Implementation of a Wi-Fi Controlled and Autonomous Vehicle

A Stepping Stone for Intelligent Navigation

World Domination

Team Members
Kwali Bunker
Alex Hege
Leah Page
Josiah Szobody

Originator: Kwali Bunker
Checked: Josiah Szobody | Released: 12/5/2025
Filename: world_domination.pdf

Title: **Implementation of a Wi-Fi Controlled and Autonomous Vehicle A Stepping Stone for Intelligent Navigation**

Date:	Document Number:	Rev:	Sheet:
12/5/2025	2-7182-818-2845-90	5.9	1 of 82

0-0002-001-0001-00 Document Format Sheet

Revision	Description	Contributors
1.0	Document updated with a Scope, describing this enterprise as a whole	Alex Hege
1.1	Drafted Test Process using data gathered in the lab	Kwali Bunker
1.2	Created Overview and Block Diagrams describing the relationships between components	Josiah Szobody Alex Hege
1.3	Added Hardware section describing the main hardware involved	Leah Page
1.4	Updated Hardware and Test Analysis with schematics and images supporting the content	Alex Hege Josiah Szobody
1.5	Created Abbreviations table defining abbreviations used in this document	Alex Hege
1.6	Quality edit: revising inaccuracies found to ensure document integrity.	Kwali Bunker Alex Hege Leah Page Josiah Szobody
2.0	Refined the Test Process to include new data and document the troubleshooting steps taken to resolve issues.	Kwali Bunker Josiah Szobody
2.1	Updated the Overview section to include new block diagrams for the motor driver system.	Alex Hege Josiah Szobody
2.2	Modified the scope to align with the current phase and objectives.	Alex Hege
2.3	Added Hardware and Test figures to support new content	Josiah Szobody
2.4	Created Hardware subsections for motor driver system components	Leah Page
2.5	Populated the abbreviation list with additional acronyms and definitions.	Alex Hege Leah Page Josiah Szobody
2.6	Quality edit: revising inaccuracies found to ensure document integrity.	Alex Hege Josiah Szobody
3.0	Added Software Listings: port.c	Kwali Bunker
3.1	Added Software Listings: ADC.c and interrupts_ADC.c	Alex Hege
3.2	Added Software Listings: timers.c and timer_interrupts.c	Leah Page
3.3	Added Software Listings: main.c	Josiah Szobody
3.4	Created Flow Charts for each of the functions included in the code additions	Alex Hege Leah Page Josiah Szobody
3.5	Updated Hardware with new sections for ADC, motor and H-bridge related components	Alex Hege Leah Page Josiah Szobody
3.6	Added Fault troubleshooting, motion and data tests.	Alex Hege Josiah Szobody

Revision	Description	Contributors
3.7	Modified Scope, Abbreviations and Overview	Alex Hege Josiah Szobody
3.8	Added Software section with descriptions of each code file	Josiah Szobody
4.0	Updated timers.c and timer_interrupts.c	Leah Page
4.1	Updated ports.c	Kwali Bunker
4.2	Updated main.c	Josiah Szobody
4.3	Added Software Listings: UART.c and UART_interrupts.c	Josiah Szobody
4.4	Created UART related flowcharts and modified flowcharts to match code structure.	Josiah Szobody
4.5	Added UART, battery monitoring and line following tests	Kwali Bunker Alex Hege
4.6	Created Power Analysis section describing the power system and specifications used	Josiah Szobody Kwali Bunker
4.7	Quality edit: revising inaccuracies found to ensure document integrity.	Alex Hege Kwali Bunker Josiah Szobody
5.0	Revised Power Analysis to include new DAC functionality and power usage	Josiah Szobody Alex Hege
5.1	Updated Hardware with new ESP32 chip and documentation	Alex Hege
5.2	Added DAC.c	Alex Hege
5.3	Added DAC flowchart and description for Software Listings	Josiah Szobody
5.4	Created new testing sections regarding the UART and Wi-Fi control system	Josiah Szobody Alex Hege
5.5	Edited the Overview and the Hardware to also include the implemented DAC motor control	Josiah Szobody Alex Hege
5.6	Updated the Battery Monitoring Test to include image verification	Kwali Bunker
5.7	Updated Abbreviations to support document content	Alex Hege Josiah Szobody
5.8	Revised Conclusion and added completed bot images to validate the successful implementation of a Wi-Fi Controlled and Autonomous Vehicle.	Kwali Bunker Alex Hege Leah Page Josiah Szobody
5.9	Quality edit: revising inaccuracies found to ensure document integrity.	Kwali Bunker Alex Hege Leah Page Josiah Szobody

Table of Contents

Table of Contents

Figures

Figures

1. Scope

This document outlines the design, implementation, and testing of an autonomous vehicle built around the MSP430FR2355 microcontroller, demonstrating real-time interaction between software and hardware to achieve autonomous motion. The system integrates multiple subsystems including power regulation, sensor input, actuator control, and user feedback into a compact, low-power platform. The software operates at a low level to manage hardware resources directly, utilizing features such as interrupts, GPIO configuration, and communication protocols such as UART, SPI and Wi-Fi.

The design serves as a foundation for intelligent navigation, incorporating basic sensor interfacing and autonomous motion control. While currently focused on local operation, it provides a framework that can be expanded for future capabilities such as obstacle detection, path planning, and real-time decision-making, offering a practical example of software and hardware working together in a low-power, autonomous system.

2. Abbreviations

Abbreviation	Definition
A	Ampere – unit of electric current
ADC	Analog-to-Digital Converter – converts analog signals to digital
AlGaAs	A semiconductor crystal made from a mixture of aluminum, gallium, and arsenic.
ASCII	American Standard Code for Information Interchange – standard for text encoding in computers
CCR	Capture/Compare Register – a timer register used to capture the timer value on an event or compare the timer value to generate interrupts or control outputs.
CCS	Code Composer Studio – an IDE developed by Texas Instruments for embedded development
cm	Centimeter – unit of length (1 cm = 0.01 meters)
CPU	Central Processing Unit – the main processor of a computer or MCU
DAC	Digital-to-Analog Converter – converts digital signals to analog
DC	Direct Current – electric current flowing in one direction
ECE	Electrical and Computer Engineering
eUSCI	Enhanced Universal Serial Communication Interface – Texas Instruments' microcontroller peripheral for serial protocols (UART, SPI, I2C).
FB	Ferrite Bead – passive component used to suppress high-frequency noise
FET	Field-Effect Transistor – a type of transistor used for amplifying or switching electronic signals
GHz	Gigahertz – unit of frequency equal to one billion hertz, commonly used to specify clock speeds, radio frequencies, and high-speed signals.
GND	Ground – common reference point in a circuit
GPIO	General-Purpose Input/Output – configurable pins on a microcontroller or IC
IC	Integrated Circuit – a chip containing multiple electronic components

Abbreviation	Definition
IDE	Integrated Development Environment – software for writing, testing, and debugging code
in	Inch – unit of length (1 in = 2.54 cm)
ISR	Interrupt Service Routine – a function that executes automatically when a hardware interrupt occurs
iOS	iPhone Operating System – the mobile operating system developed by Apple for iPhone and iPad
IoT	Internet of Things – the network of physical objects ("things") embedded with sensors and software to connect and exchange data.
I/O	Input/Output – data or signal flow to/from a system
IP	Internet Protocol – a unique address that identifies a device on a network (e.g., 192.168.1.100).
IR	Infrared – electromagnetic radiation with wavelengths longer than visible light, used for proximity sensing and remote controls.
kg	Kilogram – unit of mass (1 kg = 1000 grams)
kΩ	Kiloohm – unit of resistance (1 kΩ = 1000 ohms)
LCD	Liquid Crystal Display – a flat-panel display technology
LED	Light Emitting Diode – a semiconductor light source
mA	Milliampere – 1/1000 of an ampere (1 mA = 10^{-3} A)
MAC	Media Access Control – a unique, hardware-based identifier assigned to a network interface.
mAh	Milliampere-hour – a unit of electric charge capacity representing how much current a battery can deliver over time (1 mAh = 1 mA for 1 hour)
MCU	Microcontroller Unit – a small computer on a single IC
MHz	Megahertz – 1 million cycles per second, often used for clock speeds
N-MOSFET	N-channel Metal-Oxide-Semiconductor Field-Effect Transistor - conducts when a positive voltage is applied to the gate
NCSU	North Carolina State University
OL	Open Loop – a control system without feedback
oz	Ounce – unit of mass (1 oz ≈ 28.35 grams)
PIN	Personal Identification Number – a numeric code used for authentication or security purposes
P-MOSFET	P-channel Metal-Oxide-Semiconductor Field-Effect Transistor – conducts when a negative voltage is applied to the gate
PX. Y	Microcontroller I/O Pin – where X is the port number and Y is the specific pin on that port (e.g., P6.1)

Abbreviation	Definition
RAM	Random Access Memory – temporary data storage (volatile)
ROM	Read-Only Memory – permanent data storage (non-volatile)
PWM	Pulse Width Modulation – a technique for controlling analog devices by rapidly switching digital signals on and off, where the duty cycle (percentage of on-time) determines the average power delivered.
SAC	Single-Ended Amplifier Circuit – an electronic amplifier configuration with one input signal path.
SEPIC	Single-Ended Primary Inductor Converter – a DC-DC converter topology that can step up or step down voltage while maintaining continuous input current.
SMCLK	Sub-Main Clock – a clock source in the MSP430FR2355 used to drive timers and peripherals.
SMD	Surface-Mount Device – electronic components mounted on the surface of a PCB
SPDT	Single Pole Double Throw – a type of switch with 1 input and 2 outputs
SPI	Serial Peripheral Interface – synchronous serial communication protocol
SSID	Service Set Identifier – the public name of a Wi-Fi network.
SW	Switch – a device for making or breaking a circuit
TCP	Transmission Control Protocol – a core communication protocol that provides reliable, ordered data delivery over a network.
uA	Microampere – 1/1,000,000 of an ampere (1 μA = 10^{-6} A)
UART	Universal Asynchronous Receiver/Transmitter – serial communication protocol
UCA0	Universal Communication Interface A0 – eUSCI UART instance for ESP32 communication
UCA1	Universal Communication Interface A1 – eUSCI UART instance for PC debug communication
uH	Microhenry – unit of inductance (1 μH = 10^{-6} H)
USB	Universal Serial Bus – standard for data and power transfer
V	Volt – unit of electric potential or voltage
VBAT	Battery Voltage – the raw voltage supplied directly from the battery pack
VCC	Voltage at the Common Collector – the supply voltage for the collector terminal of bipolar junction transistors; commonly used to denote the main positive supply voltage in circuits.
Vin	Voltage Input – supply voltage input to a device or circuit
Wi-Fi	Wireless Fidelity – allows wireless communication between the system and external devices or networks
Ω	Ohm – unit of electrical resistance

3. Overview

The MSP430FR2355 microcontroller receives power from the power system, which boosts the input voltage from a power source (either a battery pack or USB) to a regulated 6V and 3.3V to supply all system components. It interfaces with the user by processing inputs from peripherals such as switches, sensors, and a thumbwheel, and provides feedback through outputs like LEDs and an LCD display.

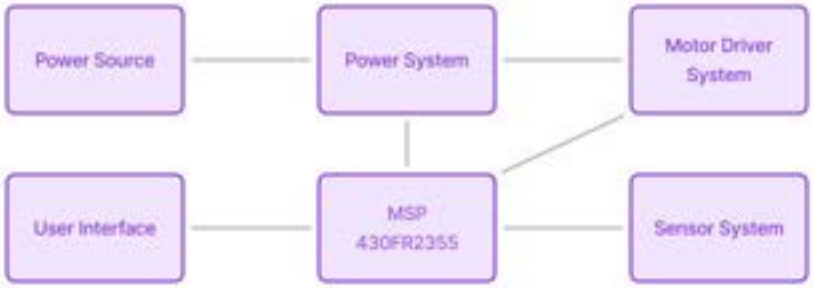

Figure 1: System Overview Block Diagram

3.1. User Interface

The user interface includes two LEDs (one green and one red), two momentary push-button switches, a slide switch SPDT, a thumbwheel, an LCD display and the iOS Magic Smoke app, that allows users to interact with the system. Inputs from the switches, thumbwheel and Magic Smoke App (via the ESP32) are read by the MSP430FR2355, while outputs, such as the LEDs and LCD, provide real-time feedback and display the system status.

Figure 2: User Interface Block Diagram

3.2. Power System

The power system is built around an LT1935ES5 regulator configured as a Single-Ended Primary Inductor Converter (SEPIC). It takes input from the 4×AA battery pack and provides a stable 3.3 V output for all electronic components, including the MSP430FR2355. A slide switch serves as the main power control, allowing the user to turn the entire system on or off.

To support the DC motors, the power system has been upgraded with a DAC that regulates a secondary SEPIC converter. This secondary converter maintains a stable voltage of approximately 6 V, which is delivered to the FET board for motor power. This design ensures consistent motor performance regardless of load variations or battery depletion by actively regulating the supply voltage.

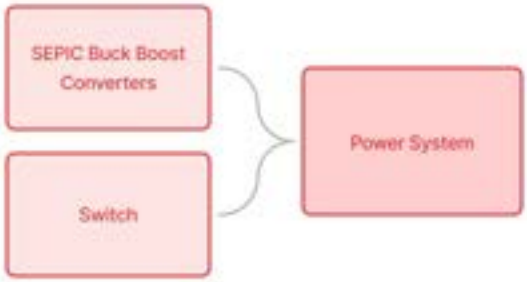

Figure 3: Power System Block Diagram

3.3. MSP430FR2355 MCU

The MSP430FR2355 microcontroller includes key internal components such as a CPU core, clock system, memory blocks (Flash and RAM), GPIO ports, timers, and communication modules (like UART, SPI, or Inter-Integrated Circuit). These modules work together to process inputs, control outputs, and manage timing and communication tasks essential to the system's operation.

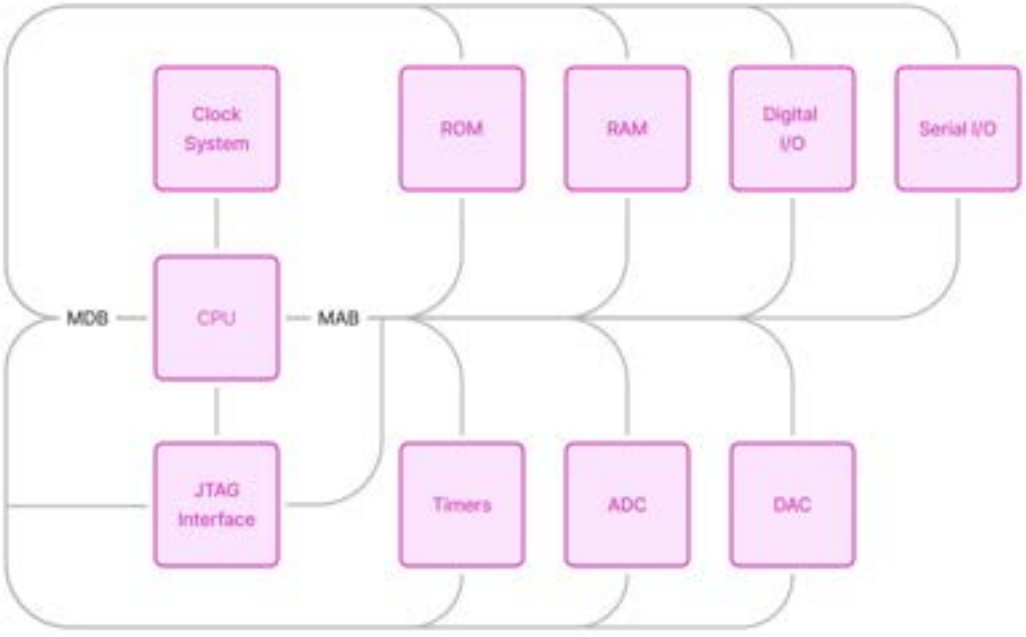

Figure 4: MSP430FR2355 Block Diagram

3.4. Motor Driver System

The motor driver system uses a full H-bridge to control the bidirectional movement of two DC motors. The H-bridge circuit controls the voltage applied to each motor, enabling it to be energized for both forward and reverse motion.

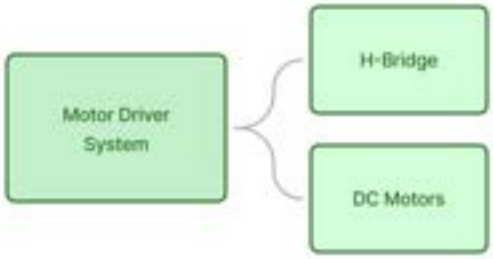

Figure 5: Motor Driver System Block Diagram

3.5. H-Bridge

The H-bridge uses a combination of two P-MOSFETs and four N-MOSFETs arranged in an "H" configuration to act as electronic switches. By opening and closing these switches, the circuit controls the direction of current flow through the motor.

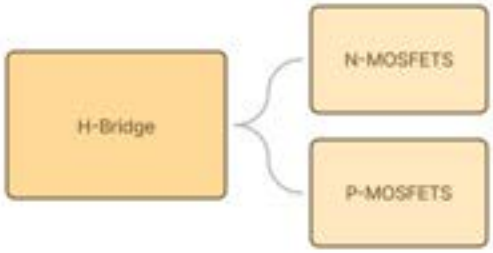

Figure 6: H-Bridge Block Diagram

3.6. Sensor System

The sensor system uses an IR emitter to illuminate a surface, while left and right detectors capture the reflected light. The combined inputs provide the data necessary for navigation and object detection.

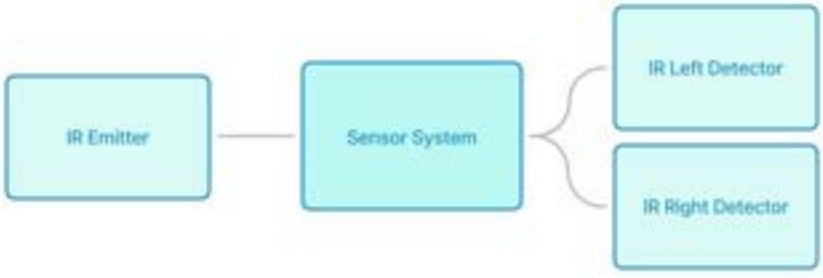

Figure 7: Sensor System Block Diagram

4. Hardware

The hardware is the physical components that are directed and controlled by internal software and code. Each component provides a specialized function that, when connected, create a fully functioning system. The integration of components such as power boards, switches, and memory storage allows the vehicle to perform its intended tasks. The sections below will examine these components in greater detail, highlighting their individual functions.

4.1. 1.2 MHz Boost DC/DC Converter

The LT1935 is a monolithic switching regulator that functions as a step-up DC/DC converter. It integrates a power switch and control circuitry, requiring only minimal external components to operate. The converter runs at a fixed 1.2 MHz switching frequency, which enables the use of small, low-profile inductors and capacitors. This high frequency operation makes it suitable for compact power supply designs where board space is limited.

