Drive the Bits

Racing the Code

Driving the Future

Revision	Description
A	Update for modifications to the vehicle to date.
B	Update for addition of black line detection and software section.
C	Update for black line navigation and serial communication.

Team Members:
Ethan Cornett
Trey Parker
Erik Seuster
Martina Viola
Drew Whitehead

Originator:

Checked: | Released: 9 - 21 - 2023

File Name: Project Write-Up - Group I2

Title:
Drive the Bits
Racing the Code

Date:	Document #	Rev:	Sheet:
12 - 07 - 2023	**0000-0001**	**C**	

Table of Contents

Figures

1. Scope

This document describes the electronic and mechanical components of the robotic car. The car's ultimate purpose is to remotely traverse a line course that is detected and followed by the car. At its core is a Texas Instruments MSP430FR2355 microprocessor running 16-bit RISC architecture from which all peripherals are attached. Communication between the user and the car is done through two avenues: The LCD screen and embedded LEDs relay relevant system and task information to the user while serial communication will be conducted through a Wi-Fi device. The backlit LCD acts as user feedback allowing for a real-time information stream for the user and eliminating the need for additional user interfaces. This helps achieve the goal of completely remote car operation and task completion. To power these peripherals a compact battery pack is attached to the underside of the car. Four AA batteries are stepped down to 5V and are more than enough to power the microprocessor and its associated peripherals. With the long-range mobility of the car as a top priority, the power supply system must be compact and long-lasting. This car's attached battery pack achieves this goal.

Our team has constructed the chassis of our five test vehicles and attached the LCD, wheels, motors, MCU boards, FET boards, and ADC boards with attached IR illuminators and detectors. These are all of the components necessary for the car to trigger motors forward and reverse, and detect the black line course. The five chassis use different front and caster wheel profiles to test the effectiveness of varying wheel diameters and depths on varying surfaces. The car's motors are connected to the installed MCU with a fully implemented H-bridge system allowing for the forward and reverse movement of the car. All test cars have a PWM system to allow for individual control of motor speed. The test cars can detect and orient themselves along a black line along the traversal terrain. This is done using the installed ADC board with an IR illuminator and detectors.

An interrupt-controlled serial communication protocol is implemented on each of the test cars. This is accomplished using two USCI interrupts in the team's software, set up to transmit and receive data from two sources simultaneously. The interrupts allow for on-the-fly adjustment of communication baud rate allowing for flexibility of use cases. The LCD is currently being used to display the status of transmissions between the two communication interrupts and display the contents of these transmissions in real time.

As of 07, Dec. 2023, The systems allowing wireless communication with the car's IOT module have been fully implemented. All of our test cars now can receive commands through their attached IOT modules. These commands can be processed in FRAM and acted upon. The startup of the cars is fully wireless and users can connect multiple wireless devices to the car to send commands using only information displayed on the LCD. The full functionality of the car that our group advertised as our final goal is complete. Users of the car can easily control its movement from a distance. A user can also remotely signal the car to follow a black line using calibrated values. These five test vehicles are now fully complete and function as described below.

The following sections of this document outline the design and assembly of the microprocessor, LCD, power supply systems, chassis, motors, FET board, PWM system, ADC board, communication interrupts, and IOT modules as of 07, Dec. 2023.

2. Abbreviations

DAC	Digital to Analog Converter
FET	Field Effect Transistor
LCD	Liquid-Crystal Display
LED	Light Emitting Diode
MCU	Microcontroller Unit
RISC	Reduced Instruction Set Computer
UI	User Interface
USB	Universal Serial Bus
SEPIC	Single-Ended Primary-Inductor Converter
PFET	Positive Channel Field-Effect Transistor
NFET	Negative Channel Field-Effect Transistor
DC	Direct Current
PCB	Printed Circuit Board
GP I/O	General Purpose Input/Output
H-Bridge	Electronic Polarity Switching Circuit
ADC	Analog-to-Digital Converter
IOT	Internet of Things
IR	Infrared
PWM	Pulse Width Modulation
FRAM	Ferroelectric Random Access Memory
TX	Transmit
RX	Receive
USCI	Universal Serial Communication Interface
eUCSI	Enhanced Universal Serial Communication Interface
MSB	Most Significant Bit
LSB	Least Significant Bit
UART	Universal Asynchronous Receiver-Transmitter
ISR	Interrupt Service Routine

3. Overview

The vehicle is connected to a power supply via a power board. The power board is connected to the microcontroller, which allows the system to run and demonstrate outputs. Upon connection to the power source, the LCD displays a message to the user, and green and red LEDs are illuminated. A user input prompts a change in state and results in a change in system output. The microcontroller is connected to the DAC board and to the motors. The user may send a message to the car from a TCP client application or IOS app. The commands sent by the user may initiate movement forward, backward, left, or right for a specified duration. The user may also issue commands to place the vehicle in autonomous mode. While in autonomous mode, the vehicle travels to the black line course, intercepts the black line, aligns itself over the black line, and travels along the path it forms. The car will continue to travel along the course until the user issues a command for the vehicle to exit the course.

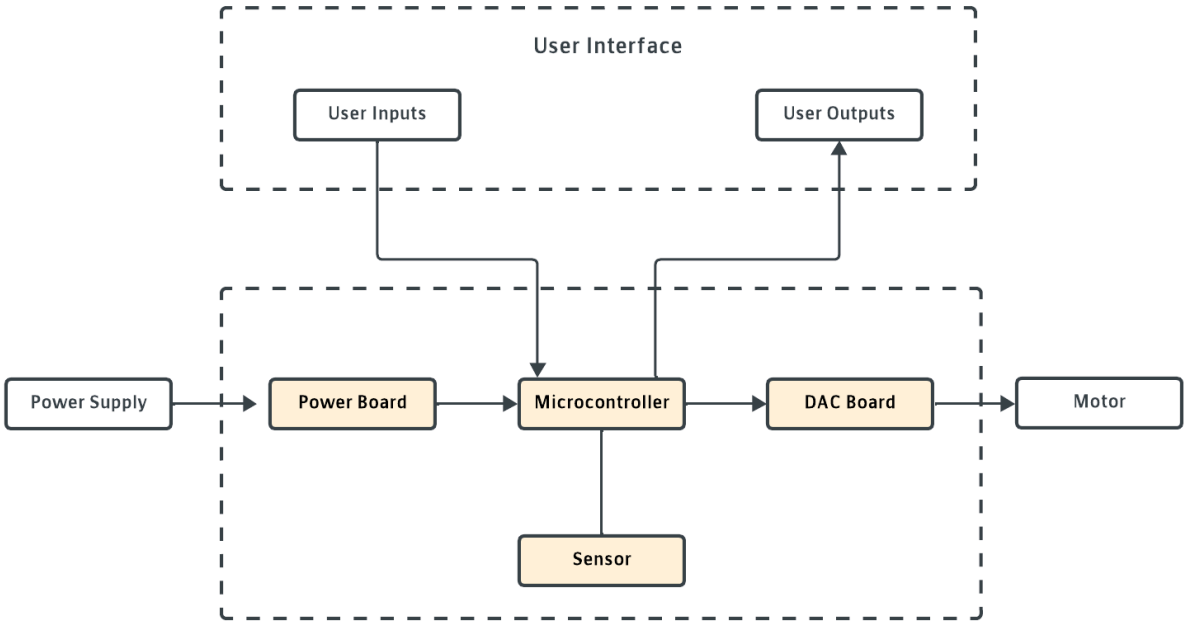

Figure 1: Block Diagram

3.1. Microcontroller

The microcontroller includes the system's processor, memory, clock system, and reset controller. The processor processes system inputs and computes system outputs, using and storing data to and from memory. The clock system synchronizes the operations of the microcontroller. The reset controller resets the processor when insufficient power is supplied to the system.

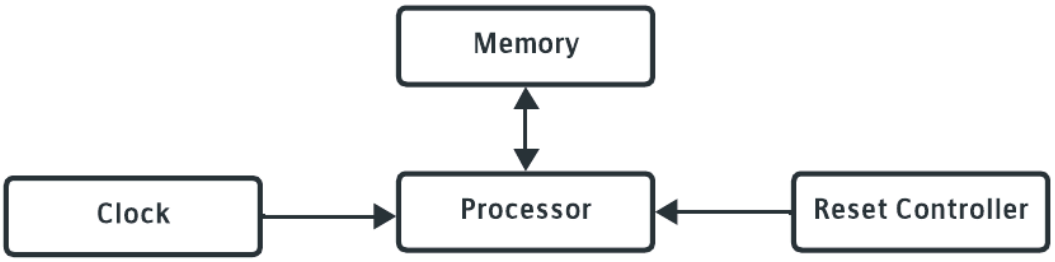

Figure 2: Microcontroller

3.2. Power Supply

Power is supplied to the vehicle using batteries. The batteries are inserted into a battery pack installed on the undercarriage of the remote vehicle. The battery pack is connected to a power board that will supply power to its connected components.

Figure 3: Power Supply

3.3. User Interface Block

The user interface of a system refers to the ways in which the user interacts with the system. The vehicle's system interfaces with the user with switches, the LCD and the LEDs. The initial LCD message, and illumination of the LCD backlight and LEDs indicate to the user that the vehicle is connected to the power supply. Upon connection to a wifi network, the SSID and IP address are displayed on the LCD. The user may send a message to the car from a TCP client application or IOS app. The car will respond with an action if the message sent matches a programmed command. The message received is displayed on the LCD. The user may use switch 2 to calibrate the vehicle for black line detection. The message will display messages to indicate the vehicle is in calibration mode, and direct the user to calibrate for white and black surfaces.

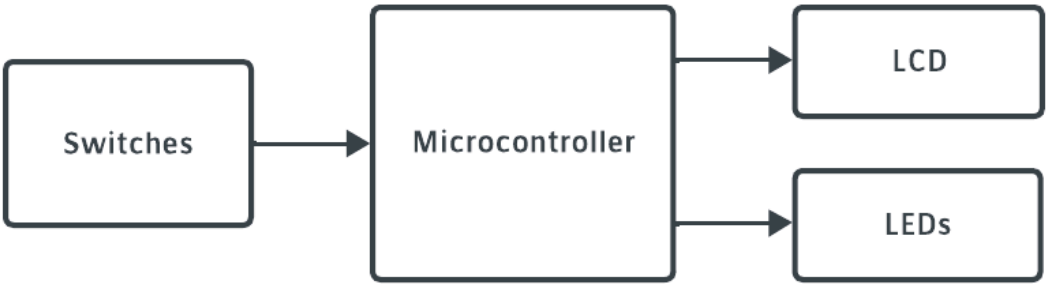

Figure 4: User Interface Block

IOT Navigation LCD Messages		
LCD Line	**Messages**	**Meaning**
Line 1	Arrived 0X	The vehicle has arrived at pad X.
	BL Start	Autonomous Mode – the vehicle has been set to autonomous mode, and the vehicle is traveling to the black line navigation course.
	Intercept	Autonomous Mode – the vehicle has detected the black line, and stopped.
	BL Turn	Autonomous Mode – the vehicle is turning to align itself over the black line.
	BL Travel	Autonomous Mode – the vehicle is traveling along the black line.
	BL Circle	Autonomous Mode – the vehicle is circling in the black line course.
	BL Exit	Autonomous Mode – the vehicle is exiting the circle in the black line course.
	BL Stop	Autonomous Mode – the vehicle has exited the circle and successfully completed the black line navigation course.
Line 2	<First Name>	The first name of the user is displayed on the third line of the LCD.
Line 3	<Last Name>	The last name of the user is displayed on the third line of the LCD.
Line 4	<Command> <Timer>	The last received command and a timer in seconds are displayed on the last line of the LCD.

Figure 5: LCD Messages

Command Character	Action
‘F’	Vehicle moves forward for a specified duration.
‘B’	Vehicle moves backward for a specified duration.
‘L’	Vehicle moves left for a specified duration.

‘R’	Vehicle moves right for a specified duration.
‘?’	Vehicle is set to autonomous mode for black line detection and navigation.

Figure 6: Commands

3.4. Sensor

The emitter is activated and analog-to-digital conversions are initiated by the user at the press of Switch 1. The emitter, the IR LED, emits infrared light which is picked up by the detectors. The values from the detectors indicate whether the vehicle has reached a black line. If the value from the left or right corresponds to that of a black line, the vehicle halts motion and turns to align both detectors over the black line. Following alignment, the vehicle follows the black line to the circle in the black line course and circles. When the user issues a command, the vehicle stops, turns, and travels away from the circle.

