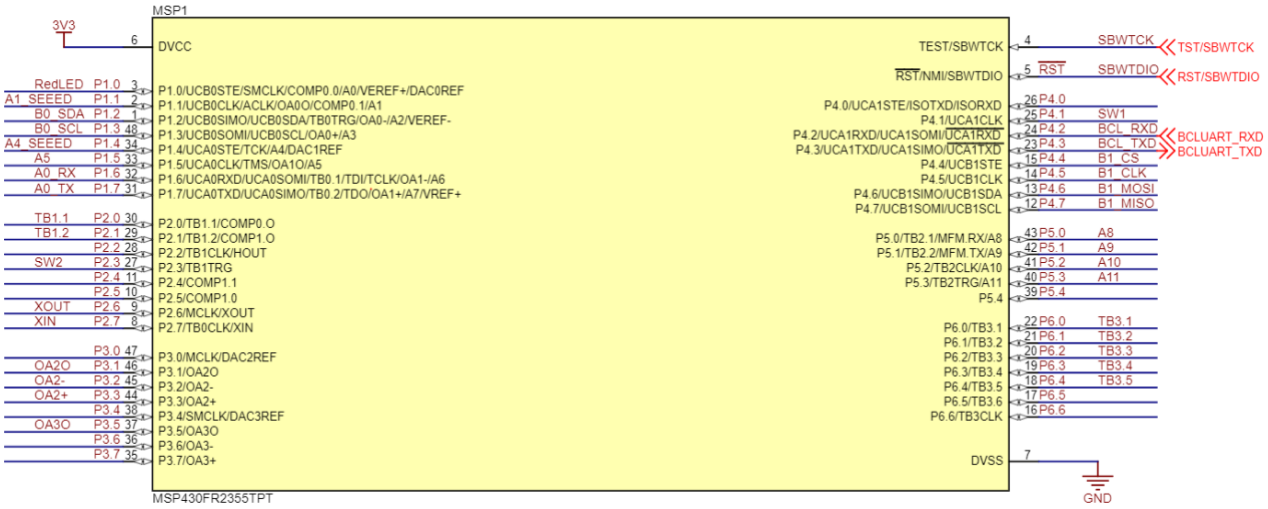


Figure 11 – MSP430 Port Schematic

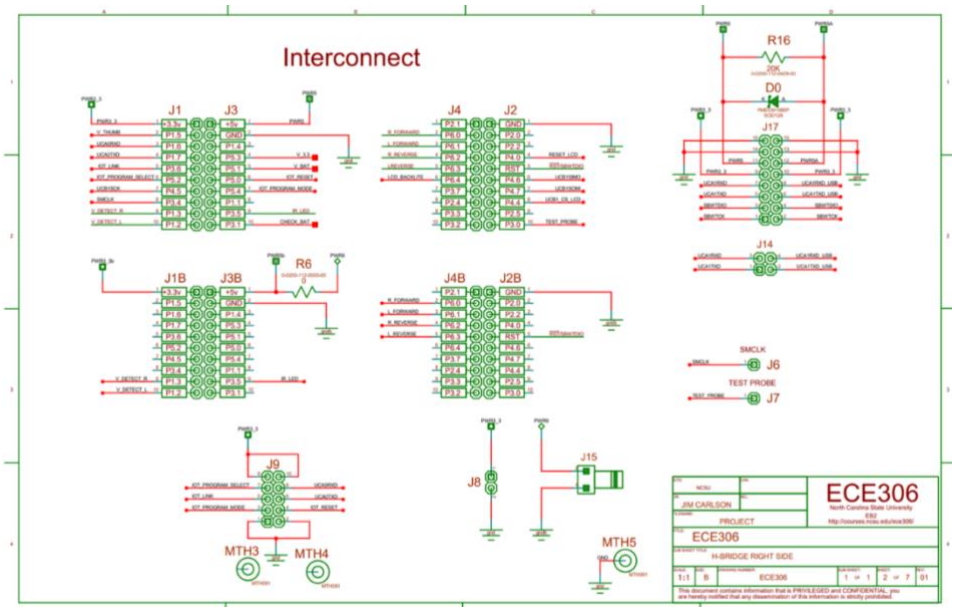


Figure 12 – Interconnect Schematic

4.4 Mechanical Framework

The chassis of the vehicle is composed of one laser cut section of 4mm thick red acrylic. The wheels are 1-inch plastic Lego hubs, with 2.5-inch rubber Lego tires. These wheels offer good traction when climbing up small ledges. The rear caster is a smaller wheel designed to swivel 360 degrees, offering smooth forward roll as well as traction while the front wheels steer the vehicle. The motors operating the front wheels are 120:1 DC Gearmotors that operate in the range of 3-6V. These motors offer 1.4 kg*cm of torque at 800 mA, plenty to drive our consciously lightweight device.

4.5 Motor Control

Attached to the Computation and Power Systems on Carisma is a PCB referred to as the “FET Board”. This piece of hardware contains a partial H-Bridge composed of two N type FET transistors. These transistors receive a digital IO signal from the Computation System and allow a voltage to be applied across either motor. (example: see Q41 on schematic below)

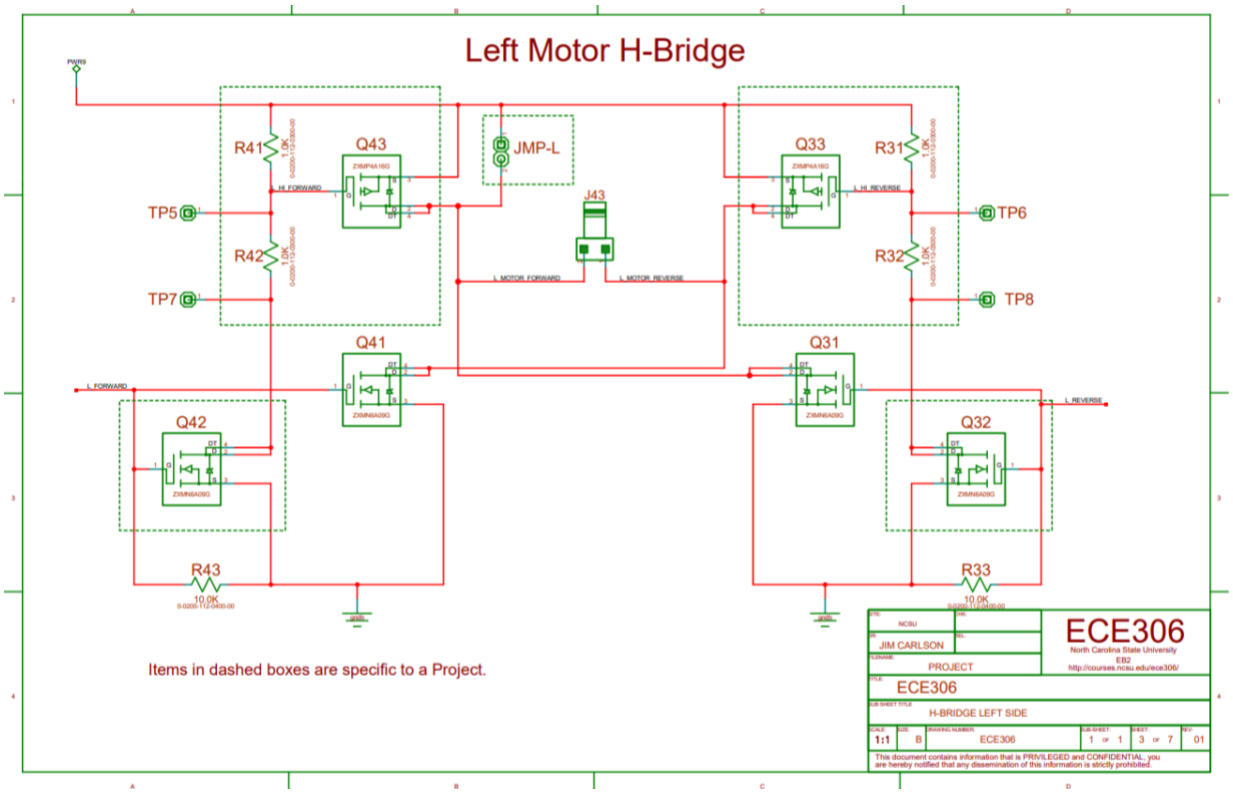


Figure 13 – Left Motor H Bridge Schematic

4.6 Black Line Detection

Black Line detection is made possible with 4 main components – a Center IR LED Emitter, one ‘Left Side’ IR Phototransistor, one ‘Right Side’ IR Phototransistor, and the MSP430’s ADC. The Emitter and Detectors is a separate module from the main control board. The Emitter/Detector control board sits between 0.5 – 1.0 inches from the ground and resides on the front-end of Carisma. As IR light is emitted, each detector receives the light signal after it bounces off of the area below the emitter. One detector representing the ‘left’ half of the signal and the other being the ‘right’ side. Depending on the color of the area, the amount of IR light reflected varies. These signals are sent to the microcontroller’s ADC, where they are converted into ‘readable’ values. See section(s) 7, 8, and 9 for further information regarding ADC configuration and usage.

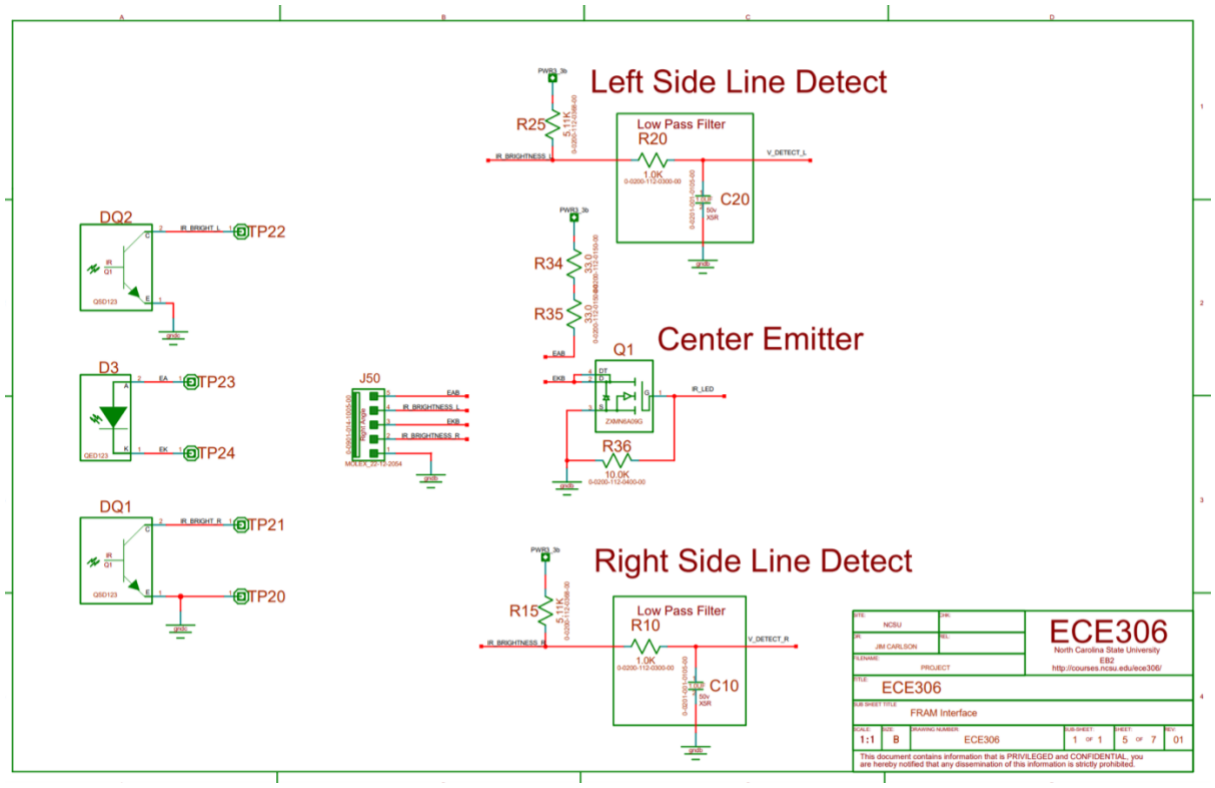


Figure 14 – IR LED Emitters (Left and Right) and IR Detector

4.7 Telit GS2101M IOT Module

The Telit GS2101MIP IOT module serves as a 'middle-man' for allowing wireless bidirectional communication between the MSP430 microcontroller and a user-controlled TCP Telnet Terminal. The IOT module resides on J9, where the MSP430's UCA0 serial port is configured to communicate at 9,200 baud. Once connected and configured for a specified router and port, the user is allowed to send commands that control different functionalities of the vehicle. Commands must be sent in a specific format, so that proper command processing can happen by the microprocessor. The general format follows as: [command specifier]+[passcode]+[command type]+[duration (if applicable)]. Where the command specifier allows the car to recognize a command has been sent. This may vary, but typically is a '^' or another character not typically used. The passcode can be any 4 digit number, but is specific to your model. Command types can be any of the following: Forward (F), Reverse (B), Right (R), Left (L), Arrived(A), Black Line Follow (L), and Exit (E). The last 3 digits of the command sent relate to the duration that the F, B, L, and R commands execute for, with exceptions including A, L, and E. Example commands include: '^####F020', '^####B100', '^####L002', '^####E'.