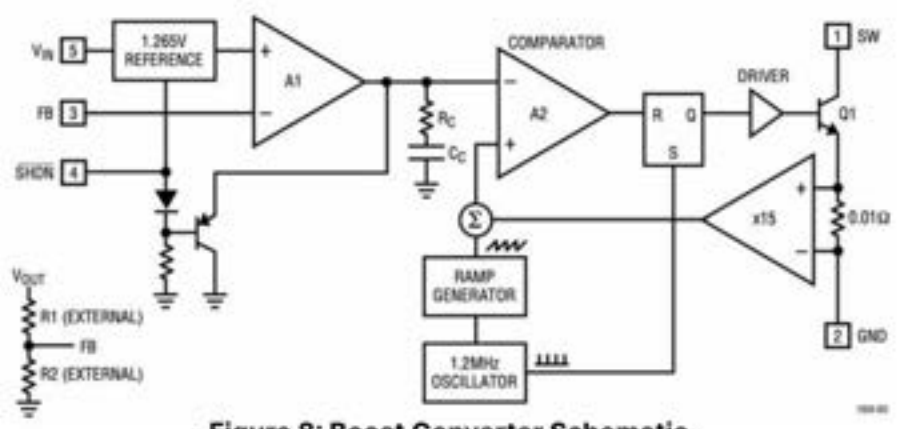

Figure 8: Boost Converter Schematic

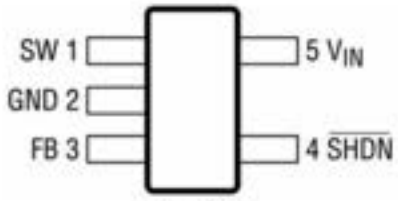

Figure 9: Boost Converter Pinout

4.2. Liquid Crystal Display

An LCD is a component that gives a visual output. It relies on the backlight to generate light, which passes through liquid crystals to create images on a pixel grid display, and each pixel adjusts light to create the image or words. In Figure 11, the current display on the LCD reads, "NCSU", "Wolfpack", "ECE 306", "GP I/O".

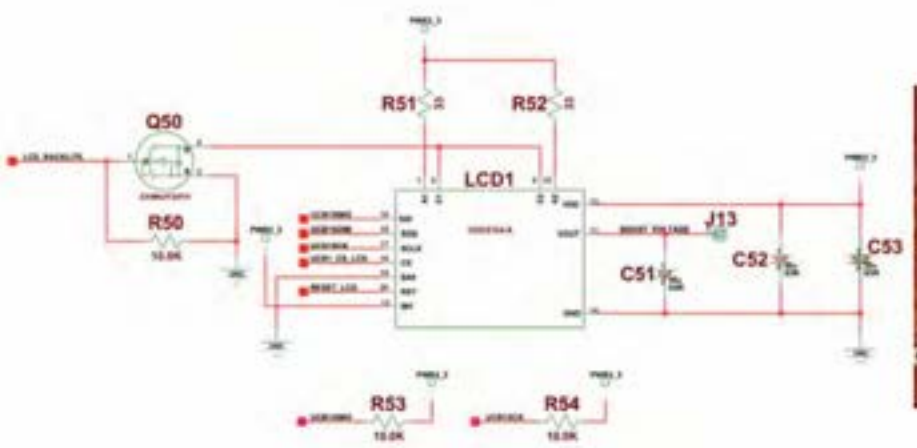

Figure 10: LCD Schematic

Figure 11: LCD Screen

4.3. Buck Boost Converter

The system utilizes two independent Single-Ended Primary Inductor Converters to regulate voltages to a stable 3.3V for digital electronics and 6.0V for the motors. This buck-boost topology is necessary because it uses a 4xAA battery pack, which starts around 6V and decreases with use. The Single-Ended Primary Inductor Converter can step the voltage down when the battery is full and step it up as the battery drains, maintaining a constant voltage supply.

The SW1 pin of the LT1935 drives a switch that controls current flow through two inductors, L1 and L2. A 1µF capacitor connects them, forming the core of the Single-Ended Primary Inductor Converter energy-transfer stage. When the switch turns on, energy is stored in the magnetic fields of the inductors and in the electric field of the coupling capacitor. When the switch turns off, this stored energy is released and delivered to the output through the diode and into the output capacitor.

The output voltage is regulated via the feedback pin, which is connected to a resistor divider of 20kΩ and 12.4kΩ. This network senses the output voltage and adjusts the peak inductor current accordingly to maintain a steady 3.3V and 6.0V output, regardless of input voltage fluctuations from the battery.

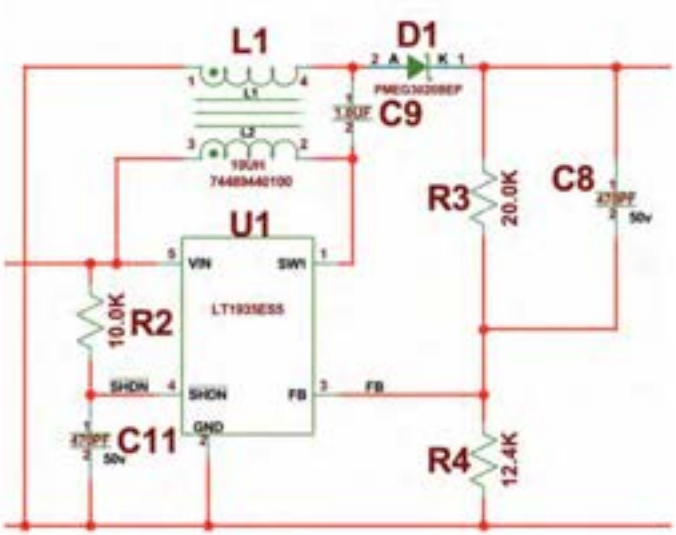

Figure 12: Buck Boost Converter Schematic

4.4. MSP430FR2355 MCU

The MSP430FR2355 microcontroller is an ultra-low power device and acts as the central control unit for the vehicle. Software is uploaded to the MCU via the Code Composer Studio IDE. It processes inputs, commands, and the connected components act accordingly. It includes smart analog combo modules, or 12-bit DACs, which handle signals such as condition inputs or sensor signals directly within the microcontroller. The MSP430FR2355 processes and carries out tasks reliably, all while functioning on limited energy.

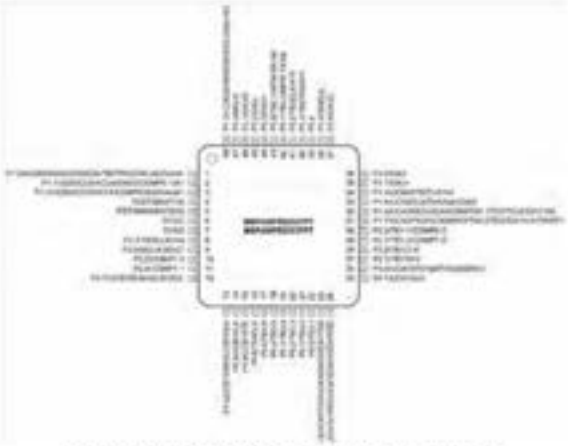

Figure 13: MSP430FR2355 Pinout

Figure 14: MSP430FR2355 LaunchPad Dev. Kit

4.5. Switch Slide SPDT

SW1 is a slide-style Single Pole Double Throw (SPDT) switch (part number OS102011MA1QN1) soldered to the power/display board used to manually toggle the circuit between two electrical paths. In this vehicle, it serves as a physical power control mechanism, allowing the user to either connect or disconnect the battery supply to the system, effectively turning the device on or off.

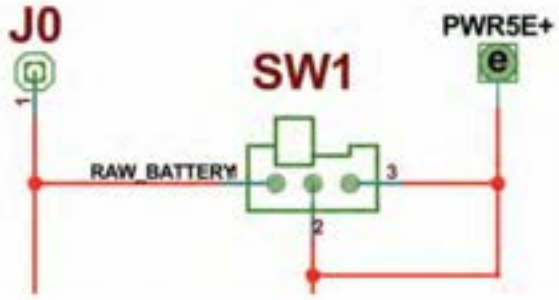

Figure 15: Switch Slide Schematic

4.6. Potentiometer Thumbwheel

The thumbwheel potentiometer serves as a manual variable resistor, which can be used to adjust signal levels or voltage output to be received through the ADC of the MSP430FR2355 microcontroller.

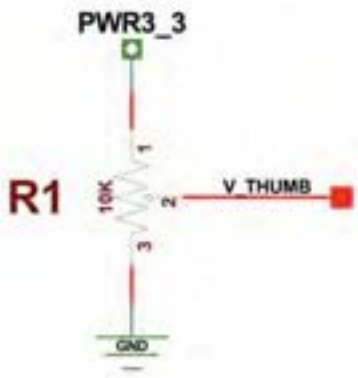

Figure 16: Thumbwheel Schematic

Figure 17: Thumbwheel Image

4.7. Power Board

The power board, or power system, serves as an attachment to the MSP430FR2355 through its pin sections. It receives 6V input from four AA batteries and distributes power to various components. It connects to the thumbwheel, LCD, two momentary push button switches, one SPDT switch, and two LEDs (red and green). The MSP430FR2355 interacts with these components via the power board, enabling control and communication. Additionally, the power board regulates and supplies 3.3V to the MSP430FR2355, the LCD, the ESP32 and all other electronics (excluding the motors).

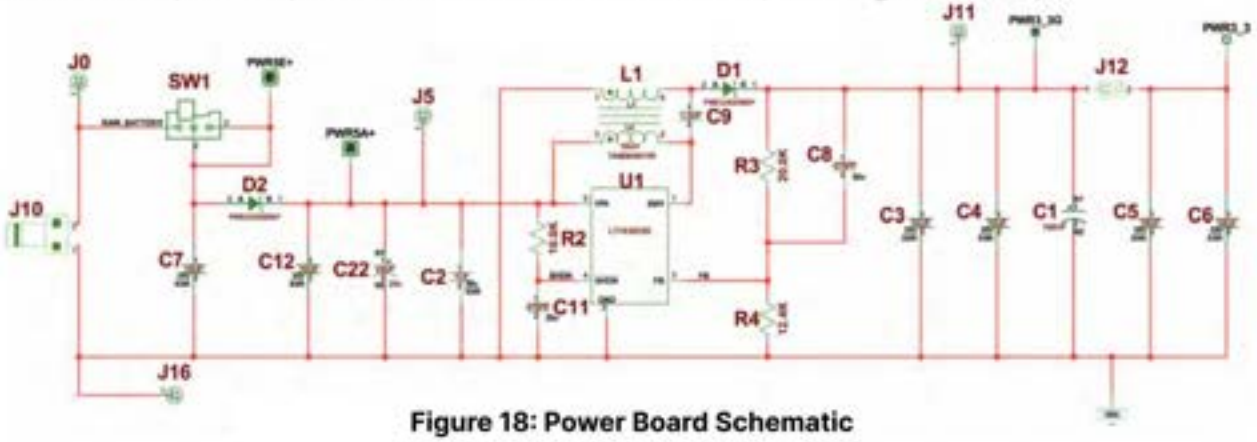

Figure 18: Power Board Schematic

Figure 19: Power Board

4.8. DAC Power Board

The DAC function is performed by the SAC3 module within the MSP430FR2355 microcontroller. This internal peripheral converts a digital value from the MCU into a precise analog voltage. This voltage is used to regulate the external SEPIC Buck-Boost converter, providing a stable 6 V supply to the motor FET board for consistent motor control.

Figure 20: DAC Board

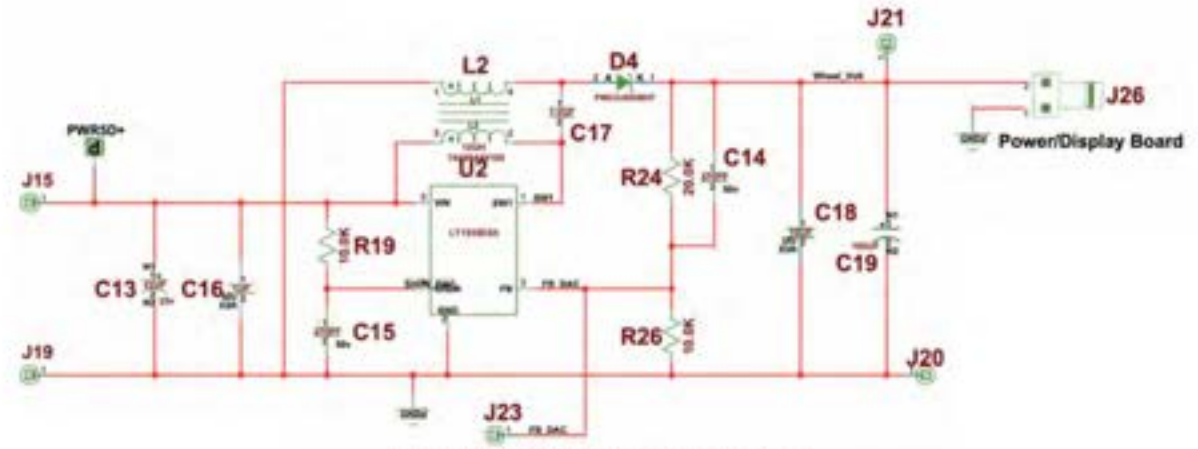

Figure 21: DAC Board Schematic

4.9. FET Board

The FET board contains two independent full H-bridges, one dedicated to each left and right DC motor. This design provides complete bidirectional control, enabling individual command of forward and reverse motion for each motor.

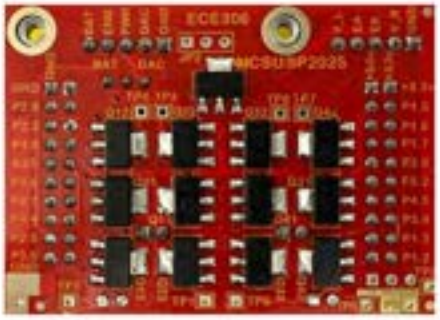

Figure 22: FET Board

4.10. Pololu Gearmotors

The Pololu gearmotors are small DC motors with built in gearboxes, ideal for powering smaller robots. The torque reduction ratio for these specific motors is 120:1, allowing the motor to run on voltages as low as 0.5 volts, but perform comfortably in the 3V to 6V range.

Figure 23: Pololu DC Motor

voltage	no-load performance	stall extrapolation
4.5V	150 RPM, 130 mA	1.8 kg·cm (25 oz·in), 1.25 A

Figure 24: Pololu Motor Key Specifications

4.11. H-Bridge

The full H-bridge circuit enables bidirectional control of a DC motor. It is built around a combination of two P-MOSFETs on the high side and four N-MOSFETs on the low side. To drive the motor forward, one high-side P-MOSFET and the diagonally opposite low-side N-MOSFET are activated, creating a current path from the power supply, through the motor, and to ground. To reverse the motor, the complementary pair of switches is turned on, reversing the current flow through the motor. This configuration provides the complete forward and reverse functionality required for the vehicle's drive system.

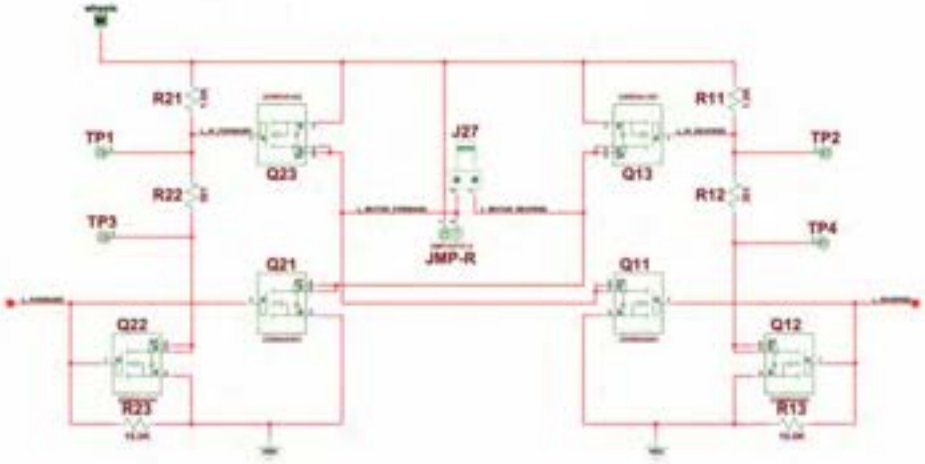

Figure 25: Right H-Bridge Schematic

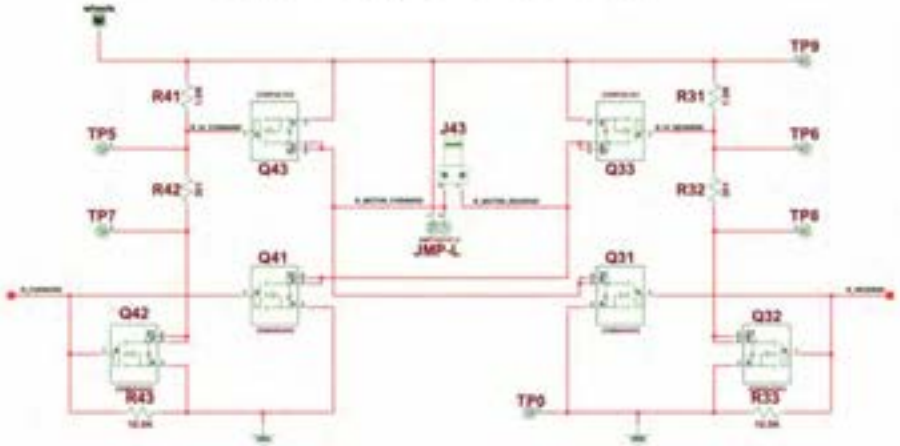

Figure 26: Left H-Bridge Schematic

4.12. N-MOSFET

The N-MOSFET is an N-channel transistor that acts as a low-side switch. It conducts current from its drain to its source when a sufficiently high voltage (typically 3.3V from the MCU) is applied to its gate relative to its source, turning the switch ON. When the gate voltage is low, the switch is OFF, blocking current flow.

Figure 27: N-MOSFET

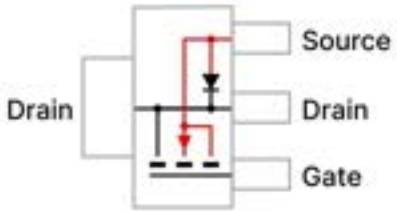

Figure 28: N-MOSFET Schematic

4.13. P-MOSFET

The P-MOSFET is a P-channel transistor that acts as a high-side switch. It conducts current from its source to its drain when a sufficiently low voltage (typically 0V/GND) is applied to its gate relative to its source, turning the switch ON. When the gate voltage is high, the switch is OFF, blocking current flow.

Figure 29: P-MOSFET

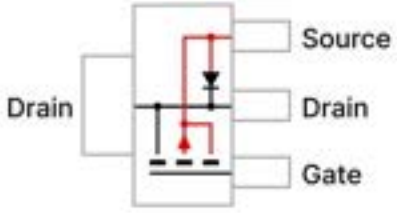

Figure 30: P-MOSFET Schematic

4.14. IR Emitter

The QED123 is a high-output infrared emitting diode that emits a narrow, focused beam of invisible light at 880 nm. Housed in a clear, peach-tinted T-1 3/4 plastic package, this AlGaAs diode features a tight 16° emission angle for efficient targeting. It is spectrally matched with the QSD123/QSD124 photosensor for optimal performance in sensing and remote control applications.