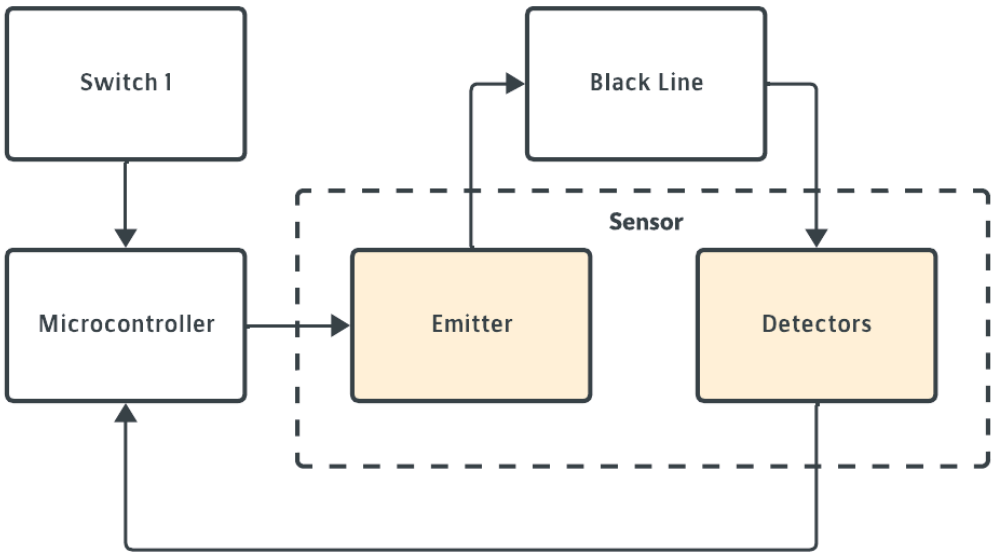

Figure 7: Sensor

3.5. Serial Communications

The car can send and receive messages to and from a PC or IOT module. Upon startup and successful connection to wifi, the vehicle automatically transmits commands to the IOT module to configure settings. The user may send a message to the car from a TCP client application or IOS app. The car will respond with an action if the message sent matches a programmed command. The message received is displayed on the LCD.

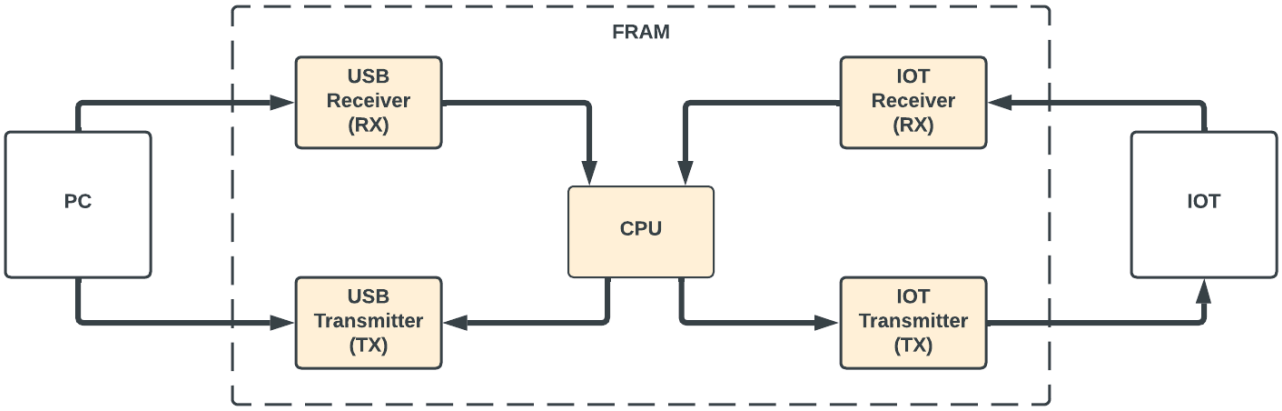

Figure 8: Serial Communication

4. Hardware

This section will describe the hardware components used in the car thus far in our project.

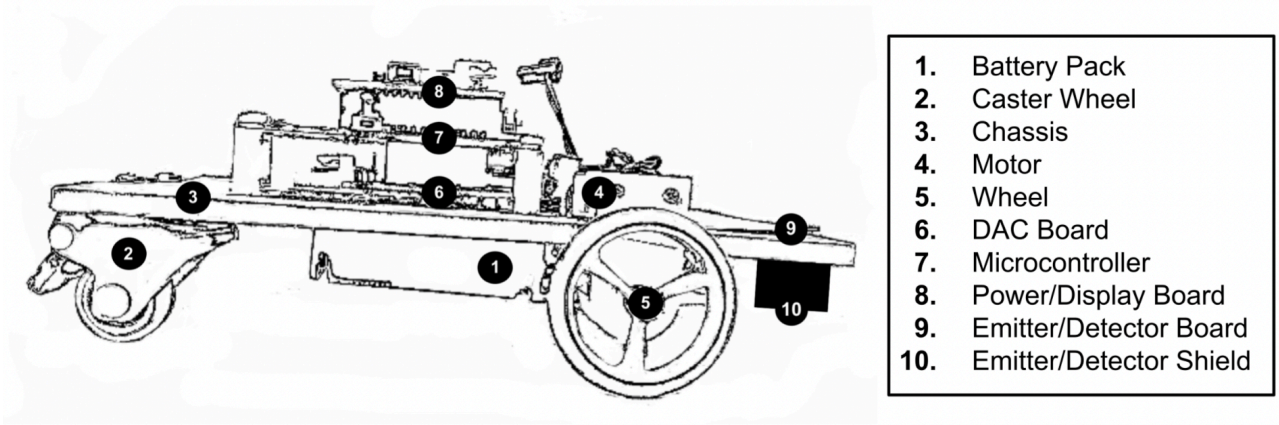

Figure 9: Hardware Overview

4.1. Subsystem 1: Microcontroller

Our car design uses the Texas Instruments MSP-EXP430FR2355 LaunchPad Development Kit. This development kit offers 6 ports (each with 6+ pins) that are programable using the select bits; these select bits are used to select the functionality of each pin. This board is the main brain of our machine and is where all of our code is loaded and executed. The microcontroller also communicates with the other hardware components via serial communication, allowing us to control the car's movement and other functions. Communication between our computers and the microcontroller is USB-A to USB Micro B (microcontroller connection).

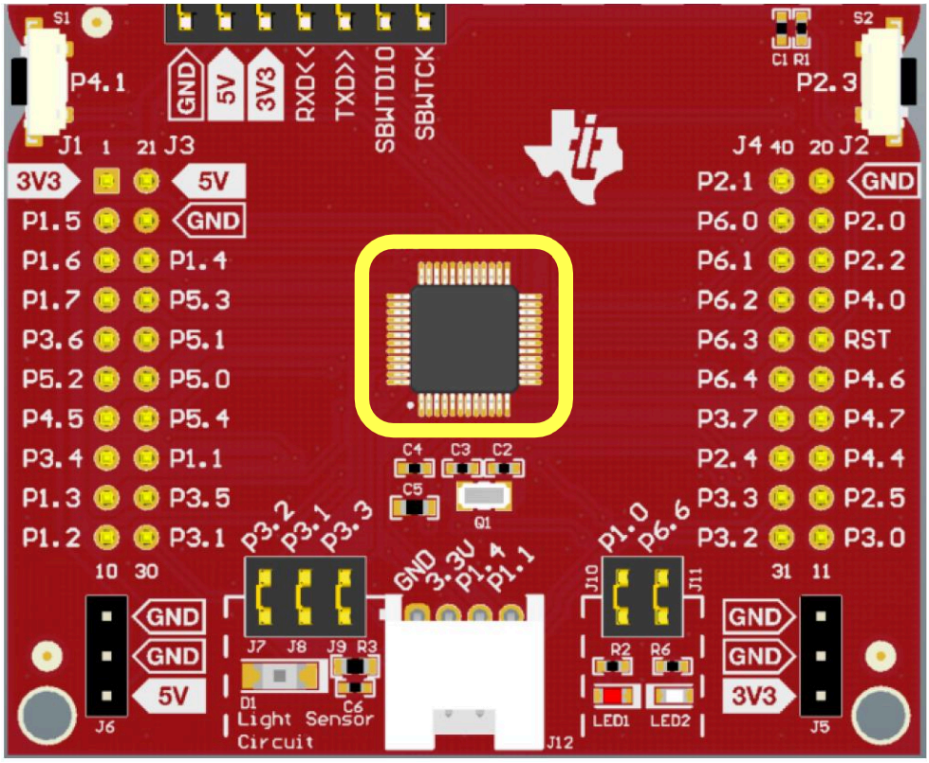

Figure 10: MSP-EXP430FR2355 LaunchPad Development Kit

4.2. Subsystem 2: Power Module

The power module on the board efficiently regulates a wide input power range (2.3V-16V) from 4 "AA" batteries, ensuring reliable and stable output power (2.4V-3.8V). Key features include power regulation for enhanced system reliability despite input power variations and a preventative diode for reverse voltage protection. The module achieves approximately 87% efficiency through synchronous rectification, optimizing power usage and prolonging battery life.

4.3. Subsystem 3: Display Module

The display module system hardware includes an LCD screen and a blacklight to improve visibility while the car is moving around. Resting on the top side of our car the display shows 3 lines of writing with 10 characters on each line.

4.4. Subsystem 4: FET Module

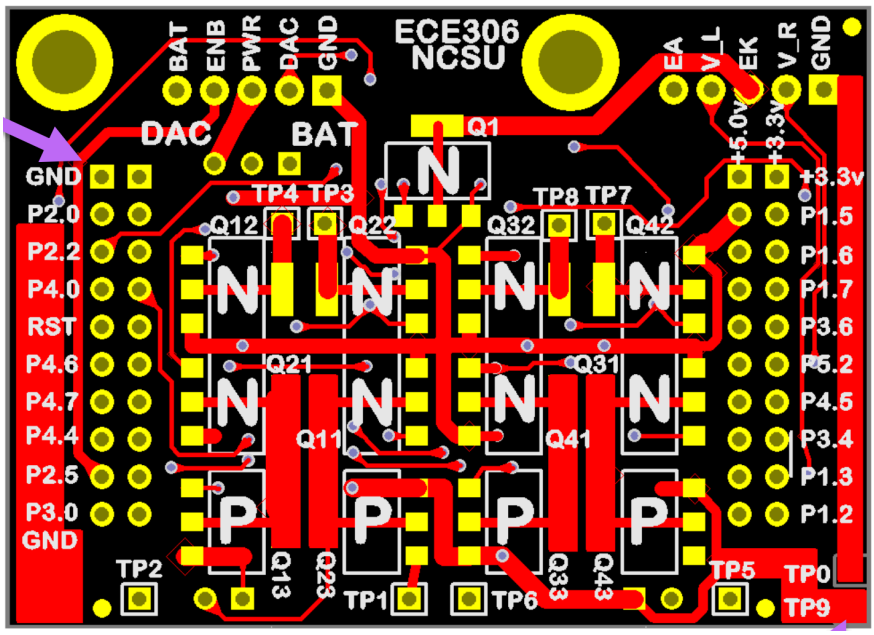

Figure 11: FET Module

The FET Module is the subsystem responsible for activating and providing power to the two motors on the car. The module consists of 8 NFETs and 4 PFETS, together these FETs create a switching circuit that provides voltage and current to the motors to turn in forward or reverse. This circuit is named “The H Bridge” and is the organization of the FETs on the PCB which you can see in Figure 11.

4.5. Supporting Hardware

Battery Pack: The 4 “AA” battery pack on our car is responsible for all power while remote operation of the car is underway.

Motors: The 2 motors are mounted on the car chassis to enable movement of the car defined by the MCU.

Chassis: The chassis for our car is a wooden board that has cutouts on it for our motors, batteries, and board to connect to.

Caster Wheel: The caster wheel acts as the back wheel of our car. It swivels to follow the motion that the front wheels direct.

Serial Communication Pins: The top of our display board has pins that let us connect directly to the FRAM board.

4.6. Power System

The power module on our board is responsible for handling all power input into our board and distributing it to the correct locations so that all systems are functioning correctly. A main advantage that is built into our power board is the ability to regulate output power despite variation of input power. The power regulation feature is a key component of our overall system because it provides increased reliability for our system even if the input power is unreliable; this lets our machine function how it is intended without worrying if the input power is reliable or not. The input power range, Vin, is 2.3V - 16V, a very wide range for a machine of this size. Our output power range is even larger at 2.4V - 38V due to the power system onboard. The input power into this module is from 4 “AA” batteries which each provide ~6V.

The power module also hosts other preventative systems to ensure optimal performance during operation cycles. One system included is a preventative diode that protects against reverse voltage if the power supply is connected incorrectly.

Lastly, our Power Module boasts roughly 87% efficiency from synchronous rectification. This ensures that our system is utilizing as much usable power as possible, increasing the lifetime of batteries while also preventing wasted energy consumption.

