PITA Car
Loose Screws Inc.

Innovating the Future of Engineering

Team Members
Keya Desai
Jennifer Jimenez
Caden Johnson
Tanishq Shendokar
Zain Utterback

Originator: Author: Loose Screws Inc.		
Checked: Loose Screws	Released:04/27/2024	
Filename: Loose Screws Inc. Write Up		
Title:	**PITA Car** **Loose Screws Inc.**	

Date:	Document Number:	Rev:	Sheet:
2/12/2025	**5-2024-306-3902-16**	**5**	**1 of 46**

Revision	Section	Author	Description
1			Individual write-ups merged in revision 2
2	**Section**	**Author**	**Description**
	Title	Tanishq Shendokar	Updated title page
	Revision	Zain Utterback	Created Revision Table
	Footer	Zain Utterback	Updated date, team members, document number, revision number, and file name.
	1	Caden Johnson	Summarized the document
	2	Tanishq Shendokar	Added abbreviations
	3	Tanishq Shendokar	Updated block diagrams for Overview
	3.1	Tanishq Shendokar	Described LCD board and added block diagram and picture
	3.2	Tanishq Shendokar	Described Power board and added block diagram and picture
	3.3	Tanishq Shendokar	Described MSP430 and added diagram
	3.4	Tanishq Shendokar	Described FET Board and included circuit diagram
	3.4	Keya Desai	Added to FET Board Description
	3.5	Keya Desai	Described User Input
	3.5	Caden Johnson	Described User Interface
	3.5	Tanishq Shendokar	Described UI and included block diagram
	4	Tanishq Shendokar	Added Hardware descriptions
	5	Keya Desai	Added Power Analysis
	6	Jennifer Jimenez	Updated section 6.1, 6.2, 6.3, and figure table
	6.1	Jennifer Jimenez	Described Power System Test and added picture
	6.2	Jennifer Jimenez	Described Switch Test and added picture
	6.3	Jennifer Jimenez	Described Wheels Error Test and added pictures
	7	Caden Johnson	Brief software summary
	8	Zain Utterback	Created Flowchart
	8.1	Zain Utterback	Described Main Block
	8.2	Zain Utterback	Described Initialization Blocks
	8.3	Zain Utterback	Described Process Blocks
	9	Zain Utterback	Pasted provided code
	10	Keya Desai	Summarized what was learned through devices creation
	10	Caden Johnson	Added to summary
3	3	Caden Johnson	Moved descriptions
	3.6	Tanishq Shendokar	DAC Board Information
	4.4	Caden Johnson	Combined descriptions
	6.4	Caden Johnson	Described FET Testing
	6.4	Jennifer Jimenez	Added Figure and reformatted
	6.5	Jennifer Jimenez	Described the testing of IR LED and detectors
	8	Keya Desai	ADC flow Chart
	8.1	Zain Utterback	Added flowchart for main
	8.1	Zain Utterback	Updated main description for project 6
	8.2.1	Caden Johnson	Ports Initializations Flowchart
	8.4.1	Jennifer Jimenez	Switch interrupt flow diagram added
	8.4.2	Keya Desai	ADC Description
	8.2.4	Tanishq Shendokar	Init_Timers

	8.4.3	Tanishq Shendokar	Timer Interrupt Explanation added
	9.1	Zain Utterback	Updated main code for project 6
	9.3	Caden Johnson	Updated ports code
	9.5	Jennifer Jimenez	Interrupt_switch.c code added
	9.5	Zain Utterback	Added more switch code
	9.6	Jennifer Jimenez	ADC.c code added
	9.7	Tanishq Shendokar	Timers.c
	9.7	Zain Utterback	Added more timers code
4	Subject	Zain Utterback	Fixed Subject Line
	6.6	Jennifer Jimenez	Serial communication testing added
	7	Tanishq Shendokar	Updated section with init_serial information
	8.2.1	Caden Johnson	Updated ports flowchart
	8.4.1	Keya Desai	Updated Switch Functionality
	8.2.5	Zain Utterback	Added Serial Initialization flowchart
	8.2.5	Tanishq Shendokar	Added serial init description
	8.4.4	Caden Johnson	Added serial communication interrupt flowchart and description
	9.3	Caden Johnson	Updated ports code
	9.8	Caden Johnson	Serial interrupt code added
	9.9	Tanishq Shendokar	Added dac code
	10	Keya Desai	Updated conclusion
5	3.6	Jennifer Jimenez	ADC Diagram
	3.6	Tanishq Shendokar	Updated IR Detector informations
	3.7	Jennifer Jimenez	Added Serial communication block
	4.8	Tanishq Shendokar	Added ADC information
	4.9	Jennifer Jimenez	Added Hardware description for IOT
	5	Zain Utterback	Added Power Calculation for FRAM, IOT, IR, and LCD
	6.7	Jennifer Jimenez	Added Command Structure testing
	6.8	Jennifer Jimenez	Added IOT Testing
	7	Caden Johnson	Updated software description for project 10
	8.1	Caden Johnson	Main flow chart and description
	8.2.1	Caden Johnson	Updated ports
	9.6	Tanishq Shendokar	Updated timers.c
	10	Keya Desai	Updated conclusion
	Misc.	Zain Utterback	Formatted document uniformly

Table of Contents

Figure List

1 Scope

This document describes the PITA Car. It goes through all aspects of the vehicle. First, the document goes through an overview of the variety of components the vehicle has. This includes details about their mounting order. Next, the document goes through the hardware of the vehicle describing each part thoroughly. From here the document talks about all the test processes the vehicle went through during its creation. Finally, the last few sections of the document go through the software behind the PITA Car. First, a description of the code organization is given followed by flow charts displaying the flow of logic for the software itself. Finally, there are copies of the actual code from specific functions within the software, and the conclusions we drew from creating this vehicle.

2 Abbreviations

	Abbreviation	Definition
1	A	Amps
2	AD2	Analog Discovery 2
3	CRO	Oscilloscope
4	DC	Direct Current
5	FET	Field Effect Transistor
6	FRAM	Ferro-Electric Random Access Memory
7	J	Jumper
8	LED	Light-emitting diode
9	MCU	Microcontroller
10	MSP430	Launchpad™ Development Kit made by Texas Instruments
11	LCD	Liquid-crystal display
12	PCB	Printed Circuit Board
13	V	Volts
14	IC	Integrated Circuit
15	P FET	P Channel MOSFET
16	N FET	N Channel MOSFET
17	S1	Switch 1 on the MSP430
18	S2	Switch 2 on the MSP430
19	SW1	Switch 1 one the power board
20	TP	Test points
21	UART	Universal Asynchronous Receiver Transmitter
22	OA2O	Operational Amplifier output
23	OA2N	Operational Amplifier output
24	OA2P	Operational Amplifier positive input
25	SMCLK	Sub-Main Clock
26	DAC	Digital to Analog Converter
27	IOT	Internet of Things
28	GRN	Green
29	EN	Enable
30	IR	Infrared
31	TX	Transmit
32	RX	Receive
33	PWM	Pulse Width Modulation

3 Overview

The system consists of a FRAM board and a Power board with an LCD screen. On the FRAM board, the code is stored, there are LEDs and 3 buttons. Each button changes the text displayed on the LCD screen. The Power board has an on/off switch and connects to the battery pack and the LCD screen.

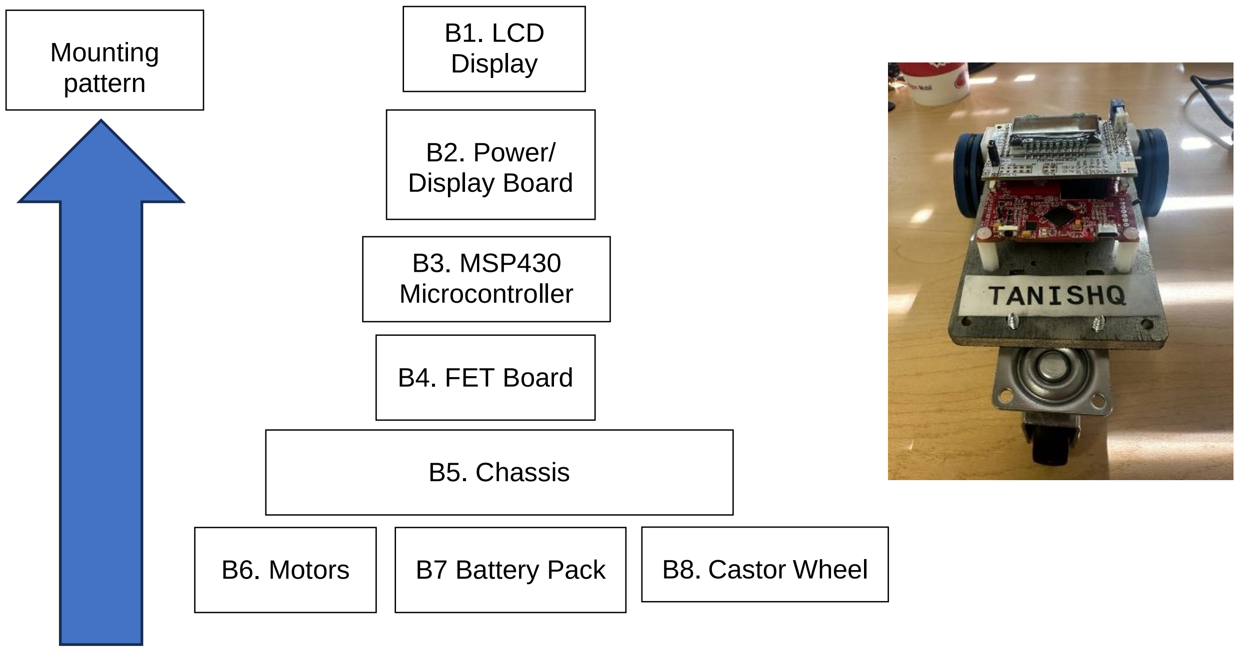

Figure 1 Block diagram.