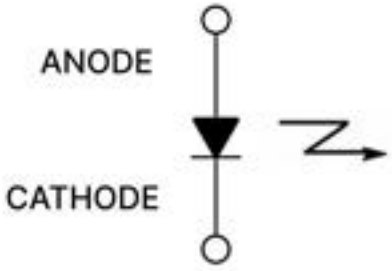

Figure 31: IR Emitter Schematic

Figure 32: IR Emitter

4.15. IR Detector

The QSD123 is a silicon phototransistor optimized to detect 880 nm infrared signals. Its black epoxy T-1 3/4 package incorporates a daylight filter and a narrow 24° reception angle, providing improved immunity to ambient light. Designed as the matched receiver for QED12X/22X/23X series emitters, it is suited for use in infrared detection circuits.

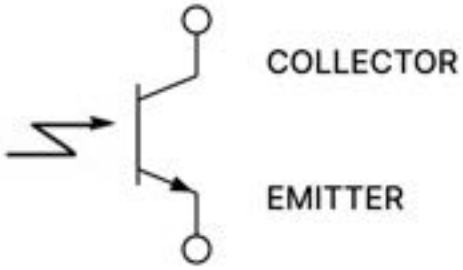

Figure 33: IR Detector Schematic

Figure 34: IR Detector

4.16. IR Emitter/Detector Board

The Emitter/Detector is the sensing module of the vehicle. It uses the infrared light from the IR Emitter and the IR Detector to register light bouncing off nearby objects and produces an electric signal to be read by the microcontroller.

Figure 35: IR Emitter/Detector Board

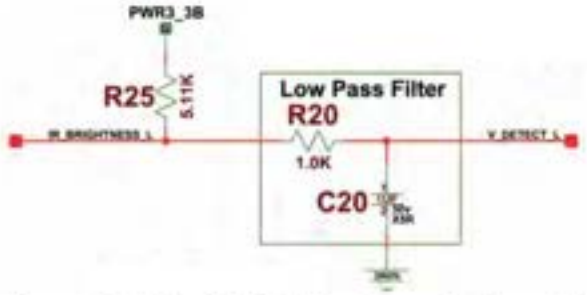

Figure 36: IR Left Side Line Detect Schematic

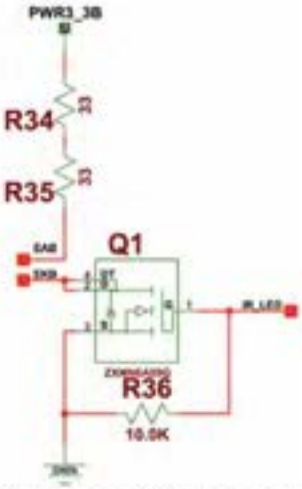

Figure 37: Center Emitter Schematic

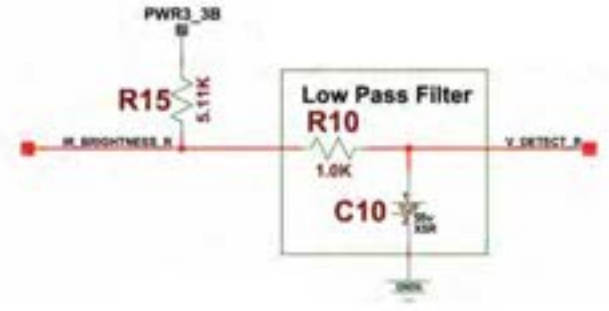

Figure 38: IR Right Side Line Detect Schematic

4.17. ESP32-WROOM-32E

The ESP32-WROOM-32E is a compact, highly integrated module that adds comprehensive wireless communication and processing capability to the embedded system. It provides both 2.4 GHz Wi-Fi and Bluetooth, allowing devices to connect to networks, send and receive data, or communicate directly with other external devices. The module includes a powerful dual-core microcontroller, onboard flash memory, and 26 versatile General-Purpose Input/Output (GPIO) pins for hardware interfacing. Additionally, it features common communication protocols such as UART, SPI, and I²C, enabling it to handle both complex processing tasks and wireless functionality within a single unit.

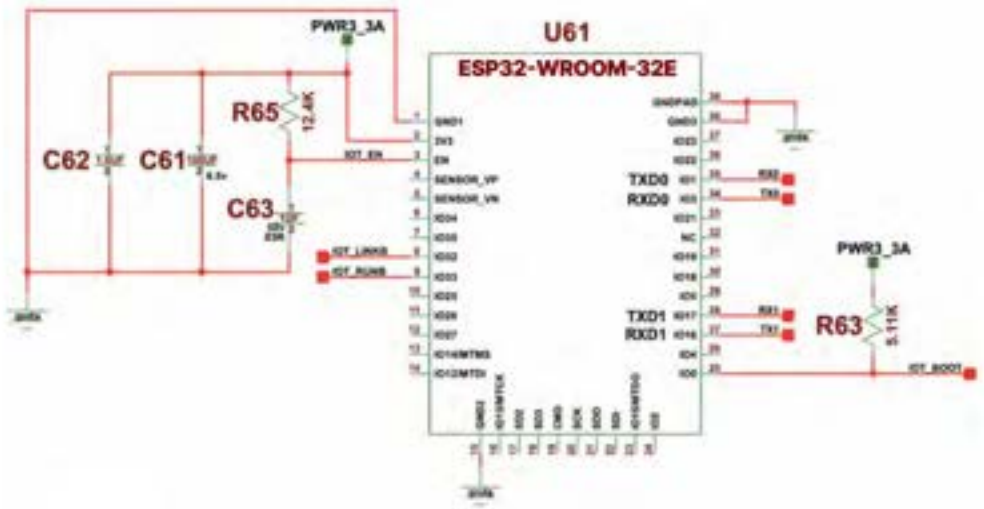

Figure 39: ESP32-WROOM-32E Schematic

Figure 40: ESP32-WROOM-32E

5. Power Analysis

This section characterizes the system's power consumption across two separate vehicle implementations to validate the power supply design and estimate operational battery life. The comparative analysis demonstrates consistent performance between bench testing and real battery operation, confirming the robustness of the power architecture for autonomous applications.

System State	Current (A)	Current (B)	Power (A)	Power (B)
MSP430FR2355 Only	13 mA	20 mA	78 mW	116 mW
+ LCD & Backlight	83 mA	81 mA	498 mW	471 mW
+ ADC & IR Emitters/LED	103 mA	104 mA	618 mW	604 mW
+ Wi-Fi (ESP32)	185 mA	186 mA	1.11 W	1.08 W
+ DAC & Motors (80% PWM)	1.04 A	1.12 A	6.24 W	2.51 W

Figure 41: System Power Measurements

Subsystem	Current (A)	Current (B)	Runtime (A)	Runtime (B)
MSP430FR2355	13 mA	20 mA	200 hrs	130 hrs
LCD & Backlight	70 mA	61 mA	37.1 hrs	42.6 hrs
ADC & IR Emitters/LED	20 mA	23 mA	130 hrs	113 hrs
Wi-Fi Module (ESP32)	82 mA	82 mA	31.7 hrs	31.7 hrs
DAC & Motors (80% PWM)	855 mA	934 mA	3.04 hrs	2.78 hrs
Everything Running	1.04 A	1.12 A	2.5 hrs	2.32 hrs

Figure 42: Subsystem Current Breakdown and Runtime Analysis

5.1. Measurement Procedure and Calculations

Current measurements were taken at the main battery input using a digital multimeter in series with the power supply. Kwali Bunker's measurements (Vehicle A) used a 6V bench supply, while Josiah Szobody's (Vehicle B) used actual 4xAA Power Max Ultra batteries with 2600 mAh capacity, measuring 6.02V open circuit and 5.81V under load.

Power calculations used the formula P = V × I, with Vehicle A at 6.00V and Vehicle B at 5.81V to reflect

actual operating conditions. Runtime projections applied the formula Runtime (hours) = 2600 mAh / I, where I is the measured current for each operating state. The subsystem currents in Figure 38 were calculated by analyzing the incremental differences between successive measurements in Figure 37.

5.2. Measured System-Level Current

The comparative analysis in Figure 41 reveals remarkably consistent power characteristics between implementations despite different measurement approaches. Both vehicles show nearly identical current draw patterns as subsystems are progressively enabled, validating the system's predictable power behavior.

The data confirms that the motor drivers account for approximately 83% of total system power during operation, representing the primary battery drain. The LCD backlight constitutes the most significant fixed load in idle states, while the ESP32 Wi-Fi module adds consistent overhead for wireless capability.

5.3. Subsystem Power Breakdown and Runtime Estimation

The detailed analysis in Figure 42 shows both vehicles achieve nearly identical runtime estimates across all operating modes. The calculations project 2-2.5 hours for full system operation, providing practical duration for extended testing, while efficient power management enables over 13 hours of continuous sensing and communication with motors disabled.

The close correlation between bench supply and actual battery measurements confirms the system performs as expected under real operating conditions, demonstrating the reliability of the power architecture for autonomous vehicle applications.

6. Test Process

Throughout testing, a digital multimeter was used to verify electrical functionality. Successful reflow and power integrity was confirmed by visually inspecting SMD solder joints, applying 4.5 V to the Power/ Display board, and probing key nodes to verify the regulated 3.3 V rail and expected pin/pad voltages. Validation of the LCD integration was also shown by applying 6 V to confirm backlight functionality, loading the provided software, and verifying that text rendered correctly and the board responded to button presses after assembly with the MSP430FR2355.

6.1. Power Board Voltage Test

To validate the 3.3 V rail on the Display board, 4.5 V was applied at the input and probed key nodes to confirm proper voltage regulation. The output measured consistently at around 3.3 V, indicating that the onboard regulator was operating correctly and providing stable power to the display circuitry.

Figure 43: Annotated Power Board

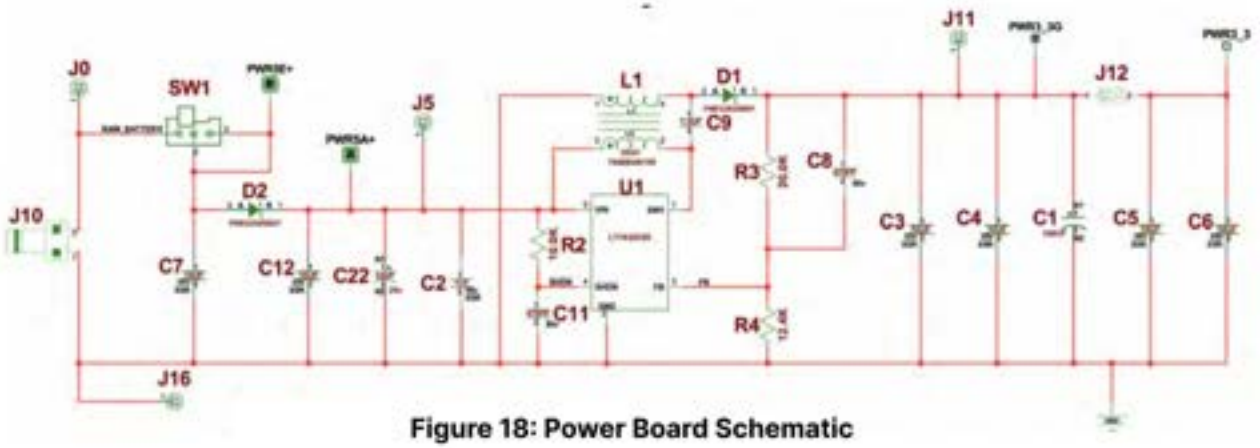

Figure 18: Power Board Schematic

6.2. DAC Voltage Test

To verify the 3.3 V rail on the DAC board, 4.5 V was supplied to the input and critical nodes were tested to validate appropriate voltage regulation. The output remained steady at approximately 3.3 V, demonstrating that the onboard regulator operated correctly and delivered stable power to subsequent components.

Figure 44: Annotated DAC Board

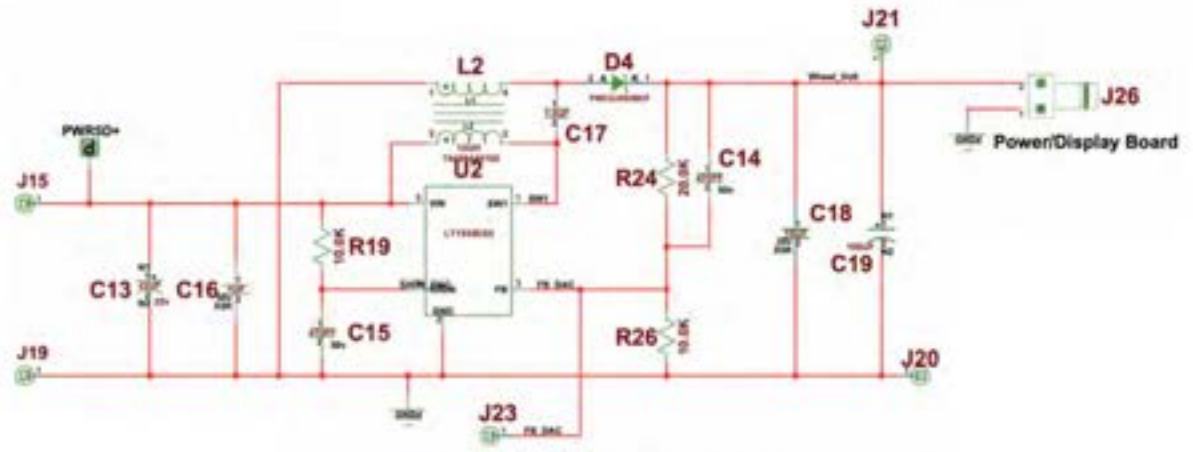

Figure 21: DAC Board Schematic

6.3. Switch Functional Test

To verify the power switch functionality, J0 (+) and J5 (−) were probed. With the switch in the OFF position, continuity mode showed OL (open), while Ohm mode measured approximately 1.377 kΩ and diode mode read ~0.148 V. These readings align with expected circuit characteristics.

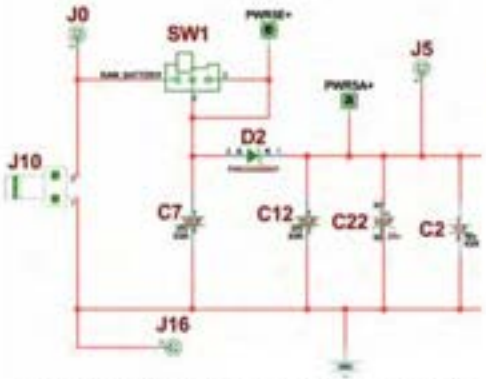

Figure 45: Switch Test Schematic

Figure 46: Switch Resistance

6.4. Battery Voltage Check

A digital multimeter was used to verify the input voltage and the operation of the voltage regulator. The battery voltage, measured at port J10 and across pads J0 (+) and J16 (−), read approximately 6.0 V. With the battery connected and the power switch in the OFF position, ~6.0 V was observed between J0 and J5, confirming continuity. When the switch was turned ON, voltages at J8 and J12 measured around 3.3 V, indicating that the regulator was functioning correctly and supplying the expected output voltage.

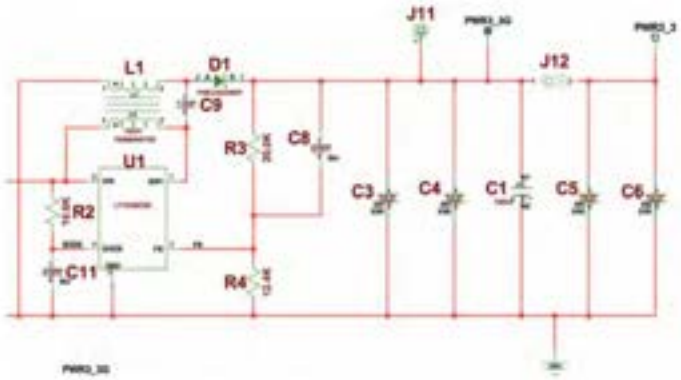

Figure 47: Battery Check Schematic

Figure 48: Battery Voltage

6.5. LCD Display Test

The display's solder joints were visually inspected to ensure proper connections. A multimeter was also used to confirm this by successfully checking continuity across the pins. After loading the provided software, the display successfully rendered text and responded correctly to inputs from the SW1 and SW2 buttons.

Figure 49: LCD Display

6.6. N and P-MOSFET Testing

The objective for the N-MOSFETs was to validate the switching functionality of two low-side switches controlled by MSP430FR2355 pins P6.0 and P6.1, confirming the drains would pull low to ground when the gates were enabled and rise to VBAT when disabled. After a visual solder inspection, software was loaded to toggle the gates. When enabling a motor channel, the GATE-to-GND voltage was probed to verify a logic-high signal of approximately 3.3V from the MCU. The DRAIN-to-GND voltage was then measured to confirm it successfully switched from VBAT (OFF state) to ~0V (ON state).

A similar validation was performed for the high-side P-MOSFETs. The same MCU pins were used for control, but through a level-shifting circuit to provide the necessary voltage range. With the software running, the GATE-to-SOURCE voltage was measured to confirm it switched between ~VBAT (to turn the FET OFF) and ~0V (to turn the FET ON). The DRAIN-to-GND voltage was then probed, verifying it switched from a high-impedance state (near 0V when OFF with no load) to approximately VBAT when the P-MOSFET was enabled and supplying power to the load.

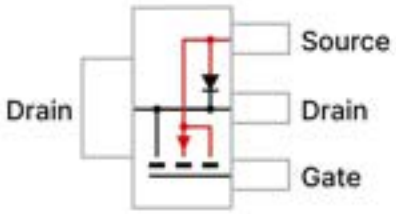

Figure 28: N-MOSFET Schematic

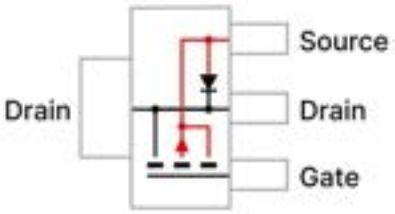

Figure 30: P-MOSFET Schematic

6.7. H-Bridge Tests

6.7.1. R_FORWARD Signal Test

With the battery connected, test points TP1 and TP3 were measured. When the R_FORWARD signal was off, both TP1 and TP3 read the full battery voltage. When R_FORWARD was activated, TP1 dropped to half the battery voltage or less, while TP3 dropped to approximately ground potential.

6.7.2. R_REVERSE Signal Test

With the battery connected, test points TP2 and TP4 were measured. When the R_REVERSE signal was off, both TP2 and TP4 read the full battery voltage. When R_REVERSE was activated, TP2 dropped to half the battery voltage or less, while TP4 dropped to approximately ground potential.

6.7.3. L_FORWARD Signal Test

With the battery connected, test points TP5 and TP7 were measured. When the L_FORWARD signal was off, both TP5 and TP7 read the full battery voltage. When L_FORWARD was activated, TP5 dropped to half the battery voltage or less, while TP7 dropped to approximately ground potential.

6.7.4. L_REVERSE Signal Test

With the battery connected, test points TP6 and TP8 were measured. When the L_REVERSE signal was off, both TP6 and TP8 read the full battery voltage. When L_REVERSE was activated, TP6 dropped to half the battery voltage or less, while TP8 dropped to approximately ground potential.