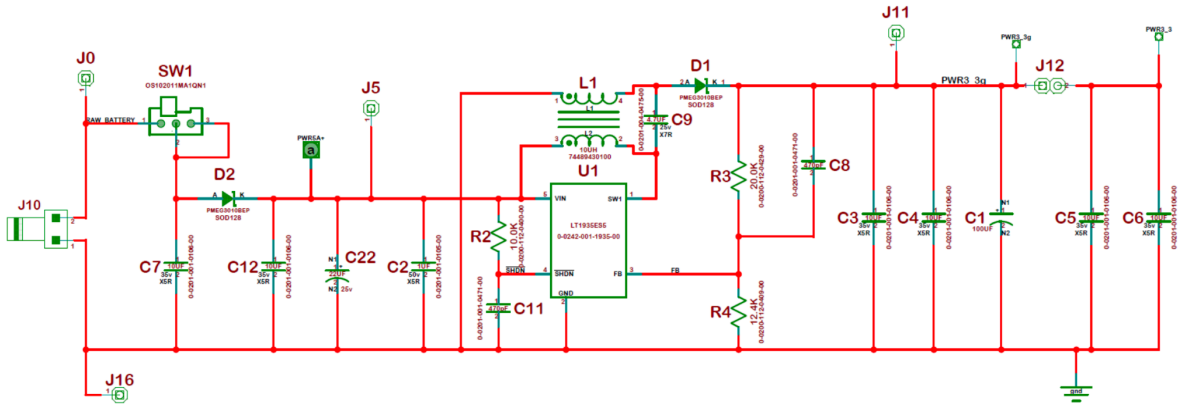

Figure 12: Power Module Schematic

4.7 Subsystem 5: Emitter & Detector Module

To enable our car to detect variances in color in front of the car we have installed a lightweight Emitter & Detector Module at the front of our car. This module is placed on top of a triangle-shaped cutout at the front of the car; having a triangle-shaped cutout allows the emitter LED (IR LED) and the two detectors to have a vantage point of the ground. The board is connected to the FET Module (Subsystem 4) via extended wires to enable communication with the board. The Emitter & Detector Module works alongside the internal ADC the MSP430 provides to detailed reading every ~50ms.

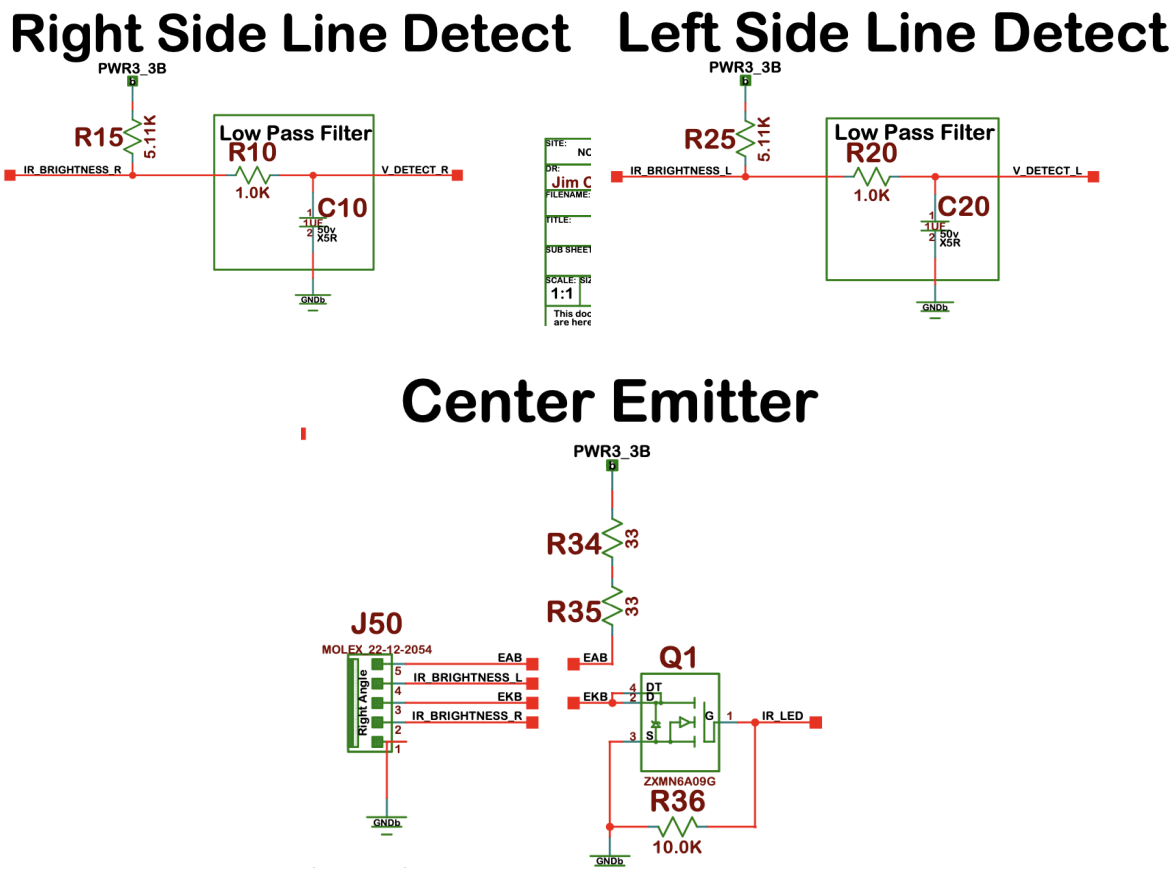

Figure 13: Emitter & Detector Module Schematic

Although the Emitter & Detector Module is lightweight it has 4 output channels that control its different functionalities. The IR LED is located in the middle of the board; this LED emits IR light. On the right and left side of the board, there are two detectors, phototransistors, that capture the light present, in their view, under the car. Combining the emitter and detector functionalities, we are able to “sense” the IR characteristic under the car. For example, when the detectors are pointed at a dark object, they will pick up a high IR characteristic, and when there's a light surface under the car they will read a lower IR characteristic. This description gives an overview of the ability this hardware provides us when programming the car.

4.8 Subsystem 6: IoT Module

The IoT module is an embedded system in PCB assembly that provides communication between our microprocessor and local networks. We use the UART interrupts that are provided by the MSP430 to send and receive messages from the IoT system. By sending commands to the IoT module we can establish a connection with the local wireless network and create a virtual server. This then allows us to connect to the IoT module wirelessly and send commands back to the IoT module. We use this system to send commands to our car and have it perform various tasks such as turning on motors, triggering interrupts, changing displays, etc.

IoT Module Circuits

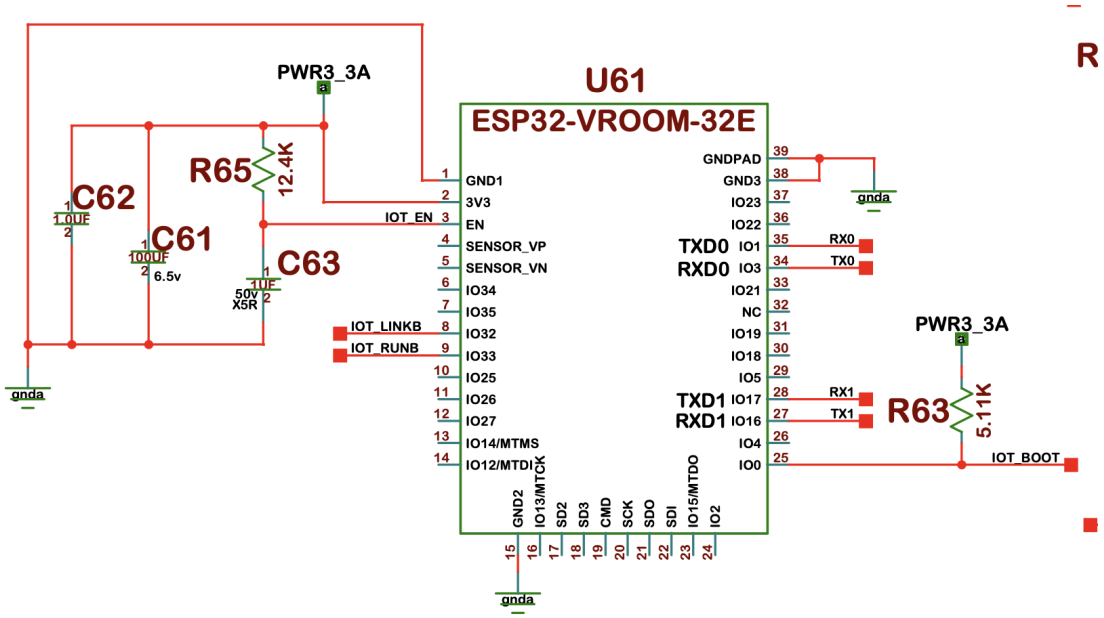

Figure 14: ESP32-VROOM

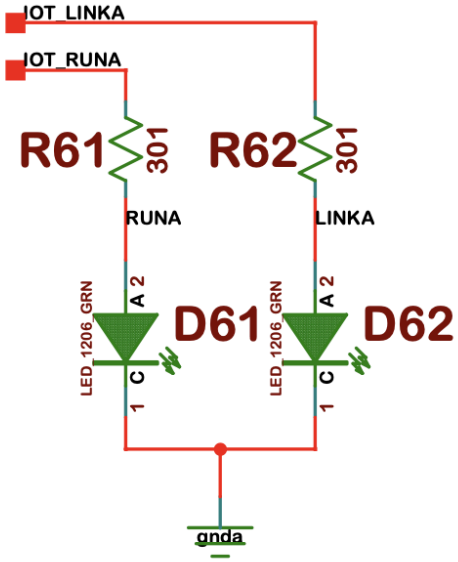

Figure 15: LED Circuit

This document contains information that is **PRIVILEGED** and **CONFIDENTIAL**; you are hereby notified that any dissemination of this information is strictly prohibited.	Date: **12 - 07 - 2023**	Document # **0000-0001**	Rev: **C**	Sheet:

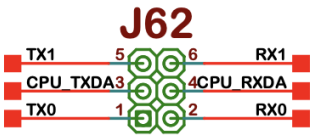

Figure 16: Connector Pins J62

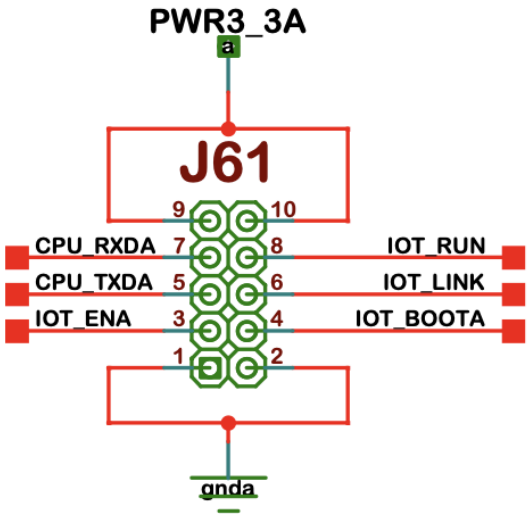

Figure 17: Connector Pins J61

6. Test Process

How are you testing the various functions and hardware? Oscilloscope / Volt meters / Custom code. This should also indicate how you tested and verified a function.

6.1. AtoD Verification

Testing the Power Supply

The battery power system was built first, which was tested using oscilloscopes and digital voltmeters in the lab. This included all the discrete resistors, capacitors, diodes, inductors, and the 3.3-volt SEPIC Converter; LT1935 that are installed on the bottom side of the board. The DAC power board was also tested using an oscilloscope and digital voltmeter with similar components already installed. Each board was tested with test settings of 5 Volts and 1milliAmp. The oscilloscope was set up to read 3.3 volts DC.

In addition to testing power, boards were inspected to make sure no components were incorrectly soldered to the board in order to prevent any potential future issues. Any components that did not pass visual inspection were re-soldered to the board and again visually inspected.

FET Testing

Once the FETs were installed they were tested to ensure the power supply to the motors worked properly. For project four, only two NFETs needed to be tested to verify forward movement. Due to a jumper installed at JMP-L and JMP-R, there was no need for PFETs in this project. Firmware to test these FETs was created and loaded to only turn the forward direction for one motor. It uses Switch 1 (SW1) and Switch 2 (SW2) to turn on each motor control pin.

With the battery on and firmware loaded, test points TP1 and TP3 were checked to verify that R_FORWARD was working properly. When R_FORWARD is off, TP1 and TP3 should equal battery voltage, and with R_FORWARD on, TP1 should show half of the battery voltage while TP3 is at ground potential.

The same procedure is used to test L_FORWARD, the only difference being that it uses the test points TP5 and TP7 instead. When L_FORWARD is off, TP5 and TP7 should equal battery voltage, and with L_FORWARD on, TP5 should be at half of the battery voltage while TP7 is at ground potential.

FET testing will become even more important in future projects, as having both forward and reverse pins on at the same time can destroy the FET board.

Analog Discovery Testing

To further test the development board, the Analog Discovery 2 was used to test pin logic when toggling the motors. As seen in Figure 10 below, R_FORWARD corresponds to P6.1 and L_FORWARD corresponds to P6.3 on the MSP430FR2355. Analog discovery jumpers labeled D0 - D3 were then connected to the pins 6.1-6.4, respectively. To test the pin logic and verify it works properly, the pins were enabled as outputs using bit masks and set to high (as if the motors were one) to simulate the car moving forward. If both of the corresponding pin logic went high after these pins were toggled, the program works and there should be no issues with the FETs once the code is implemented. However, if both pins do not go high when the forward movement function is called upon, issues could arise and must be resolved.