3.1 LCD Display Board

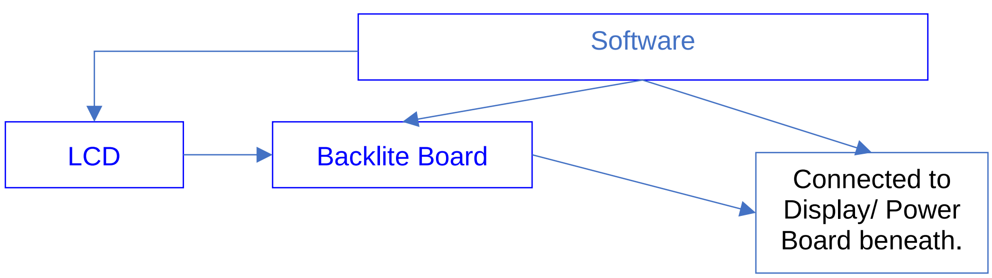

Figure 2 LCD Display Board

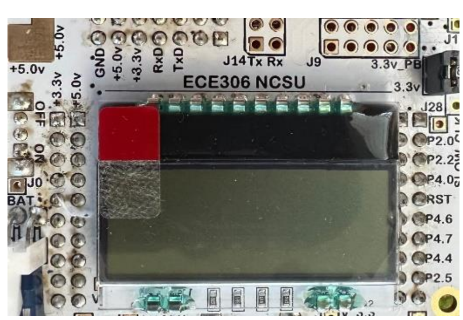

Figure 3 LCD Display Image

3.2 Power Display Board

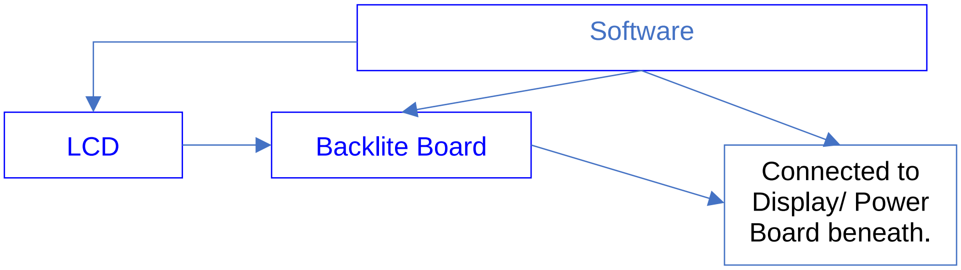

Figure 4 Power Display Board Block

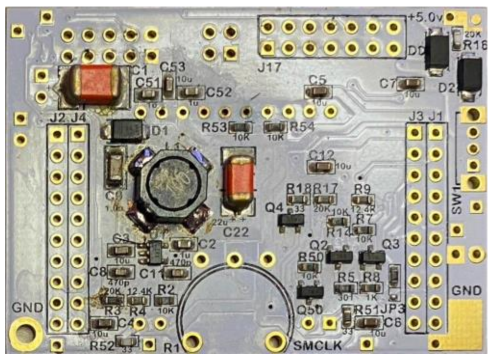

Figure 5 Power Display Board Image

3.3 MSP430 Microcontroller

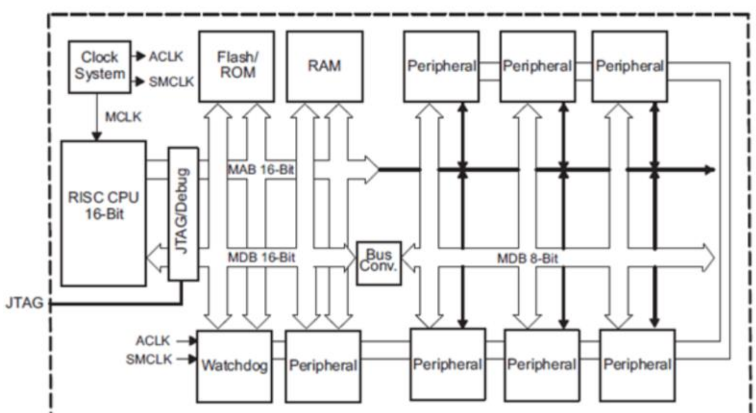

Figure 6 Input Block

3.4 FET Board

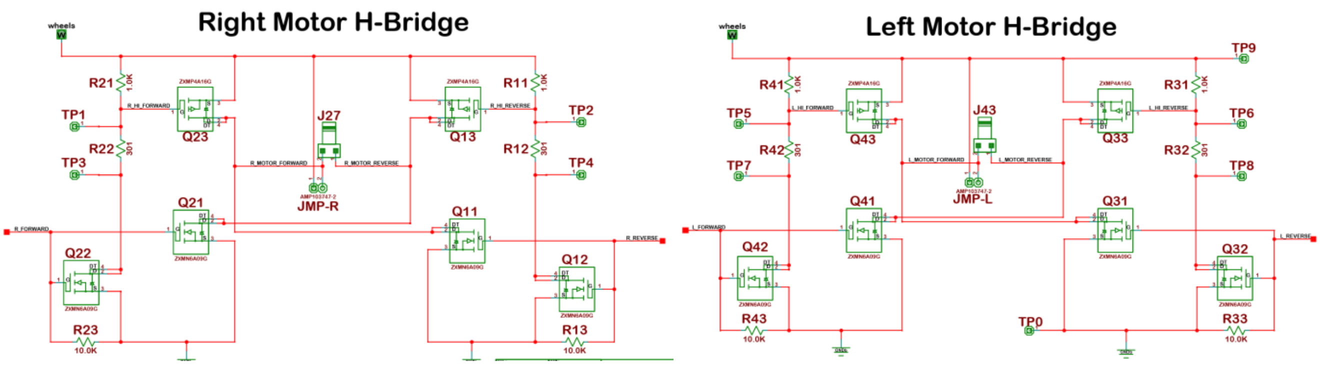

Figure 7 FET Board Circuit

3.5 User Interface

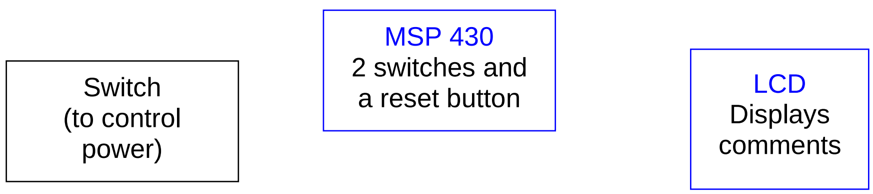

Figure 8 UI Block Diagram

3.6 ADC Board

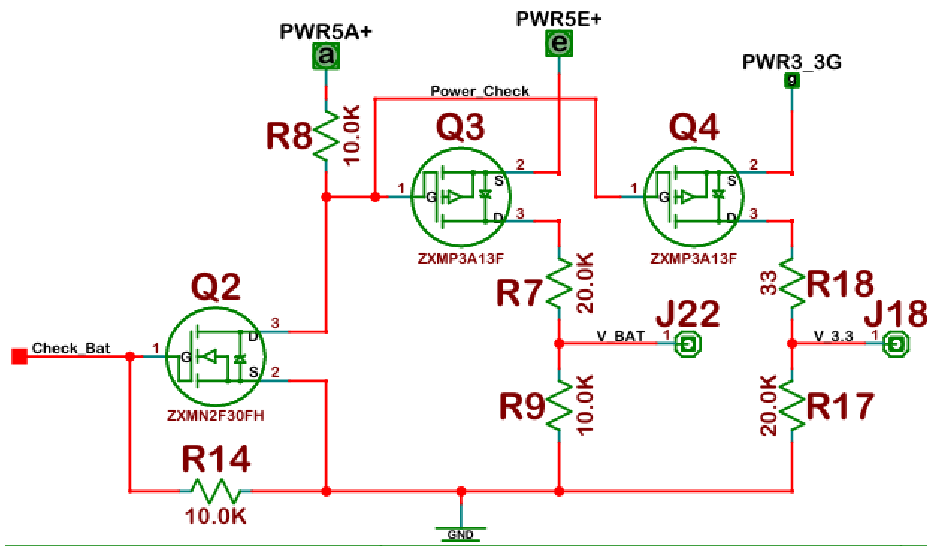

Figure 9 ADC Circuit

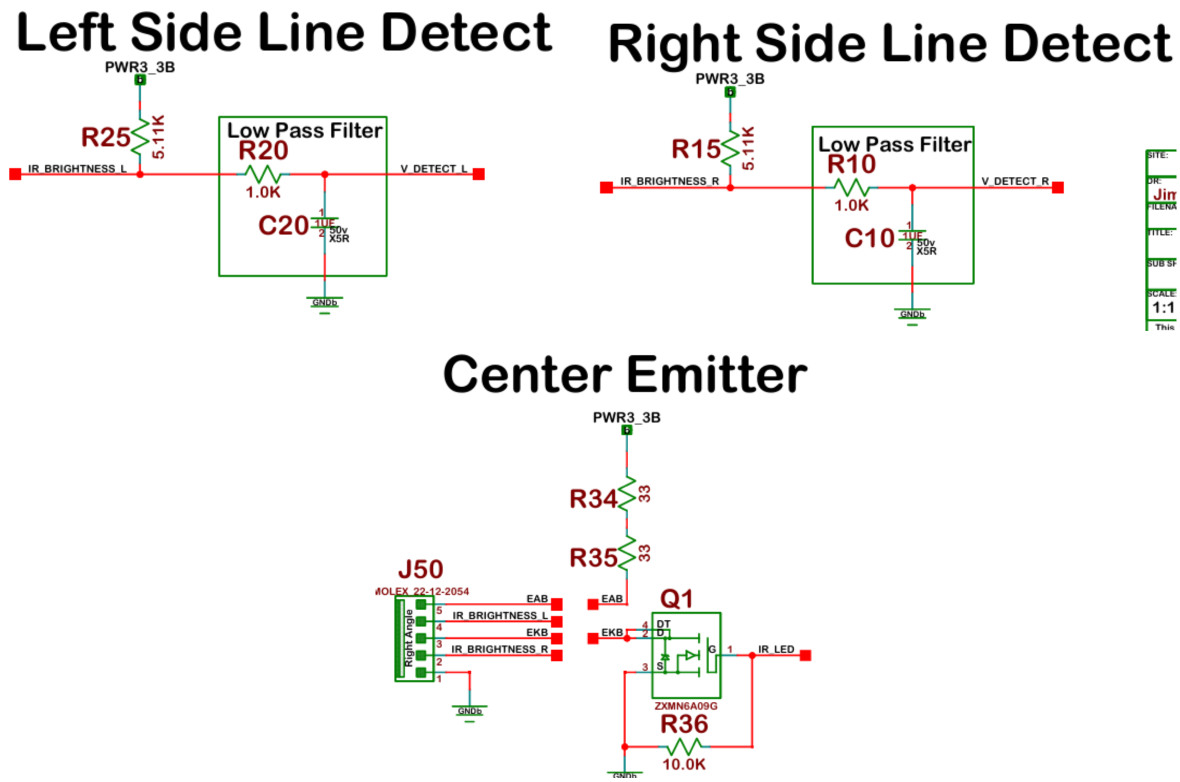

Figure 9.1 IR Detector Circuit

3.7 Serial Communications

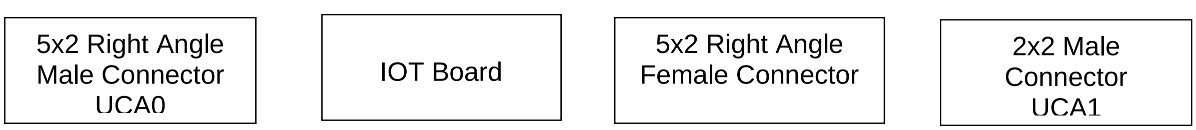

Figure 1010 Serial Communication Block

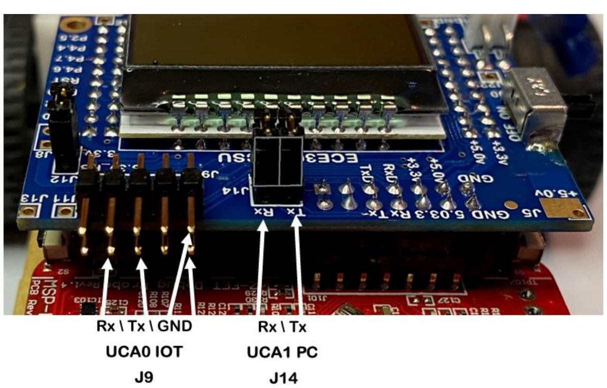

Figure 1111 IOT Board

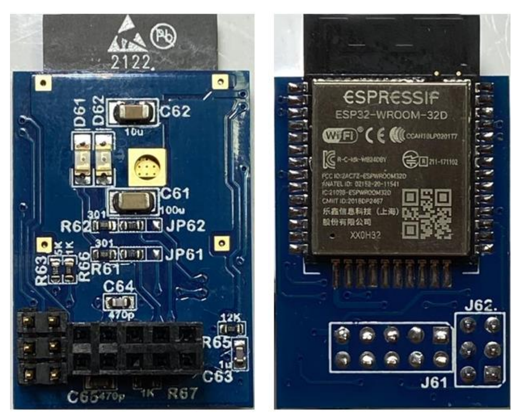

Figure 1212 Serial Communication Ports

4 Hardware

The hardware for this car consists of a laser cut wooden chassis upon which the MSP 430 is mounted and we is connected to the FET Board and Display Board. On the sides of the chassis are two motor which control the movement of the car and Coat Hanger hook to provide stability. The wheels mount to the two motors on the side of the car. Underneath the Chassis is a battery pack strapped on which powers the entire car.

4.1 LCD Display Board

The LCD Display Board consists of an LCD Screen mounted on top of a Backlite board which mounts on top of the LCD Display/ Power Board. There are 14 pins connected to the LCD Display and the Backlite board which mainly control what is displayed on the LCD and control the Backlite as well. These signals come in through the Power/ Display Board and the MSP430 pins govern the functioning of the display.

4.2 Display/ Power Board

The Power and Display board gets power from a battery pack mounted at the bottom of the car which is connected with wires to the Battery input terminal on the board. It performs two major functions; one is to regulate the power received from the battery pack and uses a Buck Boost Converter to restrict the voltage to 3.3V which is used to operate most of the car's functions. The second function is to run the signals for the Display through its necessary circuit and feed them to the LCD Display Board pins.

4.3 MSP430 Microcontroller

The MSP430 Microcontroller works at the heart of this project which provides all the output signal, computes all the calculations, and controls the peripherals.

MSP430FR235x microcontrollers (MCUs) are part of the MSP430™ MCU value line portfolio of ultra-low-power low-cost devices for sensing and measurement applications. MSP430FR235x MCUs integrate four configurable signal-chain modules called smart analog combos, each of which can be used as a 12-bit DAC or a configurable programmable-gain Op-Amp to meet the specific needs of a system while reducing the BOM and PCB size. The device also includes a 12-bit SAR ADC and two comparators. The MSP430FR215x and MSP430FR235x MCUs all support an extended temperature range from–40° up to 105°C, so higher temperature industrial applications can benefit from the devices' FRAM data-logging capabilities. (According to the datasheet for the microcontroller

4.4 FET Board

The FET Board contains the left motor H bridge and Right motor H bridge which consist of 4 P-FETS and 9 N-FETS controlling the forward and reverse travel of the motors. Field-Effect Transistors (FETs) are commonly used in motor control circuits to regulate the flow of current to the motors. The main purpose of a FET board so far is configuring a incorporate the necessary FETs and control circuitry to implement the H-bridge configuration for motor control. The FET Board is mounted at the bottom of the MSP 430 Microcontroller and connects to the two motors at the back of the car.

4.5 User Interface

The User Interface includes the SW1, S1, S2, and the LCD. SW1 is connected to the power board between the battery pack and the rest of the board, this switch decides if the rest of the board will get power from the batteries. S1 and S2 are used to take user input. The code is built to read this input and do an action. The LCD displays what the PITA Car is doing.

4.6 Chassis and Motors

The chassis provides the base for the entire car and supports rigidity for the motors and mounting points for the ICs.

4.7 Motors and Battery

We are using 4 1.5V AA batteries for powering the entire car. We are using 2 Mini Plastic Gearmotor 3mm"D" motors.

4.8 ADC Board

An analog-to-digital converter (ADC) is a crucial component for converting analog signals, like those from IR sensors, into digital values that a microcontroller, such as the MSP430FR2355, can process. The MSP430 is equipped with a high-performance 12-bit Analog Digital Converter (ADC). The ADC has a 12-bit resolution, allowing it to convert analog signals into digital values with high precision. This means it can represent analog voltages as digital values ranging from 0 to 4095. The conversion time of the ADC depends on factors such as the clock source and sampling frequency configured in your application. Our program specifically samples at about 5-20ms. The ADC12CTL0 and ADC12CTL1 registers control various aspects of the ADC operation, such as input channel selection, reference voltage, sampling frequency, and conversion mode.

4.9 Serial Communications

The serial communication block consists of a 5x2 right angle male and female connector, a 2x2 male connector, and the IOT board. The right-angle male connector is connected to the Power Display board on one end with the IOT module connected to it using the right-angle female connector. This connects the IOT to the power and ground of the Display board. The other ports are to enable and reboot the IOT board, enable the red and green LEDs on the board, and for the receive and transmit ports (UCA0). The 2x2 male connectors are used for the USB backdoor connection from the PC to the FRAM. These ports can transmit and receive (UCA1) to the FRAM and IOT depending on jumper configurations placed on the four pins on the connector. Jumpers placed parallel to the board act as a loop so that anything transmitted is received back to the PC. Jumpers placed adjacent to the board allow communication to the FRAM and IOT board.

5 Power Analysis

FRAM:

The FRAM operates at 3 volts, and at room temperature has a current draw of 3.084 mA when operating at 8 MHz, based on the FRAM's data sheet.

Since P = V x I, then : P = 3V × 3.084mA = 9.252mW

LCD Display:

The display operates between 3 and 3.6 V, with an average of 3.3 V, and has a supply current of 0.3 mA.

3.3V × 0.3mA = 0.99mW

The backlight also operates at 3.3V, with a current draw of 45mA.

3.3V × 45mA = 148.5mW

Giving us a total power draw of 149.49mW.

IOT:

The IOT module operates at 3.3 V, with a current draw of 500 mA.