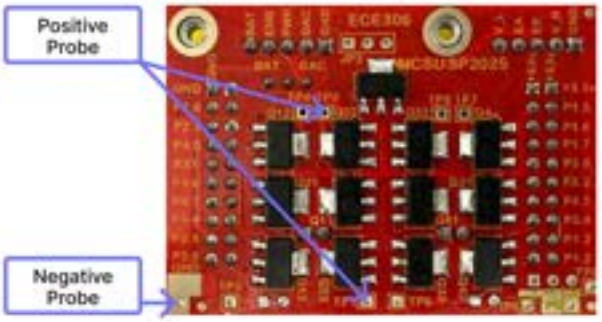

Figure 50: R_FORWARD Annotated FET Board

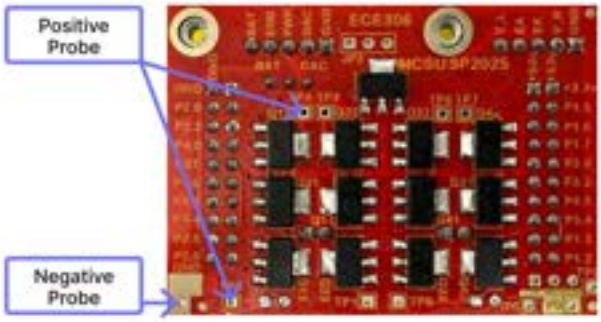

Figure 51: R_REVERSE Annotated FET Board

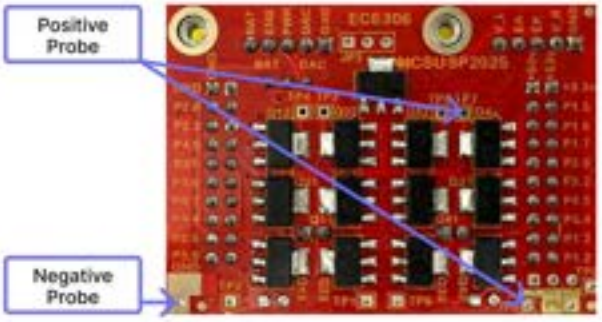

Figure 52: L_FORWARD Annotated FET Board

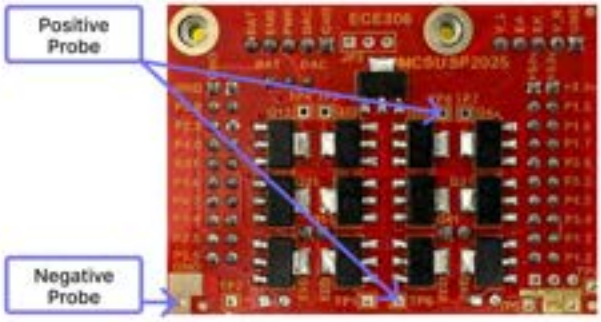

Figure 53: L_REVERSE Annotated FET Board

6.8. N-MOSFET Fault

Initially, the drain of each N-MOSFET remained near 0V regardless of the gate signal. Probing confirmed the MCU pins were correctly outputting 3.3V, which ruled out a software issue. The fault was traced to the supply-select jumper on the FET board, which was bringing in nonexistent DAC power (PWR5B+) to the wheels, preventing battery power from reaching the N-MOSFET sources. The jumper was moved to connect the battery power (PWR5C+) instead to the wheels, restoring power to the N-MOSFETs. Following this correction, both N-MOSFETs operated correctly, with the drains switching between VBAT and ground as intended.

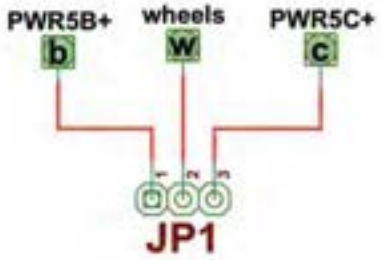

Figure 54: Motor Power Input Schematic

6.9. Forward Drive Check

The forward drive check confirmed the vehicle's ability to move under MCU control. With both motors commanded forward, the car traveled in a straight line for the expected duration, verifying correct integration of the software and hardware. This successful test established a foundation for all subsequent motion routines and validated the functionality of the half H-bridge's forward drive path.

6.10. Maneuver Test

After establishing basic forward motion, the vehicle was programmed for more complex maneuvers to demonstrate trajectory control using fixed power levels and precise timing. This involved a process of software calibration, where parameters such as drive time, turn duration, and the specific fixed power level for each motor were fine-tuned for each desired pattern.

The circle was achieved by setting one wheel to a constant high power level while pulsing the other, creating a constant arc. The figure-8 was formed by sequencing a left arc and a right arc, and the triangle was executed through three sequential forward segments with turns in between. Through these tests, the vehicle successfully demonstrated its ability to navigate a variety of precise paths, showcasing a range of motion from smooth arcs to sharp, angular turns.

6.11. Reverse Drive Check

The reverse drive check was also performed to validate the vehicle's backward motion. With the motors commanded in reverse, the car traveled backward as expected, confirming the correct functionality of the reverse drive path and completing the basic motion validation for both directions.

6.12. Motor Connector Fault

It was discovered that the blue two-pin connectors supplying power to the motors from the FET board were not making reliable contact. To address this issue, each connector was re-crimped to ensure proper mechanical and electrical connection. After re-crimping, continuity tests were performed using a digital multimeter to verify that each pin maintained a solid electrical path through the connector. All connections passed continuity testing after the repair, confirming that the connectors were properly seated and capable of reliably carrying current to their respective motors.

Figure 55: Motor Connector

6.13. ADC Verification

The MSP430FR2355 was first configured to read analog input from the thumb wheel while the motors remained disconnected to prevent noise and unintended motion. The ADC output, an integer value, was converted into ASCII characters for display, providing a direct visual representation of the ADC's operation on the LCD as the thumb wheel position changed. This initial procedure verified correct ADC sampling, proper ASCII conversion, and reliable LCD feedback before incorporating detector inputs.

The system was then tested by toggling the IR emitter on and off and verifying the display updated correctly. ADC readings from the left and right detectors were recorded over both white paper and black tape, with the emitter active and inactive. As expected, readings decreased when the emitter was on and over the white surface, and increased over the black tape. This confirms the detectors correctly responded to changes in reflected infrared light, validating the complete sensor data acquisition chain.

Figure 56: ADC Content on LCD

6.14. Line Detection with Motion Test

The vehicle was activated with switch 1 inside a 3-foot diameter circle of black tape. It advanced until its detectors identified the line, triggering a stop and a self-alignment sequence to position the sensors directly over the tape. Throughout the maneuver, detector readings and display output were monitored, showing a clear distinction between the white and black surfaces. The test successfully demonstrated the system's ability to autonomously detect a line, halt, and orient itself, confirming effective sensor and software integration.

6.15. FET Board Short Circuit Fault

During the initial power-on sequence, a critical fault occurred without any user command. The system immediately exhibited uncontrolled motor oscillation, significant overheating of the FETs, and erratic flickering of the LCD display.

Troubleshooting isolated the fault to the motor driver (FET) board, where an unintended electrical bridge was detected between critical control lines. Specifically, a non-visible short was found connecting the L_REVERSE signal to the LCD_RESET line, and the R_REVERSE signal to the LCD_BACKLIGHT line. This defect caused a direct and conflicting interaction between the motor drive circuitry and the display controller upon power application.

The issue was resolved by replacing the FET 2×10 header for J4B and J2B and cleaning the board before resoldering. This process eliminated the unintended connection, which was suspected to be caused by a microscopic solder bridge beneath the pin header. Post-repair, the system passed all power-on and operational tests, confirming the fault was corrected.

Figure 57: FET Board Replaced Header

6.16. Line Following Test

To calibrate the line-following behavior, the vehicle was first placed over white and black surfaces to record the minimum and maximum ADC values for both IR detectors. These values established distinct thresholds for "on-line" and "off-line" states for each sensor, creating a calibration profile for reliable surface detection.

The vehicle was then tested with one detector on the tape and the other off to verify the thresholds triggered correctly under real-world conditions. The control logic was tuned so that when a detector reported "on line" via comparison of calibrated black and white ADC values, the opposite motor received a higher duty cycle (higher speed), creating a proportional steering correction toward the tape. This differential steering approach allowed for smooth path correction without oversteering.

Finally, the vehicle was placed on a loop of black tape where it successfully followed the continuous path, maintaining consistent tracking and correcting drift automatically. The test confirmed successful integration of the IR sensor calibration, threshold detection logic, and proportional motor control for stable autonomous line following.

6.17. Battery Monitoring Test

The V_BAT node on the Power/Display board was configured as an analog input to monitor battery voltage and prevent system issues at low power. A test program sampled V_BAT, converted the ADC value to a pack voltage using the resistor-divider ratio, and displayed the result on the LCD for verification.

The voltage reading was calibrated using a fresh battery pack and a bench power supply. The displayed voltage was adjusted to match multimeter readings within a small margin of error across the expected operating range, ensuring accurate monitoring.

Once calibrated, conditional checks were implemented to compare the battery voltage against warning, limit, and stop thresholds. The system displays a low-battery message at the warning level, restricts motor run time at the limit level, and disables all motor movement at the stop level to protect the hardware. This test confirmed accurate voltage measurement and functional battery-protection logic.

Figure 58: Battery Monitoring Display

6.18. UCA1 Communication Test

The UART source files were verified using the USB back-channel connected to eUSCI_A1 (UCA1). UCA1 was configured for 8-N-1 operation (8 data bits per character, No parity for error checking, and 1 stop bit to signal packet completion) at 115,200 and 460,800 baud, selectable at runtime via button presses to call the serial initializations functions with different baud rate settings.

A test program was then used to transmit status strings and echoed received characters. A terminal program on a host PC confirmed that all transmitted characters appeared at both baud rates and that characters typed into the terminal were received and echoed back without errors.

To further verify timing and logic levels on the UCA1 TX, the signal was examined with a Digilent Analog Discovery UART tool to verify correct start/stop bits and bit timing at both baud rates with SMCLK = 8 MHz. These measurements confirmed the USB back-channel operates correctly and that the UART configuration supports both UCA0 and UCA1 for further expansion.

6.19. UART and Wi-Fi Communication Verification

The initial verification of UART and Wi-FI communication focused on confirming that the UCA0 serial interface on the MSP430FR2355 FRAM board could reliably transmit and receive data. The port was configured for 115,200 baud. To confirm hardware integrity, a loopback test was performed: characters sent from the PC via UCA1 were routed through the debug pass-through logic to UCA0, and immediately returned to UCA1 (PC → A1 → A0 → A1 → PC). Interrupt-driven operation was used to confirm received characters could be echoed back without error, establishing that the communication path on the MSP430FR2355 side was functioning correctly before attaching the ESP32 module.

After UCA0 operation was confirmed, the ESP32 IoT module was connected and controlled through AT commands sent from a PC terminal, routed through the FRAM board. A full set of communication checks was performed, including hard reset, soft reset, firmware identification ("AT+GMR"), UART configuration, Wi-Fi mode selection, and network scanning. Commands like "AT+CIPSTATUS", "AT+CWLAP", and "AT+CWJAP" confirmed proper responsiveness. The module was then configured with a hostname, automatic reconnect behavior, multi-connection mode, and a TCP server port. SSID and IP information were successfully retrieved and displayed on the LCD, ensuring correct parsing and formatting.

With the ESP32 configured and active on the network, communication was tested wirelessly using the Magic Smoke iOS application. The app connected to the ESP32's TCP server using the assigned IP address and port. Incoming data arrived in the expected framed format beginning with the +IPD indicator. A simple security command structure (containing a start character, a PIN, a direction command, and a time value) was used to validate control messages. These validated commands were parsed by the FRAM code and translated into vehicle actions such as forward movement, reverse travel, and timed turning maneuvers, confirming full end-to-end operation of the Wi-Fi communication path.

7. Software

The software is built around a superloop architecture in main.c that continuously executes core control logic using foreground-background architecture. This is supported by modular initialization routines (ports.c, timers.c, ADC.c) that configure the microcontroller's hardware peripherals at startup. The system's responsiveness is achieved through interrupt service routines (interrupts_timers.c, interrupts_ADC.c) that handle time-critical events, such as periodic timing and sensor data acquisition, in the background. Together, these components create a cohesive system that reads sensor inputs, makes control decisions, and provides user feedback.

7.1. main.c

The main.c file contains the main() function that serves as the application entry point, implementing a superloop architecture for continuous system operation. After enabling interrupts and unlocking GPIO ports, it calls a series of initialization functions: Init_Ports(), Init_Clocks(), Init_Timers(), Init_LCD(), Init_ADC(), Init_Serial_UCA0(), Init_Serial_UCA1(), and Init_IOT() to configure all hardware peripherals. The function then initializes system variables, sets main_state to STATE_WAITING, displays startup messages, and ensures motors are off with motor_shutdown().

The program enters its infinite while(ALWAYS) loop where it continuously executes core application logic. Each cycle calls Main_State_Machine() to process sensor data and control vehicle behavior, followed by display_handling() to format sensor information for display. Background operations include UART_Processing() to handle incoming commands, scroll() for thumbwheel-based display control, Display_Process() to refresh the LCD, and P3OUT ^= TEST_PROBE to toggle a test pin for system monitoring, ensuring responsive control while maintaining consistent user feedback and communication.

7.2. ports.c

The ports.c file implements the Init_Ports() function, which configures all General-Purpose Input/Output pins on the MSP430FR2355 microcontroller. This master function sequentially calls subroutines Init_Port1() through Init_Port6() to initialize each port. The initialization process clears output registers, defines pin directions, and selects functions (GPIO, ADC, UART or SPI) for each pin. Critical inputs like SW1 and SW2 are configured with pull-up resistors and interrupt enable, while outputs including motor control pins (L_FORWARD, R_FORWARD, L_REVERSE, R_REVERSE), status LEDs (RED_LED, GRN_LED), and LCD_BACKLITE are set to safe initial states to ensure reliable peripheral operation.

7.3. timers.c

The timers.c file contains the Init_Timers() function, which serves as the master timer initialization routine that calls Init_Timer_B0() and Init_Timer_B3() to configure the system's timing infrastructure. The Init_Timer_B0() function sets Timer B0 to use the Sub-Master Clock (SMCLK) in continuous mode with clock division through TB0CTL |= ID__8 and TB0EX0 = TBIDEX__4, while the Init_Timer_B3() function configures Timer B3 for PWM generation in up mode.

The timer's Compare Capture Register 0 is configured with TB0CCR0_INTERVAL and its interrupt is enabled via TB0CCTL0 |= CCIE. Timer B3 establishes PWM outputs for motor control pins (L_FORWARD, R_FORWARD, L_REVERSE, R_REVERSE) and LCD backlight dimming using OUTMOD_7, with initial duty cycles set to WHEEL_OFF for motors and PERCENT_80 for backlight control, enabling precise motor speed regulation and display lighting management.

7.4. ADC.c

The ADC.c file implements the Init_ADC() function, which configures the ADC12 module for 12-bit

analog-to-digital conversions. The function performs register-by-register setup: ADCCTL0 configures sample-and-hold time and enables the ADC; ADCCTL1 selects MODCLK source and single-conversion mode; ADCCTL2 sets 12-bit resolution; and ADCMCTL0 selects voltage references and initial input channel A2. After enabling the conversion complete interrupt with ADCIE |= ADCIE0, the function enables conversions with ADCCTL0 |= ADCENC and triggers the first conversion with ADCCTL0 |= ADCSC, preparing the ADC for sensor data acquisition.

7.5. UART.c

The UART.c file implements the Init_Serial_UCA0() and Init_Serial_UCA1() functions, which configure the MSP430FR2355's UART peripherals for serial communication at multiple baud rates from 9600 to 460800. These functions perform register-level configuration of clock sources, frame formats, and modulation settings while the Init_IOT() function manages ESP32 initialization through AT command sequences for Wi-Fi connection, TCP server setup, and network parameter retrieval including IP and MAC addresses.

7.6. DAC.c

The DAC.c file implements the Init_DAC() function, which configures the MSP430FR2355's Shared Analog Channel (SAC3) to operate as a Digital-to-Analog Converter with an integrated operational amplifier. The function performs a register-level setup to select VCC as the DAC's voltage reference and sets the update mode to latch immediately upon writing data to the register. It then configures the internal operational amplifier in a Buffer Mode (Gain = 1), where the DAC output is fed directly to the positive input and the Op Amp output is routed back to the negative input for stabilization. Finally, after setting an initial DAC output value (DAC_BEGIN), the function enables the Timer B0 Overflow interrupt. This interrupt is utilized to initiate a voltage ramp-down sequence to stabilize the external power supply, ensuring a reliable output voltage before motor operation begins

7.7. interrupts_timers.c

The interrupts_timers.c file contains the Timer0_B0_ISR() and TIMER0_B1_ISR() interrupt service routines that handle Timer B0 interrupts. The Timer0_B0_ISR() executes on each Timer B0 CCR0 interrupt to schedule the next interrupt by incrementing TB0CCR0 += TB0CCR0_INTERVAL and manages display timing through the display_count variable, while the TIMER0_B1_ISR() handles CCR1 and CCR2 interrupts for button debouncing and the overflow interrupt for DAC voltage ramping.

When the display_count reaches 200, the Timer0_B0_ISR() sets update_display = TRUE to request a display refresh and resets the counter, then triggers ADC sampling with ADCCTL0 |= ADCSC. The TIMER0_B1_ISR() implements software debounce counters for SW1 and SW2 that disable timer interrupts and re-enable switch interrupts after reaching DEBOUNCE_THRESHOLD, ensuring reliable button press detection. It also provides ramping for the DAC voltage to keep the motor voltage supply stable.

7.8. interrupts_ADC.c

The interrupts_ADC.c file implements the ADC_ISR() function, which handles ADC conversion complete interrupts. The ISR disables conversions with ADCCTL0 &= ~ADCENC, then uses the ADC_Channel variable to cycle through three sensor inputs. For each channel, it stores the ADCMEM0 result in the appropriate global variable (ADC_Left_Detect, ADC_Right_Detect, or ADC_Thumb) and configures ADCMCTL0 for the next channel in sequence (A2, A3, A5). After three samples, the counter resets and the ISR re-enables conversions with ADCCTL0 |= ADCENC and triggers the next conversion with ADCCTL0 |= ADCSC, maintaining continuous sensor sampling without main loop intervention.

7.9. UART_interrupts.c

The UART_interrupts.c file contains the eUSCI_A0_ISR() and eUSCI_A1_ISR() interrupt service routines that handle real-time UART data processing with circular buffer management. The ISRs manage separate transmit and receive buffers for both UART channels, implement character-by-character data flow between the ESP32 and debug terminal, and provide configurable echo functionality in debug mode while maintaining buffer integrity through wrap-around indexing and busy flags to prevent data loss during high-speed communication.

8. Flow Charts

8.1. Main Flow Chart

Shows the program's superloop architecture. After initializing all hardware, it enters an infinite loop that continuously runs the state machine, updates the display, and toggles a test probe.