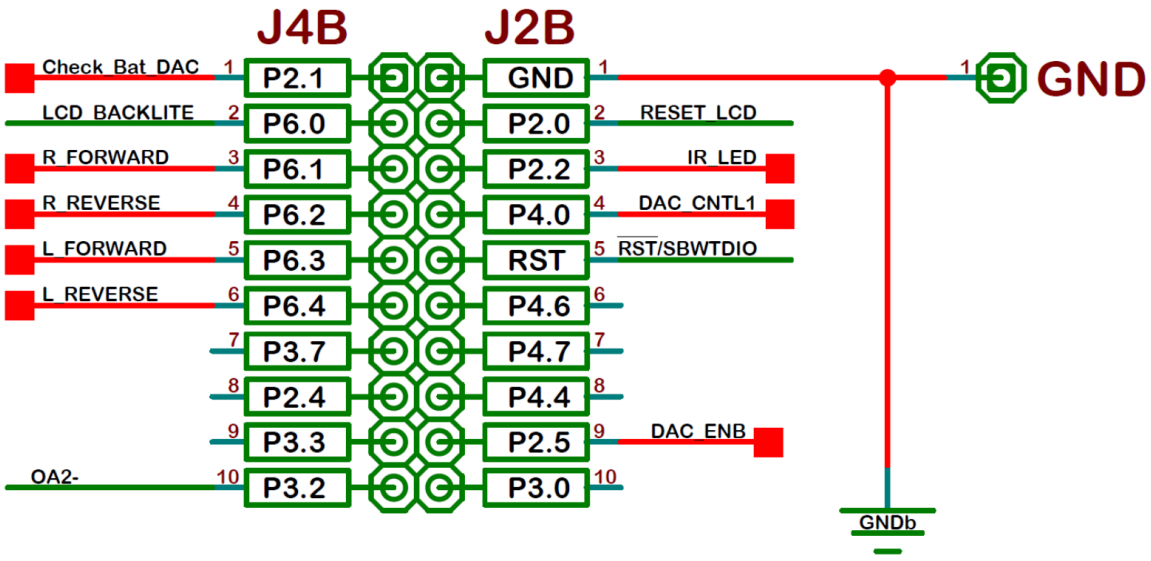

Figure 18: Circuit 2

This document contains information that is **PRIVILEGED** and **CONFIDENTIAL**; you are hereby notified that any dissemination of this information is strictly prohibited.	Date: **12 - 07 - 2023**	Document # **0000-0001**	Rev: **C**	Sheet:

6.2. Circuit 1

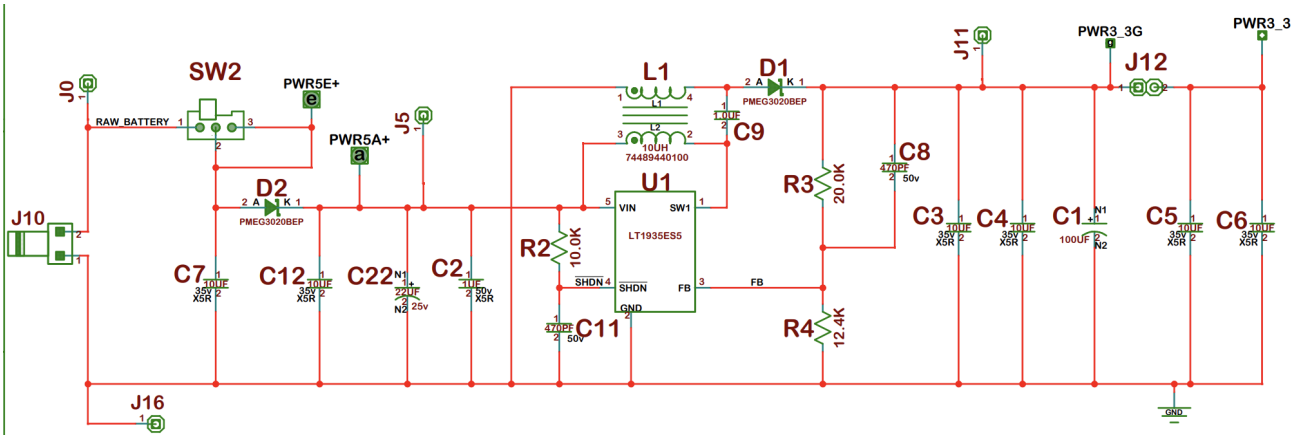

Figure 19: Circuit 1

Setting the probes:

The ground of the oscilloscope and the ground of the power supply were connected to the ground plate (GND) of the board. The positive power supply was then connected to J11, and the positive end of the oscilloscope was connected to J12, which is labeled as 3.3V. The power supply is then quickly turned on and off, while it is verified that the waveform displayed on the oscilloscope reaches the 3.3V mark.

Circuit 1:

Circuit 1 displays the SEPIC converter power system used by the power board in the project. This SEPIC converter will drop the voltage from an input of 5V to an output of 3.3V. In this case, the SEPIC converter acts as a buck converter by dropping the output voltage.

The battery power connection is at J10 (as referenced in the lab 2 notes) to supply power to the entire system. J5 is a junction and is where probing usually occurs. The diode labeled D2 is put in place in order to protect the system from an occurrence in which the batteries are installed backward. This blocks current flowing in the wrong direction, preventing serious damage to the board. Voltage output is ultimately measured at J11, and current output is measured at J12. Additionally, resistors R3 and R4 are put in place in order to make sure that the converter drops the voltage from an initial 5V to output 3.3V.

6.3. Circuit 2

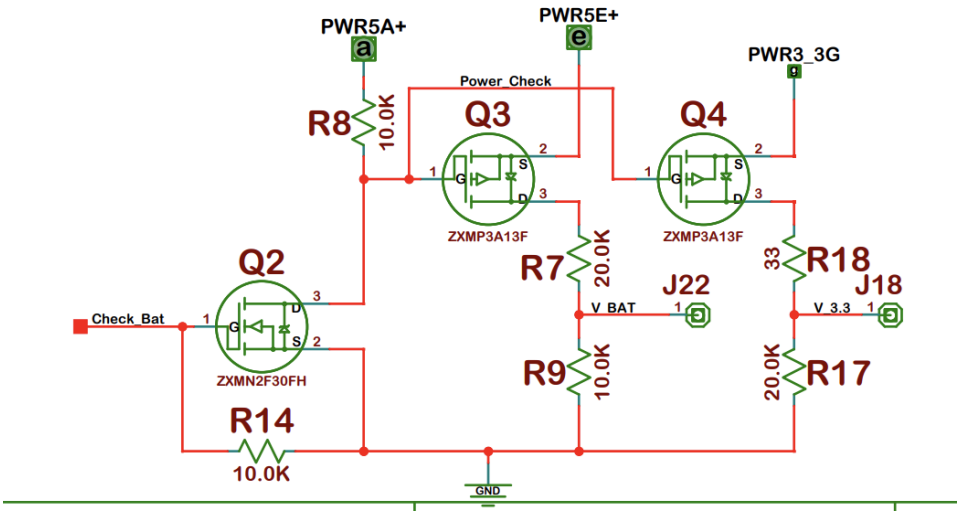

Figure 20: Circuit 2

Setting the probes:

The ground of the oscilloscope and the ground of the power supply were connected to the ground plate (GND) of the board. The positive power supply was then connected to Vin, and the positive end of the oscilloscope was connected to the pin labeled DACV. The power supply was then quickly turned on and off. After the power supply has been toggled, it is important to notice the waveform going high to around 5V, and then slowly lowering down to about 3.3V before stopping. This slow lowering of the voltage indicates that the circuit is correctly dropping the voltage to operable and non-damaging levels.

Circuit 2:

Circuit 2 includes voltage dividers in the PFETs labeled Q2 and Q3. These FETs are used to power the LCD display. Currently, the only functionality of interest for the project are the resistor dividers used for checking battery voltage.

6.4. UART Verification Testing

In order to test the UART communication features of the MSP430FR2355 board, tests were conducted with both jumpers and the Analog Discovery 2. After the UART interrupts of eUSCI0_ISR and eUSCI1_ISR are enabled and working properly, the Analog Discovery was used as a terminal for testing the receive and transmit features of the board. Firstly, to ensure proper baud rate is set on both the AD2 and the vehicle, a transmission was initiated from the AD2. Once it was received, the vehicle was to transmit what it had just received directly back to the terminal on the AD2. This verifies whether or not the data is received correctly, and that process buffers are utilized properly in the coding. Additionally, in order to avoid errors, a jumper was installed on the board, connecting the UART transmit and receive pins of the channel that was not being tested by the AD2 (either A0 or A1). In this case, to test the UART communication of UCA0, the AD2 was configured to J9 of the LCD display. This procedure was used to receive and immediately transmit back commands at two different baud rates to ensure code was working properly.

6.5. IOT Module Verification Testing

After the correct hardware additions, the IOT module needs to be tested to ensure that it is connected and functioning correctly. Using a serial terminal emulator such as Termite, add the MSP430 in the device manager. Then configure the baud rates of the terminal and UCA1 to 115,200, with no parity, 8 bits of data, and 1 stop bit. Next, jump the TXD and RXD pins on the MSP430 and ensure that the PC commands sent through the terminal show up again on the PC Terminal Emulator screen. This creates a hardware loopback with the PC using only the programming interface, excluding the code. If this works, then the PC and your serial terminal of choice are working properly, and you can begin testing the IOT module.

Afterward, use a jumper on the IOT serial port on J9. This part is meant to test the correct port setup within the software. Set up the software in your UART interrupts to take characters received on one port and transmit them to another port. The information to and from the PC should be routed through UCA1, whereas the information sent and received to and from the IOT module should be routed through UCA0. The IOT receive ISR (UCA0) should be configured to transmit the received character out to the USB PC (UCA1) Tx port. Then, using a command structure, the IOT can be tested. By creating a case statement for identifying and deciphering commands, the connection can be tested by sending commands and monitoring the responses on the PC.

Last, to test the wireless connectivity and functionality of the IOT module, the IOT needs to be freshly programmed and connected to the FRAM using a hardware reset. After this is done, the Terminal should show the message “ready”. Afterward, a series of commands listed in the project outline and IOT module manual can be used to connect the IOT to the internet, setting up various settings used in the project. With each command, the IOT should initiate a response to the terminal to ensure that it is properly connected. If at any point the message displays “ERROR”, then something needs to be fixed to ensure proper functionality.

7. Software

7.1. Main

The main routine, which is encapsulated in the main source file (main.c), holds the functionality of the MSP430 MCU for its intended embedded system implementation. In this case, the MCU is tailored for a car application that incorporates many peripherals such as ADC, LCD, timers, clocks, and LEDs. The system's operating system is implemented in a while loop, which represents a background foreground execution model.

The main routine contains the initialization of critical systems that are necessary for a functional car. The ports are initialized predominantly to GPIO to ensure proper connectivity to the peripherals which may be altered via other subroutines. The clocks are initialized to set up the necessary clocks such as SMCLK to establish the timing basis for certain subsystems of the MCU such as the clocks used in timer B0 or B3. Main includes condition initialization which sets the crucial peripheral variables to an initial state. Following the initialization of the variable conditions the necessary timers, B0 and B3, which are used in ISRs are initialized. The initialization of the ADC ensures that the ADC is in a known state and is ready to process analog data from various input channels, which is essential in the car's movement.

Main contains a while(ALWAYS) loop which governs the continuous operation of the operating system. Furthermore, the operating system manages the display process which ensures the accurate display of the system's state. The while(ALWAYS) loop also ensures that the motor connecting pins on the transistors enter a forward/reverse direction state which can result in significant damage to the FET board. The ADC is also controlled by the operating system which monitors and processes analog input from various channels which contributes to the state of the movement of the car's motors. The switching process is also monitored by the operating system and handles any input from SW1 and SW2 that can change or influence the state of the system.

7.2. Ports

The ports file configures all the ports and their corresponding pin assignments. Using bit masks, the ports were configured to default settings that enabled the required pins while saving as much power as possible. Ports are to be enabled upon use to make sure the compiler is working as efficiently as possible. Using the MSP430FR2355 manual, pins were checked for their low power states to then be set as the default values.

Each port pin has a minimum of 4 lines for their configuration, with additional configuration options for pins set as GP I/O and inputs. The first option is GP I/O, which every pin can be set as. Alternate functions can be chosen using the two select bits. Additionally, pins can be set as an input or output based on the value of PxDIR, and whether or not the pin is outputting a desired value is dependent on the PxOUT value. Using specified bit masks using “#define” for certain functions, pins are turned off by “anding” the negated value of the function that should be turned off. Conversely, settings are enabled by “oring” the macro for the value of the function that should be turned on. For example, P2OUT &= ~IR_LED turns the IR LED off, whereas P2OUT |= IR_LED turns it on.

Every port pin and its functions are listed in the MSP430 User Manual with schematics displaying pin layouts, as well as all the pin’s possible functions. Some pins support multiple actions and need to be toggled accordingly for different applications. For project 6 certain pins needed to be enabled for the ADC to work properly. The P5SELC setting needed to be enabled for ADC readings of the DAC, V_3_3, V_5_0, and VBAT functions. Additionally, the pins toggled the most are the movement pins in port 2, specifically port 6 pins 1-4 dealing with the different motor’s forward and reverse movement options.