3.3V × 500mA = 1650mW

IR Emitter:

The IR Emitter operates at 1.7V, with a current draw of 100mA.

1.7V × 100mA = 1700mW

IR Detectors:

The IR Detectors operate at 1.7V, with a current draw of 16mA per detector.

1.7V × 16mA = 27.2mW

Since there are two detectors, that gives us a total power draw of 54.4mW

Motors:

The motors are low power and operates at a voltage range of 3 to 6 V, we can use an average voltage of 4.5 V for calculations. Based on the data sheet the current draw of the motor is 800 mA (0.8A).

4.5V × 800mA = 3600mW

Total:

The total power draw for the PITA car with all peripherals enabled is as follows:

9.252mW + 149.49mW + 1650mW + 1700mW + 27.2mW + 3600mW ≈ 7136mW

Since there are 4 Batteries, 7135.942mW / 4 = 1784mW per battery

A typical AA contains about 3.9 watt-hours, or 3900mWh.

3900mWh / 1784mW = 2.186 hours

So the car will last about 2 hours running with all peripherals enabled.

6 Test Process

During the creation of our product, tests were conducted at each stage to ensure correct functionality. The first test was to verify that current flowed correctly on the power display board using an oscilloscope. The switch resistance was measured to ensure the on / off switch worked after installation. Motor error code testing was done on the FRAM to ensure the motors worked as programmed.

6.1 Power System Test

A digital storage oscilloscope and a DC power supply were used to test the power supply of the power display board. To set up the oscilloscope, the vertical scale was set to 2 V with the probe ratio being 1:1. While Channel 1 was off, the output voltage was set to 5 V and the limiting current was set to 0.02 A. The output voltage from the power supply was connected to J5 with the ground of the power supply connected to the ground plate. The positive oscilloscope probe measures the desired 3.3 V at J12 with the negative probe connected to the round ground plate as shown in Figure 6. Once Channel 1 is turned on, the capacitor output should measure approximately 3.3 V.

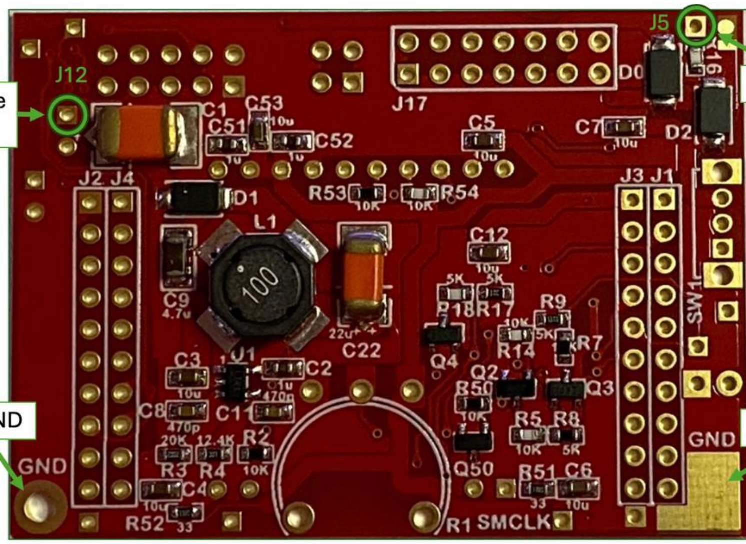

Figure 1313 Power Display Board Testing

6.2 Switch Testing

The switch testing was done to ensure that the system is off in the off position and that it powers on once the switch is flipped on. The current flow was tested on J0 and J5 using a multimeter on the diode setting on the power display board as shown in Figure 7. The positive probe was placed on J0, and the negative probe was placed on J5 to measure the resistance. The switch in the off position should show an open circuit and in the on position, it should measure about 165 ohms.

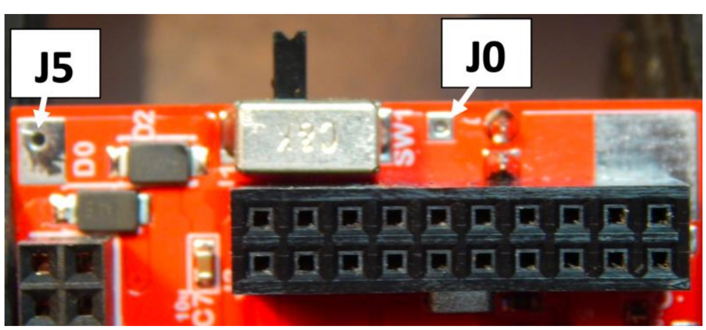

Figure 1414 Switch Testing Points

6.3 Wheels Error Testing

Once the full H-Bridge was installed on the FET board, the FRAM was programmed with a function to turn the wheels off, to move the wheels forward, and to turn the wheels in reverse. The function to turn the wheels off must be called before moving from forward to reverse and vice versa. The AD2 was used to check that the wheels weren't configured to move forward and reverse at the same time before connecting FET board. The shorting jumpers were reinstalled prior to connecting the AD2 on the FRAM. The AD2 ground was connected to GND with D0 plugged into P6.1, D1 into P6.2, D2 into P6.3, and D3 into P6.4 as shown in Figure 8. The AD2 is configured to test whether L_FORWARD and L_REVERSE or R_FORWARD and R_REVERSE were configured to turn at the same time as shown in Figure 9.

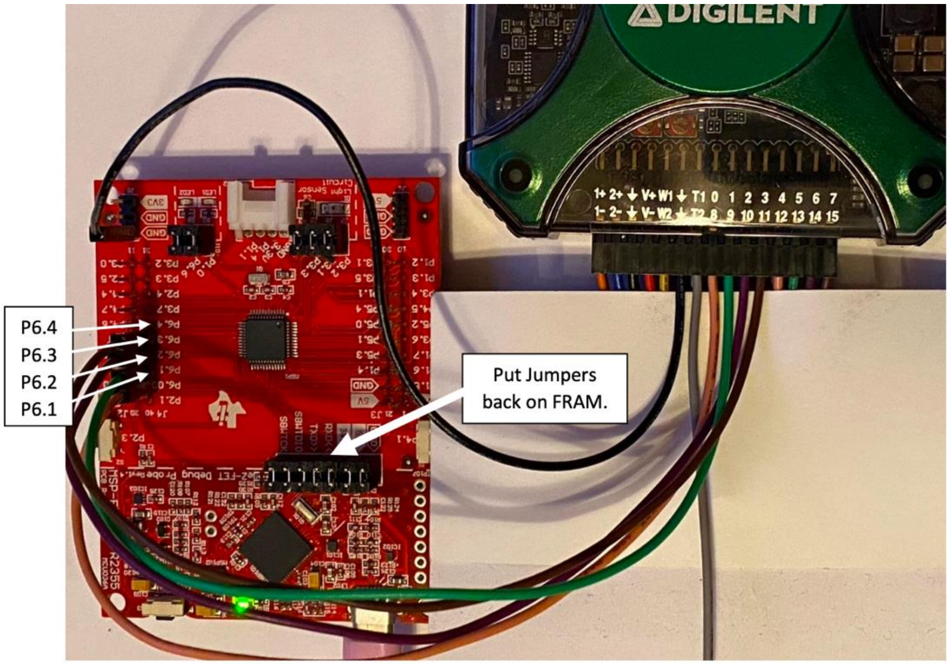

Figure 1415 FRAM Error Testing with AD2

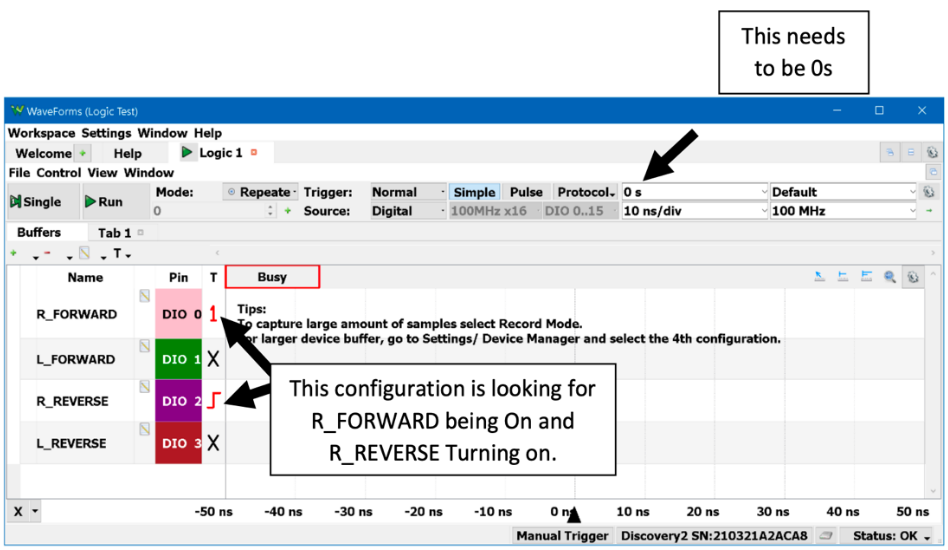

Figure 1516 AD2 Configuration for Error Testing

6.4 FET Testing

Once all the N FETs and P FETs were installed onto the FET board, the FRAM was programmed to have all motor directions set low. This was done by pressing SW1 and SW2, which were configured to turn on one motor control pin. With the FET board, Display board, and FRAM connected and the batteries switched on a voltmeter is used to check the voltage at 2 test points for each motor in all its directions: R_FORWARD, L_FORWARD, R_REVERSE, and L_ REVERSE.

- R_FORWARD is set low TP1 and TP3 should be equal to the battery voltage. When R_FORWARD is turned on TP1 should read half of the battery voltage and TP3 should be about at ground potential.
- 2. L_FORWARD is set low TP5 and TP7 should be equal to the battery voltage. When L_FORWARD turned on TP5 should read half of the battery voltage and TP7 should be about at ground potential.
- R_REVERSE is set low TP2 and TP4 should be equal to the battery voltage. When R_ REVERSE is turned on TP2 should read half of the battery voltage and TP4 should be about at ground potential.
- L_ REVERSE is set low TP6 and TP8 should be equal to the battery voltage. When L_ REVERSE is turned on TP6 should read half of the battery voltage and TP8 should be about at ground potential.

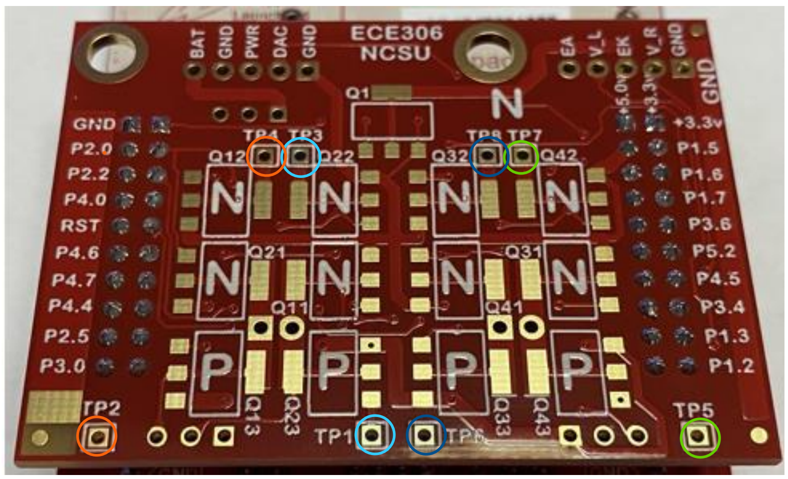

Figure 1617 FET Board

6.5 Black Line Testing

After the IR Led and the two detectors on installed, the black line test was conducted to make sure that the car can detect going from a white surface to a black line. To get accurate movements from the car, the two detectors must be close in value when placed over a black line and over a white surface. The car was placed over the black line and the values displayed on the LCD helped calibrate the sensors. The detector with the lower value was adjusted until it read to a similar value as the higher value detector. The test was repeated until both detectors displayed similar values, if they didn't, the detectors were adjusted so that they faced more toward the IR emitter LED.

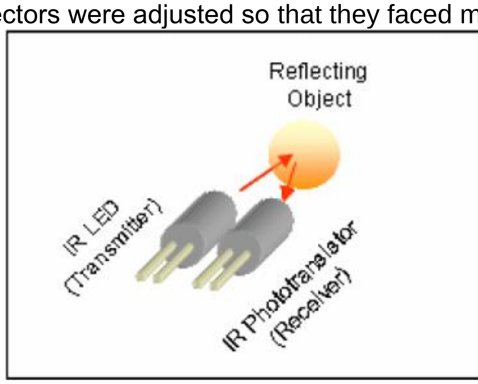

Figure 1718 IR Detection Method

6.6 AD2 Serial Communication Testing

The AD2 was used to test the serial communication of our car before adding the IOT module. The AD2 Digital I/O Signal 0 as Tx was connected to the Rx of the IOT connector. The AD2 Digital I/O Signal 1 as Rx was connected to the Tx of the IOT connector as shown in Figure 15. The ground of the AD2 is connected to the ground of the IOT connector. Six 10-character length messages were transmitted from the transmission window of the AD2. Once SW1 was pressed, the message was transmitted back to the AD2 transmission window. Once SW2 was pressed, the baud rate changes from 115,200 to 460,000 and transmits messages from the AD2 to the car and back.

Figure 1819 AD2 Connections

In the waveform application, we used the protocol window to view the transmitted serial signals. We selected the Send and Receive tab and selected TX echo with the baud rate set to 115,200. The message from the AD2 to the car is echoed in green and any message transmitted back from the car to the AD2 is in black as shown in Figure 16. If the baud rate is incorrect or the code did not function, the transmission window will display nothing or garbage characters.

Figure 1920 AD2 Protocol Settings

6.7 Command Structure Testing

Prior to installing the IOT module, we tested the communication from the IOT serial port to the FRAM using the PC backdoor with jumpers configured as shown in Figure 17. The jumper configuration allows for characters received from one port to be received and transmitted to the other port. A command structure requires a special character to identify the start of a command. In this case, the special character used is the caret ‘^’. To test whether the commands received at the IOT serial port are transmitted to the FRAM, we created a simple command structure. The Termite application was used to send the commands from the PC to the IOT to change the baud rate of our serial communications. Once “^F” is entered in the Termite window, the baud rate is changed to 115,200 and the response is displayed in Termite. Entering “^S” slows down the baud rate with a response in the Termite window of baud rate 9,600. After each command was given, we ensured the car still worked at each baud rate to confirm successful communication from the IOT serial port to the FRAM.