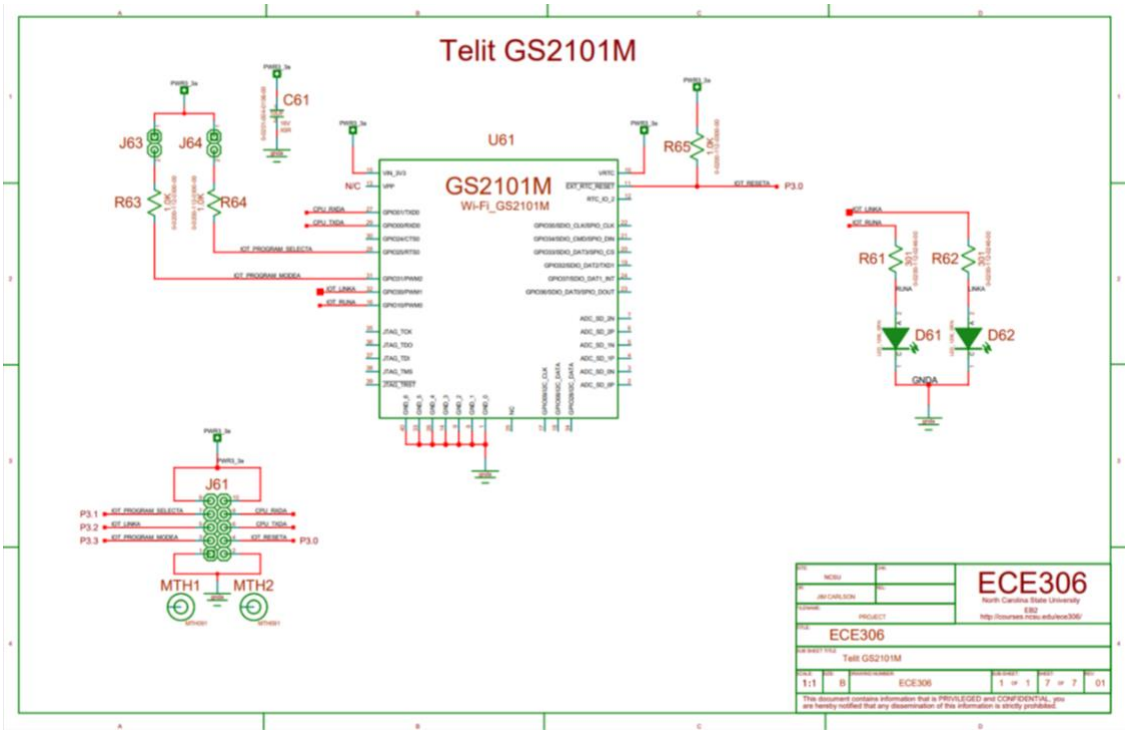


Figure 15 – IOT Module

5. Power Analysis

Carisma's power source is 4 AA batteries, with each at ~1.5 Volts, so that ~6 Volts can be supplied its' power system. Carsima allows the user to have fun every minute of every day for nearly 5 days! The order of calculating this estimate is as follows, and varies depending on usage:

Power Source: 4 1.5 Volt AA Batteries **Vcc**: 3.32 Volts **Current Consumption**: 25.2mA

Power Consumption = (3.32V)(25.2mA)

Power Consumption = 83.669mW

Power Consumption considering Synchronous Rectification Efficiency = (83.669mW) / (0.85)

Power Consumption considering Synchronous Rectification Efficiency = 98.428mW

Power Consumption per Battery = 24.607mW

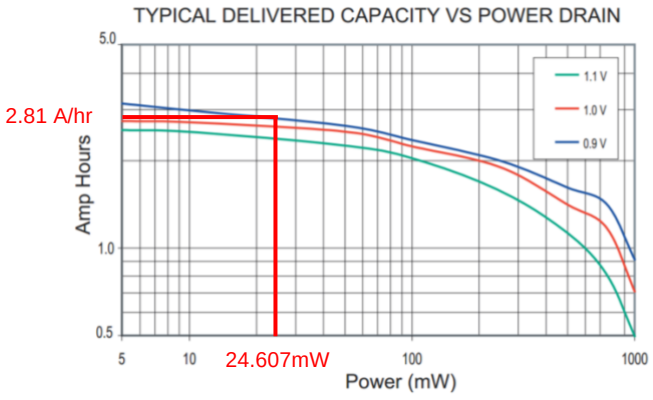


Figure 16 – Typical Delivered Capacity vs Power Drain

Typical Delivered Capacity = ~2810mA/hr

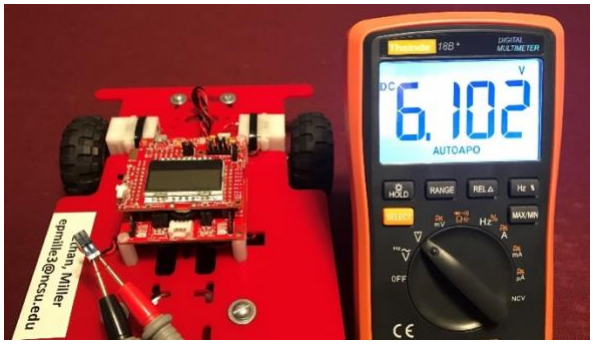
Total Hours of Usage = (2810mA/hr) / (25.2mA)

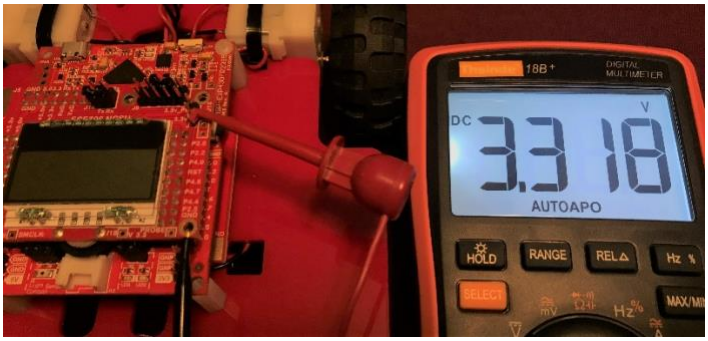
Total Usage = **111.5079 Hours or ~4.646 Days**

6. Test Process

6.1. Power Supply

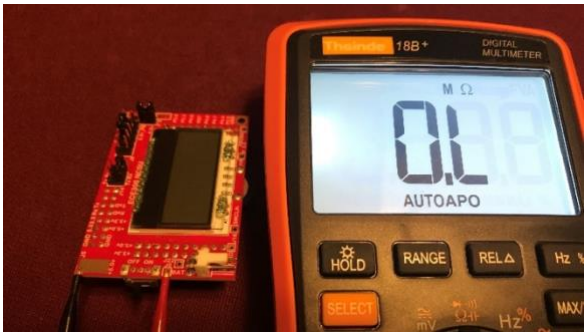
Following the soldering of the discrete components to the control board, testing must be done using a multimeter (or oscilloscope) and DC Power Supply. Making sure the power supply is supplying 4.5-6.0 volts, connect the terminals to the battery input. Using a Voltmeter with the negative lead grounded and the positive lead to probe pin 1 on J12, measured voltage at J12 should be ~3.3 Volts, this confirms that the Power System is working properly.

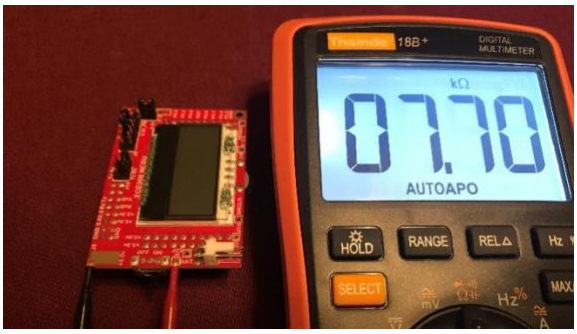




6.2. Switch 1

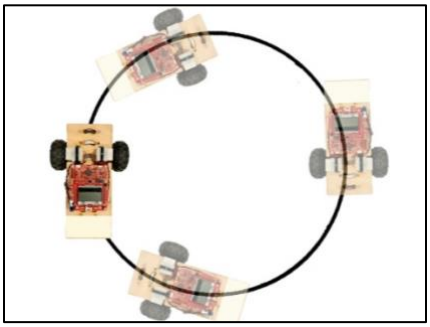
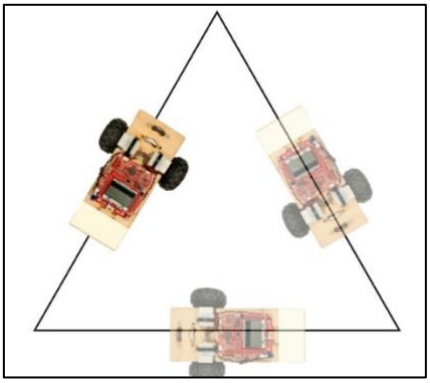
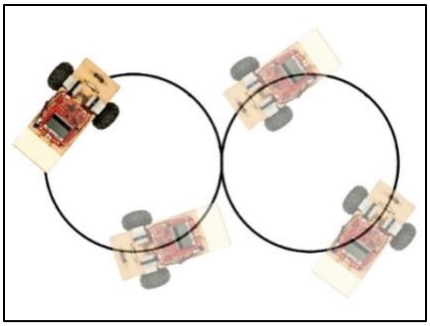
Prior to the installation of the Thumb Wheel and the LCD, one must verify that Switch 1 (SW1) is operating correctly. Doing so ensures that the power supply to the control board will function as expected. Using an Ohmmeter with the negative lead connected to J5 and the positive lead connected to J0, confirm that the switch will show an open circuit in the OFF position, and a very high (7.7 kΩ) resistance in the ON position.





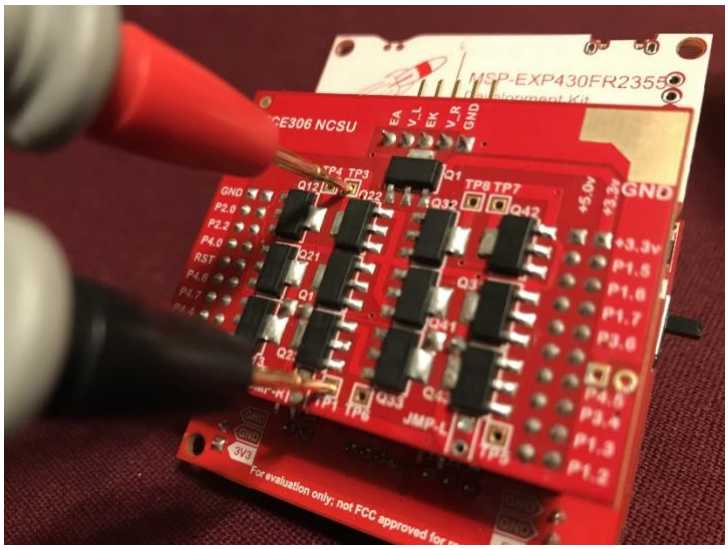
6.3. Shape Range

Once the car was able to create the circle, figure eight, and triangle movements, testing was done to ensure the movements were executed inside a specific boundary. The car was placed on a board with a 3' x 3' square outline to ensure that when the car was turned on it would not go past the square boundary. The car did not go off the board, confirming that it adhered to the required range. Images not to scale.