7.3. ADC

The analog-to-digital converter (ADC) is a peripheral of the MSP430. The sensor attached to the vehicle provides an analog signal for which the ADC produces a corresponding digital value at the output. The ADC is configured by the user to select the sampling periods, conversion clock, conversion initiation, conversion mode, and conversion channel.

When the interrupt service routine for Port 4 is triggered, corresponding to a press of Switch 1, a timer is enabled that intermittently starts conversions for channel A2. When a conversion is completed, the ADC interrupt service routine is triggered. The interrupt service routine processes and displays the data provided by the ADC. The values for the left and right detectors, channels A2 and A3 respectively, are used to determine whether the vehicle has reached the black line. When a conversion for channel A2 is completed, the conversion channel is updated to channel A3 and a conversion is started. Similarly, when a conversion for channel A2 is completed, the conversion channel is updated to channel A5 and a conversion is started. Following a conversion for channel A5, the conversion channel is reset to the first channel, A2. The next sequence of conversions for channels A2, A3, and A5 is initiated by the timer interrupt service routine.

7.4. Timers

Timers are a necessary part of our design configuration using the MSP430. Scheduling different events when using a super-loop to service requests can sometimes be hard if you need your code to wait on some execution processes. Timers give us the ability to set accurate interrupts with specified execution occurrences that can execute asynchronously to the main code.

The MSP430 provides multiple different timer options and vector spaces for interrupts. The MSP430 has timers A and B; we are using timer B which has B0, B1, B2, and B3 as options. Each of these timers has dedicated capture/compare registers, B0-B2 has R 0-3 and B3 has registers 0-7. This allows us to configure these registers to provide different functionalities to our code.

To start a timer and have it function correctly we configured, enabled, initialized, and wrote the interrupts for the specific timer. Timer configuration includes port definitions (necessary for PWM), clock settings, clock dividers, register enable, setting intervals, clearing overflow, etc. Enabling a timer is equivalent to enabling the interrupt that will handle the timer’s logic. Following the configuration and

enabling, we initialize the timer in our Main function. Lastly, we created specific interrupts in timer vectors to handle the logic of the different timers running.