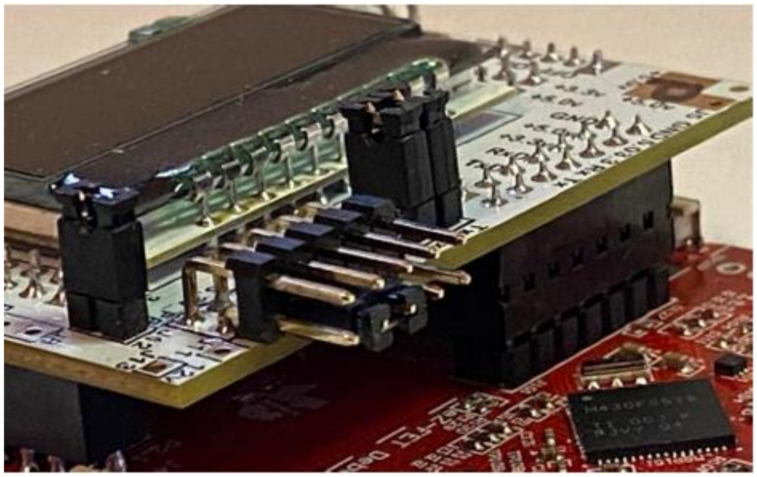

Figure 2021 IOT Serial Port Testing

6.8 IOT Testing

After the IOT module was configured and installed, testing was done to confirm that the IOT module receives commands and that commands from the IOT module to the FRAM work. Using Termite, commands were sent to the IOT module to connect it to the Wi-Fi. Once an “OK” and “Connected to Wi-Fi” was received, the IOT connection to Wi-Fi was confirmed. Additional commands were sent to receive the MAC and IP address of the IOT module. Commands to turn the wheels on and off were added to the command structure, we could test that IOT directed commands work. Using a TCP Client application, commands were sent to the IOT using its IP address to move the wheels of the car. After the car is able to move through a command, the IOT communication to the FRAM is ensured.

7 Software

The software is configured using a modular approach. The main function in main.c sets up specific configurations for the rest of the code and initializes the peripherals. After that it it calls the initialization functions which includes: Init_Ports, Init_Clocks, Init_Conditions, Init_Timers, Init_LCD, Init_Serial, Init_ADC and all over needed initializations are handled. These functions are contained within separate .c files. Next the main function sets the initial content for the display variables for the LCD display. Finally main enters a while loop that goes on forever. This loop is where the main functionality of the program is contained. Within this loop many different functions are called every cycle; these functions include Display_Process, Both_D_On, IOT_Ping, Stop_Watch, IOT_Boot, IOT_Process, and IOT_Commands.

8 Flowcharts

8.1 Main Block

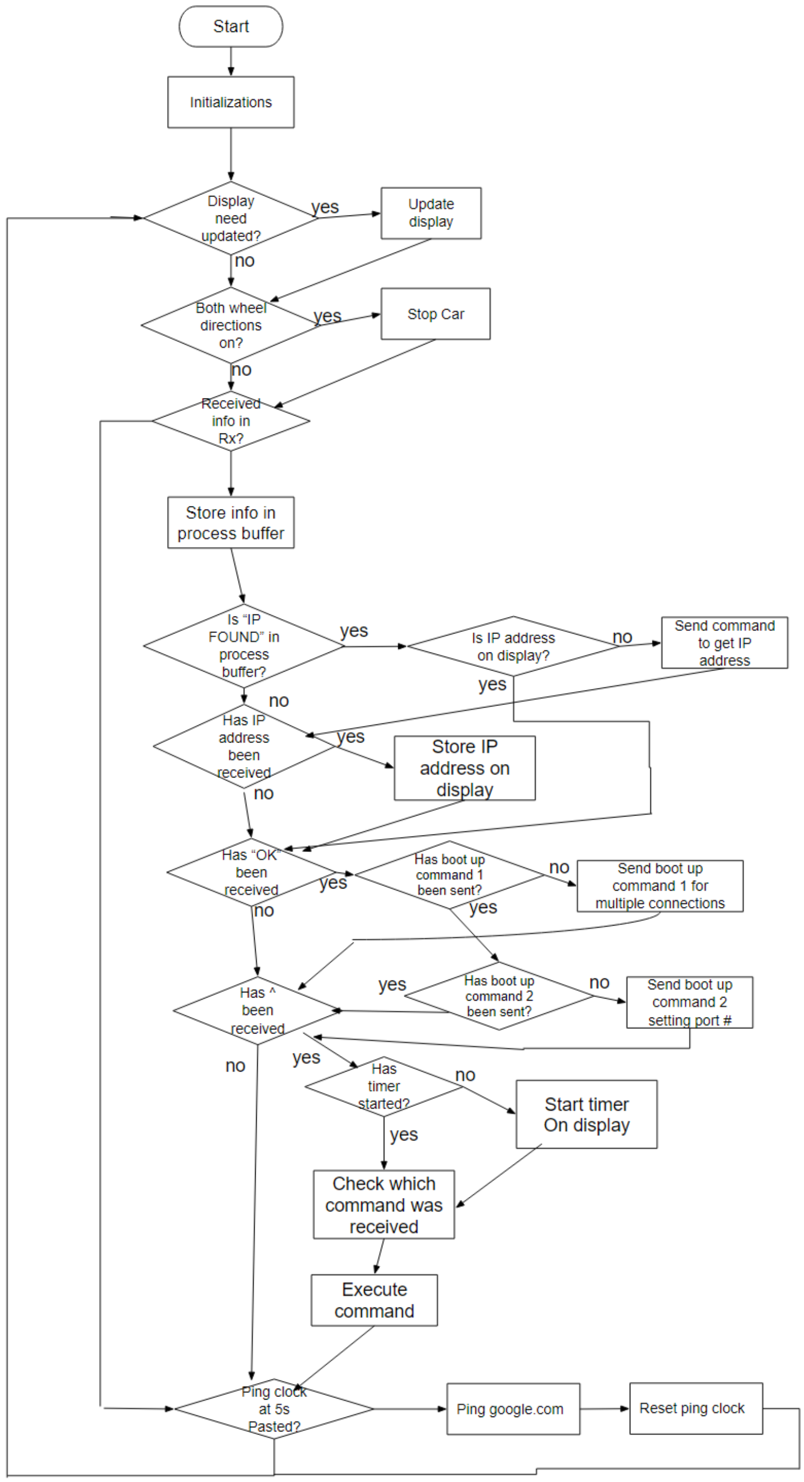

Figure 2122 Main Flowchart

The main block calls all the functions required for the car to run. First are the initialization functions, Init_Ports, Init_Clocks, Init_Conditions, Init_Timers, Init_LCD, Init_Serial, and Init_ADC.

Initializations:

- Init_Ports: Sets the port configurations for the MSP430 Fram Board.
- Init_Clocks: Sets the clock configuration for the program and disables watchdog timer.
- Init_Timers: configures the speed of the various timers used in the program. It also initializes the timer interrupts
- Init_LCD: configure and initializes the LCD display.
- Init_ADC: initializes the ADC registers and configures them for the desired use.
- Init_Serial_UCA1 and Init_Serial_UCA0: Initialize the UCA1 and UCA0 interrupts and configuring UART communication ports.
- Init_Conditions: Initializes the display line variables and reset their states.

In the main while loop many other functions are called including Display_Process, Both_D_On, IOT_Ping, Stop_Watch, IOT_Boot, IOT_Process, and IOT_Commands.

While Loop Functions

- Display_Process: Updates the display when the display variable contains new information.
- Both_D_On: Checks if both motor directions are on, if so, it turns the motor off.
- IOT_Ping: Every 5 seconds ping google.com to stay on the ncsu internet.
- Stop_Watch: Displays stop watch showing seconds passed sense turned on.
- IOT_Boot: Waits for OK messages. Then once received sends 2 commands to set up phone connection with car.
- IOT_Process: Takes the received data from Rx interrupt and stores it into process buffer.
- IOT_Commands: Checks data contained in process buffer for the special character ^. Once found it looks through the full command in process buffer and execute the command with the matching operation.

8.2 Initialization Blocks

8.2.1 Init_Ports

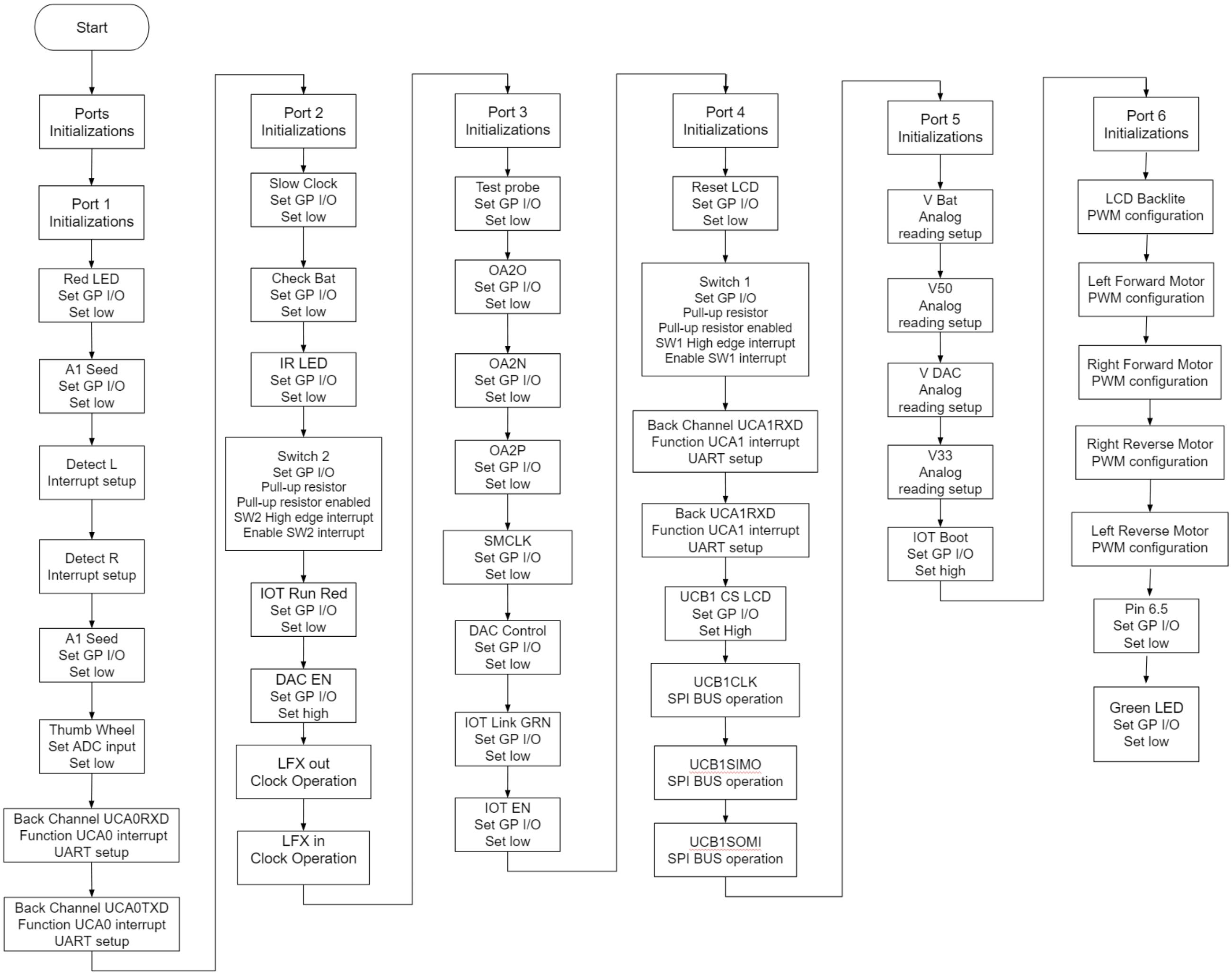

Figure 2223 Ports Flowchart

Init_Ports is called in main.c, and calls 6 functions, Init_Port_1 through 6. Port 1 Initializes 8 pins. Pin 0 is controlling the red LED on the MSP430, setting the direction to output, and setting the output to low. The rest of the pins are initialized to the following operations in order of pin number: A1_SEEED, V_DETECT_L, V_DETECT_R, A4_SEEED, V_THUMB, UCA0RXD, UCA0TXD.

Port 2 initializes 8 pins. Pins 0-2 are set up as GPIO direction output, output low, named SLOW_CLK, CHECK_BAT, and IR_LED, respectively. These will eventually be used to output a slow clock signal, check the battery power, and use the infrared LED. Pin 3 is used for switch 2, the direction is set to input, and the pullup resistor is configured and enabled. Pins 4-5 are set to GPIO direction output, output low, named IOT_RUN_RED, and DAC_ENB respectively. Pins 6-7 are used for the LFX in and out clock operation.

Port 3 initializes 8 pins. Pin 0 is set to GPIO direction output, output low, named TEST_PROBE. Pins 1-3 are initialized for the OA20, OA2-, and OA2+ operations respectively. Pin 4 is initialized to be the SMCLK output operation. Pin 5-7 are initialized to GPIO direction output, output low, and named DAC_CNTL, IOT_LINK_GRN, and IOT_EN respectively.

Port 4 initializes 8 pins. Pin 0 is set to GPIO direction output, output low, named RESET_LCD. Pin 1 is used for switch 1, the direction is set to input, and the pullup resistor is configured and enabled. Pins 2-3 are used for USCI_AI UART operation. Pin 4 is set to GPIO direction output, output high, which disables the UCB1 chip select LCD. Pins 5-7 are other UCB1 operations.

Port 5 initializes 5 pins. Pins 0-4 are used for various voltage operations. Pin 5 is used to boot the IOT module, it is set to GPIO direction output, output low.