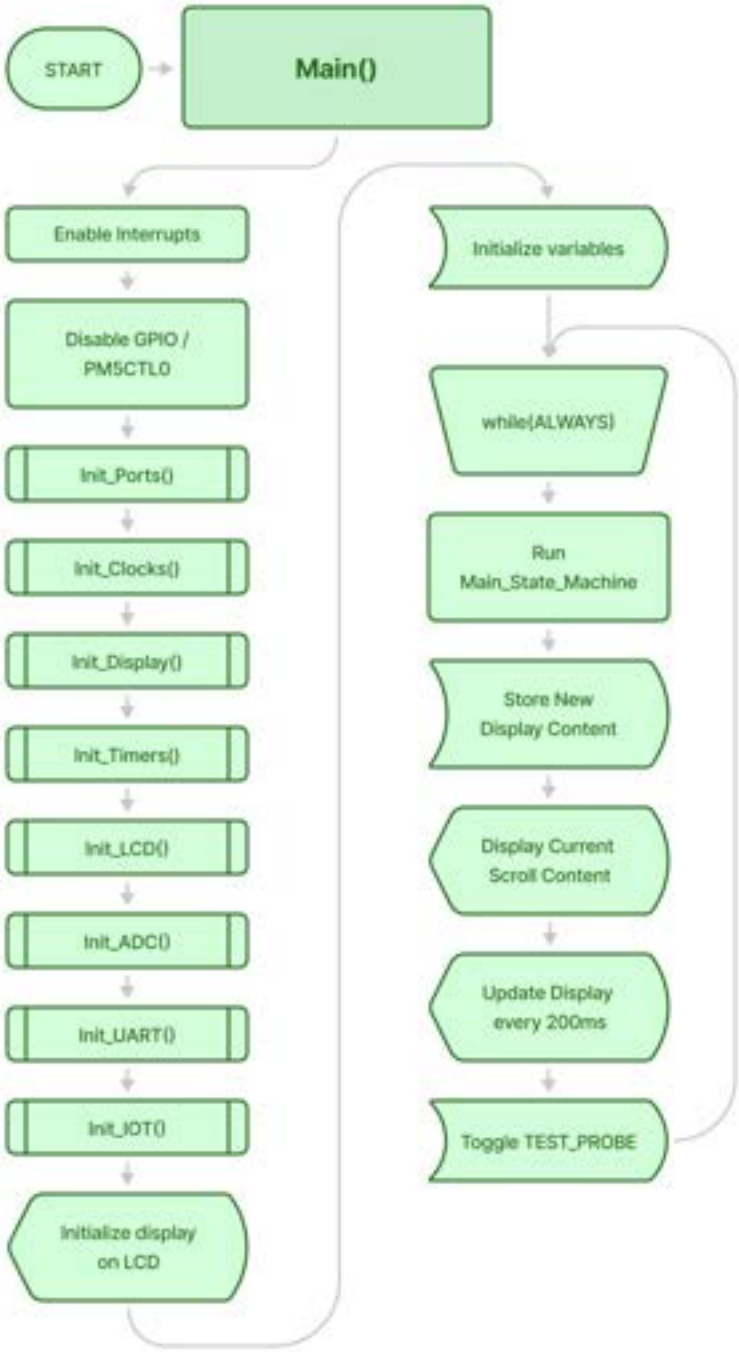

Figure 59: Main Flow Chart

8.2.1. Port Initialization Flow Chart

Illustrates the master port initialization sequence, which calls individual initialization functions for all six GPIO ports (Port 1 through Port 6) to configure the microcontroller's pins.

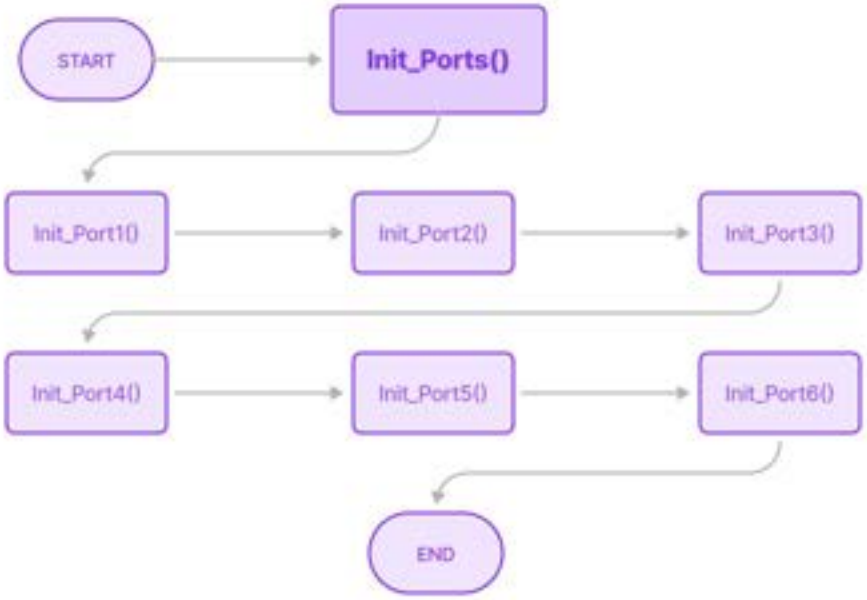

Figure 60: Init_Ports() Flow Chart

8.2.2. Port 1-3 Initialization Flow Charts

Figure 61: Port 1 Flow Chart

Figure 62: Port 2 Flow Chart

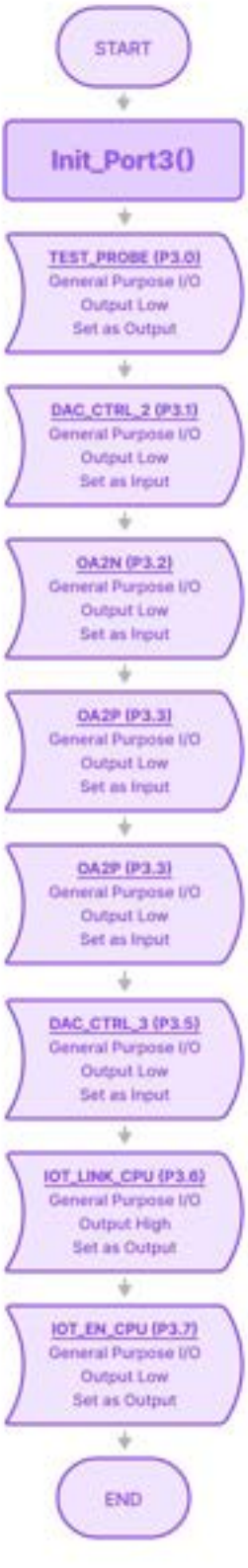

Figure 63: Port 3 Flow Chart

8.2.3. Port 4-6 Initialization Flow Charts

Figure 64: Port 4 Flow Chart

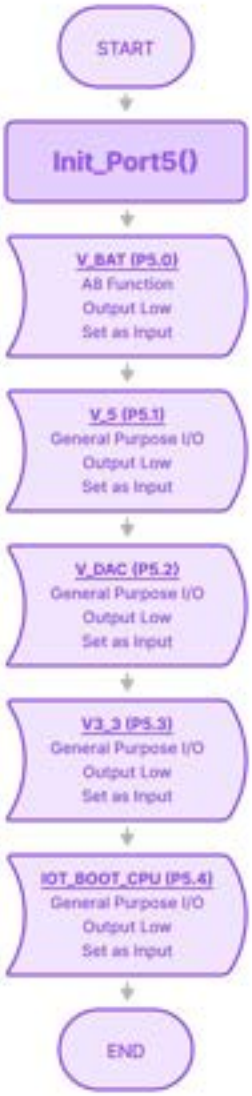

Figure 65: Port 5 Flow Chart

Figure 66: Port 6 Flow Chart

8.3.1. Timer Initialization Flow Chart

Depicts the master timer initialization routine, which calls the individual initialization functions for Timers B0 through B3 to set up the system's timing infrastructure.

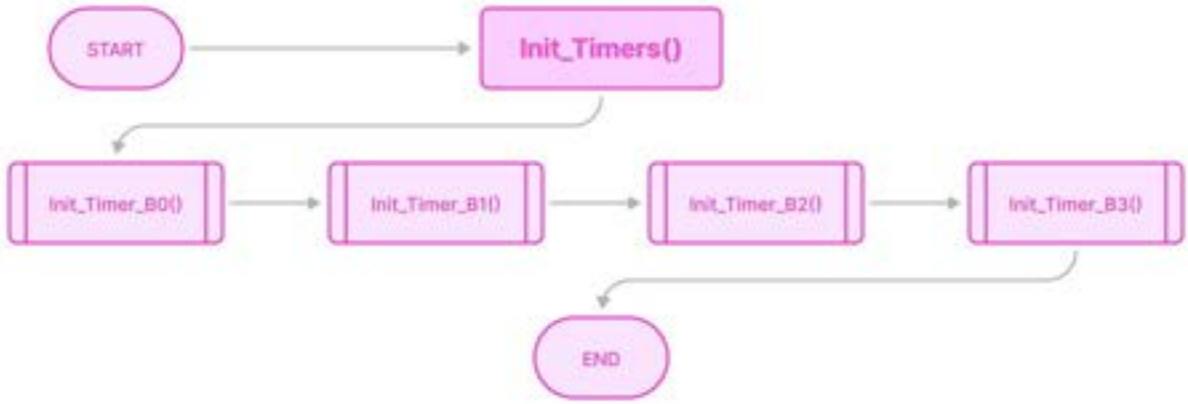

Figure 67: Init_Timers() Flow Chart

8.3.2. Timer B0-B3 Initialization Flow Charts

Figure 68:
Timer B0 Flow Chart

Figure 69:
Timer B1 Flow Chart

Figure 70:
Timer B2 Flow Chart

Figure 71:
Timer B3 Flow Chart

8.4. ADC Initialization Flow Chart

Outlines the step-by-step register configuration to set up the ADC for 12-bit conversions, including setting sample time, resolution, voltage reference, and starting the first conversion.

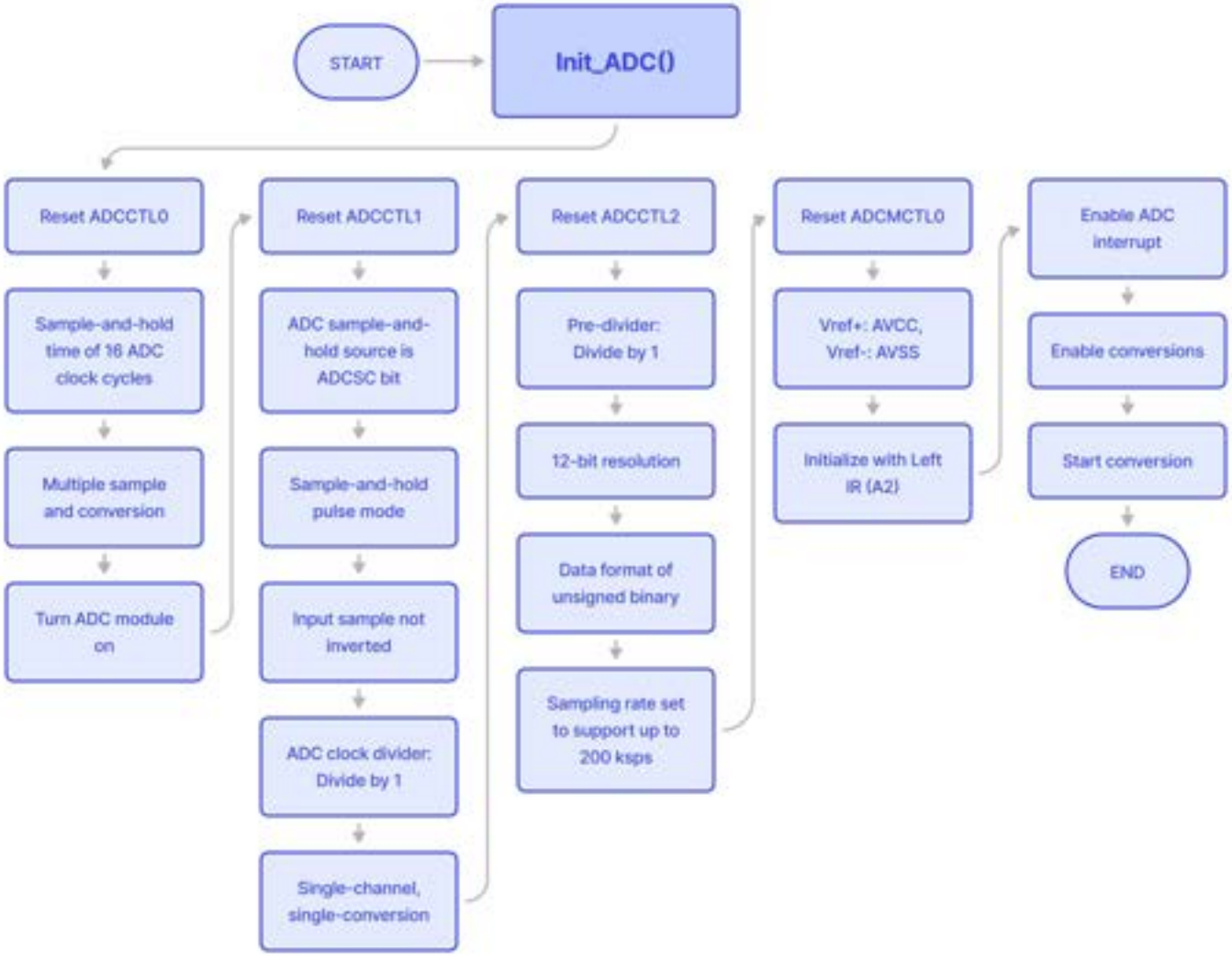

Figure 72: Init_ADC() Flow Chart

8.5. UCA0 Initialization Flow Chart

Shows the UART configuration sequence for UCA0 (ESP32). Initializes baud rate, frame format, and enables receive interrupts for serial communication.

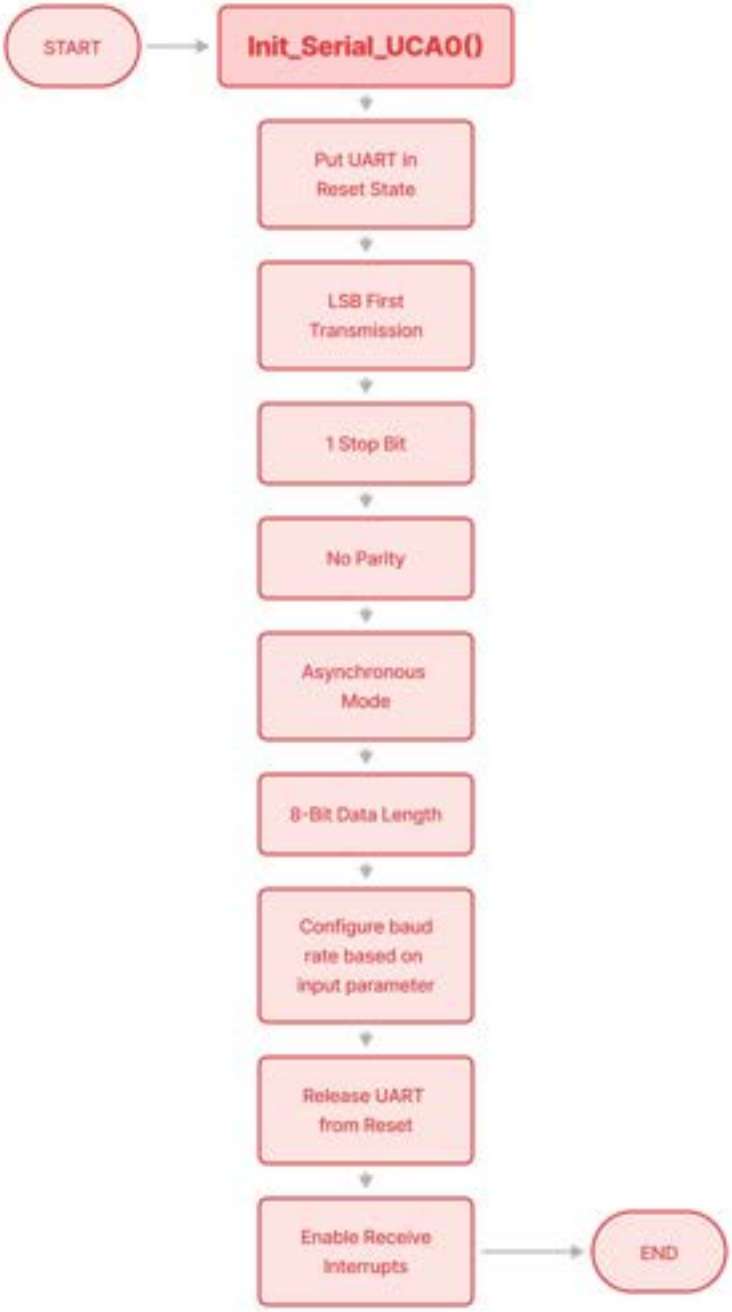

Figure 73: Init_Serial_UCA0() Flow Chart

8.6. UCA1 Initialization Flow Chart

Shows the UART configuration sequence for UCA1 (PC Debug). Initializes baud rate, frame format, and enables receive interrupts for serial communication.

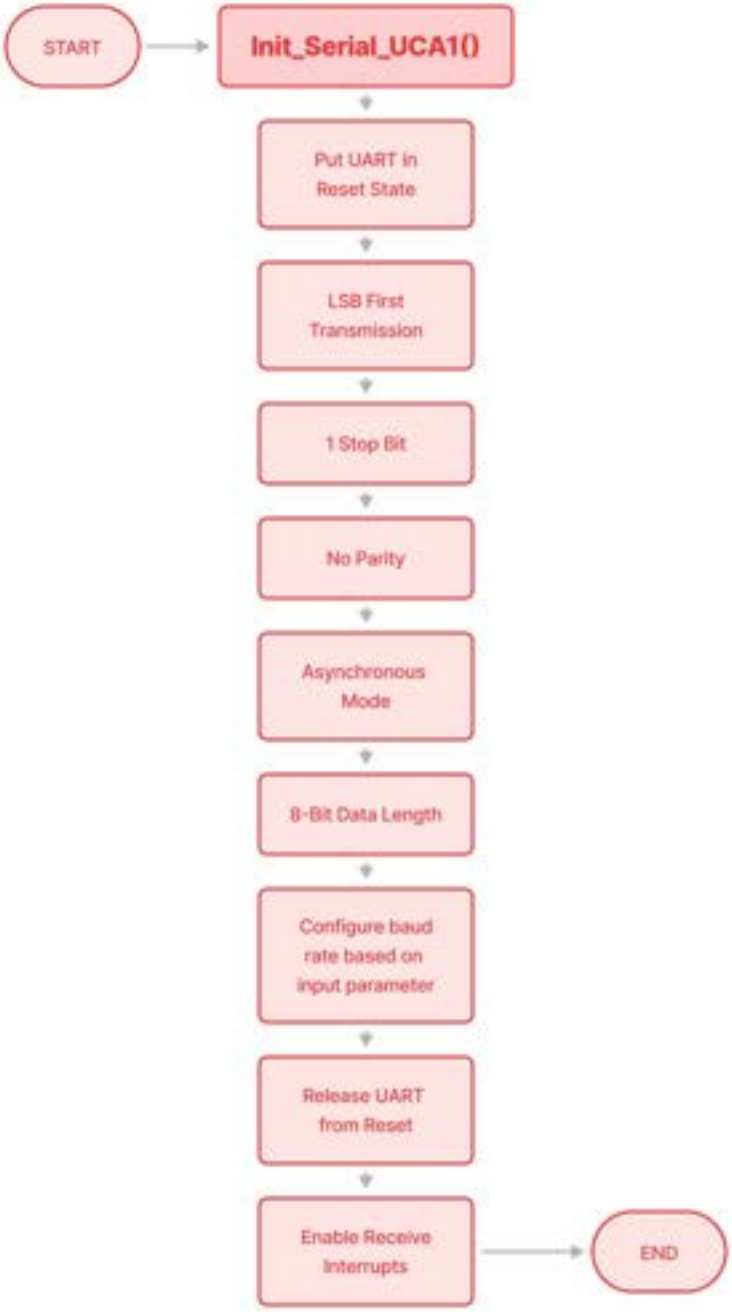

Figure 74: Init_Serial_UCA1() Flow Chart

8.7. DAC Initialization Flow Chart

Outlines the process for configuring the Digital-to-Analog Converter (DAC) and its buffer. This is followed by writing the initial output voltage and enabling the timer interrupt necessary for stabilizing the external power system.