6.4. FETs

Following the installations of the various N-FETs and P-FETs to the Control Board's H-Bridge, immediate testing must be done to prove soldering was completed correctly. Each test must include the battery supply and have SW1 turned on. First, using IAR Embedded Workbench, write code so that all movements are turned OFF. Focusing on the FETs that drive the R_FORWARD signal, probe test points TP1 and TP3 individually with a multimeter. When OFF, TP1 and TP3 should read the battery voltage (~3.3 Volts). Write code to turn ON R_FORWARD. When ON, TP1 should read half of the battery voltage (~1.6 Volts) and TP3 should be at ground potential (0 Volts). Make sure to write code to set R_FORWARD OFF before continuing. Focusing on the FETs that drive the R_REVERSE signal, probe test points TP2 and TP4 individually with a multimeter. When OFF, TP2 and TP4 should read the battery voltage (~3.3 Volts). Write code to turn ON R_REVERSE. When ON, TP2 should read half of the battery voltage (~1.6 Volts) and TP4 should be at ground potential (0 Volts). Make sure to write code to set R_REVERSE OFF before continuing. Focusing on the FETs that drive the L_FORWARD signal, probe test points TP5 and TP7 individually with a multimeter. When OFF, TP5 and TP7 should read the battery voltage (~3.3 Volts). Write code to turn ON L_FORWARD. When ON, TP5 should read half of the battery voltage (~1.6 Volts) and TP7 should be at ground potential (0 Volts). Make sure to write code to set L_FORWARD OFF before continuing. Focusing on the FETs that drive the L_REVERSE signal, probe test points TP6 and TP8 individually with a multimeter. When OFF, TP6 and TP8 should read the battery voltage (~3.3 Volts). Write code to turn ON L_REVERSE. When ON, TP6 should read half of the battery voltage (~1.6 Volts) and TP8 should be at ground potential (0 Volts). Make sure to write code to set L_REVERSE OFF before continuing.



6.5. UCA1 & UCA0 Serial Communication

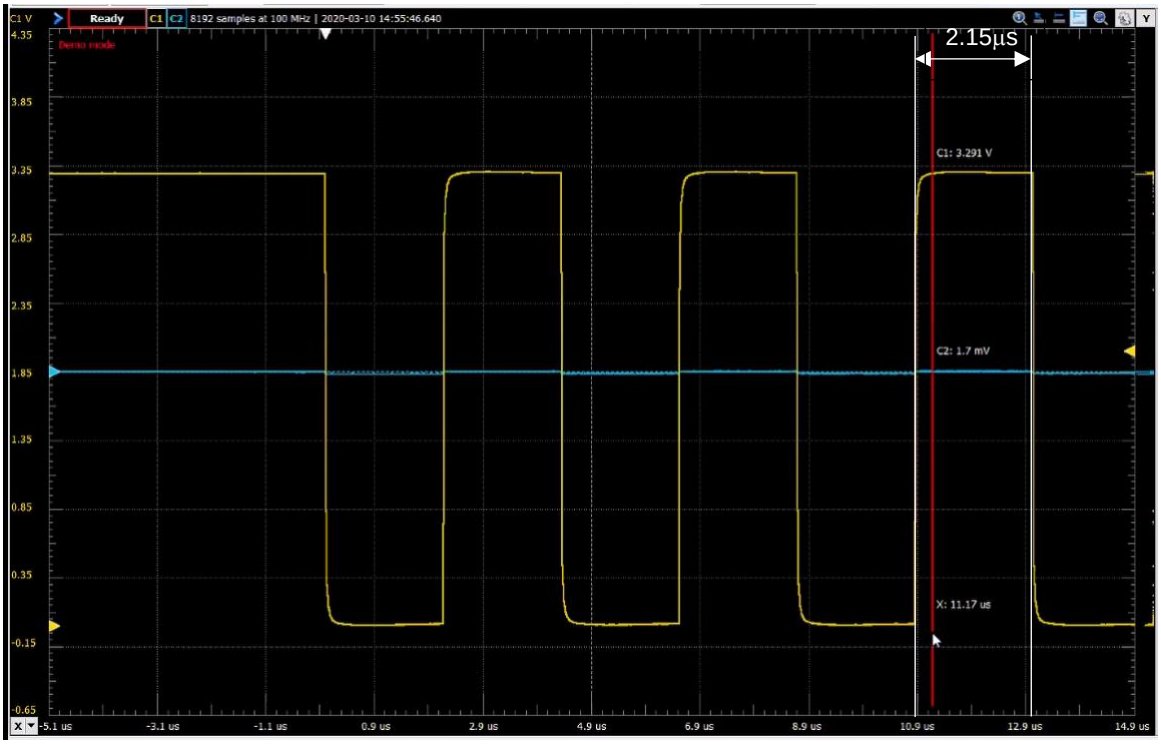


Figure 17 – Serial Communications Waveforms

Once the car can transmit and receive messages, the baud rate needs to be tested to ensure that it was transmitting at the correct rate so the message can be received accurately. To do this, an Analog Discovery 2 is connected to the transmit pin on the 2x2 jumper and connected to any ground pin on the board. Waveforms was used to measure the transmitted signal from the car, and the wave ran in scopes was analyzed. The baud rate is measured by the width of the shortest pulse. The inverse of that measured width gives you the baud rate it is transmitting at. Configured to be at 460,800 baud, we can calculate the actual baud rate of transmission by taking the inverse of the time-duration of the shortest pulse. Here, the shortest pulse is ~2.15 µsec, which calculates to ~465,116 baud, falling within the 5% margin of error and proving that transmission is working properly.

7. Software

The software is configured using a modular approach. Describe the code structure. Remember to identify the various functions and what operates when. This is a description of how your software is configured. You should be able to give this to one of your classmates and they would understand what you tried to do.

7.1. main.c

In main, the IOT is reset and then commands are transmitted to connect the IOT. The main code functions using a while loops that is always running. This while loop will check that the correct PIN was received. If it the correct PIN was entered, it will then check for which command was entered and do the necessary function associated with that command. A timer is also set to display each second throughout this while loop. Main.c also calls the initialization functions, Init_Ports, Init_Clocks, Init_Conditions, Init_Timers, and Init_LCD.

7.2. ports.c

Contains the initial configuration for all ports. There is a function for ports 1-6, where each function has the configurations for each pin associated with the ports. Each port function is called in Init_Ports. This segmentation makes it easy for Carisma's developers to find and modify information related to a specific port.

7.3. timers.c

Contains the configuration code for the timers in the MSP430. Timer B0 initializes both B0_0, B01-B0_1 and overflow. The timer function enables and disables various timers, as well as setting up the SMCLK source.

7.4. interrupt_timers.c

Contains the interrupt service routines pertaining to timers. There is an interrupt routine for TIMER0_B0_VECTOR and TIMER0_B1_VECTOR. Timer B0 is used to update the display. Timer B1 is used to handle switch presses.

7.5. ADC.c

ADC.c provides the full configuration needed to use the ADC module embedded in the MSP430 microcontroller. Here we configure a variety of attributes pertaining to the ADC module – conversion rate, conversion resolution, sampling period, references, individual external input channels, clock source, conversion mode, and a window comparator for low-power monitoring of input signals. For our use, the analog input sent from our detectors becomes quantized into a 12-bit digital value using the ADCMEM0 conversion register. The ADC uses the MODCLK as its' clock source to perform single-channel single-conversions. MODOSC is generated internally and typically is within a 4.0MHz to 5.4MHz range, depending on external variables. ADCINCH_X specifies which pins of the microcontroller are being linked to the ADC's input channels. In our case, ADCINCH_2 represents our left detector (V_DETECT_L), ADCINCH_3 represents our right detector (V_DETECT_R), and ADCINCH_5 being our thumbwheel input (V_THUMB). Each are configured as ADC channel inputs in the port configurations.

7.6. Interrupt_ADC.c

ADC interrupts are prioritized and combined to source a single interrupt vector (i.e. ADC_VECTOR). ADCIV is used to determine which ADC interrupt source requested the interrupt. For our use, we will focus on the ADCIV_ADCIFG interrupt flag. Each quantized analog signal is stored within the ADCMEM0 register and needs to be further converted into a readable string to be displayed. By calling the *HEXtoBCD* and *adc_line* functions, we can take each converted hex value, convert each into a BCD value, and then display our value onto Carisma's LCD Display.

7.7. Serial.c

In order to utilize serial communications, serial ports must be configured. Here pins 2 and 3 of port 4 are configured to be our UCA1 serial ports. Pin 2 acts as the pin responsible for the transmission of data (UCA1TX) and pin 3 as the pin responsible for receiving data (UCA1RX). Once these pins are configured to act as serial ports in the ports.c file, code within serial.c allows us to utilize serial communication. Within this, we create an 'empty' buffer (USB_Char_Rx[]) to create a record of received characters, we set the wanted baud rate, and enable the Rx interrupt. For our uses, the ability to switch between two different baud rates (460.8k and 115.2k baud) were instantiated using a user-input button press.

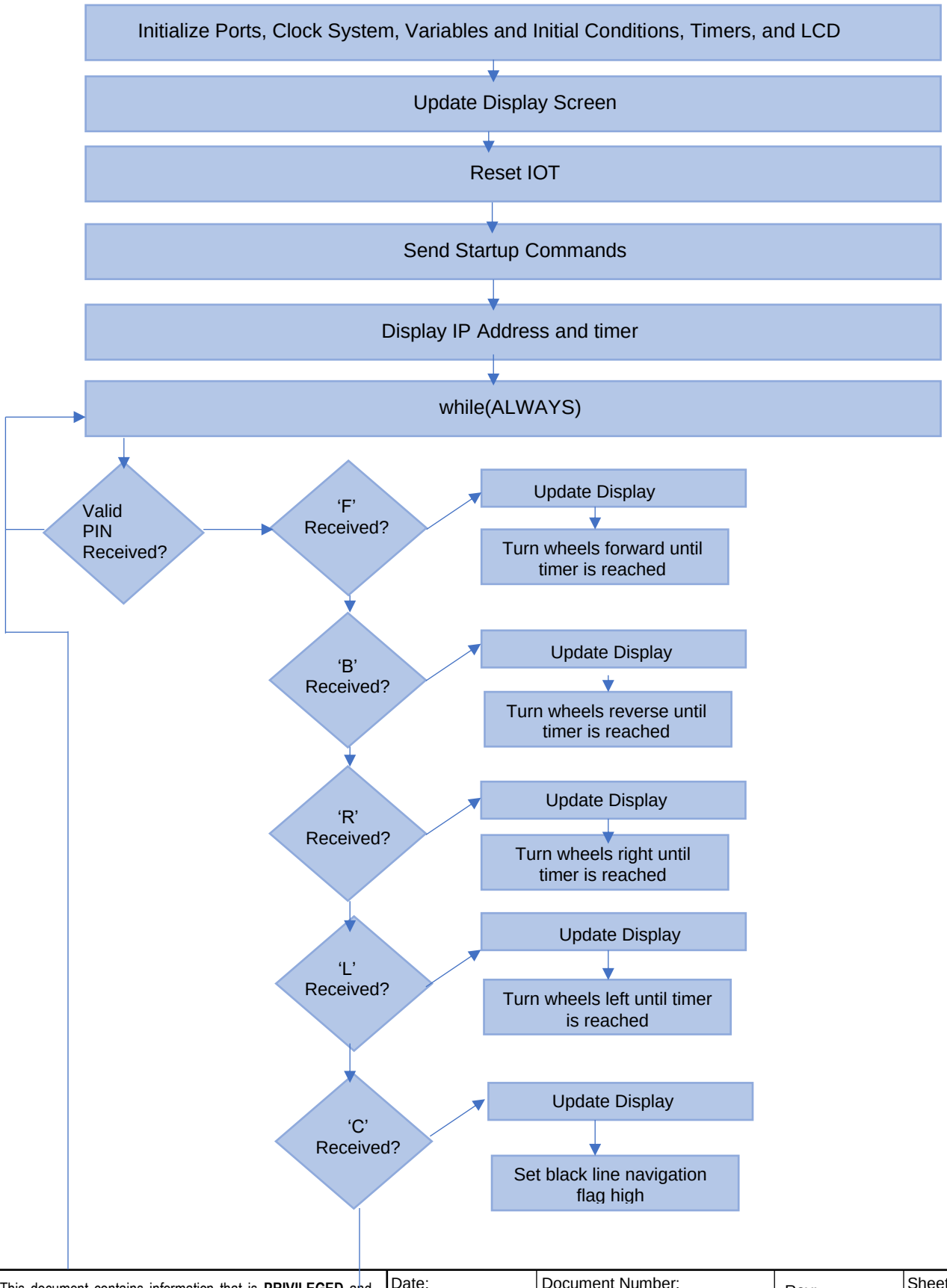
7.8. Interrupt_Serial.c

Both ports UCA0 and UCA1 have interrupt services routines for the purpose of transmitting and receiving serial data. Data being received is handled in this code using the input and output buffers of the serial ports available on Carisma's computational system. Incoming data can be placed into a ring buffer to be processed in any way that program sees fit, while outgoing data can be sent serially one byte at a time until the data is null.