Port 6 initializes 7 pins. Pin 0 controls the LCD backlight, and it is set to GPIO direction output, output low. Pins 1-4 control the 2 motors forward and reverse operations, PWM configuration. Pin 5 controls the green LED on the MSP430, set to GPIO direction output, output low.

8.2.2 Init_Clocks

Init_Clocks is called in main.c, and is used to set the initial configurations of the clocks used by the PITA Car. It runs immediately after Init_Ports. The function disables the 1ms watchdog timer, and sets MCLK to 8MHz. It then sets SMCLK to MCLK.

8.2.3 Init_Conditions

Init_Conditions is called in main.c, and is used to set the initial configurations of the array containing the text displayed on the LCD screen, as well as to enable interrupts.

8.2.4 Init_Timers

Init_Timers Consists of 4 different timers, Timer B0, Timer B1, Timer B2 and Timer B3. Currently we do not use Timer B1 and Timer B2.

Timer B0 is the main timer which has been set to run every 50 msec. To achieve this, the 8Mhz MCLK has been divided down twice by 8 which brings it to 125kHz which is further divided by the Capture Compare Register 0 interval which is set to 2500 in macros. This results in a timer which runs every 50 msec. Capture Control Register 1 and Capture Control Register 2 on Timer B0 are currently not set up as they are not required.

To display details properly on the screen, a counter which increments every 4 seconds has been set up which creates a 200msec timer which can be used to run the display.

Timer B3 is currently set up to use the SMCLK to create PWM values to control the speed of the Motors and change the brightness for the display which is all set up to control the outputs of Port 6. The wheel period is set to 50005. Here Capture Control Register 1 is set up to control the brightness of the display, Capture Control Register 2 is set up to control the Right motor forward speed, Capture Control Register 3 is set up to control the Right Motor reverse speed, Capture Control Register 4 is used to control the Left motor forward speed, and Capture Control Register 5 is used to control the Left motor reverse speed.

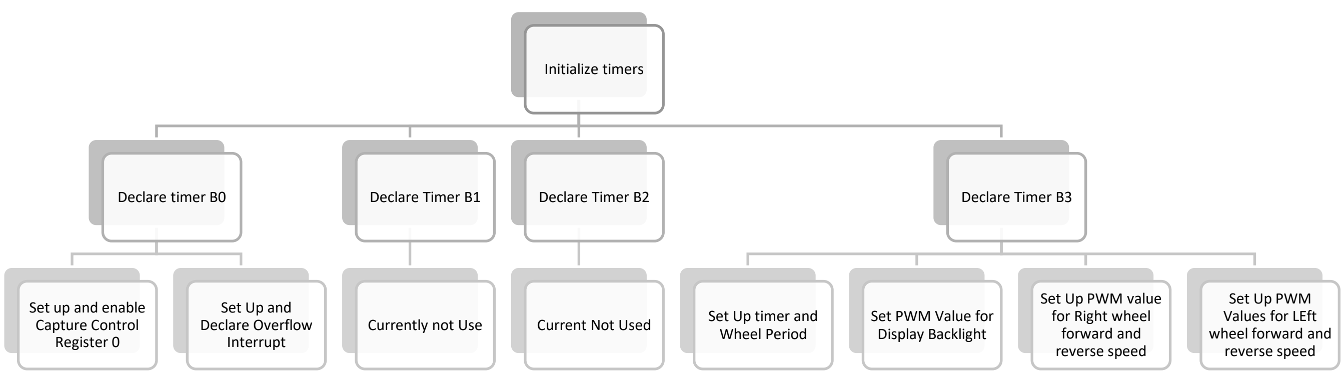

Figure 2324 Timer Flowchart

8.2.5 Init_Serial

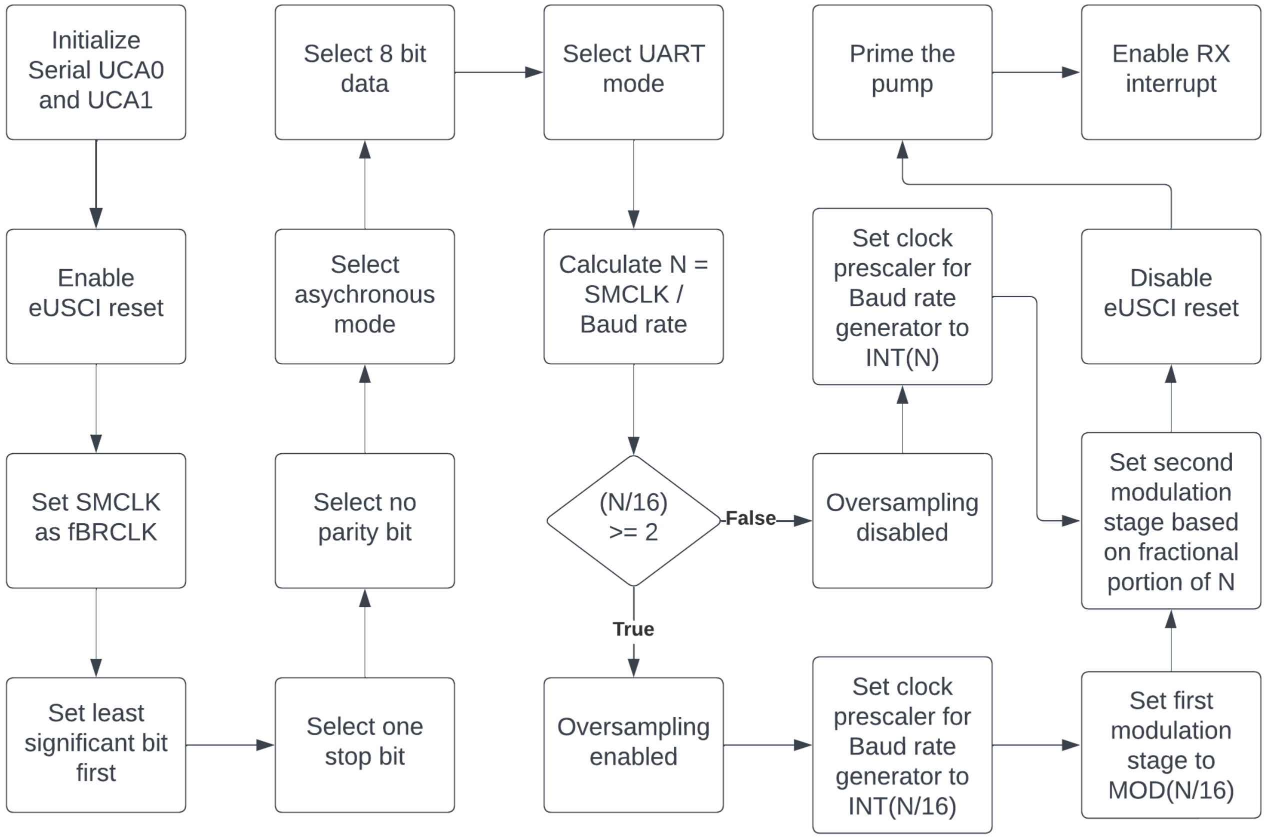

Figure 2425 Serial Initialization Flowchart

This code initializes Universal Asynchronous Receiver/Transmitter (UART) communication for two separate serial ports on a Texas Instruments MSP430 developer board. The Init_Serial_UCA0 function configures settings for UART communication on the UCA0 port, including selecting the clock source as SMCLK (sub-main clock), specifying the baud rate, setting data format parameters such as stop bits and parity, and enabling UART mode. Additionally, interrupt handling for receiving data (RX) is enabled.

Similarly, the Init_Serial_UCA1 function initializes UART communication for the UCA1 port with analogous configurations. Both functions utilize the same settings, differing only in the port number. The baud rate is set to 115,200 bits per second, a common baud rate for UART communication. This code essentially prepares the UART communication interface for sending and receiving data, typically used for communication with external devices such as sensors, displays, or other microcontrollers.

8.3 Process Blocks

8.3.1 Display_Process

Display_Process is called in the while loop in main.c, and it checks to see if it needs to update the display, and if so, it calls Display_Update() in LCD.obj.

8.4 Interrupt Blocks

8.4.1 Switches

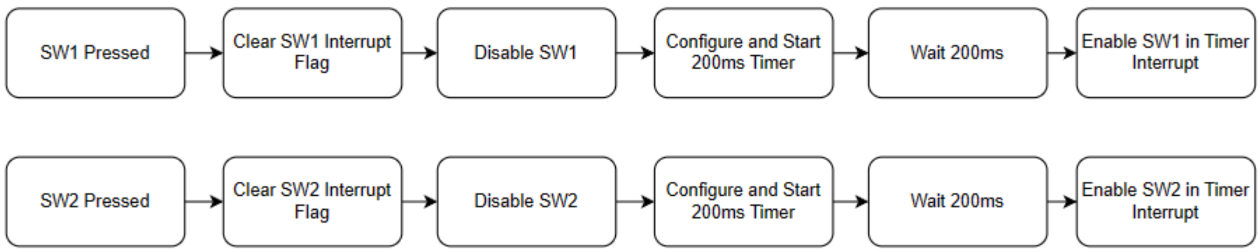

Figure 2526 Switches Interrupt Flowchart

Each switch interrupt service routine occurs when either SW1 or SW2 are pressed. Timer B0 Capture Control Register 1 and Timer B0 Capture Control Register 2 act as the debounce timer interrupts for SW1 and SW2 respectively. Once a switch is pressed, the interrupt flag for the switch is cleared and the switch is disabled. Once the time has passed for the debounce threshold, the switch interrupt is then enabled.

8.4.2 ADC

An analog-to-digital converter is a circuit that accepts an analog input signal, usually a voltage, and produces a corresponding multi-bit number at the output. The ADC core converts an analog input to its 10-bit or 12-bit digital representation and stores the result in the conversion register ADCMEM0. The core uses two programmable and selectable voltage levels (VR+ and VR–) to define the upper and lower limits of the conversion. The ADC core is configured by the control registers ADCCTL0, ADCCTL1, and ADCCTL2, the code configures the ADC module to perform single-channel single-conversion with a 12-bit resolution. It selects Pin 5 (A5) as the input channel and enables the ADC conversion complete interrupt. Finally, it enables and starts ADC conversion.

After writing the HextoBCD and adc_line functions we create an interrupt service routine (ISR) for the ADC conversion complete interrupt. It handles the ADC conversion completion and performs actions based on the converted values.

- Within the switch-case structure:
  It first disables the ADC conversion (**ADCCTL0 &= ~ADCENC**) to prepare for configuration changes.
  It switches based on the **ADC_Channel** value, which likely indicates which ADC channel has completed conversion.
- Depending on the channel: It stores the converted value from **ADCMEM0** into respective global variables (**ADC_Left_Detect**, **ADC_Right_Detect**, **ADC_Thumb_Detect**). It then calls **HEXtoBCD()** to convert the ADC value to BCD. Finally, it calls **adc_line()** to update the display with the converted value. It Repeatedly converts values from three different sources: Left Detector, Right Detector, and Thumb Wheel, changing the channel for each conversion. The conversion is triggered by an interrupt.
  After processing, it re-enables ADC conversions (**ADCCTL0 |= ADCENC**).

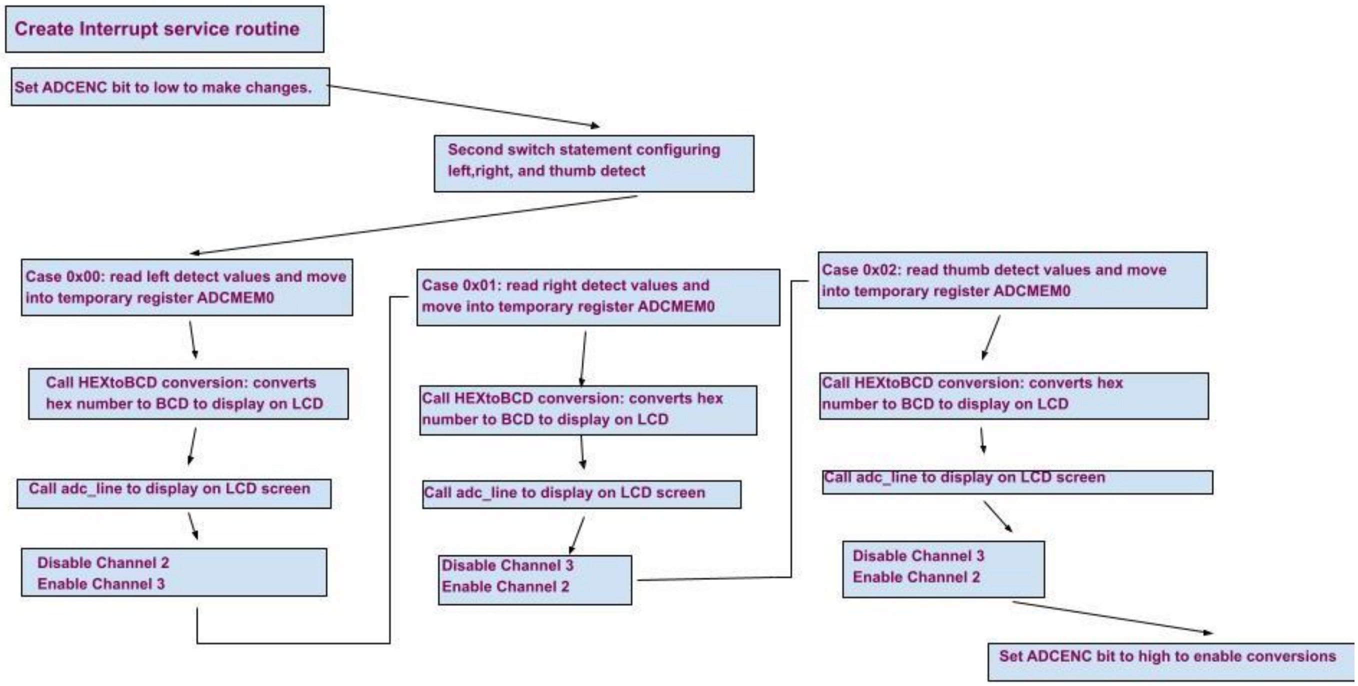

Figure 2627 ADC Interrupt Flowchart

8.4.3 Timers

This TimerB0 interrupt subroutine manages several critical tasks within its execution context. Upon invocation, it increments the Time_Sequence variable, suggesting a sequential event tracking for this clock frequency. Additionally, it controls the Analog-to-Digital Converter (ADC) by enabling conversions and initiating the next sample. Simultaneously, it increments both Time_Count and Time_Turn variables, which are used to time the duration of execution for turn made to align with the line. It also triggers a display update by setting the update_display flag to TRUE every fourth execution, to make sure that the screen is refreshed once every 200msec. Furthermore, it adjusts the Timer0_B0 Capture/Compare register (TB0CCR0) by adding an offset, potentially fine-tuning the timing behavior for subsequent interrupt cycles. This comprehensive approach to managing interrupts underscores the versatility and efficiency of the subroutine in coordinating diverse tasks within the larger program context.