Figure 75: Init_DAC() Flow Chart

8.8. Timer B0 CCR0 Interrupt Flow Chart

Describes the Timer0_B0_ISR(), which schedules the next interrupt, counts cycles to trigger a display update every 200ms, and starts the next ADC sample conversion.

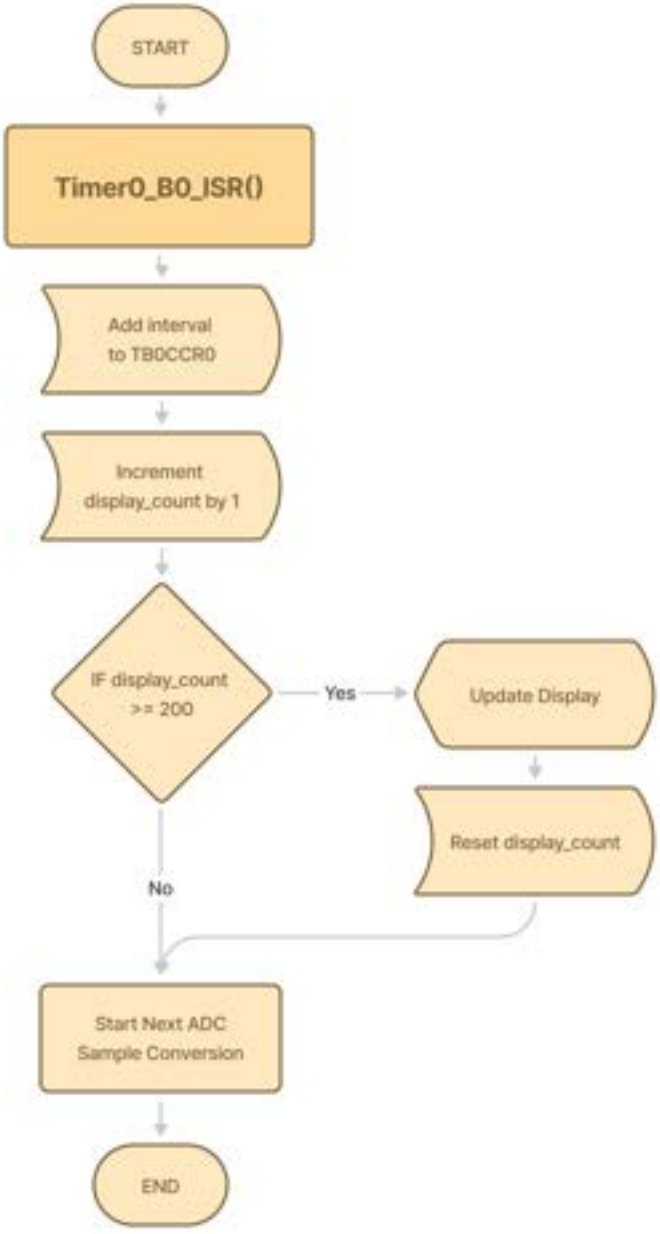

Figure 76: Timer0_B0_ISR() Flow Chart

8.9. Timer B0 CCR1, CCR2 & Overflow Interrupt Flow Chart

Handles Timer B0 CCR1, CCR2, and overflow interrupts for button debouncing functionality.

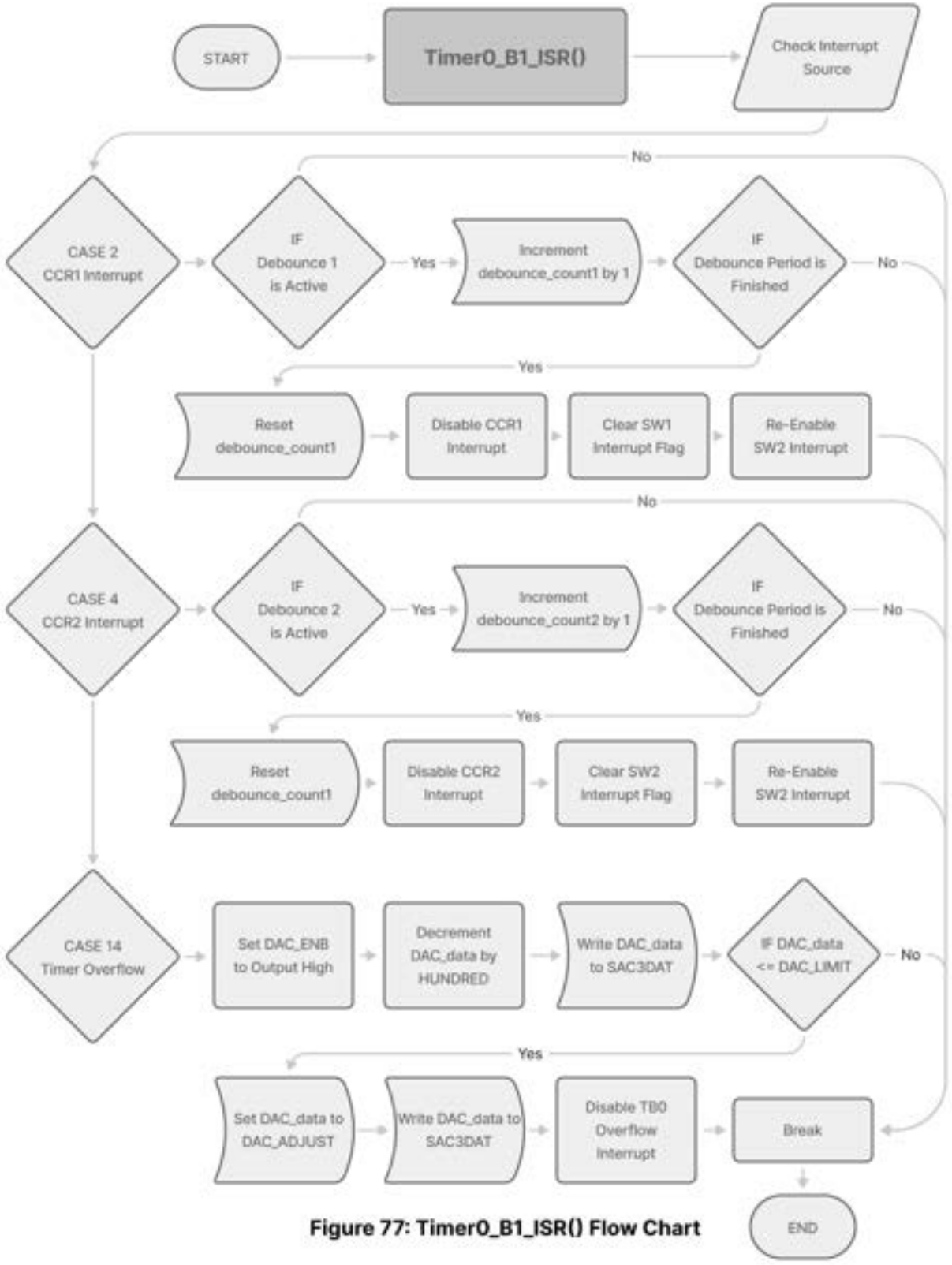

Figure 77: Timer0_B1_ISR() Flow Chart

8.10. ADC Interrupt Flow Chart

Shows the state machine inside the ADC_ISR(), which cycles through three input channels (Left IR, Right IR, Thumbwheel), stores each result, and triggers the next conversion in the sequence.

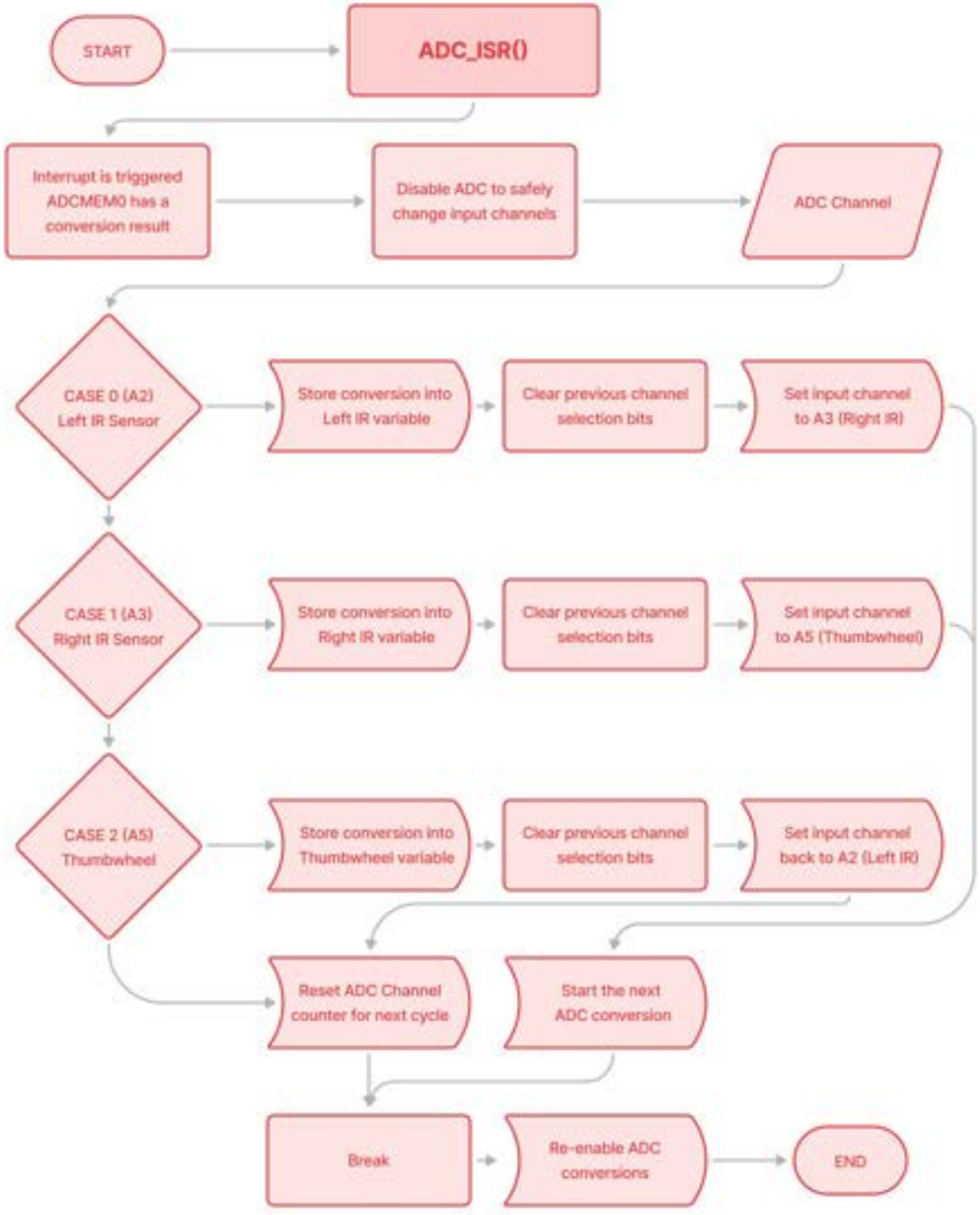

Figure 78: ADC_ISR() Flow Chart

9. Software Listing

9.1. main.c

```
1  //------------------------------------------------------------------------------
2  //  File:           main.c
3  //
4  //  Description:    Main routine with while-loop operating system
5  //
6  //  Name:           Josiah Szobody
7  //  Date:           December 2025
8  //  Version:        CCS_20.2.0.556_mac_x86
9  //------------------------------------------------------------------------------
10
11 #include  "msp430.h"
12 #include  <string.h>
13 #include  "Include/macros.h"
14 #include  "Include/ports.h"
15 #include  "Include/functions.h"
16
17 //------------------------------------------------------------------------------
18 // Global Variables
19 //------------------------------------------------------------------------------
20
21 // System State Machine Variablesd
22 iot_state_t iot_state;                      // IoT (ESP32) control state
23 system_state_t main_state;                  // Main system control state
24 line_state_t line_state;                    // Line-following control state
25
26 // State machine transition flags
27 char Next_State;                            // Flag to go to next state
28 char Prev_State;                            // Flag to go to previous state
29 char loading = FALSE;                       // General loading indicator
30
31 // General-purpose software delay/timer control
32 volatile unsigned int delay_timers[NUM_DELAY_ID] = {ZERO};   // Current time
33 volatile unsigned int delay_targets[NUM_DELAY_ID] = {ZERO};  // Target time
34 volatile unsigned char delay_active[NUM_DELAY_ID] = {FALSE}; // Active flags
35
36 // System clock flags and counters (updated by Timer B ISRs)
37 char MS_500_FLAG;                           // 500ms elapsed flag
38 char SEC_1_FLAG;                           // 1-second elapsed flag
39
```

```
40  // Display hardware buffers
41  char display_line[4][11];                   // LCD display buffer
42  char *display[4];                           // Display line pointers
43  char display_scroll[16][11];                // Scroll buffer for display
44  content
45
46  // Display control flags
47  volatile unsigned char display_changed;     // Display content changed flag
48  volatile unsigned char update_display;      // Flag for display refresh
49
50  // Display strings and status
51  char left_str[11] = "Left      \0";         // Left Sensor Display String
52  char right_str[11] = "Right     \0";        // Right Sensor Display String
53  char blank[11] = "          \0";            // Blank Display String
54  char WHITE[11] = "WHITE     \0";           // WHITE Calibration String
55  char BLACK[11] = "BLACK     \0";           // BLACK Calibration String
56  char CLOCK[11] = "CLK       \0";           // CLOCK Display String
57  char BAUD_STR[11] = "   BAUD   ";          // BAUD Rate Display String
58  char STATUS[17][11];                        // Array for status messages
59  volatile unsigned int status_index;         // Current status message index
60  volatile unsigned int boot_index;           // Index used during boot up
61
62  // UART/IoT Configuration
63  char debug_mode_active = TRUE;              // UART debug passthrough
64  enabled (PC to ESP32)
65  char reboot_IOT = FALSE;                    // ESP32 factory reset flag
66  char UNCA = TRUE;                           // Wi-Fi Connection selection
67
68  // IoT data strings
69  char ip[DATA_LENGTH] = {0};                 // ESP32 IP address string
70  char mac[DATA_LENGTH] = {0};                // ESP32 MAC address string
71  char ssid[DATA_LENGTH] = {0};               // Connected Wi-Fi SSID string
72  char command[COMMAND_LENGTH] = {0};         // Last received command text
73  char A0_Command_Buffer[MAX_COMMAND][COMMAND_LENGTH]; // Command storage array
74
75  // UART command processing flags
76  volatile unsigned char A0_command;          // Current command index
77  volatile unsigned char new_A0_command;      // New A0 command ready to parse
78
79  // UART buffers (UCA1: Debug/PC, UCA0: ESP32/IoT)
80  char A1_rx_buffer[RX_BUFFER_SIZE];          // UCA1 receive circular buffer
81  char A1_tx_buffer[TX_BUFFER_SIZE];          // UCA1 transmit circular buffer
```

```
82  char A0_rx_buffer[RX_BUFFER_SIZE];          // UCA0 receive circular buffer
83  char A0_tx_buffer[TX_BUFFER_SIZE];          // UCA0 transmit circular buffer
84
85  // UCA1 buffer indices (PC Debug)
86  volatile unsigned int A1_rx_write;          // UCA1 RX write pointer
87  volatile unsigned int A1_rx_read;           // UCA1 RX read pointer
88  volatile unsigned int A1_tx_write;          // UCA1 TX write pointer
89  volatile unsigned int A1_tx_read;           // UCA1 TX read pointer
90
91  // UCA0 buffer indices (IoT)
92  volatile unsigned int A0_rx_write;          // UCA0 RX write pointer
93  volatile unsigned int A0_rx_read;           // UCA0 RX read pointer
94  volatile unsigned int A0_tx_write;          // UCA0 TX write pointer
95  volatile unsigned int A0_tx_read;           // UCA0 TX read pointer
96  volatile char A0_tx_busy = FALSE;           // UCA0 TX status flag
97
98  // ADC measurement values
99  volatile unsigned int ADC_Left_Detect;      // Left IR sensor ADC value
100 volatile unsigned int ADC_Right_Detect;     // Right IR sensor ADC value
101 volatile unsigned int ADC_Thumb;            // Thumbwheel ADC value
102
103 // ADC control and calibration
104 volatile unsigned int ADC_Channel;          // Current ADC sensor
105 char ADC_sampling;                          // ADC active sampling flag
106 volatile unsigned int BLACK_VAL;            // Black surface reference value
107 volatile unsigned int WHITE_VAL;            // White surface reference value
108 char BLDETECT;                              // Line detection status flag
109 char BL_LEFT;                               // Line detection direction flag
110
111 // DAC voltage ramp variable
112 volatile unsigned int DAC_data;             // Voltage value for DAC output
113
114 // Button press flags
115 volatile unsigned char SW2_Pressed;         // SW2 button press flag
116 volatile unsigned char SW1_Pressed;         // SW1 button press flag
117 volatile unsigned char Num_SW2_Pressed;     // SW2 press counter
118
119 // Button debounce counters
120 volatile unsigned int SW1_debounce_count;   // SW1 debounce counter
121 volatile unsigned int SW2_debounce_count;   // SW2 debounce counter
122
123 // Motor control variables
```

```
124 volatile unsigned char motor_direction;      // Current movement direction
125 volatile unsigned int motor_time;            // Movement duration (ms)
126 char motor_timed;                            // Flag for timed movement
127
128 //-------------------------------------------------------------------------
129 // Function:        main
130 //
131 // Description:     Main application entry point with superloop architecture
132 //-------------------------------------------------------------------------
133 void main(void){
134     //---------------------------------------------------------------------
135     // Phase 1: Core System Initialization
136     //---------------------------------------------------------------------
137     // Essential setup
138     enable_interrupts();                    // Enable global interrupts
139     PM5CTL0 &= ~LOCKLPM5;                   // Enable I/O pins
140
141     Init_Ports();                           // Initialize all GPIO ports
142     Init_Clocks();                          // Initialize Clock System
143     Init_Timers();                          // Initialize Timers
144
145     // Visual boot indicator (Blink 1)
146     P1OUT ^= RED_LED;       // LED on
147     __delay_cycles(SEC_1_CYCLES);
148
149     // Display setup
150     Init_LCD();                             // Initialize LCD display hardware
151     Init_Display();                         // Initialize display buffers
152
153     // Initial boot display message
154     strcpy(display_line[0], "          "); // Line 0
155     strcpy(display_line[1], "Booting Up"); // Line 1
156     strcpy(display_line[2], "          "); // Line 2
157     strcpy(display_line[3], "          "); // Line 3
158     display_changed = TRUE;
159     update_display = TRUE;
160     Display_Process();                      // Force display update
161
162     // Visual boot indicator (Blink 2)
163     P1OUT ^= RED_LED;       // LED off
164     __delay_cycles(SEC_1_CYCLES);
165
```

```
166     Init_ADC();                             // Init ADC for sensor reading
167
168     // Visual boot indicator (Blink 3)
169     P1OUT ^= RED_LED;       // LED on
170     __delay_cycles(SEC_1_CYCLES);
171
172     //------------------------------------------------------------------------
173     // Phase 2: UART and ESP32/IoT Initialization
174     //------------------------------------------------------------------------
175     Init_Serial_UCA0(BAUD_115200);          // Initialize UCA0 for IoT (ESP32)
176     Init_Serial_UCA1(BAUD_115200);          // Initialize UCA1 for PC (debug)
177
178     // Visual boot indicator (Blink 4)
179     P1OUT ^= RED_LED;       // LED off
180     __delay_cycles(SEC_1_CYCLES);
181
182     Init_IOT();                             // Initialize and configure ESP32
183
184     // Visual boot indicator (Blink 5)
185     P1OUT ^= RED_LED;       // LED on
186     __delay_cycles(SEC_1_CYCLES);
187
188     Init_DAC();                             // Initialize DAC
189
190     //------------------------------------------------------------------------
191     // Phase 3: Application-Specific Variable Initialization
192     //------------------------------------------------------------------------
193
194     // Initialize display scroll buffer
195     display_ip(0);                            // Display IP address
196     strcpy(display_scroll[2], "SSID:     "); // Static SSID label
197     strncpy(display_scroll[3], ssid, 10);    // Copy current SSID
198     display_changed = TRUE;                   // Trigger display update
199
200     // Initialize UART command processing
201     A0_command = BEGINNING;                   // Start command index
202     new_A0_command = FALSE;                   // Clear new command flag
203
204     // Initialize system timing variables
205     MS_500_FLAG = FALSE;
206     SEC_1_FLAG = FALSE;
207
```

```
208     // Initialize user input variables
209     SW1_Pressed = FALSE;                    // Clear SW1 flag
210     SW2_Pressed = FALSE;                    // Clear SW2 flag
211     Num_SW2_Pressed = ZERO;                 // Reset SW2 counter
212
213     // Initialize ADC/DAC measurement variables
214     ADC_Left_Detect = ZERO;                 // Clear left sensor value
215     ADC_Right_Detect = ZERO;                // Clear right sensor value
216     ADC_Thumb = ZERO;                       // Clear thumbwheel value
217     ADC_Channel = ZERO;                     // Start with channel 0
218     ADC_sampling = FALSE;                   // Clear ADC sampling flag
219     DAC_data = DAC_BEGIN;                   // Initialize DAC voltage value
220
221     // Initialize button debounce counters
222     SW1_debounce_count = ZERO;              // Reset SW1 debounce timer
223     SW2_debounce_count = ZERO;              // Reset SW2 debounce timer
224
225     // Initialize motor control variables
226     motor_direction = STILL;                // Motors stopped
227     motor_time = ZERO;                      // No movement time
228     motor_timed = TRUE;
229
230     // Initialize UART circular buffer indices
231     A1_rx_write = BEGINNING;                // A1 RX write pointer
232     A1_rx_read = BEGINNING;                 // A1 RX read pointer
233     A1_tx_write = BEGINNING;                // A1 TX write pointer
234     A1_tx_read = BEGINNING;                 // A1 TX read pointer
235     A0_rx_write = BEGINNING;                // A0 RX write pointer
236     A0_rx_read = BEGINNING;                 // A0 RX read pointer
237     A0_tx_write = BEGINNING;                // A0 TX write pointer
238     A0_tx_read = BEGINNING;                 // A0 TX read pointer
239
240     init_status();                          // Initialize status messages
241     status_index = ZERO;
242
243     // Initialize system state machine
244     main_state = STATE_IDLE;                // Start in IDLE state
245     motor_shutdown();                       // Ensure motors are off
246     Next_State = FALSE;
247     Prev_State = FALSE;
248     BLDETECT = FALSE;
249     BL_LEFT = FALSE;
```

```
    //-------------------------------------------------------------------------
    // Main Superloop
    //-------------------------------------------------------------------------
    while(ALWAYS) {
        // 1. EXECUTE PENDING ACTIONS
        Move(motor_time, motor_direction, motor_timed);  // Control motors

        // 2. PROCESS INPUTS
        UART_Processing();                  // Handle incoming commands

        // 3. STATE AND OUTPUTS
        Main_State_Machine();               // Main control logic
        display_handling();                 // Update LCD content
        scroll();                           // Manage display scrolling

        // Background tasks
        Ping();                             // Keep-alive function
        Display_Process();                  // Update Display hardware
        P3OUT ^= TEST_PROBE;                // Toggle probe pin
    }
}
```