8. Flow Charts

8.1. Main

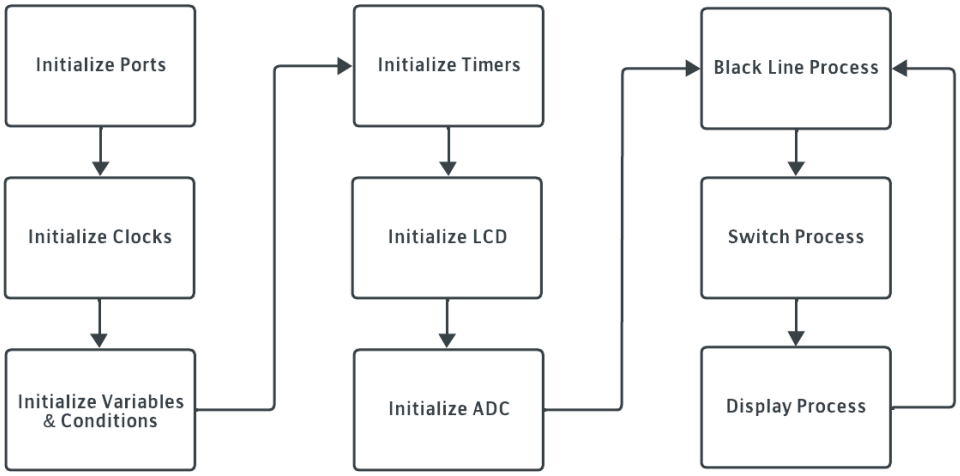

Figure 21: Main Flow Chart

8.2. Ports

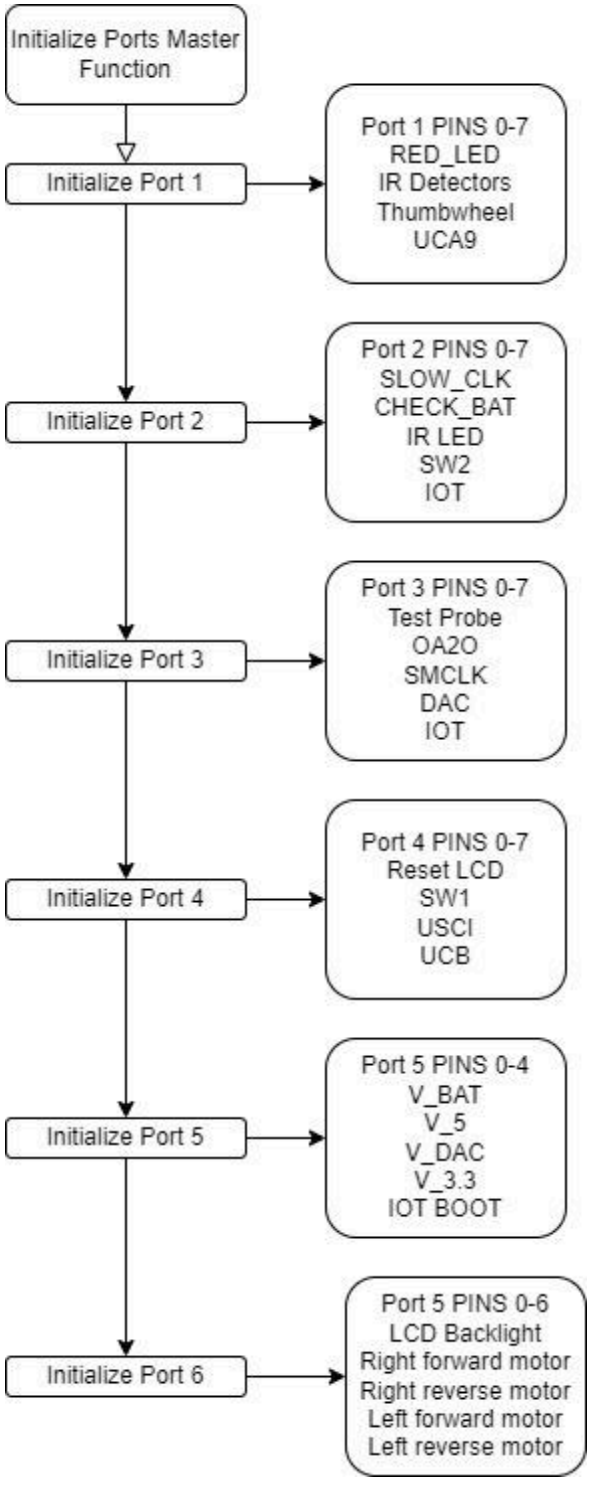

Figure 22: Port Configurations Flow Chart

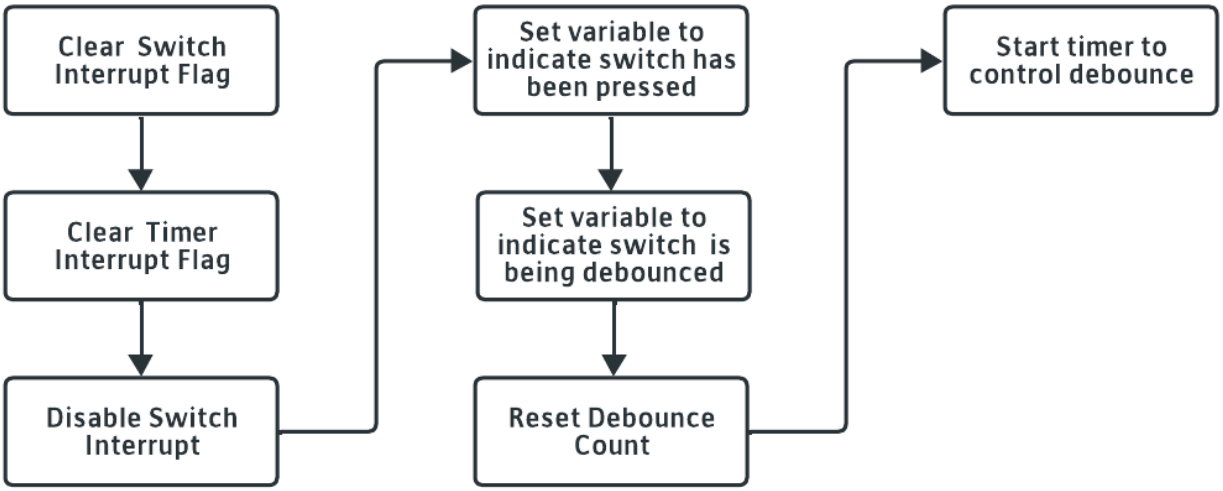

Figure 23: Port Interrupt Service Routines for Switch 1 & Switch 2

8.3. ADC

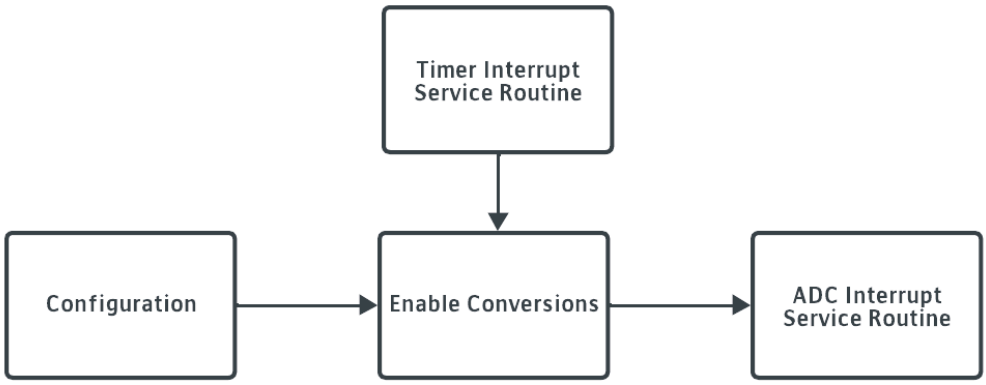

Figure 24: ADC Overview

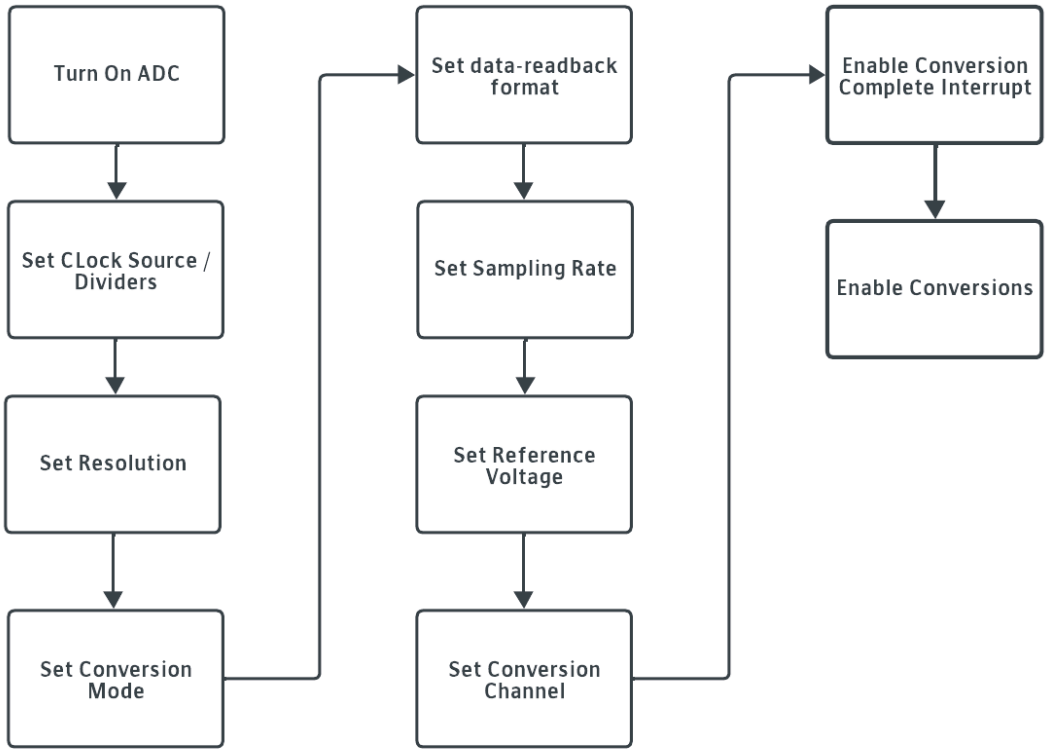

Figure 25: ADC Configuration

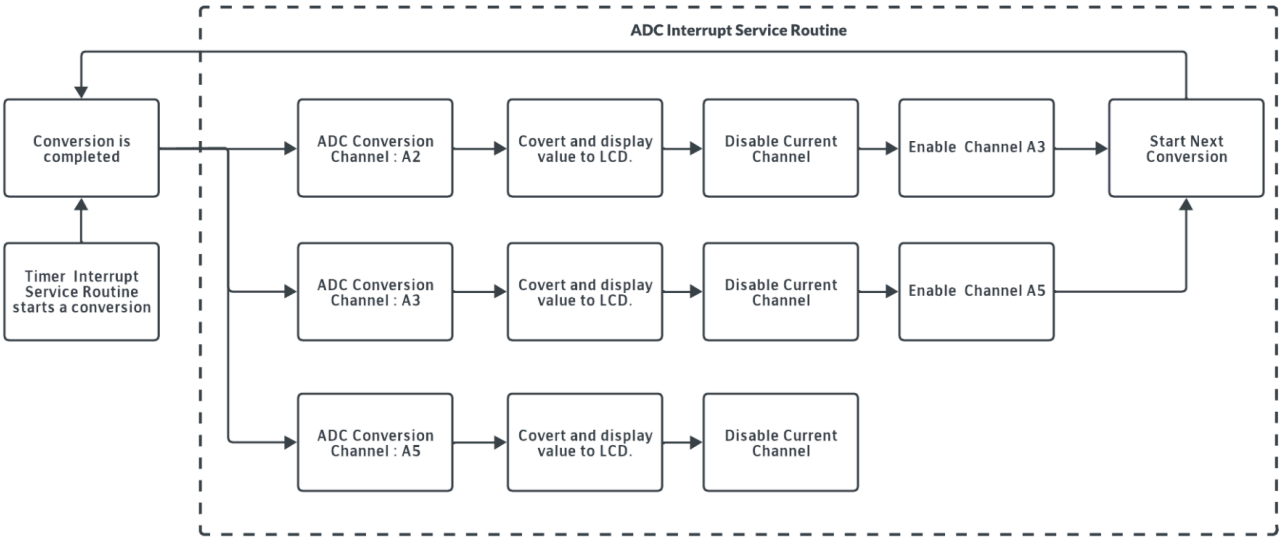

Figure 26: ADC Interrupt Service Routine

8.4. Timers

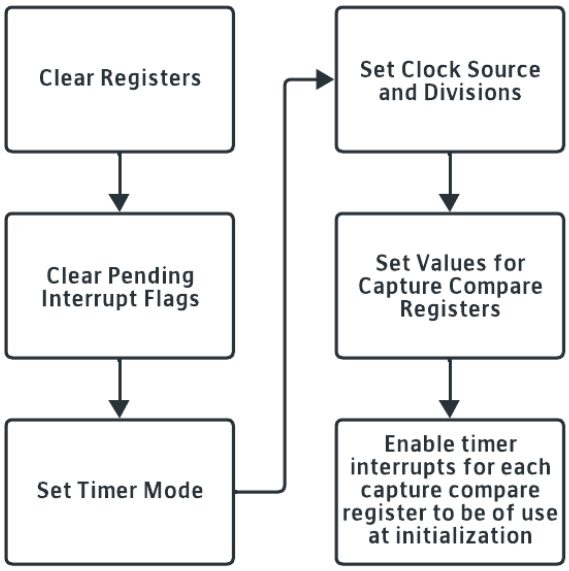

Figure 27: Timer Configurations

8.5. Serial Communication

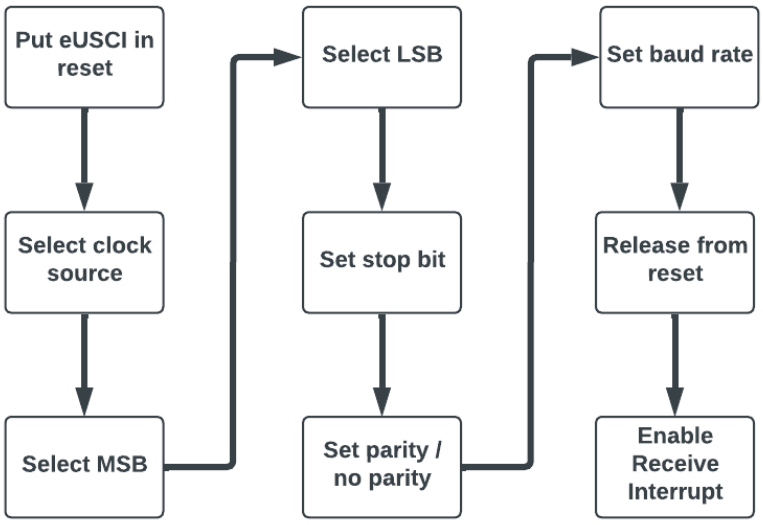

Figure 28: Serial Port Configuration

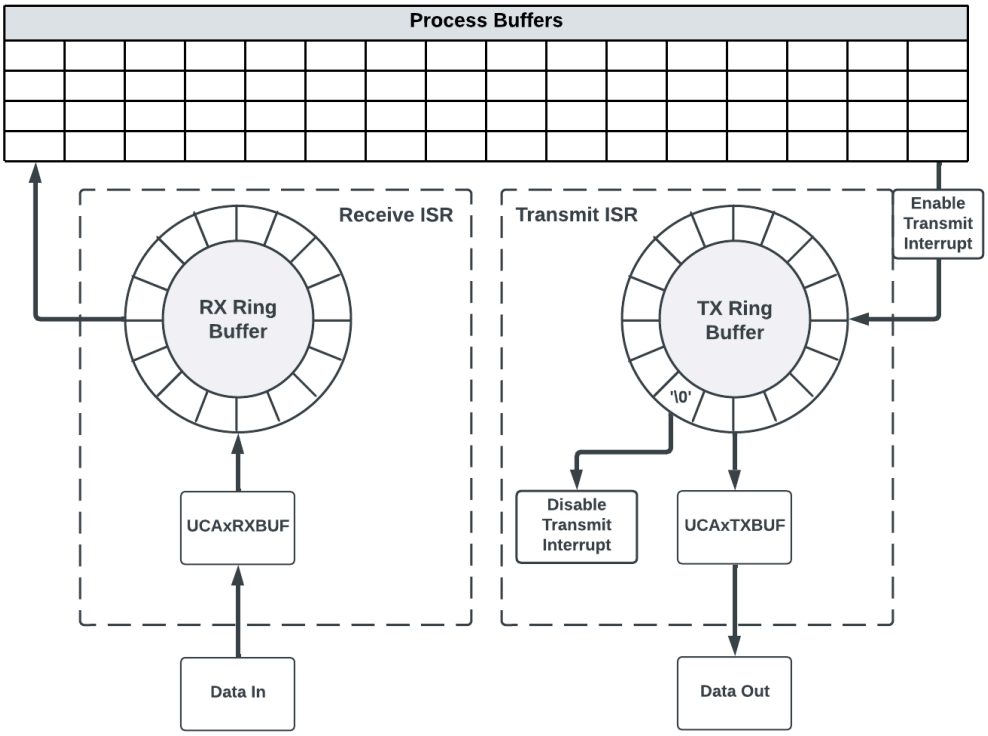

Figure 29: Serial Interrupt Service Routines

9. Software Listing

9.1. Main.c

```
//-------------------------------------------------------------------------------
//
//  Description: This file contains the Main Routine - "While" Operating System
//
//  Jim Carlson
//  Jan 2023
//  Built with Code Composer Version: CCS12.4.0.00007_win64
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
#include  "msp430.h"
#include  <string.h>
#include  "include/functions.h"
          "Include/LCD.h"
#include  "include/ports.h"
#include  "Include/macros.h"
// Function Prototypes
void main(void);
// project 3 functions
//----------------------
//void stop(void);
//void move_forward(void);
//----------------------
// ---------------------------------------
volatile char slow_input_down;
char display_line[4][11];
char *display[4];
volatile unsigned char display_changed;
extern volatile unsigned char update_display;
extern volatile unsigned int update_display_count;
unsigned int movement_state;
// timer var from interrupt.c
//unsigned int timer;
// ADC global refrenced variables
unsigned int ADC_Channel;
unsigned int ADC_Thumb;
unsigned int ADC_Right_Det;
unsigned int ADC_Left_Det;
unsigned int count_adc;
//-------------------------------------------------------------------------------------------------------
----------------------------
//void main(void){
void main(void){
    //    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    //-------------------------------------------------------------------------
    // Main Program
```

This document contains information that is **PRIVILEGED** and **CONFIDENTIAL**; you are hereby notified that any dissemination of this information is strictly prohibited.	Date: **12 - 07 - 2023**	Document # **0000-0001**	Rev: **C**	Sheet:

```
    // This is the main routine for the program. Execution of code starts here.
    // The operating system is Back Ground Fore Ground.
    //
    //-------------------------------------------------------------------------
      PM5CTL0 &= ~LOCKLPM5;
    // Disable the GPIO power-on default high-impedance mode to activate
    // previously configured port settings
      Init_Ports();                       // Initialize Ports
      Init_Clocks();                      // Initialize Clock System
      Init_Conditions();                  // Initialize Variables and Initial Conditions
      Init_LCD();                         // Initialize LCD
      Init_Timer_B0();                    // initialize ports for timer B0
      Init_Timer_B3();                    // initialize ports for timer B3
      Init_ADC();                         // initialize ports for ADC
      Init_LEDs();                        // initialize ports for LEDs / LCD
    //P2OUT &= ~RESET_LCD;
      // Place the contents of what you want on the display, in between the quotes
    // Limited to 10 characters per line
      strcpy(display_line[0], "          ");
      strcpy(display_line[1], "          ");
      strcpy(display_line[2], "          ");
      strcpy(display_line[3], "          ");
      display_changed = TRUE;
      // Set Port pin High [Wheel On] // Set Port pin High [Wheel On] // Set Red LED On
      // Set Port pin High [Wheel On] // Set Port pin Low [Wheel Off] // Set Red LED Off
//-------------------------------------------------------------------------
// Begining of the "While" Operating System
//-------------------------------------------------------------------------
        while(ALWAYS) {                     // Can the Operating system run
           Display_Process();
           P3OUT ^= TEST_PROBE;
           display_changed = TRUE;
           forward_reverse_check();
           if(count_adc){
               movement_state=1;
               Line_State_Machine();
               count_adc = 0;
           }
           Switches_Process();
        }
}
```

This document contains information that is **PRIVILEGED** and **CONFIDENTIAL**; you are hereby notified that any dissemination of this information is strictly prohibited.	Date: **12 - 07 - 2023**	Document # **0000-0001**	Rev: **C**	Sheet:

9.2. Ports.c

```
// ports.c

//-------------------------------------------------------------------------------
//
//  DESCRIPTION: PORTS.C CONTAINS CODE THAT INITIALIZES PORTS 1-6.
//               CURRENTLY FUNCTIONS FOR PROJECT 10
//               UPDATES MUST BE DONE HERE TO PIN SETUPS FOR FUTURE ADDITIONS
//
//  ANDREW WHITEHEAD
//  DEC 2023
//  Built with Code Composer Version: CCS12.4.0
//-------------------------------------------------------------------------------



#include  "msp430.h"
#include  <string.h>
#include  "functions.h"
#include  "LCD.h"
#include  "ports.h"
#include "macros.h"
#include "global.h"


void Init_Ports(){ // Called in main, initializes ports 1-6 by calling functions bellow
    Init_Port1();
    Init_Port2();
    Init_Port3();
    Init_Port4();
    Init_Port5();
    Init_Port6();
}



void Init_Port1(void){ // Configure Port 1
//-------------------------------------------------------------------------------
    P1OUT = 0x00;
    P1DIR = 0x00;

    P1SELC |= A1_SEEED; // ADC input for A1_SEEED
   P1SELC |= V_DETECT_L; // ADC input for V_DETECT_L
   P1SELC |= V_DETECT_R; // ADC input for V_DETECT_R
   P1SELC |= A4_SEEED; // ADC input for V_A4_SEEED
   P1SELC |= V_THUMB; // ADC input for V_THUMB


    // P1 PIN 0
    P1SEL0 &= ~RED_LED;     //RED_LED GPIO
    P1SEL1 &= ~RED_LED;     //RED_LED GPIO
    P1DIR |= RED_LED;       //RED_LED SET DIRECTION TO OUTPUT
    P1OUT |= RED_LED;       //RED_LED SET INITIAL BALUE TO HIGH


    //P1 PIN 1
```

```
    P1SEL0 |=A1_SEEED;  // A1 Function Select
    P1SEL1 |=A1_SEEED;  // A1 Function Select


    //P1 PIN 2              //ADC Left IR Detector
    P1SEL0 |=V_DETECT_L;    //A2 Function Select
    P1SEL1 |=V_DETECT_L;    //A2 Function Select


    //P1 PIN 3              //ADC Right IR Detector
    P1SEL0 |=V_DETECT_R;    //A3 Function Select
    P1SEL1 |=V_DETECT_R;    //A3 Function Select


    //P1 PIN 4
    P1SEL0 |=A4_SEEED;  //A4 Function Select
    P1SEL1 |=A4_SEEED;  //A4 Function Select


    //P1 PIN 5          //ADC Thumbwheel
    P1SEL0 |=V_THUMB;   //A5 Function Select
    P1SEL1 |=V_THUMB;   //A5 Function Select


    //P1 PIN 6          //A0 serial receive pin (IOT)
    P1SEL0 |=UCA0RXD;   //GPIO select
    P1SEL1 &= ~UCA0RXD; //GPIO select


    //P1 PIN            //A0 Transmit pin (IOT)
    P1SEL0 |=UCA0TXD;   //GPIO select
    P1SEL1 &=~UCA0TXD;  //GPIO select
}

void Init_Port2(void){ // Configure Port 2
//------------------------------------------------------------------------------
     P2OUT = 0x00; // P2 set Low
     P2DIR = 0x00; // Set P2 direction to output

     //P2 PIN 0
     P2SEL0 &= ~SLOW_CLK; // SLOW_CLK GPIO operation
     P2SEL1 &= ~SLOW_CLK; // SLOW_CLK GPIO operation
     P2OUT &= ~SLOW_CLK; // Initial Value = Low / Off
     P2DIR |= SLOW_CLK; // Direction = output

     //P2 PIN 1
     P2SEL0 &= ~CHECK_BAT; // CHECK_BAT GPIO operation
     P2SEL1 &= ~CHECK_BAT; // CHECK_BAT GPIO operation
     P2OUT &= ~CHECK_BAT; // Initial Value = Low / Off
     P2DIR |= CHECK_BAT; // Direction = output

     //P2 PIN 2
     P2SEL0 &= ~IR_LED; // P2_2 GPIO operation
     P2SEL1 &= ~IR_LED; // P2_2 GPIO operation

    // P2OUT &= ~IR_LED; // Initial Value = Low / Off
     P2OUT |= IR_LED;
     P2DIR |= IR_LED; // Direction = input
```

```
    //P2 PIN 3
    P2SEL0 &= ~SW2; // SW1 GPIO operation
    P2SEL1 &= ~SW2; // SW1 GPIO operation
    P2DIR &= ~SW2; // Direction = input
   //Interrupt initialization for switch 2
    P2PUD |= SW2;
    P2IES |= SW2;
    P2IFG &= ~SW2;
    P2IE |= SW2;
    P2REN |= SW2;

    //P2 PIN 4
    P2SEL0 &= ~IOT_RUN_RED; // IOT_RUN_CPU GPIO operation
    P2SEL1 &= ~IOT_RUN_RED; // IOT_RUN_CPU GPIO operation
    P2OUT &= ~IOT_RUN_RED; // Initial Value = Low / Off
    P2DIR |= IOT_RUN_RED; // Direction = input

    //P2 PIN 5
    P2SEL0 &= ~DAC_ENB; // DAC_ENB GPIO operation
    P2SEL1 &= ~DAC_ENB; // DAC_ENB GPIO operation
    P2OUT |= DAC_ENB; // Initial Value = High
    P2DIR |= DAC_ENB; // Direction = output

    //P2 PIN 6
    P2SEL0 &= ~LFXOUT; // LFXOUT Clock operation
    P2SEL1 |= LFXOUT; // LFXOUT Clock operation

    //P2 PIN 7
    P2SEL0 &= ~LFXIN; // LFXIN Clock operation
    P2SEL1 |= LFXIN; // LFXIN Clock operation
 //--------------------------------------------------------------------------
}

void Init_Port3(void){ // Configure Port 3
//--------------------------------------------------------------------------
    P3OUT = 0x00;
    P3DIR = 0x00;

    //P3 PIN 0
    P3SEL0 &= ~TEST_PROBE;  //GPIO operation
    P3SEL1 &= ~TEST_PROBE;  //GPIO operation
    P3DIR |= TEST_PROBE;    //Output

    //P3 PIN 1
    P3SEL0 |= OA2O;     //OA20O operation
    P3SEL1 |= OA2O;     //OA2O operation

    //P3 PIN 2
    P3SEL0 |= OA2N;     //OA2- operation
    P3SEL1 |= OA2N;     //OA2- operation

    //P3 PIN 3
    P3SEL0 |= OA2P;     //OA2+ operation
    P3SEL1 |= OA2P;     //OA2+ operation

    //P3 PIN 4
    P3SEL0 &= ~SMCLK;   //SMCLK operation
```

```
    P3SEL1 &= ~SMCLK;   //SMCLK operation
    P3DIR &= ~SMCLK;    //Input Direction

    //P3 PIN 5
    P3SEL0 |= DAC_CNTL; //DAC Setup
    P3SEL1 |= DAC_CNTL; //DAC Setup

    //P3 PIN 6
    P3SEL0 &= ~IOT_LINK_GRN;    //IOT Link operation GPIO
    P3SEL1 &= ~IOT_LINK_GRN;    //IOT Link operation GPIO

    //P3 PIN 7
    P3SEL0 &= ~IOT_EN;      //IOT Enable GPIO
    P3SEL1 &= ~IOT_EN;      //IOT Enable GPIO
}

void Init_Port4(void){ // Configure Port 4
//------------------------------------------------------------------------------

     P4OUT = 0x00; // P1 set Low
     P4DIR = 0x00; // Set P1 direction to output
    // P4 PIN 0
     P4SEL0 &= ~RESET_LCD; // RESET_LCD GPIO operation
     P4SEL1 &= ~RESET_LCD; // RESET_LCD GPIO operation
     P4OUT &= ~RESET_LCD; // Set P4_0 initial value Low
     P4DIR |= RESET_LCD; // Set P4_0 direction to output

     // P4 PIN 1
     P4SEL0 &= ~SW1; // SW1 GPIO operation
      P4SEL1 &= ~SW1; // SW1 GPIO operation
      P4DIR &= ~SW1; // Direction = input
      //Interrupt setup for switch 1
      P4PUD |= SW1;
      P4IES |= SW1;
      P4IFG &= ~SW1;
      P4IE |= SW1;
      P4REN |= SW1;

    // P4 PIN 2
     P4SEL0 |= UCA1TXD; // USCI_A1 UART operation
     P4SEL1 &= ~UCA1TXD; // USCI_A1 UART operation
    // P4 PIN 3
     P4SEL0 |= UCA1RXD; // USCI_A1 UART operation
     P4SEL1 &= ~UCA1RXD; // USCI_A1 UART operation

    // P4 PIN 4
     P4SEL0 &= ~UCB1_CS_LCD; // UCB1_CS_LCD GPIO operation
     P4SEL1 &= ~UCB1_CS_LCD; // UCB1_CS_LCD GPIO operation
     P4OUT |= UCB1_CS_LCD; // Set SPI_CS_LCD Off [High]
     P4DIR |= UCB1_CS_LCD; // Set SPI_CS_LCD direction to output

    // P4 PIN 5
     P4SEL0 |= UCB1CLK; // UCB1CLK SPI BUS operation
     P4SEL1 &= ~UCB1CLK; // UCB1CLK SPI BUS operation

    // P4 PIN 6
     P4SEL0 |= UCB1SIMO; // UCB1SIMO SPI BUS operation
     P4SEL1 &= ~UCB1SIMO; // UCB1SIMO SPI BUS operation
```

```
    // P4 PIN 7
     P4SEL0 |= UCB1SOMI; // UCB1SOMI SPI BUS operation
     P4SEL1 &= ~UCB1SOMI; // UCB1SOMI SPI BUS operation
 //-------------------------------------------------------------------
     }

void Init_Port5(void){ // Configure Port 5
//-------------------------------------------------------------------------------
   P5OUT = 0x00;
   P5DIR = 0x00;

   P5SELC |= V_BAT; // ADC input for V_BAT
   P5SELC |= V_5_0; // ADC input for V_5_0
   P5SELC |= V_DAC; // ADC input for V_DAC
   P5SELC |= V_3_3; // ADC input for V_3_3

   //P5 PIN 0
    P5SEL0 |= V_BAT;    //A8 function
    P5SEL1 |= V_BAT;    //A8 function

    //P5 PIN 1
    P5SEL0 |= V_5_0;    //A9 function
    P5SEL1 |= V_5_0;    //A9 function

    //P5 PIN 2
    P5SEL0 |= V_DAC;    //A10 function
    P5SEL1 |= V_DAC;    //A10 function

    //P5 PIN 3
    P5SEL0 |= V_3_3;    //A11 function
    P5SEL1 |= V_3_3;    //A11 function

    //P5 PIN 4
    P5SEL0 &= ~IOT_BOOT_CPU;    //GPIO
    P5SEL1 &= ~IOT_BOOT_CPU;    //GPIO
    P5OUT |= IOT_BOOT_CPU;  //output High
    P5DIR |= IOT_BOOT_CPU;  //output direction
}

void Init_Port6(void){ // Configure Port 6
//-------------------------------------------------------------------------------

    P6OUT = 0x00;
    P6DIR = 0x00;

    //P6 PIN 0
    P6SEL0 |= LCD_BACKLITE;          //LCD_BACKLITE SET GPIO
    P6SEL1 &= ~LCD_BACKLITE;        //LCD_BACKLITE SET GPIO
    P6DIR |= LCD_BACKLITE;        //LCD_BACKLITE SET DIRECTION TO OUTPUT


    //P6 PIN 1
    P6SEL0 |= R_FORWARD;    //Right Forward H-Bridge
    P6SEL1 &= ~R_FORWARD;

    P6DIR |= R_FORWARD; //output direction
```

```
    //P6 PIN 2
    P6SEL0 |= R_REVERSE;    //Right reverse H-Bridge
    P6SEL1 &= ~R_REVERSE;

    P6DIR |= R_REVERSE; //output direction

    //P6 PIN 3
    P6SEL0 |= L_FORWARD;    //Left forward H-Bridge
    P6SEL1 &= ~L_FORWARD;

    P6DIR |= L_FORWARD; //output direction

    //P6 PIN 4
    P6SEL0 |= L_REVERSE; //Left Reverse H-Bridge
    P6SEL1 &= ~L_REVERSE;

    P6DIR |= L_REVERSE; //output direction

    //P6 PIN 5
    P6SEL0 &= ~P6_5;    //GPIO
    P6SEL1 &= ~P6_5;
    P6DIR &= ~P6_5; //Input Direction

    //P6 PIN 6
    P6SEL0 &= ~GRN_LED;       //RED_LED SET GPIO
    P6SEL1 &= ~GRN_LED;       //RED_LED SET GPIO
    P6DIR |= GRN_LED;       //RED_LED SET DIRECTION TO OUTPUT
    P6OUT |= GRN_LED;       //RED_LED SET INITIAL VAL TO HIGH
}
```


Date:	Document #	Rev:	Sheet:
12 - 07 - 2023	**0000-0001**	**C**	

9.3. ADC.c

```
/*
 *  ADC.c
 *
 *  Description: ADC configuration
 *
 *  globals defined: ADC_Channel, ADC_Right_Detect, ADC_Left_Detect, ADC_Thumb, adc_char
 *
 *  Built with Code Composer Version: CCS12.4.0.00007_win64
 *  Created on: Oct 11, 2023
 *      Author: Martina Viola
 */

// Include Directives
#include  "msp430.h"
#include  <string.h>
#include  "functions.h"
#include  "LCD.h"
#include  "ports.h"
#include  "macros.h"


//-------------------------------------------------------------------------------------------------
/*
 * Function: Init_ADC(void);
 * globals used: none
 * passed variables: none
 * local variables: none
 * returned values: none
 *
 * Description: configuration for ADC
 */
//-------------------------------------------------------------------------------------------------
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 = 0;                    // Reset
    ADCCTL0 |= ADCSHT_2;            // 16 ADC clocks
    ADCCTL0 |= ADCMSC;              // MSC
    ADCCTL0 |= ADCON;               // ADC ON
// ADCCTL1 Register
    ADCCTL1 = 0;                    // 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 - 10b
    ADCCTL1 |= ADCCONSEQ_0;           // ADC conversion sequence 11b = Single-channel single-conv
                                      // ADCCTL1 & ADCBUSY  identifies a conversion is in process
// ADCCTL2 Register
    ADCCTL2 = 0;                      // Reset
    ADCCTL2 |= ADCPDIV_0;             // ADC pre-divider 10b = Pre-divide by 64
    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
    ADCIE |= ADCIE0;                  // Enable ADC conversion complete interrupt
    ADCCTL0 |= ADCENC;                // ADC enable conversion.
}
```