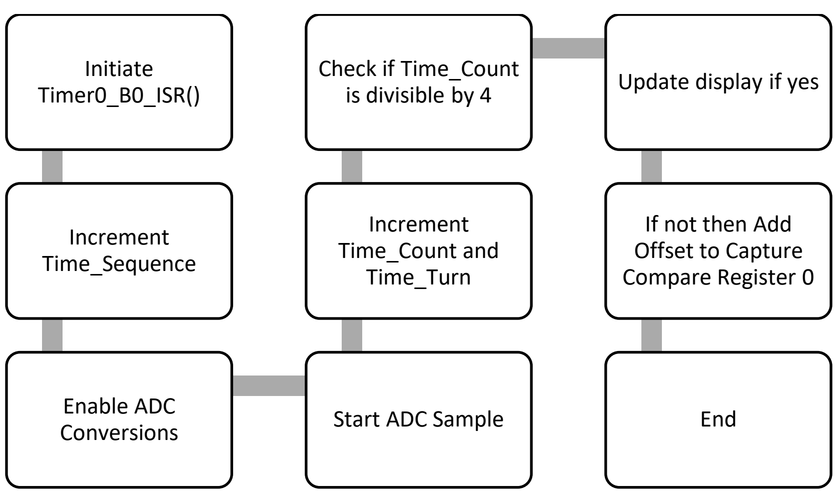

Figure 2728 Timer Interrupt Flowchart

8.4.4 Serial Communication

The serial communication utilized has 8 data bits, 1 stop bit, and no parity. There are 2 interrupt service routines used for receiving and transmitting data. Each of the interrupts works with a different section of pins. The first interrupt is UCA0 which receives serial communications on pin 1.6 and transmits on pin 1.7. These pins are used through J14 on the schematic pins 1 and 3 on the power board. The second interrupt is UCA1 which receives serial communications on pin 4.2 and transmits on pin 4.3. These pins are used through J9 on the schematic pins 5 and 7 on the power board. While the RX interrupt is enabled when serial communication is received the interrupt will trigger causing the characters to be written into a 16-bit character ring buffer; USB_Char_Rx and IOT_Char_Rx. When ready to transmit a character or string the data is moved into a 16-bit character array USB_Char_Tx (UCA1) or IOT_Char_Tx (UCA0). After the data is stored, the TX interrupt is enabled and the data is transmitted to the pin. Once done the TX interrupt is disabled.

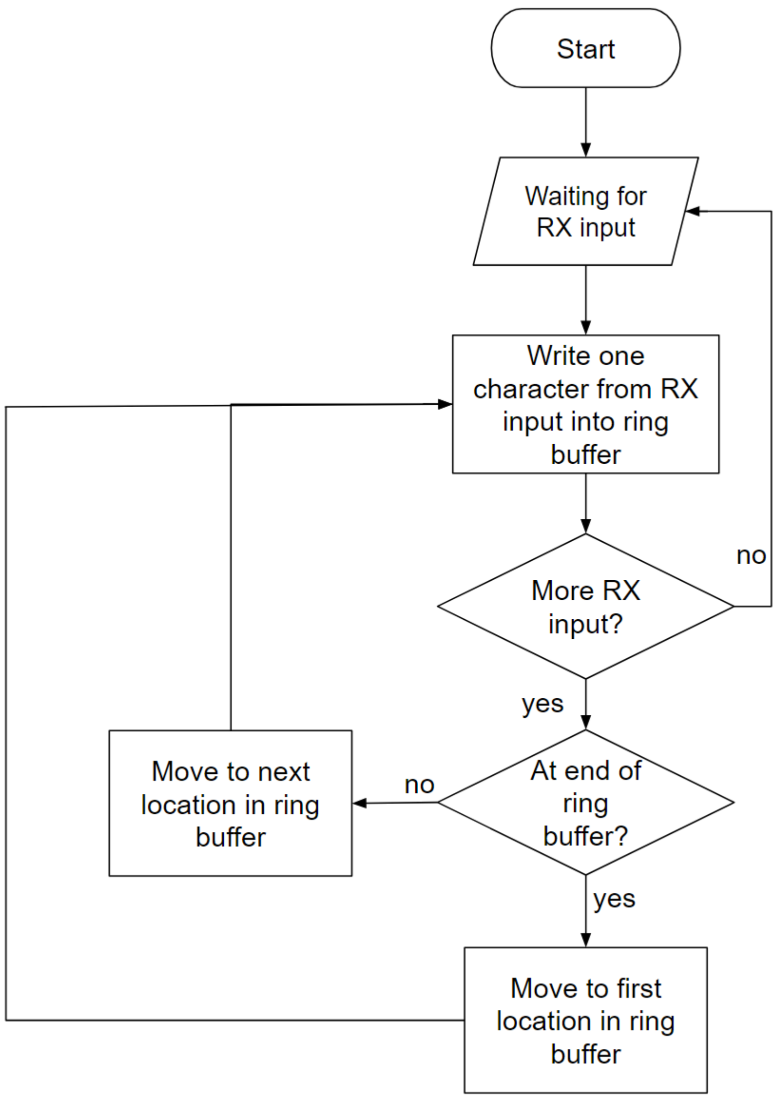

Figure 2829 UCA1 and UCA0 Rx Interrupt Flowchart

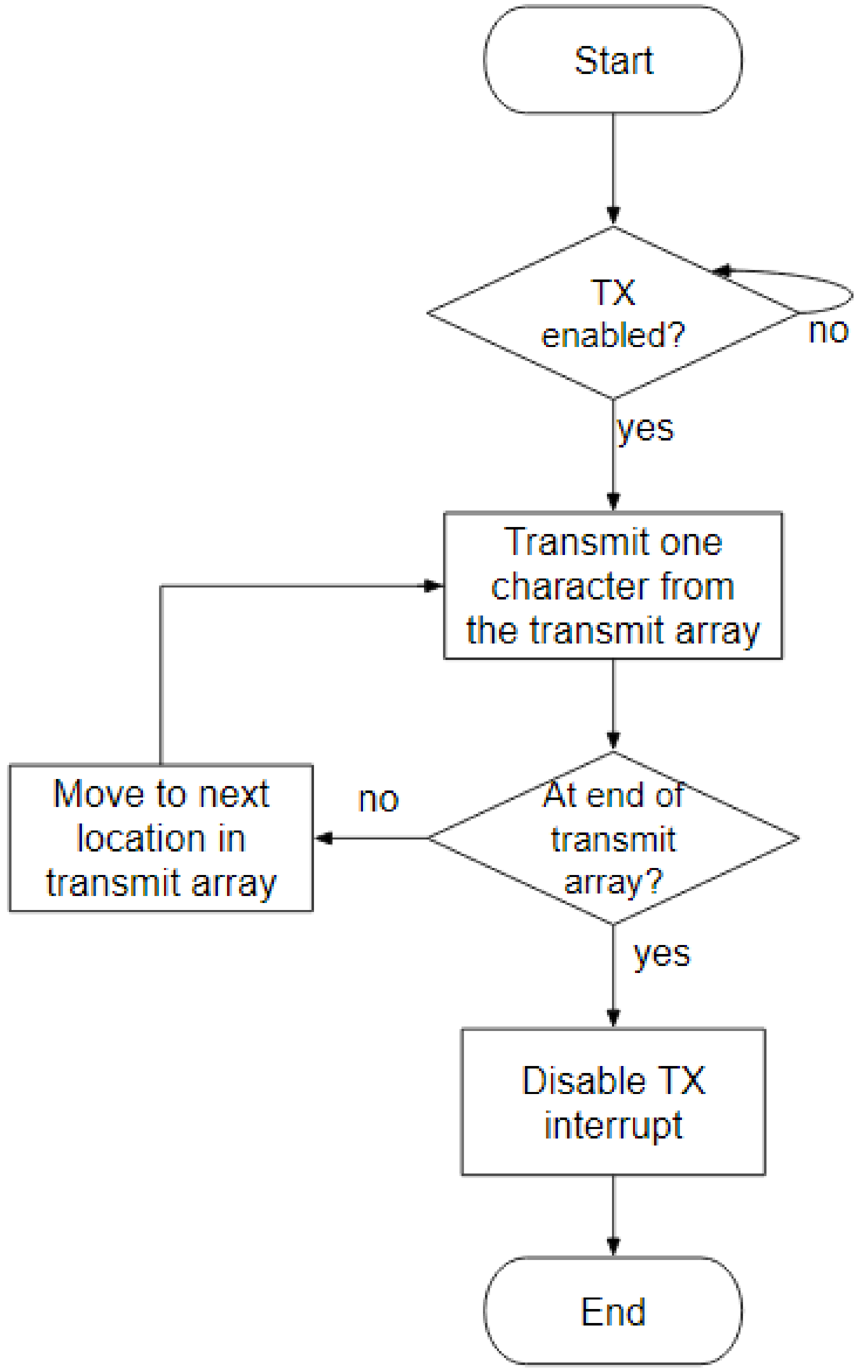

Figure 2930 UCA1 and UCA0 TX Interrupt Flowchart

9 Software Listing

9.1 main.c

```
void main(void){
//    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

//------------------------------------------------------------------------------
// Main Program
// 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_ADC();
    Init_Ports();                        // Initialize Ports
    Init_Clocks();                       // Initialize Clock System
    Init_Conditions();                   // Initialize Variables and Initial Conditions
    Init_LEDs();                         //turns off both LEDs
    Init_Timers();                       // Initialize Timers
    Init_LCD();                          // Initialize LCD
    Init_Serial_UCA1();                  //Initialize UCA1 communication
    Init_Serial_UCA0();                  //Initialize UCA0 communication



  //Initial Display
  strcpy(display_line[0], "Waiting   ");
  strcpy(display_line[1], "          ");
  strcpy(display_line[2], "          ");
  strcpy(display_line[3], "          ");

  display_changed = TRUE;
  update_display = TRUE;




//------------------------------------------------------------------------------
// Begining of the "While" Operating System
//------------------------------------------------------------------------------
  while(ALWAYS) {                       // Can the Operating system run


    Display_Process();                  // Update Display
    Both_D_On();                        // both directions turned on STOP!!!!

    IOT_Ping();                         //ping wifi every 5s

    Stop_Watch();                       //Displayed clock

    IOT_Boot();                         //iot boot up messages
```

```
    IOT_Process();                          //iot process: bring in received messages from the
device
    IOT_Commands();                         //check processed messages for actionable commands


    P3OUT ^= TEST_PROBE;                   // Change State of TEST_PROBE OFF
    }
  }
```