9.2. ports.c

```c
/*
 * File:          ports.c
 * Description:   Initialize the function of each pin on the MSP430FR2355.
 * Created on:    October 2025
 * Author:        Kwali Bunker
 */

#include <msp430.h>
#include "Includes/functions.h"
#include "Includes/ports.h"

//-------------------- Master Init --------------------
void Init_Ports(void) {
  Init_Port1();
  Init_Port2();
  Init_Port3();
  Init_Port4();
  Init_Port5();
  Init_Port6();
}

//-------------------- Port 1 --------------------
void Init_Port1(void) {
  P1OUT = 0x00;                         // Clear output register
  P1DIR = 0x00;                         // Set all pins as inputs initially

  // P1.0 RED LED
  P1DIR  |=  RED_LED;                   // Set as output
  P1SEL0 &= ~RED_LED;                   // GPIO function
  P1SEL1 &= ~RED_LED;                   // GPIO function
  P1OUT  |=  RED_LED;                   // Turn on initially

  // P1.1 (0x02)
  P1SEL0 &= ~A1_SEEED;                  // GPIO function
  P1SEL1 &= ~A1_SEEED;                  // GPIO function
  P1OUT  &= ~A1_SEEED;                  // Output low

  // P1.2 (A2 Left), P1.3 (A3 Right), P1.5 (A5 Thumb) -> analog inputs
  P1DIR  &= ~( V_DETECT_L | V_DETECT_R | V_THUMB); // Set as inputs
  P1SEL0 |=  ( V_DETECT_L | V_DETECT_R | V_THUMB); // Analog function
  P1SEL1 |=  ( V_DETECT_L | V_DETECT_R | V_THUMB); // Analog function
```

```
42    P1OUT  &= ~( V_DETECT_L | V_DETECT_R | V_THUMB); // Output low
43
44    // P1.4 (0x10)
45    P1SEL0 &= ~A4_SEEED;                // GPIO function
46    P1SEL1 &= ~A4_SEEED;                // GPIO function
47    P1OUT  &= ~A4_SEEED;                // Output low
48
49    // UART on UCA0 (P1.6 RXD, P1.7 TXD)
50    P1SEL0 |= (UCA0RXD | UCA0TXD);      // GPIO function
51    P1SEL1 &= ~(UCA0RXD | UCA0TXD);     // GPIO function
52    P1REN  &= ~(UCA0RXD | UCA0TXD);     // Enable pull-up/down
53    P1DIR  &= ~UCA0RXD;                 // Set as input
54    P1DIR  |= UCA0TXD;                  // Set as output
55  }
56
57  //-------------------- Port 2 --------------------
58  void Init_Port2(void) {
59    P2OUT = 0x00;                       // Clear output register
60    P2DIR = 0x00;                       // Set all pins as inputs initially
61
62    // P2.0 SLOW_CLK (safe as input)
63    P2DIR  &= ~SLOW_CLK;                // Set as input
64    P2SEL0 &= ~SLOW_CLK;                // GPIO function
65    P2SEL1 &= ~SLOW_CLK;                // GPIO function
66
67    // P2.1 CHECK_BAT (input)
68    P2DIR  &= ~CHECK_BAT;               // Set as input
69    P2SEL0 &= ~CHECK_BAT;               // GPIO function
70    P2SEL1 &= ~CHECK_BAT;               // GPIO function
71
72    // P2.2 IR_LED (GPIO output, start low)
73    P2DIR  |=  IR_LED;                  // Set as output
74    P2SEL0 &= ~IR_LED;                  // GPIO function
75    P2SEL1 &= ~IR_LED;                  // GPIO function
76    P2OUT  |=  IR_LED;                  // Turn on initially
77
78    // P2.3 SW2 (input with pull-up)
79    P2DIR  &= ~SW2;                     // Set as input
80    P2SEL0 &= ~SW2;                     // GPIO function
81    P2SEL1 &= ~SW2;                     // GPIO function
82    P2REN  |=  SW2;                     // Enable pull-up/down
83    P2OUT  |=  SW2;                     // Set as pull-up
```

```
84    // Interrupt on falling edge (press), clear flag, then enable
85    P2IES |=  SW2;                        // High->low triggers when pressed
86    P2IFG &= ~SW2;                        // Clear any stale flag
87    P2IE  |=  SW2;                        // Enable SW2 interrupt
88
89    // P2.4 IOT_RUN_CPU (input)
90    P2DIR  &= ~IOT_RUN_CPU;               // Set as input
91    P2SEL0 &= ~IOT_RUN_CPU;               // GPIO function
92    P2SEL1 &= ~IOT_RUN_CPU;               // GPIO function
93
94    // P2.5 DAC_ENB (GPIO output, start low)
95    P2DIR  |=  DAC_ENB;                   // Set as output
96    P2SEL0 &= ~DAC_ENB;                   // GPIO function
97    P2SEL1 &= ~DAC_ENB;                   // GPIO function
98    P2OUT  |=  DAC_ENB;                   // Output high
99
100   // P2.6 LFXOUT, P2.7 LFXIN
101   P2SEL0 &= ~(LFXOUT | LFXIN);          // Set for crystal
102   P2SEL1 |=  (LFXOUT | LFXIN);          // Set for crystal
103   P2DIR  &= ~(LFXOUT | LFXIN);          // Set as inputs
104 }
105
106 //-------------------- Port 3 --------------------
107 void Init_Port3(void) {
108   P3OUT = 0x00;                         // Clear output register
109   P3DIR = 0x00;                         // Set all pins as inputs initially
110
111   // P3.0 TEST_PROBE (GPIO output low)
112   P3DIR  |=  TEST_PROBE;              // Commented out - remains input
113   P3SEL0 &= ~TEST_PROBE;                // GPIO function
114   P3SEL1 &= ~TEST_PROBE;                // GPIO function
115   P3OUT  &= ~TEST_PROBE;                // Output low
116
117   // P3.1 DAC_CTRL_2 (input)
118   P3DIR  &= ~DAC_CTRL_2;                // Set as input
119   P3SEL0 &= ~DAC_CTRL_2;                // GPIO function
120   P3SEL1 &= ~DAC_CTRL_2;                // GPIO function
121
122   // P3.2 OA2N, P3.3 OA2P
123   P3SEL0 &=  ~(OA2N | OA2P);            // GPIO function
124   P3SEL1 &=  ~(OA2N | OA2P);            // GPIO function
125   P3OUT &=  ~(OA2N | OA2P);             // Output low
```

```
126
127     // P3.4 SMCLK
128     P3SEL0 &= ~SMCLK;                   // GPIO function
129     P3SEL1 &= ~SMCLK;                   // GPIO function
130
131     // P3.5 DAC_CTRL_3 (GPIO input)
132     P3DIR  &= ~DAC_CTRL_3;              // Set as input
133     P3SEL0 &= ~DAC_CTRL_3;              // GPIO function
134     P3SEL1 &= ~DAC_CTRL_3;              // GPIO function
135
136     // P3.6 IOT_LINK_CPU, P3.7 IOT_EN_CPU (inputs)
137     P3SEL0 &= ~(IOT_LINK_CPU | IOT_EN_CPU); // GPIO function
138     P3SEL1 &= ~(IOT_LINK_CPU | IOT_EN_CPU); // GPIO function
139     P3DIR  |=  (IOT_LINK_CPU | IOT_EN_CPU); // Set as outputs
140     P3OUT  |=  (IOT_LINK_CPU | IOT_EN_CPU); // enable/link CPU<->J9 path
141 }
142
143 //-------------------- Port 4 --------------------
144 void Init_Port4(void) {
145     P4OUT = 0x00;                       // Clear output register
146     P4DIR = 0x00;                       // Set all pins as inputs initially
147
148     // P4.0 RESET_LCD (GPIO output low)
149     P4DIR  |=  RESET_LCD;               // Set as output
150     P4SEL0 &= ~RESET_LCD;               // GPIO function
151     P4SEL1 &= ~RESET_LCD;               // GPIO function
152     P4OUT  &= ~RESET_LCD;               // Output low
153
154     // P4.1 SW1 (input with pull-up)
155     P4DIR  &= ~SW1;                     // Set as input
156     P4SEL0 &= ~SW1;                     // GPIO function
157     P4SEL1 &= ~SW1;                     // GPIO function
158     P4REN  |=  SW1;                     // Enable pull-up/down
159     P4OUT  |=  SW1;                     // Set as pull-up
160     // Interrupt on falling edge (press), clear flag, then enable
161     P4IES |=  SW1;                      // High->low triggers when pressed
162     P4IFG &= ~SW1;                      // Clear any stale flag
163     P4IE  |=  SW1;                      // Enable SW1 interrupt
164
165     // UART on UCA1 (P4.2 RXD, P4.3 TXD)
166     P4SEL0 |=  (UCA1RXD | UCA1TXD);     // Set for UART function
167     P4SEL1 &= ~(UCA1RXD | UCA1TXD);     // Set for UART function
```

```
168     P4REN  &= ~(UCA1RXD | UCA1TXD);     // no pull-ups/downs on UART pins
169     P4DIR  &= ~UCA1RXD;                 // RX must be input
170     P4DIR  |=  UCA1TXD;                 // TX as output (module will drive it)
171
172     // P4.4 LCD -
173     P4DIR  |=  UCB1_CS_LCD;             // Set as output
174     P4SEL0 &= ~UCB1_CS_LCD;             // GPIO function
175     P4SEL1 &= ~UCB1_CS_LCD;             // GPIO function
176     P4OUT  |=  UCB1_CS_LCD;             // Output high
177
178     // P4.5 SCLK, P4.6 SIMO, P4.7 SOMI
179     P4SEL0 |=  (UCB1CLK | UCB1SIMO | UCB1SOMI); // Set for SPI function
180     P4SEL1 &= ~(UCB1CLK | UCB1SIMO | UCB1SOMI); // Set for SPI function
181   }
182
183   //-------------------- Port 5 --------------------
184   void Init_Port5(void) {
185     P5OUT = 0x00;                       // Clear output register
186     P5DIR = 0x00;                       // Set all pins as inputs initially
187
188     // P5.0 = V_BAT node -> analog to ADC A8
189     P5SEL0 |=  V_BAT;                   // route P5.0 to peripheral (ADC A8)
190     P5SEL1 |=  V_BAT;
191
192     // P5.1-P5.3
193     P5SEL0 &= ~(V_5 | V_DAC | V3_3); // GPIO function
194     P5SEL1 &= ~(V_5 | V_DAC | V3_3); // GPIO function
195
196     // P5.4 IOT_BOOT_CPU (input)
197     P5DIR  &= ~IOT_BOOT_CPU;            // Set as input
198     P5SEL0 &= ~IOT_BOOT_CPU;            // GPIO function
199     P5SEL1 &= ~IOT_BOOT_CPU;            // GPIO function
200   }
201
202   //-------------------- Port 6 --------------------
203   void Init_Port6(void) {
204     P6OUT = 0x00;                       // Clear output register
205     P6DIR = 0x00;                       // Set all pins as inputs initially
206
207     // Motor control pins
208     P6SEL0 &= ~(L_FORWARD | R_FORWARD | L_REVERSE | R_REVERSE); // GPIO
209     P6SEL1 &= ~(L_FORWARD | R_FORWARD | L_REVERSE | R_REVERSE); // GPIO
```

```
210     P6DIR  |=  (L_FORWARD | R_FORWARD | L_REVERSE | R_REVERSE); // Outputs
211     P6OUT  &= ~(L_FORWARD | R_FORWARD | L_REVERSE | R_REVERSE); // Start low
212
213     // Backlight as GPIO (ON)
214     P6SEL0 &= ~LCD_BACKLITE;            // GPIO function
215     P6SEL1 &= ~LCD_BACKLITE;            // GPIO function
216     P6DIR  |=  LCD_BACKLITE;            // Set as output
217     P6OUT  |=  LCD_BACKLITE;            // Turn on backlight
218
219     // P6.5 input GPIO
220     P6SEL0 &= ~P6_5;                    // GPIO function
221     P6SEL1 &= ~P6_5;                    // GPIO function
222     P6DIR &= ~P6_5;                     // Set as input
223
224     // P6.6 GRN_LED: GPIO output low
225     P6SEL0 &= ~GRN_LED;                 // GPIO function
226     P6SEL1 &= ~GRN_LED;                 // GPIO function
227     P6DIR |= GRN_LED;                   // Set as output
228     P6OUT &= ~GRN_LED;                  // Output low initially
229 }
```

9.3. timers.c

```c
//=====================================================================
// File Name    : timers.c
//
// Description  : Timer initialization functions for MSP430FR2355
//
// Author       : Leah Page
// Date         : October 2025
//=====================================================================

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

//=====================================================================
// Function Name : Init_Timers
//
// Description   : Master timer initialization function
//=====================================================================
void Init_Timers(void){
    Init_Timer_B0();
    Init_Timer_B3();
}

//=====================================================================
// Function Name : Init_Timer_B0
//
// Description   : Initialize Timer B0 with SMCLK in continuous mode
//=====================================================================
void Init_Timer_B0(void){
    TB0CTL = TBSSEL__SMCLK;           // SMCLK source
    TB0CTL |= TBCLR;                  // Reset timer
    TB0CTL |= MC__CONTINOUS;          // Continuous mode
    TB0CTL |= ID__8;                  // Divide by 8

    TB0EX0 = TBIDEX__4;               // Divide by additional 4

    TB0CCR0 = TB0CCR0_INTERVAL;       // Set CCR0 interval
    TB0CCTL0 |= CCIE;                 // Enable CCR0 interrupt
```

```
42      // TB0CCR1 = TB0CCR1_INTERVAL;    // CCR1 (disabled)
43      // TB0CCTL1 |= CCIE;              // CCR1 interrupt (disabled)
44
45      // TB0CCR2 = TB0CCR2_INTERVAL;    // CCR2 (disabled)
46      // TB0CCTL2 |= CCIE;              // CCR2 interrupt (disabled)
47
48      TB0CTL &= ~TBIE;                  // Disable overflow interrupt
49      TB0CTL &= ~TBIFG;                 // Clear overflow flag
50  }
51
52  //=============================================================================
53  // Function Name : Init_Timer_B3
54  //
55  // Description   : Initialize Timer B3 with SMCLK in up mode
56  //=============================================================================
57  void Init_Timer_B3(void) {
58      //-----------------------------------------------------------------------
59      // SMCLK source, up count mode, PWM Right Side
60      // TB3.1 P6.1 L_FORWARD
61      // TB3.2 P6.2 R_FORWARD
62      // TB3.3 P6.3 L_REVERSE
63      // TB3.4 P6.4 R_REVERSE
64      // TB3.5 P6.5 LCD_BACKLITE
65      //-----------------------------------------------------------------------
66      TB3CTL = TBSSEL__SMCLK;           // SMCLK
67      TB3CTL |= MC__UP;                 // Up Mode
68      TB3CTL |= TBCLR;                  // Clear TAR
69
70      PWM_PERIOD = WHEEL_PERIOD;        // PWM Period [Set this to 50005]
71
72      TB3CCTL1 = OUTMOD_7;              // CCR1 reset/set
73      LEFT_FORWARD_SPEED = WHEEL_OFF;   // P6.1 Left Forward PWM duty cycle
74
75      TB3CCTL2 = OUTMOD_7;              // CCR2 reset/set
76      RIGHT_FORWARD_SPEED = WHEEL_OFF;  // P6.2 Right Forward PWM duty cycle
77
78      TB3CCTL3 = OUTMOD_7;              // CCR3 reset/set
79      LEFT_REVERSE_SPEED = WHEEL_OFF;   // P6.3 Left Reverse PWM duty cycle
80
81      TB3CCTL4 = OUTMOD_7;              // CCR4 reset/set
82      RIGHT_REVERSE_SPEED = WHEEL_OFF;  // P6.4 Right Reverse PWM duty cycle
83
```

```
84      TB3CCTL5 = OUTMOD_7;                // CCR5 reset/set
85      LCD_BACKLITE_DIMING = PERCENT_80;  // P6.5 LCD_BACKLITE on dimming %
86      //-------------------------------------------------------------------------
87  }
```