9.4. Timers.c

```
// timers.c
//------------------------------------------------------------------------------
//  Description: This file contains timer configs & enabling.
//
//  Ethan Cornett
//  Sep 2023
//  Built with Code Composer Version: 12.4.0.00007
//------------------------------------------------------------------------------

//      INCLUDES
#include  "msp430.h"
#include  <string.h>
#include  "Include/functions.h"
#include  "Include/ports.h"
#include  "Include/macros.h"

//------------------------------------------------------------------------------
//  GLOBALS
//------------------------------------------------------------------------------

//---------- Functions -------------//
void Init_Timer_B0(void){
    TB0CTL = TBSSEL__SMCLK;       //Set SMCLK as source clock
    TB0CTL |= MC__CONTINOUS;      //Continuous up
    TB0CTL |= ID__8;              //Divide clock by 2
    TB0EX0 = TBIDEX__8;           //Divide clock by an additional 8
    TB0CTL |= TBCLR;              //Resets TB0R, Clock divider, Count Direction
    TB0CCR0 = TB0CCR0_INTERVAL_25ms;
    TB0CCTL0 |= CCIE;             //Enable CCR0 interrupt
    TB0CCTL0 &= ~CCIFG;
    TB0CCR1 = TB0CCR2_INTERVAL;   //CCR1  Set debounce time
    TB0CCTL1 &= ~CCIE;             //Initially disabled CCR1 interrupt
    TB0CCR2 = TB0CCR2_INTERVAL;   //CCR2  Set debounce time
    TB0CCTL2 &= ~CCIE;             //Initially Disabled CCR2 interrupt
    TB0CTL &= ~TBIE;              //Disable Overflow Interrupt
    TB0CTL &= ~TBIFG;             //Clear overflow interrupt flag
}

void Init_Timer_B1(void){
    TB1CTL = TBSSEL__SMCLK;        //Set clock to SMCLK
    TB1CTL |= MC__CONTINOUS;      //Set clock to continuous
    TB1CTL |= ID__8;              //Divide clock by 2
    TB1EX0 = TBIDEX__8;           //Divide clock by an additional 8
    TB1CTL |= TBCLR;              //Resets TB0R, Clock divider, Count Direction
    TB1CCR0 = TB1CCR0_Interval_200ms;
    TB1CCTL0 |= CCIE;             //Disable CCR0 interrupt
    TB1CCTL0 &= ~CCIFG;           //Clear B1_R0 Interrupt flag
    TB1CCR1 = TB0CCR2_INTERVAL;   //CCR1  Set debounce time
```

```
    TB1CCTL1 &= ~CCIE;              //Initially disabled CCR1 interrupt
    TB1CCR2 = TB0CCR2_INTERVAL;    //CCR2  Set debounce time
    TB1CCTL2 &= ~CCIE;              //Initially Disabled CCR2 interrupt
    TB1CTL &= ~TBIE;               //Disable Overflow Interrupt
    TB1CTL &= ~TBIFG;              //Clear overflow interrupt flag
}
```

9.5. Serial.c

```c
/*
 *  serial.c
 *
 *  Description: Initializes serial ports UCA0 and UCA1
 *
 *  Built with Code Composer Version: CCS12.4.0.00007_win64
 *  Created on: Nov 2, 2023
 *      Author: Martina Viola
 */

// Include Directives
#include  "msp430.h"
#include  <string.h>
#include  "functions.h"
#include  "LCD.h"
#include  "ports.h"
#include  "macros.h"
#include "variables.h"

//-------------------------------------------------------------------------------------------------------
/*
 *  Function: Init_Serial_UCA0(void);
 *
 *  Description: Initialization for UCA0
*/
//-------------------------------------------------------------------------------------------------------
void Init_Serial_UCA0(char speed){

    // UART0 Configuration
    UCA0CTLW0 = 0;
    UCA0CTLW0 |= UCSWRST;           // Put eUSCI in reset
    UCA0CTLW0 |= UCSSEL__SMCLK;     // Set SMCLK as fBRCLK
    UCA0CTLW0 &= ~UCMSB;            // MSB, LSB select
    UCA0CTLW0 &= ~UCSPB;            // MSB, LSB select
    UCA0CTLW0 &= ~UCPEN;            // UCSPB = 0(1 stop bit) OR 1(2 stop bits)
    UCA0CTLW0 &= ~UCSYNC;           // No Parity
    UCA0CTLW0 &= ~UC7BIT;
    UCA0CTLW0 |= UCMODE_0;

    // set baud rate
    if (speed == '1') {
        UCA0BRW = 4;                // 115, 200 baud
        UCA0MCTLW = 0x5551;
    }
    else if (speed == '2'){
        UCA0BRW = 17;               // 460,800
        UCA0MCTLW = 0x4A00;
    }
```

```
    else {
        UCA0BRW = 4;                // 115, 200 baud
        UCA0MCTLW = 0x5551;
    }
    UCA0CTLW0 &= ~UCSWRST;          // release from reset
    UCA0TXBUF = 0x00;               // Prime the Pump
    UCA0IE |= UCRXIE;               // enable receive interrupt
}

//-------------------------------------------------------------------------------------------------
/*
 *  Function: Init_Serial_UCA1(void);
 *
 *  Description: Initialization for UCA1
*/
//-------------------------------------------------------------------------------------------------
void Init_Serial_UCA1(char speed){

    // Configure eUSCI_A0 for UART mode
    UCA1CTLW0 = 0;                  // Use word register
    UCA1CTLW0 |= UCSWRST;           // Put eUSCI in reset
    UCA1CTLW0 |= UCSSEL__SMCLK;     // Set SMCLK as fBRCLK
    UCA1CTLW0 &= ~UCMSB;            // MSB, LSB select
    UCA1CTLW0 &= ~UCSPB;            // MSB, LSB select
    UCA1CTLW0 &= ~UCPEN;            // UCSPB = 0(1 stop bit) OR 1(2 stop bits)
    UCA1CTLW0 &= ~UCSYNC;           // No Parity
    UCA1CTLW0 &= ~UC7BIT;
    UCA1CTLW0 |= UCMODE_0;

    // set baud sate
    if (speed == '1') {
        UCA1BRW = 4;                // 115, 200 baud rate
        UCA1MCTLW = 0x5551;
    }
    else if (speed == '2'){
        UCA1BRW = 17;               // 460,800 baud rate
        UCA1MCTLW = 0x4A00;
    }
    else if (speed == '3') {
        UCA1BRW = 26;               // 19,200 baud rate
        UCA1MCTLW = 0xB601;
        speed = '1';
    }
    else {
        UCA1BRW = 4;
        UCA1MCTLW = 0x5551;
    }
    UCA1CTLW0 &= ~UCSWRST;          // release from reset
    UCA1TXBUF = 0x00;               // Prime the Pump
    UCA1IE |= UCRXIE;               // enable receive interrupt
}
```

9.6. Interrupt_ADC.c

```
/*
 *  Interrupt_ADC.c
 *
 *  Description: ADC configuration
 *
 *  globals defined: ADC_Channel, ADC_Right_Detect, ADC_Left_Detect, ADC_Thumb, adc_char
 *
 *  Built with Code Composer Version: CCS12.4.0.00007_win64
 *  Created on: Oct 11, 2023
 *      Author: Martina Viola
 */

// Include Directives
#include  "msp430.h"
#include  <string.h>
#include  "functions.h"
#include  "LCD.h"
#include  "ports.h"
#include  "macros.h"

// Global Variables
unsigned int ADC_Channel;
unsigned int ADC_Right_Detect;
unsigned int ADC_Left_Detect;
unsigned int ADC_Thumb;
char adc_char[4];

// External Global Variables
extern char display_line[4][11];
extern char *display[4];
extern volatile unsigned char display_changed;

//-------------------------------------------------------------------------------------------------
// Hex to BCD Conversion
// Convert a Hex number to a BCD for display on an LCD or monitor //
//-------------------------------------------------------------------------------------------------
void HEXtoBCD(int hex_value){
    int i;
    int value;
    for(i=0; i < 4; i++) {
        adc_char[i] = '0';
    }
    while (hex_value > 999){
      hex_value = hex_value - 1000;
      value = value + 1;
      adc_char[0] = 0x30 + value;
    }
```

```
    value = 0;
    while (hex_value > 99){
        hex_value = hex_value - 100;
        value = value + 1;
        adc_char[1] = 0x30 + value;
    }
    value = 0;
    while (hex_value > 9){
        hex_value = hex_value - 10;
        value = value + 1;
        adc_char[2] = 0x30 + value;
    }
    adc_char[3] = 0x30 + hex_value;
}

//-------------------------------------------------------------------------------------------------
// ADC Line insert
// Take the HEX to BCD value in the array adc_char and display on the LCD
//-------------------------------------------------------------------------------------------------
void adc_line(unsigned int line, unsigned int location){
    int i;
    unsigned int real_line;
    real_line = line - 1;
    for(i=0; i < 4; i++) {
        display_line[real_line][i+location] = adc_char[i];
    }
}


//-------------------------------------------------------------------------------------------------
// ADC Interrupt Routine
#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
                                            // before its previous conversion result was read.
        break;
    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 0x00:                                  // Channel A2 Interrupt
```

```
            ADC_Left_Detect = ADCMEM0;                  // Move result into Global Values
            ADC_Left_Detect = ADC_Left_Detect >> 2;     // Divide result by 4
            HEXtoBCD(ADC_Left_Detect);                  // Convert result to string
            adc_line(3, 1);                             // Place string in display
            display_changed = TRUE;
            ADCMCTL0 &= ~ADCINCH_2;                     // Disable Last channel A2
            ADCMCTL0 |=  ADCINCH_3;                     // Enable Next channel A3
            ADCCTL0 |= ADCSC;                           // Start next sample
            break;
        case 0x01:                                      // Channel A3 Interrupt
            ADC_Right_Detect = ADCMEM0;                 // Move result into Global Values
            ADC_Right_Detect = ADC_Right_Detect >> 2;   // Divide result by 4
            HEXtoBCD(ADC_Right_Detect);                 // Convert result to string
            adc_line(3, 6);                             // Place string in display
            display_changed = TRUE;
            ADCMCTL0 &= ~ADCINCH_3;                     // Disable Last channel A3
            ADCMCTL0 |= ADCINCH_5;                      // Enable Next channel A5
            ADCCTL0 |= ADCSC;                           // Start next sample
            break;
        case 0x02:
            ADC_Thumb = ADCMEM0;                        // Move result into Global Values
            ADC_Thumb = ADC_Thumb >> 2;                 // Divide result by 4
            HEXtoBCD(ADC_Thumb);                        // Convert result to string
            adc_line(4, 6);                             // Place string in display
            display_changed = TRUE;
            ADCMCTL0 &= ~ADCINCH_5;                     // Disable Last channel A5
            ADCMCTL0 |= ADCINCH_2;                      // Enable first channel A2
            ADC_Channel = 0;                            // Set state back to beginning channel
            break;
        default:
                break;
        }
        ADCCTL0 |= ADCENC;                              // Enable Conversions
        break;
    default:
      break;
    }
}
```

This document contains information that is **PRIVILEGED** and **CONFIDENTIAL**; you are hereby notified that any dissemination of this information is strictly prohibited.	Date: **12 - 07 - 2023**	Document # **0000-0001**	Rev: **C**	Sheet:

9.7. Interrupt_Serial.c

```
/*
 *  interrupt_serial.c
 *
 *  Description: Defines interrupt service routines for transmitting and receiving data through the
 *               serial communications port
 *  globals defined: none
 *
 *  Built with Code Composer Version: CCS12.4.0.00007_win64
 *  Created on: Nov 2, 2023
 *      Author: Martina Viola
 */

// Include Directives
#include  "msp430.h"
#include  <string.h>
#include  "functions.h"
#include  "LCD.h"
#include  "ports.h"
#include  "macros.h"
#include "variables.h"


//-------------------------------------------------------------------------------------------------
/*
 * Function: eUSCI_A0_ISR(void);
 *
 * Description: Interrupt for to serial port UCA0 (IOT); transmits and receives through UCA0
*/
//-------------------------------------------------------------------------------------------------

#pragma vector=EUSCI_A0_VECTOR
__interrupt void eUSCI_A0_ISR(void){
    char iot_receive;
    unsigned int temp;
    switch(__even_in_range(UCA0IV,0x08)){
        case 0:                                         // Vector 0 - no interrupt
            break;
        case 2:
            iot_receive = UCA0RXBUF;                    // char copied to temp variable
            IOT_Ring_Rx[iot_rx_wr++] = iot_receive;     // character added to the next position
            if(iot_rx_wr >= sizeof(IOT_Ring_Rx)){
                  iot_rx_wr = BEGINNING;                // reset the write pointer
            }
            break;
        case 4:
            UCA0TXBUF = IOT_Tx_buf[iot_tx];             // copy char from transmit buff
            IOT_Tx_buf[iot_tx++] = NULL;                // erase character from transmit buff
            if(IOT_Tx_buf[iot_tx] == NULL){             // if the next character is null
                UCA0IE &= ~UCTXIE;                      // disable transmit interrupt
```

```
                iot_tx = 0;
            }
            break;
        default:
            break;
    }
}

//=====================================================================================================
/*
 * Function: eUSCI_A1_ISR(void);
 *
 * Description: Interrupt for to serial port UCA1 (USB); transmits and receives through UCA1
 */
//=====================================================================================================
#pragma vector = EUSCI_A1_VECTOR
__interrupt void eUSCI_A1_ISR(void){
    char usb_value;
    unsigned int temp;
    switch(__even_in_range(UCA1IV,0x08)){
        case 0:
            break;                                      // Vector 0 - no interrupt

        case 2:                                         // receive interrupt
            usb_value = UCA1RXBUF;                      // character copied to temp variable
            USB_Ring_Rx[usb_rx_wr++] = usb_value;       // character is added to ring buffer
            if(usb_rx_wr >= sizeof(USB_Ring_Rx)){
                usb_rx_wr = BEGINNING;
            }

            Break;

        case 4:                                         // Vector 4 - TX1IFG: transmit
            UCA1TXBUF = USB_Tx_buf[usb_tx];             // copy transmit buff char to UCA0TXBUF
            USB_Tx_buf[usb_tx++] = 0;                   // erase character from transmit buff
            if(USB_Tx_buf[usb_tx] == NULL){
                UCA1IE &= ~UCTXIE;                      // disable transmit interrupt
                usb_tx = 0;
            }
            break;
        default:
            break;
    }
}
```