8. Flow Charts

The following charts describe the functionality of each portion of Carisma's code.

8.1. Main Blocks



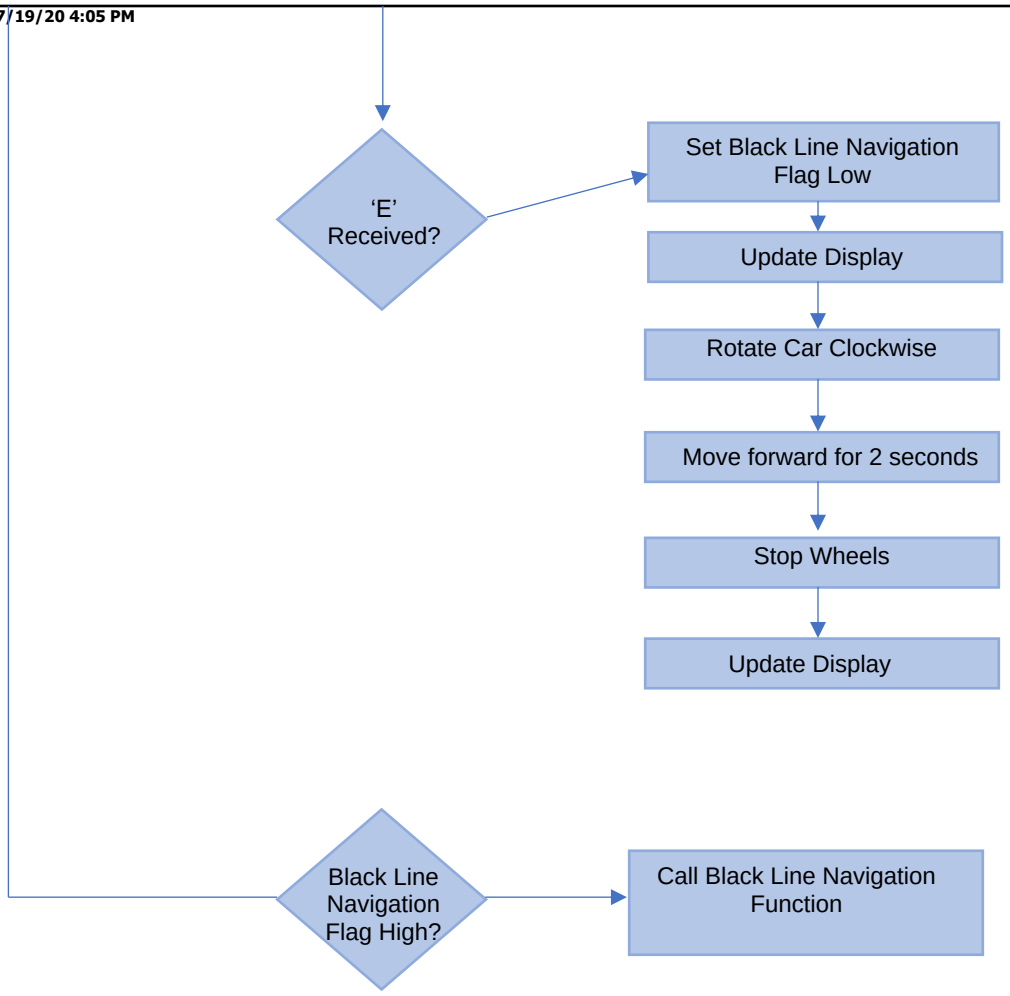


Figure 18 – Main Flowchart

8.2. Ports

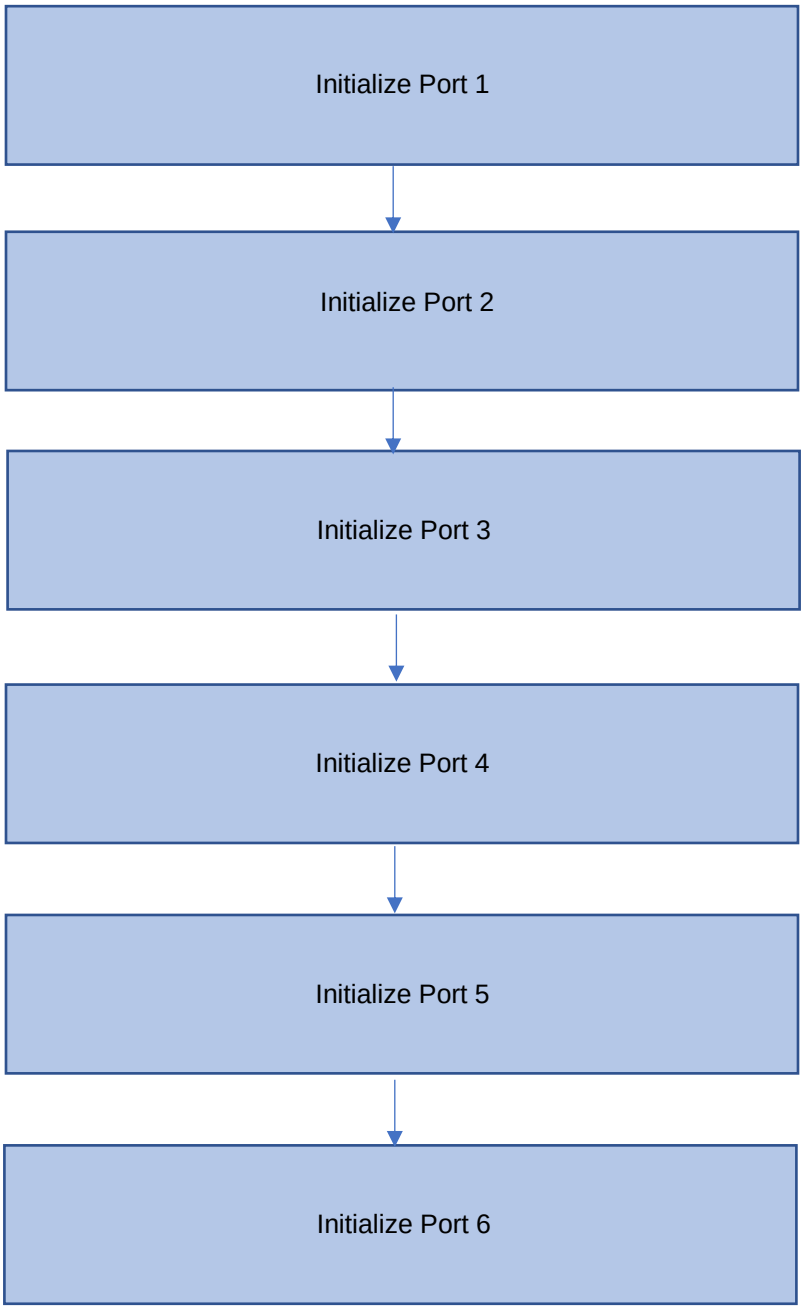


Figure 19 – Ports Flowchart

8.3. Timer Blocks

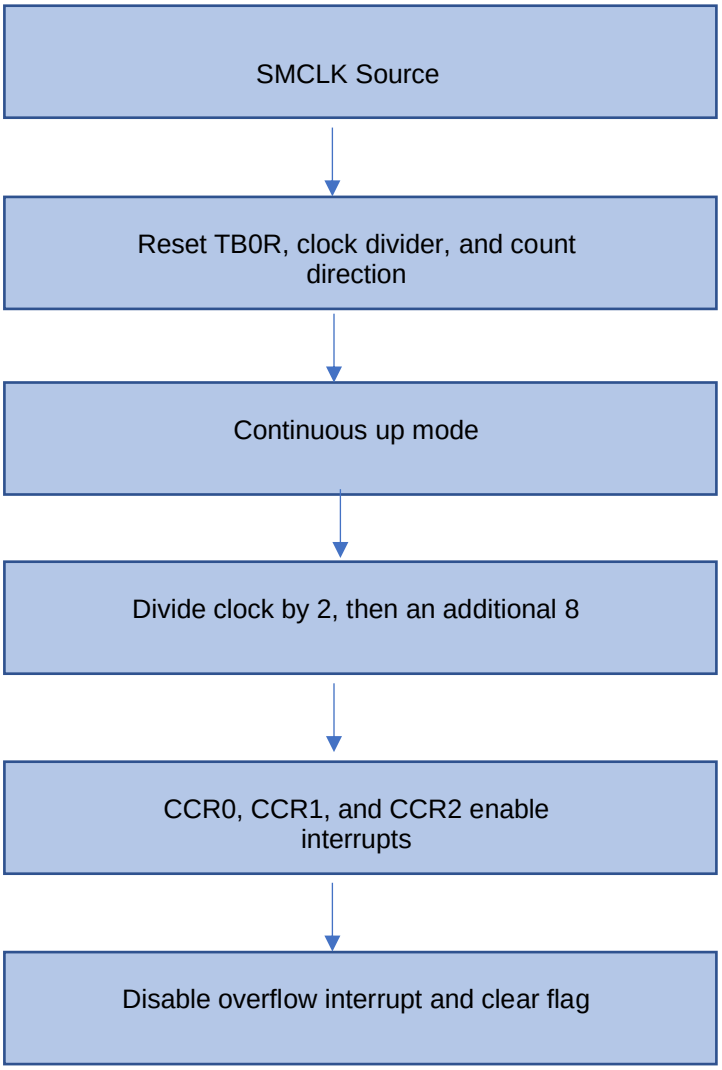


Figure 20 – Timers Flowchart

8.4. Interrupt Blocks

Interrupts are the best part about Carisma's software. Rather than constantly polling values to decide when to take a specific action, Carisma utilizes efficient, quick Interrupt Service Routines to interject program flow and execute many different functions.