9.2 ports.c

```
void Init_Ports(void){

    Init_Port_1();
    Init_Port_2();
    Init_Port_3();
    Init_Port_4();
    Init_Port_5();
    Init_Port_6();


}

void Init_Port_1(void){
    //-------------------------------------------------------

    //Configure Port 1
    // Port 1 Pins
    // RED_LED (0x01) // 0 RED LED 0
    // V_A1_SEEED (0x02) // 1 A1_SEEED
    // V_DETECT_L (0x04) // 2 V_DETECT_L
    // V_DETECT_R (0x08) // 3 V_DETECT_R
    // V_A4_SEEED (0x10) // 4 V_A4_SEEED
    // V_THUMB (0x20) // 5 V_THUMB
    // UCA0RXD (0x40) // 6 Back Channel UCA0RXD
    // UCA0TXD (0x80) // 7 Back Channel UCA0TXD
    //-------------------------------------------------------

P1OUT = 0x00; //set low
P1DIR = 0x00;

P1SEL0 &= ~RED_LED; //REDLED GPIO operation
P1SEL1 &= ~RED_LED;
P1OUT  &= ~RED_LED;
P1DIR  |=  RED_LED;


P1SEL0 &= ~V_A1_SEEED; //
P1SEL1 &= ~V_A1_SEEED;
P1OUT  &= ~V_A1_SEEED;
P1DIR  |=  V_A1_SEEED;


P1SELC |=  V_DETECT_L; // ADC input for V_DETECT_L
```

```
P1SELC |=  V_DETECT_R; // ADC input for V_DETECT_R

P1SEL0 &= ~V_A4_SEEED;
P1SEL1 &= ~V_A4_SEEED;
//P1SELC |= V_A4_SEEED; // ADC input for V_A4_SEEED
P1OUT  &= ~V_A4_SEEED;
P1DIR  |=  V_A4_SEEED;


P1SELC |=  V_THUMB; // ADC input for V_THUMB

P1SEL0 |=  UCA0RXD;
P1SEL1 &= ~UCA0RXD;


P1SEL0 |=  UCA0TXD;
P1SEL1 &= ~UCA0TXD;



    }

void Init_Port_2(void){ // Configure Port 2
//-------------------------------------------------------------------------------
 P2OUT = 0x00; // P2 set Low
 P2DIR = 0x00; // Set P2 direction to output

 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

 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

 P2SEL0 &= ~IR_LED; // P2_2 GPIO operation
 P2SEL1 &= ~IR_LED; // P2_2 GPIO operation
 P2OUT  &= ~IR_LED; // Initial Value = Low / Off
 P2DIR  |=  IR_LED; // Direction = input

 P2SEL0 &= ~SW2; // SW2 set as I/0
 P2SEL1 &= ~SW2; // SW2 set as I/0
 P2DIR  &= ~SW2; // SW2 Direction = input
 P2OUT  |=  SW2; // Configure pull-up resistor SW1
 P2REN  |=  SW2; // Enable pull-up resistor SW1
 P2IES  |=  SW2; // SW2 Hi/Lo edge interrupt
 P2IFG  &= ~SW2; // IFG SW1 cleared
 P2IE   |=  SW2; // SW1 interrupt Enabled

 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

 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

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

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


void Init_Port_3(void){
P3OUT = 0x00; //set low
P3DIR = 0x00;

P3SEL0 &= ~TEST_PROBE;
P3SEL1 &= ~TEST_PROBE;
P3OUT  &= ~TEST_PROBE;
P3DIR   |=  TEST_PROBE;

P3SEL0 &= ~OA2O;
P3SEL1 &= ~OA2O;
P3OUT  &= ~OA2O;
P3DIR   |=  OA2O;

P3SEL0 &= ~OA2N;
P3SEL1 &= ~OA2N;
P3OUT  &= ~OA2N;
P3DIR   |=  OA2N;

P3SEL0 &= ~OA2P;
P3SEL1 &= ~OA2P;
P3OUT  &= ~OA2P;
P3DIR   |=  OA2P;


P3SEL0 &= ~SMCLK_OUT;
P3SEL1 &= ~SMCLK_OUT;
P3OUT  &= ~SMCLK_OUT;
P3DIR   |=  SMCLK_OUT;

//P3SEL0 &= ~DAC_CNTL;
//P3SEL1 &= ~DAC_CNTL;
P3SELC |=  DAC_CNTL;
//P3OUT  &= ~DAC_CNTL;
//P3DIR |= DAC_CNTL;

P3SEL0 &= ~IOT_LINK_GRN;
P3SEL1 &= ~IOT_LINK_GRN;
P3OUT  &= ~IOT_LINK_GRN;
P3DIR   |=  IOT_LINK_GRN;
```

```
P3SEL0 &= ~IOT_EN;
P3SEL1 &= ~IOT_EN;
P3OUT  &= ~IOT_EN;
P3DIR  |=  IOT_EN;

    }


void Init_Port_4(void){ // Configure PORT 4
//-------------------------------------------------------------------------------
 P4OUT = 0x00; // P4 set Low
 P4DIR = 0x00; // Set P4 direction to output

 P4SEL0 &= ~RESET_LCD; // RESET_LCD GPIO operation
 P4SEL1 &= ~RESET_LCD; // RESET_LCD GPIO operation
 P4OUT  &= ~RESET_LCD; // Initial Value = Low / Off
 P4DIR  |=  RESET_LCD; // Direction = output

 P4SEL0 &= ~SW1; // SW1 set as I/0
 P4SEL1 &= ~SW1; // SW1 set as I/0
 P4DIR  &= ~SW1; // SW1 Direction = input
 P4OUT  |=  SW1; // Configure pull-up resistor SW1
 P4REN  |=  SW1; // Enable pull-up resistor SW1
 P4IES  |=  SW1; // SW1 Hi/Lo edge interrupt
 P4IFG  &= ~SW1; // IFG SW1 cleared
 P4IE   |=  SW1; // SW1 interrupt Enabled

 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
 P4OUT  |=  UCB1_CS_LCD; // Set SPI_CS_LCD Off [High]
 P4DIR  |=  UCB1_CS_LCD; // Set SPI_CS_LCD direction to output

 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
 //-------------------------------------------------------------------------------
}
void Init_Port_5(void){
P5OUT = 0x00; //set low
P5DIR = 0x00;

P5SELC |= V_BAT;

P5SELC |= V_5_0;
```

```
P5SELC |= V_DAC;

P5SELC |= V_3_3;

P5SEL0 &= ~IOT_BOOT;
P5SEL1 &= ~IOT_BOOT;
P5OUT  |=  IOT_BOOT;
P5DIR  |=  IOT_BOOT;

    }


void Init_Port_6(void){
P6OUT = 0x00; //set low
P6DIR = 0x00;


P6SEL0 |=  LCD_BACKLITE;
P6SEL1 &= ~LCD_BACKLITE;
P6DIR  |=  LCD_BACKLITE;

P6SEL0 |=  R_FORWARD;
P6SEL1 &= ~R_FORWARD;
P6DIR  |=  R_FORWARD;

P6SEL0 |=  L_FORWARD; // Set Select 0 For GP I/O
P6SEL1 &= ~L_FORWARD; // Set Select 0 For GP I/O
P6DIR  |=  L_FORWARD; // Set Direction to Output

P6SEL0 |=  R_REVERSE;
P6SEL1 &= ~R_REVERSE;
P6DIR  |=  R_REVERSE;

P6SEL0 |=  L_REVERSE;
P6SEL1 &= ~L_REVERSE;
P6DIR  |=  L_REVERSE;

P6SEL0 &= ~P6_5;
P6SEL1 &= ~P6_5;
P6OUT  &= ~P6_5;
P6DIR  |=  P6_5;

P6SEL0 &= ~GRN_LED;
P6SEL1 &= ~GRN_LED;
P6OUT  &= ~GRN_LED;
P6DIR  |=  GRN_LED;

    }
```

9.3 clocks.c

```
void Init_Clocks(void){
// -----------------------------------------------------------------------------
// Clock Configurtaions
// This is the clock initialization for the program.
```

```
// Initial clock configuration, runs immediately after port configuration.
// Disables 1ms watchdog timer,
// Configure MCLK for 8MHz and XT1 sourcing ACLK and FLLREF.
//
// Description: Configure ACLK = 32768Hz,
//                        MCLK = DCO + XT1CLK REF = 8MHz,
//                        SMCLK = MCLK = 8MHz.
// Toggle LED to indicate that the program is running.
//
// --------------------------------------------------------------------------------
  WDTCTL = WDTPW | WDTHOLD;  // Disable watchdog

  do{
    CSCTL7 &= ~XT1OFFG;      // Clear XT1 fault flag
    CSCTL7 &= ~DCOFFG;       // Clear DCO fault flag
    SFRIFG1 &= ~OFIFG;
  } while (SFRIFG1 & OFIFG); // Test oscillator fault flag
  __bis_SR_register(SCG0);   // disable FLL

  CSCTL1 = DCOFTRIMEN_1;
  CSCTL1 |= DCOFTRIM0;
  CSCTL1 |= DCOFTRIM1;       // DCOFTRIM=3
  CSCTL1 |= DCORSEL_3;       // DCO Range = 8MHz

  CSCTL2 = FLLD_0 + 243;     // DCODIV = 8MHz

  CSCTL3 |= SELREF__XT1CLK;  // Set XT1CLK as FLL reference source
  __delay_cycles(3);
  __bic_SR_register(SCG0);   // enable FLL
  Software_Trim();           // Software Trim to get the best DCOFTRIM value

  CSCTL4 = SELA__XT1CLK;     // Set ACLK = XT1CLK = 32768Hz
  CSCTL4 |= SELMS__DCOCLKDIV;// DCOCLK = MCLK and SMCLK source

//  CSCTL5 |= DIVM__4;         // MCLK = DCOCLK / 4  = 2MHZ,
//  CSCTL5 |= DIVS__4;         // SMCLK = MCLK / 4 = 500KHz
  CSCTL5 |= DIVM__1;        // MCLK = DCOCLK = 8MHZ,
  CSCTL5 |= DIVS__1;        // SMCLK = MCLK = 8MHz

  PM5CTL0 &= ~LOCKLPM5;      // Disable the GPIO power-on default high-impedance mode
                             // to activate previously configured port settings
}
```

9.4 interrupts_switches.c

```
// Port 4 interrupt. For switches, they are disabled for the duration
// of the debounce timer. Flag is set that user space can check.
#pragma vector=PORT4_VECTOR
__interrupt void switchP4_interrupt(void){
    // Switch 1
    if (P4IFG & SW1) {
        strcpy(display_line[0], "IRLED: ON ");
        hold = counter;
        event = BLK_LN;
        display_changed = TRUE;
        P2OUT |= IR_LED;
        TB0CCTL0 &= ~CCIE; // CCR0 disable interrupt
```

```
        P4IFG &= ~SW1; // IFG SW1 cleared
        P4IE &= ~SW1; // SW1 interrupt disabled
        TB0CCTL1 &= ~TBIFG; // Clear CCR1 interrupt flag
        count_debounce_SW1 = RESET_STATE; // Reset debounce counter
        debounce_SW1 = TRUE; // Set debounce detector to TRUE
        TB0CCR1 += TB0CCR1_INTERVAL; // Add Offset to TBCCR1
        TB0CCTL1 |= CCIE; // CCR1 enable interrupt
    }
}

// Port 2 interrupt. For switches, they are disabled for the duration
// of the debounce timer. Flag is set that user space can check.
#pragma vector=PORT2_VECTOR
__interrupt void switchP2_interrupt(void){
    // Switch 2
    if (P2IFG & SW2) {
        strcpy(display_line[0], "IRLED: OFF");
        display_changed = TRUE;
        P2OUT &= ~IR_LED;
        TB0CCTL0 &= ~CCIE; // CCR0 disable interrupt
        P2IFG &= ~SW2; // IFG SW2 cleared
        P2IE &= ~SW2; // SW2 interrupt disabled
        TB0CCTL2 &= ~TBIFG; // Clear CCR2 interrupt flag
        count_debounce_SW2 = RESET_STATE; // Reset debounce counter
        debounce_SW2 = TRUE; // Set debounce detector to TRUE
        TB0CCR2 += TB0CCR2_INTERVAL; // Add Offset to TBCCR1
        TB0CCTL2 |= CCIE; // CCR1 enable interrupt
    }
}
```

9.5 adc.c

```
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
    ADCCTL1 |= ADCCONSEQ_0; // ADC conversion sequence 00b = Single-channel single
conversion
// ADCCTL1 & ADCBUSY identifies a conversion is in process

// ADCCTL2 Register
    ADCCTL2 = 0;                // 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_5;       // V_THUMB (0x20) Pin 5 A5

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

#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 the result by 4
                   HEXtoBCD(ADC_Left_Detect);    // Convert result to String
                   adc_line(1,3);                // Place String in Display
                   ADCMCTL0 &= ~ADCINCH_2;       // Disable Last channel A2
                   ADCMCTL0 |= ADCINCH_3;        // Enable Next channel A3
                   break;
               case 0x01:                      // Channel A3 Interrupt
                   ADC_Right_Detect = ADCMEM0;   // Move result into Global Values
                   ADC_Right_Detect = ADC_Right_Detect >> 2; // Divide the result by 4
                   HEXtoBCD(ADC_Right_Detect);   // Convert result to String
                   adc_line(2,3);                // Place String in Display
                   ADCMCTL0 &= ~ADCINCH_3;       // Disable Last channel A2
                   ADCMCTL0 |= ADCINCH_5;        // Enable Next channel ????
                   break;
               case 0x02:                      // Channel ??? Interrupt
                   ADC_Thumb_Detect = ADCMEM0;   // Move result into Global Values
                   ADC_Thumb_Detect = ADC_Thumb_Detect >> 2; // Divide the result by 4
                   HEXtoBCD(ADC_Thumb_Detect);   // Convert result to String
                   adc_line(3,3);                // Place String in Display
                   ADCMCTL0 &= ~ADCINCH_5;       // Disable Last channel A?
```

```
                    ADCMCTL0 |= ADCINCH_2;       // Enable First channel 2
                    ADC_Channel = 0;
                    break;
                default:
                    break;
              }
              ADCCTL0 |= ADCENC; // Enable Conversions
              ADCCTL0 |= ADCSC;
              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;
    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 place it
// in the desired location on the desired line of the display.
// char line => Specifies the line 1 thru 4
// char location => Is the location 0 thru 9
//
//---------------------------------------------------------------
void adc_line(char line, char 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];
    }
}
//--------------------------------------------------------------
```