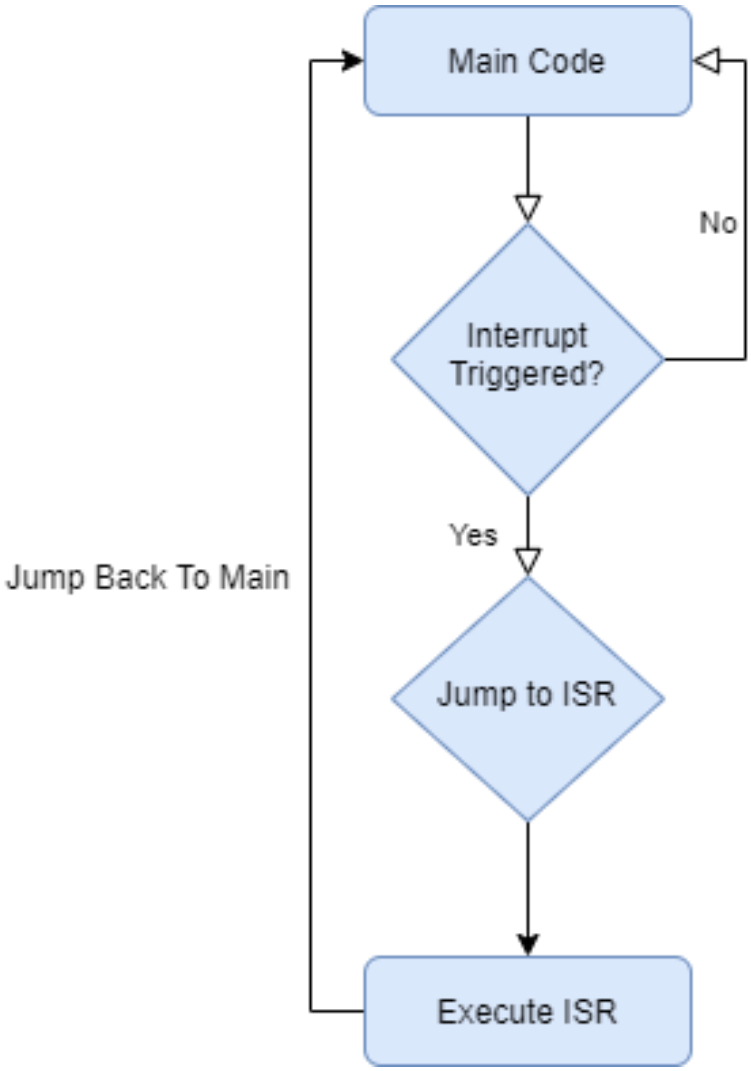


Figure 21 – Interrupts Flowchart

8.5. ADC Blocks

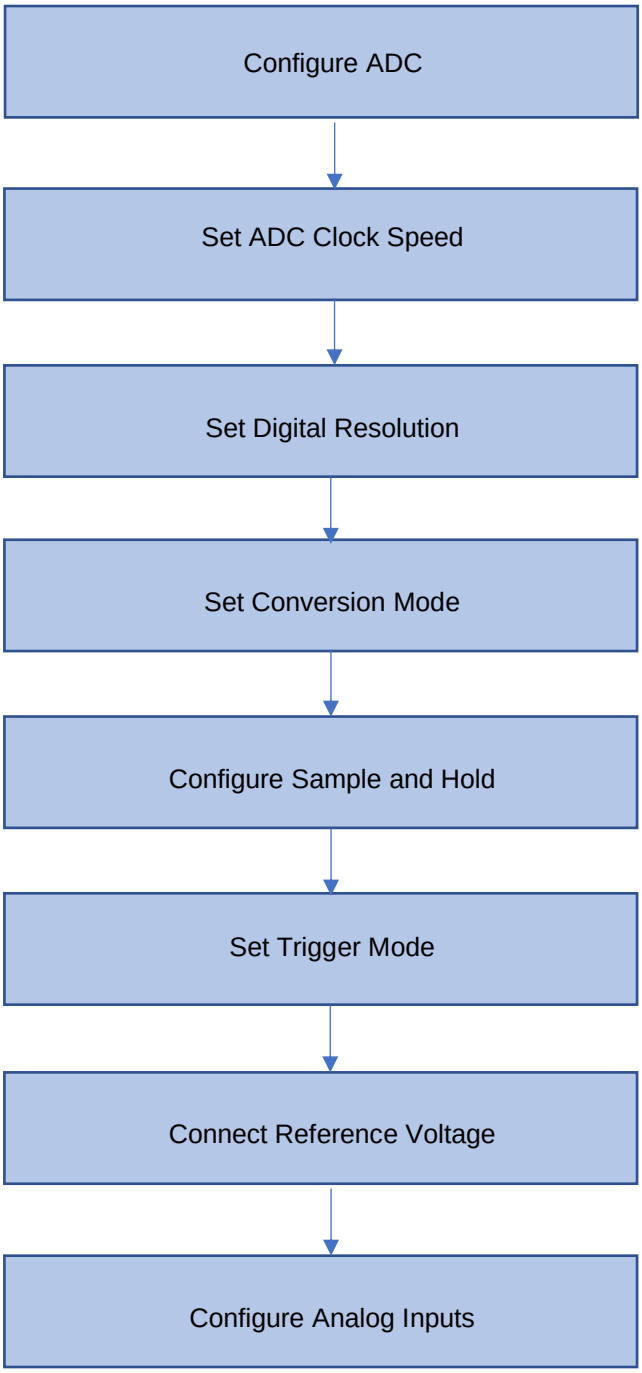


Figure 22 – ADC Configuration Flowchart

8.6. ADC Blocks

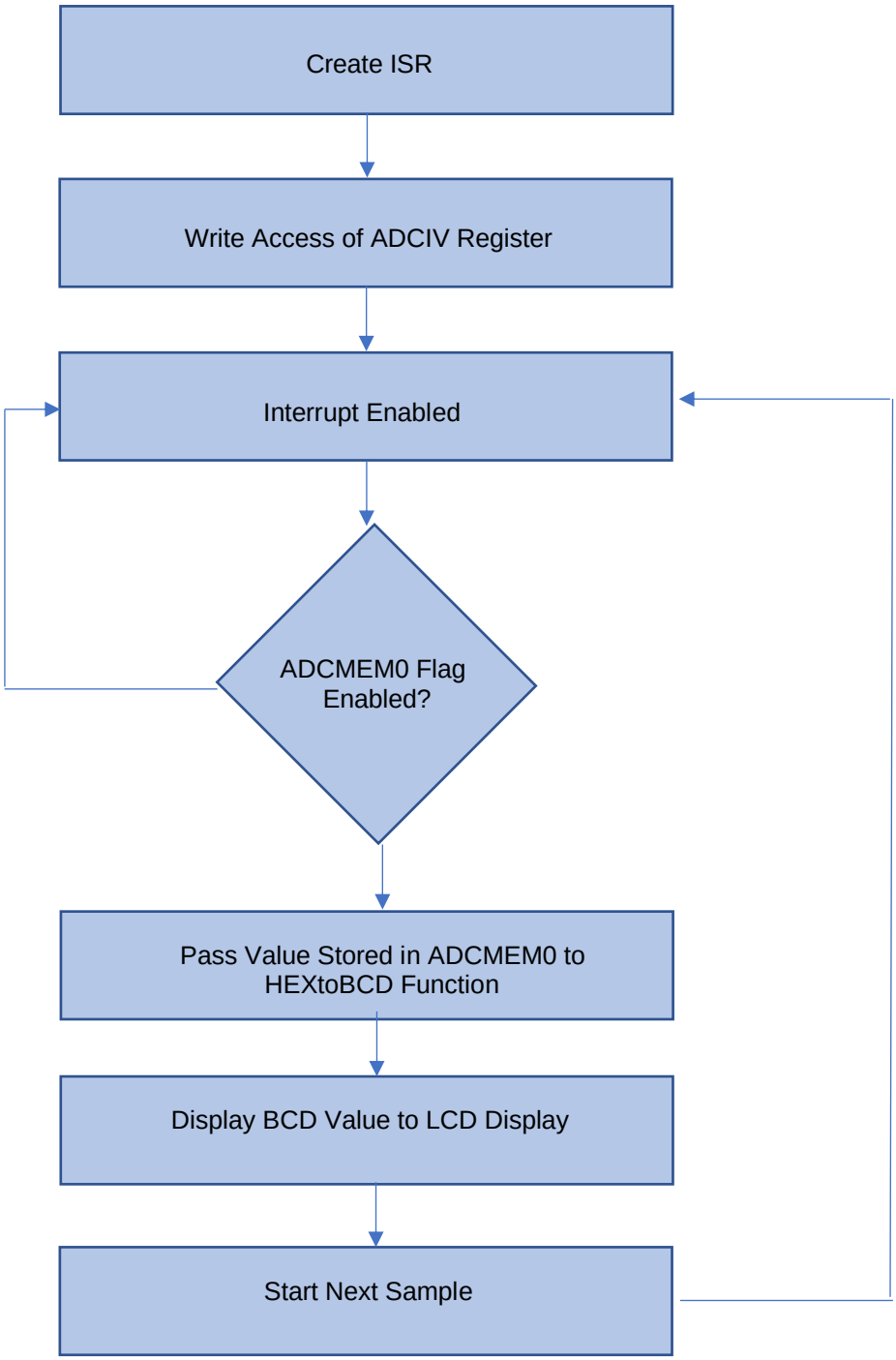


Figure 23 – ADC Interrupt Flowchart

8.7. Serial Blocks

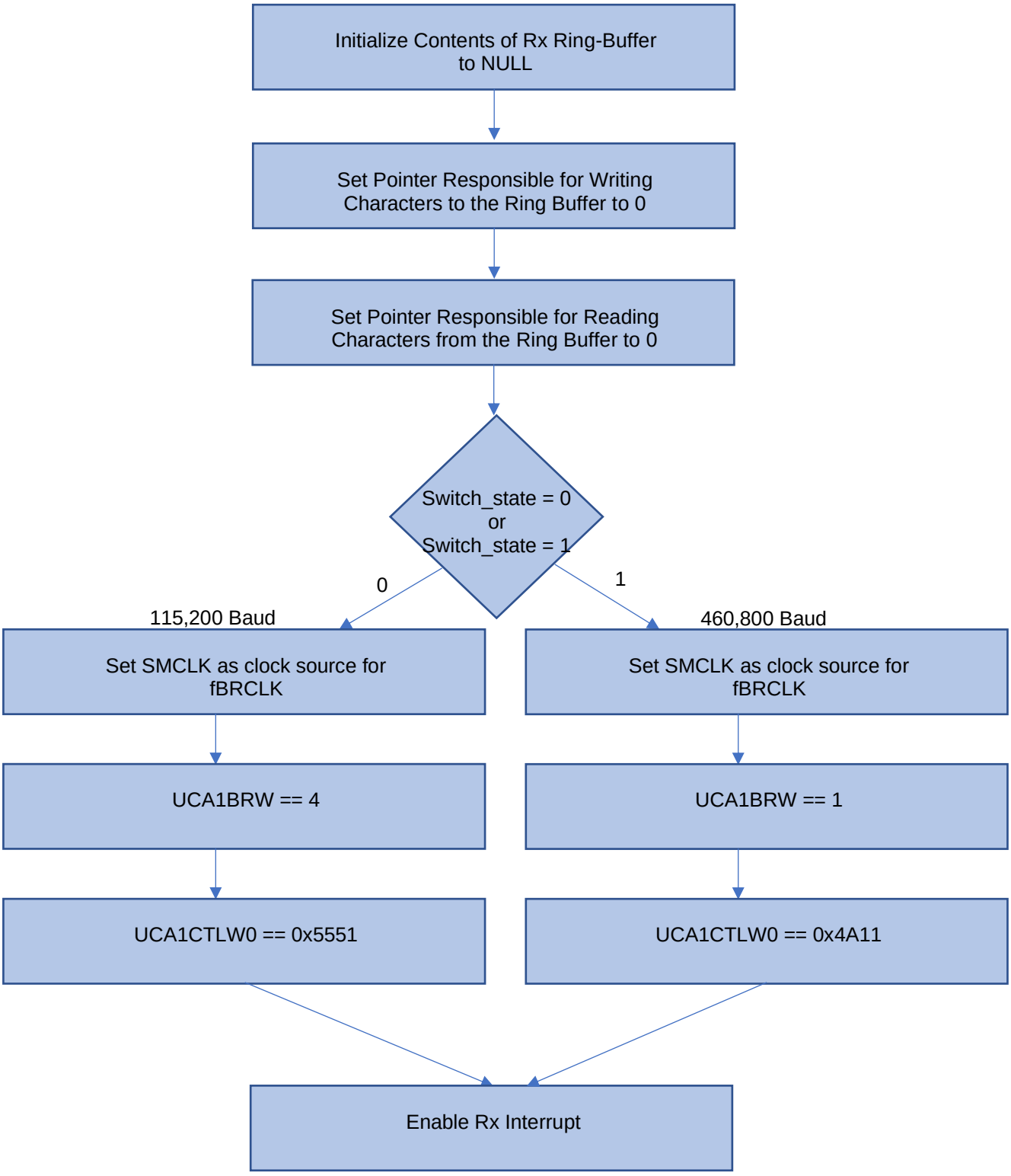


Figure 24 – Serial Configuration Flowchart

8.8. Serial Blocks

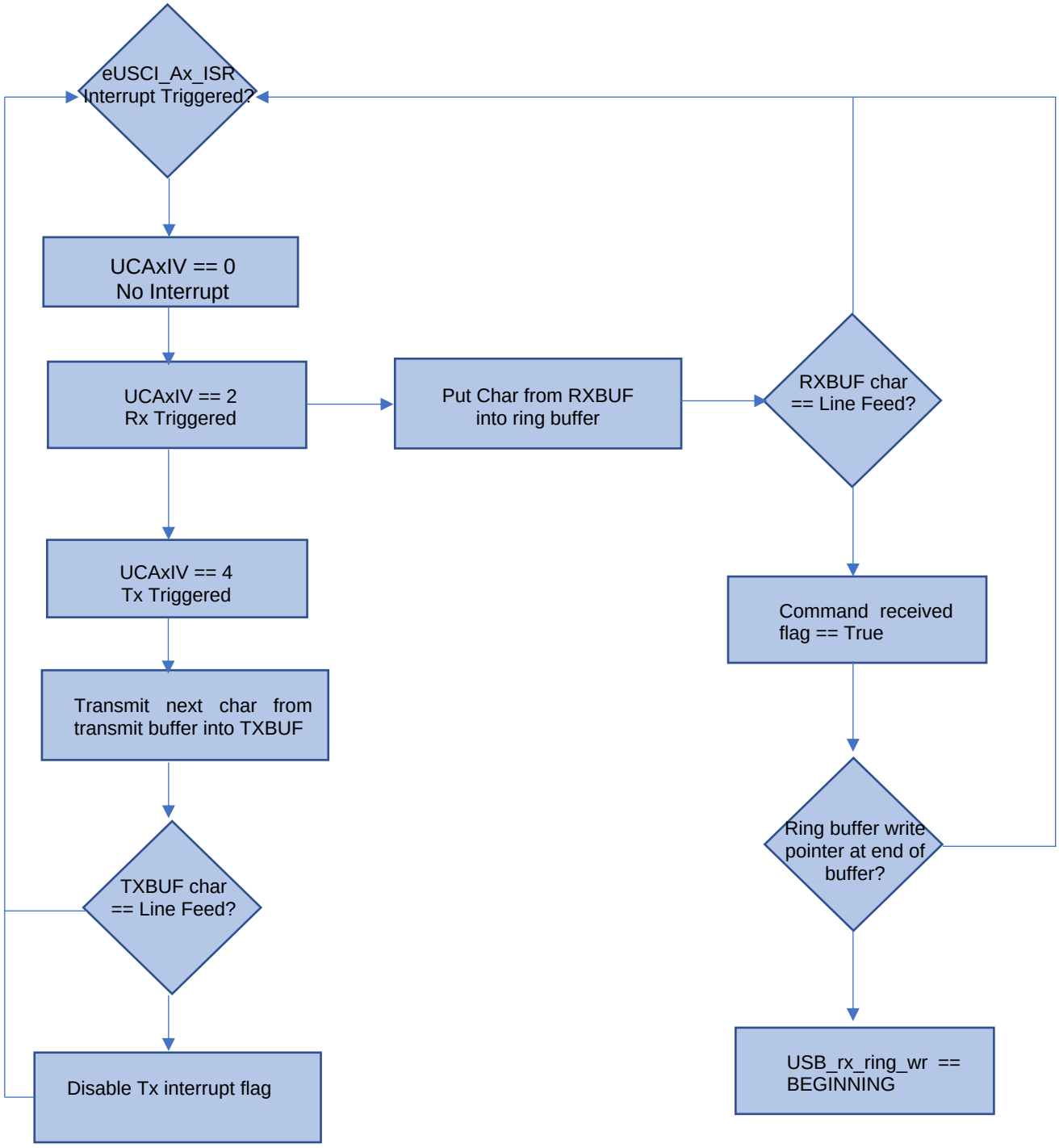


Figure 25 – Serial Interrupt Flowchart

9. Software Listing

This is a printout of the actual code, with each file in its own section.

9.1. main.c

```
// Begining of the "While" Operating System
//-------------------------------------------------------------------------------
  while(ALWAYS) {

    // ALL project 7 code commented out below for Project 8 purposes

    // SW1 Handler - transmit most recent command back to AD2 when pressed
    // move the command from the bottom line of the display to the second line of the display
    if (pressFlag1 == TRUE) {

      strcpy(display_line[DISPLAY_LINE0], " TRANSMIT ");
      update_string(display_line[DISPLAY_LINE0], DISPLAY_LINE0);

      // move the command being transmit to line 2
      strcpy(display_line[DISPLAY_LINE1], recieved);
      update_string(display_line[DISPLAY_LINE1], DISPLAY_LINE1);
      strcpy(display_line[DISPLAY_LINE3], "          ");
      update_string(display_line[DISPLAY_LINE3], DISPLAY_LINE3);
      display_changed = TRUE;


      pressFlag1 = FALSE;
      sendFlag = TRUE;
      Time_Sequence = RESET_STATE;
    }

    // SW2 Handler - toggle between baud rates and display the current baud rate on line 3
    if (pressFlag2 == TRUE) {
      if (baudState == TRUE) {

        // baud is currently 460,800 make 115,200
        Init_Serial_UCA0(BRW115, CTL115);
        strcpy(display_line[DISPLAY_LINE2], baud_115);
        update_string(display_line[DISPLAY_LINE2], DISPLAY_LINE2);
        display_changed = TRUE;
        baudState = RESET_STATE;
      }
    else {
      // baud is currently 115,200 make 460,800
      Init_Serial_UCA0(TRUE, CTL460);
      strcpy(display_line[DISPLAY_LINE2], baud_460);
      update_string(display_line[DISPLAY_LINE2], DISPLAY_LINE2);
      display_changed = TRUE;
      baudState = TRUE;
    }
    pressFlag2 = FALSE;
  }

  if (sendFlag == TRUE) {
    UCA0TXBUF = recieved[RESET_STATE];
    UCA0_index = RESET_STATE;
    UCA0IE |= UCTXIE;
    sendFlag = FALSE;
  }

  if (cmdFlag == TRUE) {

    for (int i = RESET_STATE; i < sizeof(recieved); i++) {
      recieved[i] = USB_Char_Rx[i];
      USB_Char_Rx[i] = '\0';
    }
```

```
      usb_rx_ring_wr = RESET_STATE;
      cmdFlag = FALSE;

      // say recieved on line 1, put command onto line 4
      strcpy(display_line[DISPLAY_LINE0], " RECEIVED ");
      update_string(display_line[DISPLAY_LINE0], DISPLAY_LINE0);
      strcpy(display_line[DISPLAY_LINE3], recieved);
      update_string(display_line[DISPLAY_LINE3], DISPLAY_LINE3);
      display_changed = TRUE;

      Time_Sequence = RESET_STATE;
    }

    if (Time_Sequence > DELAY) {
      // just say youre waiting
      strcpy(display_line[DISPLAY_LINE0], "  WAITING ");
      update_string(display_line[DISPLAY_LINE0], DISPLAY_LINE0);
      display_changed = TRUE;
    }

    Display_Process();
  }
//---------------------------------------------------------------------------
}
```

9.2. ports.c

```
#include "macros.h"
#include  "msp430.h"
#include "functions.h"

//call all port functions
void Init_Ports(void){
  Init_Port1();
  Init_Port2();
  Init_Port4();
  Init_Port5();
  Init_Port6();
  Init_Port3(USE_SMCLK);
  Init_Port3(USE_GPIO);
}

// initialize port 1 and defines its pins
void Init_Port1(void){
  P2DIR = OUTPUT_DIRECTION;
  P2OUT = LOW_OUTPUT;

  P1SEL0 &= ~RED_LED; //Red LED operation
  P1SEL1 &= ~RED_LED; //Red LED operation
  P1DIR |= RED_LED; // Direction = Output

  P1SEL0 |= A1_SEED; // P1_1 operation
  P1SEL1 |= A1_SEED; // P1_1 operation

  P1SEL0 |= V_DETECT_L; // P1_2 operation
  P1SEL1 |= V_DETECT_L; // P1_2 operation

  P1SEL0 |= V_DETECT_R; // P1_3 operation
  P1SEL1 |= V_DETECT_R; // P1_3 operation

  P1SEL0 |= A4_SEED; // P1_4 operation
  P1SEL1 |= A4_SEED; // P1_4 operation

  P1SEL0 |= V_THUMB; // P1_5 operation
  P1SEL1 |= V_THUMB; // P1_5 operation

  P1SEL1 &= ~UCA0RXD; // P1_6 operation
  P1SEL0 |= UCA0RXD; // P1_6 operation

  P1SEL1 &= ~UCA0TXD; // P1_7 operation
  P1SEL0 |= UCA0TXD; // P1_7 operation
}
//initialize port 2 and defines its pins
void Init_Port2(void){

  P2DIR = OUTPUT_DIRECTION; //set P2 direction to output
  P2OUT = LOW_OUTPUT; //P2 set Low

  P2SEL0 &= ~P2_0; // P2_0 GPIO operation
  P2SEL1 &= ~P2_0; // P2_0 GPIO operation
  P2DIR &= ~P2_0; // Direction = input

  P2SEL0 &= ~P2_1; // P2_1 GPIO operation
  P2SEL1 &= ~P2_1; // P2_1 GPIO operation
  P2DIR &= ~P2_1; // Direction = input

  P2SEL0 &= ~P2_2; // P2_2 GPIO operation
  P2SEL1 &= ~P2_2; // P2_2 GPIO operation
  P2DIR &= ~P2_2; // Direction = input

  P2SEL0 &= ~SW2; // SW2 Operation
  P2SEL1 &= ~SW2; // SW2 Operation
  P2DIR &= ~SW2; // Direction = input
  P2OUT |= SW2; // Configure pullup resistor
  P2REN |= SW2; // Enable pullup resistor
  P2IE |= SW2;

  P2SEL0 &= ~P2_4; // P2_4 GPIO operation
  P2SEL1 &= ~P2_4; // P2_4 GPIO operation
  P2DIR &= ~P2_4; // Direction = input

  P2SEL0 &= ~P2_5; // P2_5 GPIO operation
  P2SEL1 &= ~P2_5; // P2_5 GPIO operation
  P2DIR &= ~P2_5; // Direction = input
```

```
  P2SEL0 &= ~P2_4; // P2_4 GPIO operation
  P2SEL1 &= ~P2_4; // P2_4 GPIO operation
  P2DIR &= ~P2_4; // Direction = input

  P2SEL0 &= ~P2_5; // P2_5 GPIO operation
  P2SEL1 &= ~P2_5; // P2_5 GPIO operation
  P2DIR &= ~P2_5; // Direction = input

  P2SEL0 &= ~LFXOUT; // LFXOUT Clock operation
  P2SEL1 |= LFXOUT; // LFXOUT Clock operation

  P2SEL0 &= ~LFXIN; // LFXIN Clock operation
  P2SEL1 |= LFXIN; // LFXIN Clock operation
}
//initialize Port 3 and defines pins
void Init_Port3(int arg){
  P3DIR = OUTPUT_DIRECTION; //Set P3 direction to output
  P3OUT = LOW_OUTPUT; //P3 set to low

  P3SEL0 &= ~TEST_PROBE;
  P3SEL1 &= ~TEST_PROBE;
  P3DIR &= ~TEST_PROBE;

  P3SEL0 |= CHECK_BAT;
  P3SEL1 |= CHECK_BAT;

  P3SEL0 &= ~OA2N;
  P3SEL1 &= ~OA2N;
  P3DIR &= ~OA2N;

  P3SEL0 &= ~OA2P;
  P3SEL1 &= ~OA2P;
  P3DIR &= ~OA2P;

  //P3SEL0 &= ~SMCLK_OUT;
  //P3SEL1 &= ~SMCLK_OUT;
  //P3DIR &= ~SMCLK_OUT;

  if(arg == USE_SMCLK){
   P3SEL0 |= SMCLK_OUT;
   P3SEL1 &= ~SMCLK_OUT;
   //P3DIR |= SMCLK_OUT;
  }else{
   P3SEL0 &= ~SMCLK_OUT;
   P3SEL1 &= ~SMCLK_OUT;
   //P3DIR |= SMCLK_OUT;
  }

  P3SEL0 &= ~IR_LED;
  P3SEL1 &= ~IR_LED;
  P3DIR |= IR_LED;
//      P3OUT |= IR_LED;

  P3SEL0 &= ~IOT_LINK;
  P3SEL1 &= ~IOT_LINK;
  P3DIR &= ~IOT_LINK;

  P3SEL0 &= ~P3_7;
  P3SEL1 &= ~P3_7;
}
//Initialize Port 4 and defines pins
void Init_Port4(void){

  P4DIR = OUTPUT_DIRECTION;
  P4OUT = LOW_OUTPUT;

  P4SEL0 &= ~RESET_LCD; // RESET_LCD GPIO operation
  P4SEL1 &= ~RESET_LCD; // RESET_LCD GPIO operation
  P4DIR |= RESET_LCD; // Set RESET_LCD direction to output
  P4OUT |= RESET_LCD; // Set RESET_LCD Off [High]

  P4SEL0 &= ~SW1; // SW1 GPIO operation
  P4SEL1 &= ~SW1; // SW1 GPIO operation
  P4DIR &= ~SW1; // Direction = input
  P4OUT |= SW1; // Configure pullup resistor
  P4REN |= SW1; // Enable pullup resistor
  //P4IE |= SW1;

  P4SEL0 |= UCA1TXD; // USCI_A1 UART operation
  P4SEL1 &= ~UCA1TXD; // USCI_A1 UART operation
```

```
  P4SEL0 |= UCA1RXD; // USCI_A1 UART operation
  P4SEL1 &= ~UCA1RXD; // USCI_A1 UART operation

  P4SEL0 &= ~UCB1_CS_LCD; // UCB1_CS_LCD GPIO operation
  P4SEL1 &= ~UCB1_CS_LCD; // UCB1_CS_LCD GPIO operation
  P4DIR |= UCB1_CS_LCD; // Set SPI_CS_LCD direction to output
  P4OUT |= UCB1_CS_LCD; // Set SPI_CS_LCD Off [High]

  P4SEL0 |= UCB1CLK; // UCB1CLK SPI BUS operation
  P4SEL1 &= ~UCB1CLK; // UCB1CLK SPI BUS operation

  P4SEL0 |= UCB1SIMO; // UCB1SIMO SPI BUS operation
  P4SEL1 &= ~UCB1SIMO; // UCB1SIMO SPI BUS operation
  P4SEL0 |= UCB1SOMI; // UCB1SOMI SPI BUS operation
  P4SEL1 &= ~UCB1SOMI; // UCB1SOMI SPI BUS operation

}
//Initialize Port 5 and defines pins
void Init_Port5(void){

  P5DIR = OUTPUT_DIRECTION;
  P5OUT = LOW_OUTPUT;

  P5SEL0 &= ~IOT_RESET;
  P5SEL1 &= ~IOT_RESET;
  P5DIR &= ~IOT_RESET;

  P5SEL0 |= V_BAT;
  P5SEL1 |= V_BAT;

  P5SEL0 &= ~IOT_PROG_SEL;
  P5SEL1 &= ~IOT_PROG_SEL;
  P5DIR &= ~IOT_PROG_SEL;

  P5SEL0 |= V_3_3;
  P5SEL1 |= V_3_3;

  P5SEL0 &= ~IOT_PROG_MODE;
  P5SEL1 &= ~IOT_PROG_MODE;
  P5DIR &= ~IOT_PROG_MODE;
}
//Initialize Port 6 and defines pins
void Init_Port6(void){

  P6DIR = OUTPUT_DIRECTION;
  P6OUT = LOW_OUTPUT;

  P6SEL0 |= R_FORWARD;
  P6SEL1 &= ~R_FORWARD;
  P6DIR |= R_FORWARD;
  //P6OUT |= R_FORWARD; //on
  //P6OUT &= ~R_FORWARD; //OFF

  P6SEL0 |= L_FORWARD;
  P6SEL1 &= ~L_FORWARD;
  P6DIR |= L_FORWARD;
  //P6OUT |= L_FORWARD; //ON
  //P6OUT &= ~L_FORWARD; //OFF

  P6SEL0 |= R_REVERSE;
  P6SEL1 &= ~R_REVERSE;
  P6DIR |= R_REVERSE;
 // P6OUT |= R_REVERSE; //ON
  //P6OUT &= ~R_REVERSE; //OFF

  P6SEL0 |= L_REVERSE;
  P6SEL1 &= ~L_REVERSE;
  P6DIR |= L_REVERSE;
  //P6OUT |= L_REVERSE; //ON
  //P6OUT &= ~L_REVERSE; //OFF

  P6SEL0 &= ~LCD_BACKLITE;
  P6SEL1 &= ~LCD_BACKLITE;
  //P6OUT |= LCD_BACKLITE; //backlight on
  //P6DIR |= LCD_BACKLITE;
  //P6OUT &= ~LCD_BACKLITE; //backlight off
  //P6DIR &= ~LCD_BACKLITE;

  P6SEL0 &= ~P6_5;
  P6SEL1 &= ~P6_5;
  P6DIR &= ~P6_5;

  P6SEL0 &= ~GRN_LED;
  P6SEL1 &= ~GRN_LED;
  P6DIR |= GRN_LED;
}
```