9.6 timers.c

```
//function for timers
void Init_Timers(){
    Init_Timer_B0();
    Init_Timer_B1();
    //Init_Timer_B2();
    Init_Timer_B3();
}


// 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__4; // Divide clock by 4
TB0EX0 = TBIDEX__8; // Divide clock by an additional 8


TB0CCR0 = TB0CCR0_INTERVAL; // CCR0
TB0CCTL0 |= CCIE; // CCR0 enable interrupt
TB0CCTL0 &= ~CCIFG; // Clearing Interrupt flag

//TB0CCR1 = TB0CCR1_INTERVAL; // CCR1
//TB0CCTL1 |= CCIE; // CCR1 enable interrupt

//TB0CCR2 = TB0CCR2_INTERVAL; // CCR2
//TB0CCTL2 |= CCIE; // CCR2 enable interrupt

TB0CTL |= TBIE; // Enable Overflow Interrupt
TB0CTL &= ~TBIFG; // Clear Overflow Interrupt flag
}
//--------------------------------------------------------------------------------
void Init_Timer_B1(void) {
TB1CTL = TBSSEL__SMCLK; // SMCLK source
TB1CTL |= TBCLR; // Resets TB0R, clock divider, count direction
TB1CTL |= MC__CONTINOUS; // Continuous up
TB1CTL |= ID__4; // Divide clock by 4
TB1EX0 = TBIDEX_7; // Divide clock by an additional 8


TB1CCR0 = TB1CCR0_INTERVAL; // CCR0
TB1CCTL0 |= CCIE; // CCR0 enable interrupt
TB1CCTL0 &= ~CCIFG; // Clearing Interrupt flag

//TB1CCR1 = TB1CCR1_INTERVAL; // CCR1
//TB1CCTL1 |= CCIE; // CCR1 enable interrupt
//TB1CCTL1 &= ~CCIFG; // Clearing Interrupt flag

//TB0CCR2 = TB0CCR2_INTERVAL; // CCR2
```

```
//TB0CCTL2 |= CCIE; // CCR2 enable interrupt

//TB1CTL |= TBIE; // Enable Overflow Interrupt
//TB1CTL &= ~TBIFG; // Clear Overflow Interrupt flag
}
//------------------------------------------------------------------------------
void Init_Timer_B3(void) {
//-----------------------------------------------------------------------------
// SMCLK source, up count mode, PWM Right Side
// TB3.1 P6.0 LCD_BACKLITE
// TB3.2 P6.1 R_FORWARD
// TB3.3 P6.2 R_REVERSE
// TB3.4 P6.3 L_FORWARD
// TB3.5 P6.4 L_REVERSE
//-----------------------------------------------------------------------------
TB3CTL = TBSSEL__SMCLK; // SMCLK
TB3CTL |= MC__UP; // Up Mode
TB3CTL |= TBCLR; // Clear TAR
PWM_PERIOD = WHEEL_PERIOD; // PWM Period [Set this to 50005]
TB3CCTL1 = OUTMOD_7; // CCR1 reset/set
LCD_BACKLITE_DIMING = PERCENT_80; // P6.0 Right Forward PWM duty cycle
TB3CCTL2 = OUTMOD_7; // CCR2 reset/set
RIGHT_FORWARD_SPEED = WHEEL_OFF; // P6.1 Right Forward PWM duty cycle
TB3CCTL3 = OUTMOD_7; // CCR3 reset/set
RIGHT_REVERSE_SPEED = WHEEL_OFF; // P6.2 Right Reverse PWM duty cycle
TB3CCTL4 = OUTMOD_7; // CCR4 reset/set
LEFT_FORWARD_SPEED = WHEEL_OFF; // P6.3 Left Forward PWM duty cycle
TB3CCTL5 = OUTMOD_7; // CCR5 reset/set
LEFT_REVERSE_SPEED = WHEEL_OFF; // P6.4 Left Reverse PWM duty cycle
//-----------------------------------------------------------------------------
}

#pragma vector=TIMER0_B1_VECTOR
__interrupt void TIMER0_B1_ISR(void){
//----------------------------------------------------------------------------
// TimerB0 1-2, Overflow Interrupt Vector (TBIV) handler
//----------------------------------------------------------------------------
    switch(__even_in_range(TB0IV,14)){
        case 0: break; // No interrupt
        case 2: // CCR1 not used
            ++count_debounce_SW1;
            if(count_debounce_SW1 >= DEBOUNCE_TIME){
                TB0CCTL1 &= ~CCIE; // CCR1 disable interrupt
                P4IFG &= ~SW1; // IFG SW1 cleared
                P4IE |= SW1; // SW1 interrupt enabled
                debounce_SW1 = RESET_STATE; //Reset debounce detector
                TB0CCR0 += TB0CCR0_INTERVAL; // Add Offset to TBCCR0
                TB0CCTL0 |= CCIE; // CCR0 enable interrupt
            }
            TB0CCR1 += TB0CCR1_INTERVAL; // Add Offset to TBCCR1
            break;
        case 4: // CCR2 not used
            ++count_debounce_SW2;
            if(count_debounce_SW2 >= DEBOUNCE_TIME){
                TB0CCTL2 &= ~CCIE; // CCR2 disable interrupt
                P2IFG &= ~SW2; // IFG SW2 cleared
```

```
                P2IE |= SW2; // SW2 interrupt enabled
                debounce_SW2 = RESET_STATE; // Reset debounce detector
                TB0CCR0 += TB0CCR0_INTERVAL; // Add Offset to TBCCR0
                TB0CCTL0 |= CCIE; // CCR0 enable interrupt
            }
            TB0CCR2 += TB0CCR2_INTERVAL; // Add Offset to TBCCR2
            break;
        case 14: // overflow
            break;
        default: break;
    }
//-----------------------------------------------------------------------------
}
```

9.7 serial.c

```
void Init_Serial_UCA1(void){
//-------------------------------------------------------------------------------
// TX error (%) RX error (%)
// BRCLK Baudrate UCOS16 UCBRx UCFx UCSx neg pos neg pos
// 8000000 4800 1 104 2 0xD6 -0.08 0.04 -0.10 0.14
// 8000000 9600 1 52 1 0x49 -0.08 0.04 -0.10 0.14
// 8000000 19200 1 26 0 0xB6 -0.08 0.16 -0.28 0.20
// 8000000 57600 1 8 10 0xF7 -0.32 0.32 -1.00 0.36
// 8000000 115200 1 4 5 0x55 -0.80 0.64 -1.12 1.76
// 8000000 460800 0 17 0 0x4A -2.72 2.56 -3.76 7.28
//-------------------------------------------------------------------------------
// Configure eUSCI_A0 for UART mode
 UCA1CTLW0 = 0;
 UCA1CTLW0 |= UCSWRST ; // Put eUSCI in reset
 UCA1CTLW0 |= UCSSEL__SMCLK; // Set SMCLK as fBRCLK
 UCA1CTLW0 &= ~UCMSB; // MSB, LSB select
 //UCA1CTLW0 &= ~UCSPB; // UCSPB = 0(1 stop bit) OR 1(2 stop bits)
 UCA1CTLW0 &= ~UCPEN; // No Parity
 UCA1CTLW0 &= ~UCSYNC;
 UCA1CTLW0 &= ~UC7BIT;
 UCA1CTLW0 |= UCMODE_0;
// BRCLK Baudrate UCOS16 UCBRx UCFx UCSx neg pos neg pos
// 8000000 115200 1 4 5 0x55 -0.80 0.64 -1.12 1.76
// UCA?MCTLW = UCSx + UCFx + UCOS16
 UCA1BRW = 4 ; // 115,200 baud
 UCA1MCTLW = 0x5551 ;
 UCA1CTLW0 &= ~UCSWRST ; // release from reset
 //UCA1TXBUF = 0x00; // Prime the Pump
 UCA1IE |= UCRXIE; // Enable RX interrupt
//-------------------------------------------------------------------------------
}

void Init_Serial_UCA0(void){
//-------------------------------------------------------------------------------
// TX error (%) RX error (%)
// BRCLK Baudrate UCOS16 UCBRx UCFx UCSx neg pos neg pos
// 8000000 4800 1 104 2 0xD6 -0.08 0.04 -0.10 0.14
// 8000000 9600 1 52 1 0x49 -0.08 0.04 -0.10 0.14
// 8000000 19200 1 26 0 0xB6 -0.08 0.16 -0.28 0.20
// 8000000 57600 1 8 10 0xF7 -0.32 0.32 -1.00 0.36
```

```
// 8000000 115200 1 4 5 0x55 -0.80 0.64 -1.12 1.76
// 8000000 460800 0 17 0 0x4A -2.72 2.56 -3.76 7.28
//------------------------------------------------------------------------------
// Configure eUSCI_A0 for UART mode
 UCA0CTLW0 = 0;
 UCA0CTLW0 |= UCSWRST ; // Put eUSCI in reset
 UCA0CTLW0 |= UCSSEL__SMCLK; // Set SMCLK as fBRCLK
 UCA0CTLW0 &= ~UCMSB; // MSB, LSB select
 //U0A0CTLW0 &= ~UCSPB; // UCSPB = 0(1 stop bit) OR 1(2 stop bits)
 UCA0CTLW0 &= ~UCPEN; // No Parity
 UCA0CTLW0 &= ~UCSYNC;
 UCA0CTLW0 &= ~UC7BIT;
 UCA0CTLW0 |= UCMODE_0;
// BRCLK Baudrate UCOS16 UCBRx UCFx UCSx neg pos neg pos
// 8000000 115200 1 4 5 0x55 -0.80 0.64 -1.12 1.76
// UCA?MCTLW = UCSx + UCFx + UCOS16
 UCA0BRW = 4 ; // 115,200 baud
 UCA0MCTLW = 0x5551 ;
 UCA0CTLW0 &= ~UCSWRST ; // release from reset
 //UCA0TXBUF = 0x00; // Prime the Pump
 UCA0IE |= UCRXIE; // Enable RX interrupt
//------------------------------------------------------------------------------
}

// Global Variables
char process_buffer[25]; // Size for appropriate Command Length
extern char pb_index; // Index for process_buffer
void USCI_A1_transmit(void){ // Transmit Function for USCI_A0
// Contents must be in process_buffer
// End of Transmission is identified by NULL character in process_buffer
// process_buffer includes Carriage Return and Line Feed
 pb_index = 0; // Set Array index to first location
 UCA1IE |= UCTXIE; // Enable TX interrupt
}
#pragma vector=EUSCI_A1_VECTOR
__interrupt void eUSCI_A1_ISR(void){
//------------------------------------------------------------------------------
// Echo back RXed character, confirm TX buffer is ready first
 //unsigned int temp;
    switch(__even_in_range(UCA1IV,0x08)){
     case 0: // Vector 0 - no interrupt
     break;
     case 2: // Vector 2 - RXIFG


          USB_Char_Rx[usb_rx_ring_wr++] = UCA1RXBUF; // Rx -> IOT_2_PC character array

          if (usb_rx_ring_wr >= sizeof(USB_Char_Rx)){
              usb_rx_ring_wr = BEGINNING; // Circular buffer back to beginning
          }

          //UCA1IE |= UCTXIE; // Enable Tx interrupt
          break;


     case 4: // Vector 4 - TXIFG
```

```
          UCA1TXBUF = USB_Char_Tx[usb_tx_ring_wr]; // Transmit Current Indexed value

          USB_Char_Tx[usb_tx_ring_wr++] = NULL_RxTx; // Null Location of Transmitted value
          if(USB_Char_Tx[usb_tx_ring_wr] == NULL_RxTx){ // Is the next pb_index location
NULL - End of Command
          UCA1IE &= ~UCTXIE; // Disable TX interrupt
          }

      break;

 default: break;
 }
//------------------------------------------------------------------------------
}




#pragma vector=EUSCI_A0_VECTOR
__interrupt void eUSCI_A0_ISR(void){
//------------------------------------------------------------------------------
// Echo back RXed character, confirm TX buffer is ready first
 //unsigned int temp;
    switch(__even_in_range(UCA0IV,0x08)){
     case 0: // Vector 0 - no interrupt
     break;
     case 2: // Vector 2 - RXIFG


          IOT_Char_Rx[iot_rx_ring_wr++] = UCA0RXBUF; // Rx -> IOT_2_PC character array

          if (iot_rx_ring_wr >= sizeof(IOT_Char_Rx)){
              iot_rx_ring_wr = BEGINNING; // Circular buffer back to beginning
          }
          //UCA0IE |= UCTXIE; // Enable Tx interrupt
          break;


     case 4: // Vector 4 - TXIFG
          UCA0TXBUF = IOT_Char_Tx[iot_tx_ring_wr]; // Transmit Current Indexed value
          IOT_Char_Tx[iot_tx_ring_wr++] = NULL_RxTx; // Null Location of Transmitted value

          if(IOT_Char_Tx[iot_tx_ring_wr] == NULL_RxTx){ // Is the next pb_index location
NULL - End of Command
          UCA0IE &= ~UCTXIE; // Disable TX interrupt
          }

          break;

 default: break;
 }
//------------------------------------------------------------------------------
}


//------------------------------------------------------------------------------
```

Conclusion

The document provides a comprehensive overview of the PITA Car project, detailing its components, hardware software, and testing processes. The project comprises a FRAM board, a Power board, and a variety of peripherals, including an LCD display, switches, and microcontroller units (MCUs). The hardware components include the chassis, motors, battery pack, FET board, and power display board, each playing a crucial role in the car's functionality, from providing structural support to regulating power distribution and controlling motor movements. User interaction with the PITA Car is facilitated through switches and the LCD display, as well as the magic smoke app using serial communication. The switches trigger specific actions in the car's control system. While the LCD display provides real-time feedback on the car's status and actions.

Going through the creation of the PITA Car we learned so much about embedded systems and how they operate. Working on this coding project taught me several valuable lessons about embedded systems programming and robotics. Firstly, we gained a deeper understanding of how to interface with hardware components and peripherals using microcontroller ports and timers. By initializing and configuring ports for input and output operations, we learned how to control external devices such as switches, LEDs, and motors. We also learned how important thorough testing is to catch mistakes as early as possible. Furthermore, implementing the movement patterns of the robotic car, such as moving in a shapes as well as moving per instructions of the magic smoke app provided hands-on experience in translating high-level instructions into low-level commands. By defining constants like CIRCL_WHEEL_COUNT_TIME, CIRCLE_RIGHT_COUNT_TIME, CIRCLE_LEFT_COUNT_TIME, and CIRCLE_TRAVEL_DISTANCE, we learned how to parameterize the behavior of the car's movements, allowing for easy adjustments of its trajectory and speed. Additionally, serial communication was integral part of communication with our and vise versa.

Serial communication was central to the project's success, enabling seamless data exchange between different components and with external systems. This communication allowed us to send instructions to the car, receive real-time feedback through the LCD display, and even remotely adjust operational parameters. By establishing reliable serial communication, we ensured that the car's control system could respond accurately to user commands and adapt to changing conditions.

Throughout the project, we learned the intricacies of interfacing with hardware using serial communication protocols. This involved initializing microcontroller ports for transmission and reception, managing baud rates, and ensuring error-free data transfer. Serial communication not only enabled us to communicate with the robotic car but also provided a pathway to diagnose issues, making troubleshooting and debugging more manageable.

Testing and iteration played a crucial role in the project's success. We faced challenges such as synchronizing data transmission, handling interrupts, and ensuring stable communication links. Through rigorous testing, we identified potential errors, like incorrect baud rates or improper initialization, that could lead to communication breakdowns or erratic behavior. Serial communication was also instrumental in helping us verify that our software was correctly controlling the car's hardware components.

During the creation of the PITA Car the test process we went through prevented errors that if went unnoticed could have been catastrophic. Some of the errors that we faced are setting extreme values for these constants could lead to boundary errors. For instance, setting a very high value for Circle_Travel_Distance could result in the car attempting to move beyond its physical constraints, potentially causing damage to the car or its surroundings. Failing to properly initialize variables or settings related to the car's movement could cause unexpected behavior. For example, if the motor control pins are not configured correctly or if the motor driver is not properly initialized, the car may not respond to the commands. Lastly, Inadequate handling of interrupts, timing settings, and asynchronous events could lead to timing discrepancies or interruptions in motor control routines. Additionally configuring our IOT to the correct ip address and port number was cruical. However, through experimentation, testing, and iteration, we were able to enhance our skills in embedded systems.

The PITA Car project served as an invaluable learning experience, highlighting the complexities and nuances of embedded systems development. It underscored the importance of careful planning, thorough testing, and a deep understanding of hardware-software interactions.