9.3. timers.c

```
#include  "functions.h"
#include  "msp430.h"
#include <string.h>
#include "macros.h"


// Timer B0 initialization sets up both B0_0, B0_1-B0_2 and overflow
void Init_Timer_B0(void) {
  TB0CTL = TBSSEL__SMCLK; // SMCLK source
  TB0CTL |= TBCLR; // Resets TB0R, clock divider, count direction
  TB0CTL |= MC__CONTINOUS; // Continuous up
  TB0CTL |= ID__2; // Divide clock by 2
  TB0EX0 = TBIDEX__8; // Divide clock by an additional 8
  TB0CCR0 = TB0CCR0_INTERVAL; // CCR0
  TB0CCTL0 |= CCIE; // CCR0 enable interrupt
  TB0CCR1 = TB0CCR1_INTERVAL; // CCR1
  TB0CCTL1 &= ~CCIE; // CCR1 enable interrupt
  // TB0CCR2 = TB0CCR2_INTERVAL; // CCR2
  // TB0CCTL2 |= CCIE; // CCR2 enable interrupt
  TB0CTL &= ~TBIE; // Disable Overflow Interrupt
  TB0CTL &= ~TBIFG; // Clear Overflow Interrupt flag
}

void Init_Timers(void){
  Init_Timer_B0();
  Init_Timer_B3();
}
```

```
//------------------------------------------------------------------------------
void Init_Timer_B3(void) {
//------------------------------------------------------------------------------
// SMCLK source, up count mode, PWM Right Side
// TB3.1 P6.0 R_FORWARD
// TB3.2 P6.1 L_FORWARD
// TB3.3 P6.2 R_REVERSE
// TB3.4 P6.3 L_REVERSE
//------------------------------------------------------------------------------
    TB3CTL = TBSSEL__SMCLK;             // SMCLK
    TB3CTL |= MC__UP;                   // Up Mode
    TB3CTL |= TBCLR;                    // Clear TAR

    TB3CCR0 = WHEEL_PERIOD;             // PWM Period

    TB3CCTL1 = OUTMOD_7;                // CCR1 reset/set
    RIGHT_FORWARD_SPEED = WHEEL_OFF;    // P6.0 Right Forward PWM duty cycle

    TB3CCTL2 = OUTMOD_7;                // CCR2 reset/set
    LEFT_FORWARD_SPEED = WHEEL_OFF;     // P6.1 Left Forward PWM duty cycle

    TB3CCTL3 = OUTMOD_7;                // CCR3 reset/set
    RIGHT_REVERSE_SPEED = WHEEL_OFF;    // P6.2 Right Reverse PWM duty cycles

    TB3CCTL4 = OUTMOD_7;                // CCR4 reset/set
    LEFT_REVERSE_SPEED = WHEEL_OFF;     // P6.3 Left Reverse PWM duty cycle
//------------------------------------------------------------------------------
}
```

9.4. interrupt_timers.c

```
// Timer B0 ISR ------------------------------------------------------------------
#pragma vector = TIMER0_B0_VECTOR
__interrupt void Timer0_B0_ISR(void) {

    // Increment timers
    Time_Sequence++;
    displayTimerCount++;
    cmdTimer++;

    if ((Time_Sequence > 20) && !(P5OUT & IOT_RESET)) { P5OUT |= IOT_RESET; }

    // Refresh the LCD every 200 ms
    if (displayTimerCount == REFRESHRATE) {
        update_display = TRUE;
        displayTimerCount = RESET_STATE;
    }


    TB0CCR0 += TB0CCR0_INTERVAL;        // Add offset to TBCCR0
}
//-------------------------------------------------------------------------------
```

```
// Timer B1 ISR ------------------------------------------------------------------
#pragma vector = TIMER0_B1_VECTOR
__interrupt void TIMER0_B1_ISR(void) {
    // Timer B0 1-2, Overflow Interrupt Vector (TBIV) handler

    switch (__even_in_range(TB0IV, TIMERRANGE)) {
        case FALSE: break;      // No interrupt
        case CCR1: // CCR1

            // Increment debounce timers
            if (debouncing_SW1) { SW1_DebounceTimer++; }

            if (debouncing_SW2) { SW2_DebounceTimer++; }

            // wake the switches back up
            if (SW1_DebounceTimer > DEBOUNCEDONE) {
                SW1_DebounceTimer = FALSE;
                debouncing_SW1 = FALSE;
                P4IE |= SW1;
            }

            if (SW2_DebounceTimer > DEBOUNCEDONE) {
                SW2_DebounceTimer = FALSE;
                debouncing_SW2 = FALSE;
                P2IE |= SW2;
            }

            if ((!debouncing_SW1) && (!debouncing_SW2)) { TB0CCTL1 &= ~CCIE; }

            TB0CCR1 += TB0CCR1_INTERVAL;     // Add offset to TBCCR1
            break;
        case CCR2: // CCR2
            TB0CCR2 += TB0CCR2_INTERVAL;     // Add Offset to TBCCR2
            break;
        case TIMERRANGE: // Overflow
            break;
        default: break;
    }
}
//-------------------------------------------------------------------------------
```

9.5. ADC.c

```
//Macros
#include  "functions.h"
#include  "msp430.h"
#include  "macros.h"


void Init_ADC(void){
//------------------------------------------------------------------------------
// V_DETECT_L           (0x04) // Pin 2 A2
// V_DETECT_R           (0x08) // Pin 3 A3
// V_THUMB              (0x20) // Pin 5 A5
//------------------------------------------------------------------------------
// ADCCTL0 Register
  ADCCTL0 = RESET_STATE;              // Reset
  ADCCTL0 |= ADCSHT_2;                // 16 ADC clocks
  ADCCTL0 |= ADCMSC;                  // MSC
  ADCCTL0 |= ADCON;                   // ADC ON

// ADCCTL1 Register
  ADCCTL2 = RESET_STATE;              // Reset
  ADCCTL1 |= ADCSHS_0;                // 00b = ADCSC bit
  ADCCTL1 |= ADCSHP;                  // ADC sample-and-hold SAMPCON signal from sampling timer.
  ADCCTL1 &= ~ADCISSH;                // ADC invert signal sample-and-hold.
  ADCCTL1 |= ADCDIV_0;                // ADC clock divider - 000b = Divide by 1
  ADCCTL1 |= ADCSSEL_0;               // ADC clock MODCLK
  ADCCTL1 |= ADCCONSEQ_0;             // ADC conversion sequence 00b = Single-channel single-conversion
  // ADCCTL1 & ADCBUSY                // identifies a conversion is in process

// ADCCTL2 Register
  ADCCTL2 = RESET_STATE;              // Reset
  ADCCTL2 |= ADCPDIV0;                // ADC pre-divider 00b = Pre-divide by 1
  ADCCTL2 |= ADCRES_2;                // ADC resolution 10b = 12 bit (14 clock cycle conversion time)
  ADCCTL2 &= ~ADCDF;                  // ADC data read-back format 0b = Binary unsigned.
  ADCCTL2 &= ~ADCSR;                  // ADC sampling rate 0b = ADC buffer supports up to 200 ksps

// ADCMCTL0 Register
  ADCMCTL0 |= ADCSREF_0;              // VREF - 000b = {VR+ = AVCC and VR- = AVSS }
  ADCMCTL0 |= ADCINCH_2;              // V_DETECT_L (0x04) Pin 2 A2
  ADCMCTL0 |= ADCINCH_3;
  ADCMCTL0 |= ADCINCH_5;

  ADCIE |= ADCIE0;                    // Enable ADC conv complete interrupt
  ADCCTL0 |= ADCENC;                  // ADC enable conversion.
  ADCCTL0 |= ADCSC;                   // ADC start conversion.
}
```

9.6. interrupt_ADC.c

```
//Macros
#include  "functions.h"
#include  "msp430.h"
#include  "macros.h"
#include  <string.h>

//Functions
void HEXtoBCD(int hex_value);
void adc_line(int line);

//Globals
extern volatile unsigned int ADC_Channel = RESET_STATE;
extern volatile unsigned int ADC_Left_Detect = RESET_STATE;
extern volatile unsigned int ADC_Right_Detect = RESET_STATE;
extern volatile unsigned int ADC_Thumb = RESET_STATE;

//configure display
#pragma vector=ADC_VECTOR
__interrupt void ADC_ISR(void){
  switch(__even_in_range(ADCIV,ADCIV_ADCIFG)){
  case ADCIV_NONE:
    break;
  case ADCIV_ADCOVIFG:     // When a conversion result is written to the ADCMEM0
    break;                 // before its previous conversion result was read.
  case ADCIV_ADCTOVIFG:    // ADC conversion-time overflow
    break;
  case ADCIV_ADCHIIFG:     // Window comparator interrupt flags
    break;
  case ADCIV_ADCLOIFG:     // Window comparator interrupt flag
    break;
  case ADCIV_ADCINIFG:     // Window comparator interrupt flag
    break;
  case ADCIV_ADCIFG:       // ADCMEM0 memory register with the conversion result
    ADCCTL0 &= ~ADCENC;    // Disable ENC bit
    switch (ADC_Channel++){
    case CHANNEL_A2:                                  // Channel A2 Interrupt
      ADCMCTL0 &= ~ADCINCH_2;                         // Disable Last channel A2
      ADCMCTL0 |= ADCINCH_3;                          // Enable Next channel A3
      ADC_Left_Detect = ADCMEM0;                      // Move result into Global
      ADC_Left_Detect = ADC_Left_Detect >> SHIFT_2;   // Divide the result by 4
      HEXtoBCD(ADC_Left_Detect);                      // Convert result to String
      adc_line(DISP_LINE3);                           // Place String in Display
      break;
    case CHANNEL_A3:
      ADCMCTL0 &= ~ADCINCH_3;                         // Disable Last channel A3
      ADCMCTL0 |= ADCINCH_5;                          // Enable Next channel A5
      ADC_Right_Detect = ADCMEM0;                     // Move result into Global
      ADC_Right_Detect = ADC_Right_Detect >> SHIFT_2; // Divide the result by 4
      HEXtoBCD(ADC_Right_Detect);                     // Convert result to String
      adc_line(DISP_LINE2);                           // Place String in Display
      break;
    case CHANNEL_A5:
      ADCMCTL0 &= ~ADCINCH_5;                         // Disable Last channel A5
      ADCMCTL0 |= ADCINCH_2;                          // Enable channel A2
      ADC_Thumb = ADCMEM0;                            // Move result into Global
      ADC_Thumb = ADC_Thumb >> SHIFT_2;               // Divide the result by 4
      HEXtoBCD(ADC_Thumb);                            // Convert result to String
```

```
      HEXtoBCD(ADC_Left_Detect);                          // Convert result to String
      adc_line(DISP_LINE3);                               // Place String in Display
      break;
    case CHANNEL_A3:
      ADCMCTL0 &= ~ADCINCH_3;                             // Disable Last channel A3
      ADCMCTL0 |= ADCINCH_5;                              // Enable Next channel A5
      ADC_Right_Detect = ADCMEM0;                         // Move result into Global
      ADC_Right_Detect = ADC_Right_Detect >> SHIFT_2;     // Divide the result by 4
      HEXtoBCD(ADC_Right_Detect);                         // Convert result to String
      adc_line(DISP_LINE2);                               // Place String in Display
      break;
    case CHANNEL_A5:
      ADCMCTL0 &= ~ADCINCH_5;                             // Disable Last channel A5
      ADCMCTL0 |= ADCINCH_2;                              // Enable channel A2
      ADC_Thumb = ADCMEM0;                                // Move result into Global
      ADC_Thumb = ADC_Thumb >> SHIFT_2;                   // Divide the result by 4
      HEXtoBCD(ADC_Thumb);                                // Convert result to String
      adc_line(DISP_LINE1);                               // Place String in Display
      ADC_Channel = RESET_STATE;                          // Set state back to beginning channel
      break;
    default:
      break;
    }
    ADCCTL0 |= ADCENC;                  // Enable Conversions
    ADCCTL0 |= ADCSC;                   // Start next sample
    break;
  default:
    break;
  }
}


//-------------------------------------------------------------------------------
// Hex to BCD Conversion
// Convert a Hex number to a BCD for display on an LCD or monitor
//
//-------------------------------------------------------------------------------
void HEXtoBCD(int hex_value){
  int value = RESET_STATE;
  adc_char[INDEX0] = '0';
  while (hex_value > TH_LIMIT){
    hex_value = hex_value - THOUS;
    value = value + ALWAYS;
    adc_char[INDEX0] = ASCII_OFFSET + value;
  }
  value = RESET_STATE;
  while (hex_value > H_LIMIT){
    hex_value = hex_value - HUNDRED;
    value = value + ALWAYS;
    adc_char[INDEX1] = ASCII_OFFSET + value;
  }
  value = RESET_STATE;
  while (hex_value > TEN_LIMIT){
    hex_value = hex_value - TEN;
    value = value + ALWAYS;
    adc_char[INDEX2] = ASCII_OFFSET + value;
  }
  adc_char[INDEX3] = ASCII_OFFSET + hex_value;
}
//-------------------------------------------------------------------------------
```

```
//-------------------------------------------------------------------------
// Display converted Hex->BCD values to LCD Display
//
//-------------------------------------------------------------------------
void adc_line(int line){
  //ADC_Thumb
  if(line == DISP_LINE1){
    adc_char[INDEX9] = ' ';
    adc_char[INDEX7] = ' ';
    strcpy(display_line[MID_LINEA], adc_char);
    update_string(display_line[MID_LINEA], MID_LINEA);
    display_changed = TRUE;

  }
  //ADC_Right_Detect
  if(line == DISP_LINE2){
    adc_char[INDEX7] = 'R';
    strcpy(display_line[MID_LINEB], adc_char);
    update_string(display_line[MID_LINEB], MID_LINEB);
    display_changed = TRUE;

  }
  //ADC_Left_Detect
  if(line == DISP_LINE3){
    adc_char[INDEX7] = 'L';
    strcpy(display_line[BOT_LINE], adc_char);
    update_string(display_line[BOT_LINE], BOT_LINE);
    display_changed = TRUE;

  }
    adc_char[INDEX0] = '0';
    adc_char[INDEX1] = '0';
    adc_char[INDEX2] = '0';
    adc_char[INDEX3] = '0';
    adc_char[INDEX4] = ' ';
    adc_char[INDEX5] = ' ';
    adc_char[INDEX6] = ' ';
    adc_char[INDEX7] = ' ';
    adc_char[INDEX8] = ' ';
    adc_char[INDEX9] = 'W';
    adc_char[INDEX10] = '\0';

}
```

9.7. serial.c

```
//Macros
#include  "functions.h"
#include  "msp430.h"
#include  <string.h>
#include  "macros.h"
#include  "ports.h"

//Globals
extern volatile char USB_Char_Rx[SMALL_RING_SIZE] = {' '};
extern volatile unsigned int usb_rx_ring_wr = RESET_STATE;
extern volatile unsigned int usb_rx_ring_rd = RESET_STATE;
extern volatile int switch_state;


//-------------------------------------------------------------------------------
void Init_Serial_UCA1(int switch_state){
  int i;
  for(i=RESET_STATE; i<SMALL_RING_SIZE; i++){
    USB_Char_Rx[i] = CLEAR;              // USB Rx Buffer
  }
  usb_rx_ring_wr = BEGINNING;
  usb_rx_ring_rd = BEGINNING;

  // Configure UART 1
  //Baud Rate 115200 -- switch_state == 0
  if(switch_state == BAUD_115){
    //BRCLK    Baudrate UCOS16 UCBRx UCFx UCSx
    //8000000  115200     1      4    5   0x55
    UCA1CTLW0 = RESET_STATE;                        // Use word register
    UCA1CTLW0 |= UCSWRST;                  // Set Software reset enable
    UCA1CTLW0 |= UCSSEL__SMCLK;            // Set SMCLK as fBRCLK
    UCA1BRW = BRW4;
    UCA1MCTLW = TLW55;
    UCA1CTLW0 &= ~ UCSWRST;                // Set Software reset enable
    UCA1IE |= UCRXIE;                      // Enable RX interrupt
  }
  //Baud Rate 460800 -- switch_state == 1
  if(switch_state == BAUD_460){
    //BRCLK    Baudrate UCOS16 UCBRx UCFx UCSx
    //8000000  460800     1      1    1   0x4A
    UCA1CTLW0 = RESET_STATE;                        // Use word register
    UCA1CTLW0 |= UCSWRST;                  // Set Software reset enable
    UCA1CTLW0 |= UCSSEL__SMCLK;            // Set SMCLK as fBRCLK
    UCA1BRW = BRW1;
    UCA1MCTLW = TLW4A;
    UCA1CTLW0 &= ~ UCSWRST;                // Set Software reset enable
    UCA1IE |= UCRXIE;                      // Enable RX interrupt
  }
}
```

9.8. Interrupt_Serial.c

```
//-------------------------------------------------------------------------------
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void) {
    unsigned int temp;
    switch(__even_in_range(UCA0IV,RANGE)) {
        case RESET_STATE: break;
        case RECEIVING:
            temp = usb_rx_ring_wr++;
            USB_Char_Rx[temp] = UCA0RXBUF;
            if (USB_Char_Rx[temp] == LF) {
                cmdFlag = TRUE;
            }
            if (usb_rx_ring_wr >= (sizeof(USB_Char_Rx))) {
                usb_rx_ring_wr = BEGINNING;
            }
            break;
        case SENDING:
            UCA0TXBUF = command[UCA0_index++];
            if (command[UCA0_index] == NULLTERM) {
                UCA0IE &= ~UCTXIE;
            }
            break;
            break;
        default: break;
    }
}
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
#pragma vector=USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void) {
    unsigned int temp;
    switch(__even_in_range(UCA1IV,RANGE)) {
        case RESET_STATE: break;
        case RECEIVING:
            temp = usb_rx_ring_wr++;
            USB_Char_Rx[temp] = UCA1RXBUF;
            if (USB_Char_Rx[temp] == LF) {
                cmdFlag = TRUE;
            }
            if (usb_rx_ring_wr >= (sizeof(USB_Char_Rx))) {
                usb_rx_ring_wr = BEGINNING;
            }
            break;
        case SENDING:
            UCA1TXBUF = command[UCA1_index++];
            if (command[UCA1_index] == NULLTERM) {
                UCA1IE &= ~UCTXIE;
            }
            break;
            break;
        default: break;
    }
}
//-------------------------------------------------------------------------------
```

9.9 Hex to BCD Function

```
#include  "functions.h"
#include  "msp430.h"
#include <string.h>
#include "macros.h"

extern char display_line[HEIGHTFOUR][ELEVENTH_CHARACTER];
extern volatile unsigned char display_changed;
extern volatile unsigned char update_display;
extern volatile unsigned int update_display_count;


// globals
char thousands;
char hundreds;
char tens;
char ones;

void HEXtoBCD_RIGHT(unsigned int value_passed){
  thousands = RESET_STATE;
  hundreds = RESET_STATE;
  tens = RESET_STATE;
  ones = RESET_STATE;
  if(value_passed >= thousand){
    value_passed = value_passed - thousand;
    thousands = one;
  }
  while(value_passed >= hundred){
    value_passed = value_passed - hundred;
    hundreds = hundreds + one;
  }
  while(value_passed >= ten){
    value_passed = value_passed - ten;
    tens = tens + one;
  }
  while(value_passed >= one){
    value_passed = value_passed - one;
    ones = ones + one;
  }
  thousands |= offset;
  hundreds |= offset;


  tens |= offset;
  ones |= offset;

    display_line[LINETHREE][LINETHREE]= thousands;
    display_line[LINETHREE][HEIGHTFOUR] = hundreds;
    display_line[LINETHREE][HEIGHTFIVE] = tens;
    display_line[LINETHREE][HEIGHTSIX] = ones;
    update_string(display_line[LINETHREE], LINETHREE);

 }
```

10. Conclusions

Lauren: After taking this class it is clear to me why it is a requirement for CPE majors. This class teaches students how to work through challenging or unforeseen problems, complete work by strict deadlines, and the essentials to embedded systems and electrical/software skills that we will use in our future careers. Each project and homework were a learning process that allowed me to practice the skills we had been learning in previous courses in a hands-on way. A lot went wrong throughout the course of this semester, such as broken components, dealing with monster truck size wheels, and much more, but I was able to work through all those problems with the support of the TA's and find a solution.

Ethan: In this class I learned a lot about the fundamentals of embedded systems. The pragmatic mindset surrounding design under the motto "the electron is king" stuck with me during this class, and I found that even when writing code for other classes where efficiency and storage conservation was of no concern, I was subconsciously making efforts to make my code smaller and faster. While this class helped me master certain hands on electronics skills like soldering and oscilloscope usage, I found that the most valuable take away was the "under the hood" experience I received by building and debugging every aspect of my car piece by piece. Coming into the class I couldn't really tell you what an interrupt was, or what serial was, or what a round robin operating system does, or why you shouldn't use blocking code, and the list goes on. After completing the course and looking back I can tell that my mindset, while still that of an amateur, has shifted into a much more organized, in depth, "this is what's going on" approach to embedded systems.

Evan: 306 had everything you could ask for in a class: soldering, hardware assembly, coding, making things move, etc. Most courses are solely within a classroom setting, and you get in a bit of a routine by going to lectures, doing homework or projects, and taking exams. 306 definitely threw a wrench in the works (in a good way). Being in a lab setting, creating something tangible, and being under the constant stress of falling behind, all taught me valuable lessons and it's something that I'm pretty appreciative of. If there was confusion on my end, or something hadn't worked the way I had intended, I always had my peers, the TAs, and Mr. Carlson to rely on for help. I really am glad all CpE majors have to take this course, it's a class I've learned a lot in, and it's definitely one I'll remember. I wish we could've done Project 10 demos in EBII ;(