9.4. ADC.c

```
//===============================================================================
// File Name    : ADC.c
//
// Description  : Implements ADC setup, channel reading, and conversion to
//                BCD ASCII digits. Supports Thumb Wheel input and IR
//                detectors for line sensing.
//
// Author       : Alex Hege
// Date         : October 2025
//===============================================================================

#include <msp430.h>
#include "Includes/ADC.h"
#include "Includes/macros.h"

//===============================================================================
// Function Name : Init_ADC
//
// Description   : Initializes MSP430FR2355 ADC12 module for 12-bit
//                 conversions. Configures ADC for sequential reading of
//                 Left IR (A2), Right IR (A3), and Thumbwheel (A5) channels.
//
//===============================================================================

void Init_ADC(void) {
    //--------------------------
    // ADCCTL0: Control Register 0
    //--------------------------
    ADCCTL0 = CLEAR;              // Reset register
    ADCCTL0 |= ADCSHT_2;          // Sample-and-hold time: 16 ADC clocks
    ADCCTL0 |= ADCMSC;            // Multiple sample and conversion
    ADCCTL0 |= ADCON;             // ADC ON

    //--------------------------
    // ADCCTL1: Control Register 1
    //--------------------------
    ADCCTL1 = CLEAR;              // Reset register
    ADCCTL1 |= ADCSHS_0;          // ADC sample-and-hold source: ADCSC bit
    ADCCTL1 |= ADCSHP;            // Sample-and-hold pulse mode
    ADCCTL1 &= ~ADCISSH;          // Input sample not inverted
    ADCCTL1 |= ADCDIV_0;          // ADC clock divider: /1
```

```
42      ADCCTL1 |= ADCSSEL_0;          // Clock source: MODCLK (~5 MHz)
43      ADCCTL1 |= ADCCONSEQ_0;        // Single-channel, single-conversion
44
45      //--------------------------
46      // ADCCTL2: Control Register 2
47      //--------------------------
48      ADCCTL2 = CLEAR;               // Reset register
49      ADCCTL2 |= ADCPDIV0;           // Pre-divider: /1
50      ADCCTL2 |= ADCRES_2;           // 12-bit resolution
51      ADCCTL2 &= ~ADCDF;             // Data format: unsigned binary
52      ADCCTL2 &= ~ADCSR;             // Sampling rate: supports up to 200 ksps
53
54      //--------------------------
55      // ADCMCTL0: Memory Control 0
56      //--------------------------
57      ADCMCTL0 = CLEAR;              // Reset register
58      ADCMCTL0 |= ADCSREF_0;         // Vref+: AVCC, Vref-: AVSS
59      ADCMCTL0 |= ADCINCH_2;         // Start with Left IR (A2)
60
61      //--------------------------
62      // ADC Interrupt and Start
63      //--------------------------
64      ADCIE |= ADCIE0;               // Enable ADC conversion complete interrupt
65      ADCCTL0 |= ADCENC;             // Enable conversion
66      ADCCTL0 |= ADCSC;              // Start Conversion
67  }
```

9.5. UART.c

```
1  //-------------------------------------------------------------------------
2  //  File:           UART.c
3  //
4  //  Description:    UART driver for MSP430FR2355 - handles both UCA0 and UCA1
5  //
6  //  Name:           Josiah Szobody
7  //  Date:           November 2025
8  //  Version:        CCS_20.2.0.556_mac_x86
9  //-------------------------------------------------------------------------
10
11 #include "msp430.h"
12 #include <string.h>
13 #include "Include/macros.h"
14 #include "Include/ports.h"
15 #include "Include/functions.h"
16 #include "Include/globals.h"
17
18 //-------------------------------------------------------------------------
19 // Function:        Init_Serial_UCA0
20 //
21 // Description:     Initialize UCA0 with specified baud rate
22 //-------------------------------------------------------------------------
23 void Init_Serial_UCA0(baud_rate_t baud_rate) {
24     //---------------------------------------------------------------------
25     // Baud Rate Configuration Table for 8MHz SMCLK
26     //
27     // BRCLK   Baudrate  UCOS16  UCBRx  UCFx  UCSx  TX error(%)  RX error(%)
28     //         (Hz)              (dec)  (hex) (hex)  neg   pos    neg   pos
29     //---------------------------------------------------------------------
30     // 8000000  4800     1       104     2    0xD6  -0.08  0.04  -0.10  0.14
31     // 8000000  9600     1       52      1    0x49  -0.08  0.04  -0.10  0.14
32     // 8000000  19200    1       26      0    0xB6  -0.08  0.16  -0.28  0.20
33     // 8000000  57600    1       8       10   0xF7  -0.32  0.32  -1.00  0.36
34     // 8000000  115200   1       4       5    0x55  -0.80  0.64  -1.12  1.76
35     // 8000000  460800   0       17      0    0x4A  -2.72  2.56  -3.76  7.28
36     //---------------------------------------------------------------------
37
38     // Configure eUSCI_A0 Control Registers
39     UCA0CTLW0 = CLEAR;                      // Clear control register
40     UCA0CTLW0 |= UCSWRST;                   // Put eUSCI in reset for config
41     UCA0CTLW0 |= UCSSEL__SMCLK;             // Use SMCLK as clock source
```

```
    UCA0CTLW0 &= ~UCMSB;                    // LSB first transmission
    UCA0CTLW0 &= ~UCSPB;                    // 1 stop bit
    UCA0CTLW0 &= ~UCPEN;                    // No parity
    UCA0CTLW0 &= ~UCSYNC;                   // Asynchronous mode
    UCA0CTLW0 &= ~UC7BIT;                   // 8-bit data length
    UCA0CTLW0 |= UCMODE0;                   // UART mode

    // Configure baud rate based on selection
    switch(baud_rate) {
        case BAUD_9600:
            UCA0BRW = BAUD_9600_DIVISOR;        // 9600 baud divisor
            UCA0MCTLW = BAUD_9600_MODULATION;   // OS16=1, BRF=1, BRS=0x49
            break;
        case BAUD_19200:
            UCA0BRW = BAUD_19200_DIVISOR;       // 19200 baud divisor
            UCA0MCTLW = BAUD_19200_MODULATION;  // OS16=1, BRF=0, BRS=0xB6
            break;
        case BAUD_57600:
            UCA0BRW = BAUD_57600_DIVISOR;       // 57600 baud divisor
            UCA0MCTLW = BAUD_57600_MODULATION;  // OS16=1, BRF=10, BRS=0xF7
            break;
        case BAUD_115200:                       // Default case
        default:
            UCA0BRW = BAUD_115200_DIVISOR;      // 115200 baud divisor
            UCA0MCTLW = BAUD_115200_MODULATION; // OS16=1, BRF=5, BRS=0x55
            break;
        case BAUD_460800:
            UCA0BRW = BAUD_460800_DIVISOR;      // 460800 baud divisor
            UCA0MCTLW = BAUD_460800_MODULATION; // OS16=0, BRF=0, BRS=0x4A
            break;
    }

    // Activate module and enable interrupts
    UCA0CTLW0 &= ~UCSWRST;                  // Release from reset
    UCA0IE |= UCRXIE;                       // Enable receive interrupts
}

//------------------------------------------------------------------------
// Function:        Init_Serial_UCA1
//
// Description:     Initialize UCA1 with specified baud rate
//------------------------------------------------------------------------
```

```
84  void Init_Serial_UCA1(baud_rate_t baud_rate) {
85      //-------------------------------------------------------------------
86      // Baud Rate Configuration Table for 8MHz SMCLK
87      //
88      // BRCLK   Baudrate  UCOS16  UCBRx  UCFx  UCSx  TX error(%)  RX error(%)
89      //         (Hz)              (dec)  (hex) (hex)  neg   pos    neg   pos
90      //-------------------------------------------------------------------
91      // 8000000  4800     1       104     2    0xD6  -0.08  0.04  -0.10  0.14
92      // 8000000  9600     1       52      1    0x49  -0.08  0.04  -0.10  0.14
93      // 8000000  19200    1       26      0    0xB6  -0.08  0.16  -0.28  0.20
94      // 8000000  57600    1       8       10   0xF7  -0.32  0.32  -1.00  0.36
95      // 8000000  115200   1       4       5    0x55  -0.80  0.64  -1.12  1.76
96      // 8000000  460800   0       17      0    0x4A  -2.72  2.56  -3.76  7.28
97      //-------------------------------------------------------------------
98
99      // Configure eUSCI_A0 Control Registers
100     UCA1CTLW0 = CLEAR;                      // Clear control register
101     UCA1CTLW0 |= UCSWRST;                   // Put eUSCI in reset for config
102     UCA1CTLW0 |= UCSSEL__SMCLK;             // Use SMCLK as clock source
103     UCA1CTLW0 &= ~UCMSB;                    // LSB first transmission
104     UCA1CTLW0 &= ~UCSPB;                    // 1 stop bit
105     UCA1CTLW0 &= ~UCPEN;                    // No parity
106     UCA1CTLW0 &= ~UCSYNC;                   // Asynchronous mode
107     UCA1CTLW0 &= ~UC7BIT;                   // 8-bit data length
108     UCA1CTLW0 |= UCMODE0;                   // UART mode
109
110     // Configure baud rate based on selection
111     switch(baud_rate) {
112         case BAUD_9600:
113             UCA1BRW = BAUD_9600_DIVISOR;        // 9600 baud divisor
114             UCA1MCTLW = BAUD_9600_MODULATION;   // OS16=1, BRF=1, BRS=0x49
115             break;
116         case BAUD_19200:
117             UCA1BRW = BAUD_19200_DIVISOR;       // 19200 baud divisor
118             UCA1MCTLW = BAUD_19200_MODULATION;  // OS16=1, BRF=0, BRS=0xB6
119             break;
120         case BAUD_57600:
121             UCA1BRW = BAUD_57600_DIVISOR;       // 57600 baud divisor
122             UCA1MCTLW = BAUD_57600_MODULATION;  // OS16=1, BRF=10, BRS=0xF7
123             break;
124         case BAUD_115200:                       // Default case
125         default:
```

```
126             UCA1BRW = BAUD_115200_DIVISOR;      // 115200 baud divisor
127             UCA1MCTLW = BAUD_115200_MODULATION; // OS16=1, BRF=5, BRS=0x55
128             break;
129         case BAUD_460800:
130             UCA1BRW = BAUD_460800_DIVISOR;      // 460800 baud divisor
131             UCA1MCTLW = BAUD_460800_MODULATION; // OS16=0, BRF=0, BRS=0x4A
132             break;
133     }
134
135     // Activate module and enable interrupts
136     UCA1CTLW0 &= ~UCSWRST;                  // Release from reset
137     UCA1IE |= UCRXIE;                       // Enable receive interrupts
138 }
```

9.6. DAC.c

```c
//=============================================================================
// File Name    : DAC.c
//
// Description  : Initializes and configures the SAC3 DAC module on the
MSP430.
//                Sets reference source, enables operational amplifier
routing,
//                configures the PGA buffer, loads initial DAC output value,
//                and enables TimerB0 overflow interrupt for DAC stepping.
//
// Author       : Alex Hege
// Date         : December 2025
// Compiler     : TI Code Composer Studio v20.3.0 (MSP430FR2355 Compiler)
//=============================================================================

#include "msp430.h"             // MSP4302355 core header
#include "Includes/macros.h"    // Macro definitions
#include "Includes/DAC.h"       // DAC definitions/prototypes

volatile unsigned int DAC_Data = RESET;

//=============================================================================
// Function Name : Init_DAC
//
// Description   : Initializes SAC3 for DAC output. Configures reference
//                 selection, operational amplifier routing, buffer mode,
//                 initial DAC output value, and enables TimerB0 overflow
//                 interrupts for DAC update timing.
//=============================================================================
void Init_DAC(void){
    SAC3DAC = DACSREF_0;    // Select VCC as DAC reference
    SAC3DAC |= DACLSEL_0;   // DAC latch loads when DACDAT written

    SAC3OA = NMUXEN;        // SAC Negative input MUX control
    SAC3OA |= PMUXEN;       // SAC Positive input MUX control
    SAC3OA |= PSEL_1;       // 12-bit reference DAC source selected
    SAC3OA |= NSEL_1;       // Select negative pin input
    SAC3OA |= OAPM;         // Low-speed, low-power mode

    SAC3PGA = MSEL_1;       // Set OA as buffer mode
    SAC3OA |= SACEN;        // Enable SAC module
```

```
42
43      SAC3OA |= OAEN;          // Enable operational amplifier
44
45      DAC_Data = DAC_Begin;    // Initial DAC output value (e.g., ~2V)
46      SAC3DAT = DAC_Data;      // Load DAC data register
47
48      TB0CTL |= TBIE;          // Enable Timer B0 overflow interrupt
49      P1OUT |= RED_LED;        // Indicate timer overflow started
50      SAC3DAC |= DACEN;        // Enable DAC
51  }
```

9.7. timer_interrupts.c

```c
//===============================================================================
// File Name    : timer_interrupts.c
//
// Description : Handles Timer B0 interrupts for the MSP430FR2355.
//
// Author       : Leah Page
// Date         : November 2025
//===============================================================================

#include "msp430.h"
#include "Include/functions.h"
#include "Include/LCD.h"
#include "Include/ports.h"
#include "Include/macros.h"
#include "Include/timers.h"

unsigned int display_count;
volatile unsigned int wheels_clock;

//===============================================================================
// ISR Name     : Timer0_B0_ISR
//
// Description  : CCR0 interrupt for Timer B0. Manages display refresh timing
//                and triggers ADC sampling. Updates display every 200ms.
//===============================================================================
#pragma vector = TIMER0_B0_VECTOR
__interrupt void Timer0_B0_ISR(void){
    TB0CCR0 += TB0CCR0_INTERVAL;        // Add Offset to TBCCR0
    if (++display_count >= MS_200) {
        update_display = TRUE;          // Request display refresh
        display_count = ZERO;           // Reset counter
    }
    ADCCTL0 |= ADCSC;                   // Start next ADC sample conversion
}

//===============================================================================
// ISR Name     : TIMER0_B1_ISR
//
// Description  : Handles Timer B0 CCR1, CCR2, and overflow interrupts.
//===============================================================================
#pragma vector=TIMER0_B1_VECTOR
```

```
42  __interrupt void TIMER0_B1_ISR(void){
43      switch(__even_in_range(TB0IV,14)){
44          case 0:  // No interrupt
45              break;
46          case 2:  // CCR1 interrupt - SW1 debouncing
47              if (debounce_active1) {
48                  debounce_count1++;
49                  if (debounce_count1 >= DEBOUNCE_THRESHOLD) {
50                      debounce_active1 = FALSE;     // Debouncing complete
51                      TB0CCTL1 &= ~CCIE;            // Disable CCR1 interrupt
52                      P4IFG &= ~SW1;                // Clear SW1 interrupt flag
53                      P4IE  |=  SW1;                // Re-enable SW1 interrupt
54                  }
55              }
56              TB0CCR1 += TB0CCR1_INTERVAL;          // Add Offset to TBCCR1
57              break;
58          case 4:  // CCR2 interrupt - SW2 debouncing
59              if (debounce_active2) {
60                  debounce_count2++;
61                  if (debounce_count2 >= DEBOUNCE_THRESHOLD) {
62                      debounce_active2 = FALSE;     // Debouncing complete
63                      TB0CCTL2 &= ~CCIE;            // Disable CCR2 interrupt
64                      P2IFG &= ~SW2;                // Clear SW2 interrupt flag
65                      P2IE  |=  SW2;                // Re-enable SW2 interrupt
66                  }
67              }
68              TB0CCR2 += TB0CCR2_INTERVAL;          // Add Offset to TBCCR2
69              break;
70          case 14:  // Overflow (Used for DAC stabilization sequence)
71              P2OUT |= DAC_ENB;                     // Set output to high
72              DAC_data = DAC_data - HUNDRED;        // Step down DAC data value
73              SAC3DAT = DAC_data;                   // Write new DAC data
74              if(DAC_data <= DAC_LIMIT){
75                  DAC_data = DAC_ADJUST;            // Set to final stable value
76                  SAC3DAT = DAC_data;
77                  TB0CTL &= ~TBIE;                  // Disable Timer B0 overflow
78              }
79              break;
80          default:
81              break;
82      }
83  }
```

9.8. interrupts_ADC.c

```
//===============================================================================
// File Name    : interrupt_ADC.c
//
// Description  : Handles interrupts for the ADC
//
// Author       : Alex Hege
// Date         : October 2025
// Compiler     : TI Code Composer Studio v20.3.0 (MSP430FR2355 Compiler)
//===============================================================================

#include <msp430.h>
#include "Includes/macros.h"
#include "Includes/switches.h"
#include "Includes/interrupts.h"
#include "Includes/display.h"

//===============================================================================
// ISR Name     : ADC_ISR
//
// Description  : ADC conversion complete ISR. Cycles through three channels:
//                Left IR (A2), Right IR (A3), and Thumbwheel (A5). Stores
//                readings in global variables and triggers the next ADC
//                conversion.
//
// Globals Modified:
//   - ADC_Left_Detect, ADC_Right_Detect, ADC_Thumb
//   - ADC_Channel
//===============================================================================
#pragma vector=ADC_VECTOR
__interrupt void ADC_ISR(void) {
    switch (__even_in_range(ADCIV, ADCIV_ADCIFG)) {
        case ADCIV_ADCIFG:  // ADCMEM0 has a conversion result

            ADCCTL0 &= ~ADCENC;      // Disable ADC for channel switching

            switch (ADC_Channel++) {
                case 0:  // Left IR (A2)
                    ADC_Left_Detect = ADCMEM0;    // Store left IR reading
                    ADCMCTL0 &= ~ADCINCH_2;       // Clear previous channel
                    ADCMCTL0 |= ADCINCH_3;        // Set to Right IR (A3)
                    break;
```

```
42
43                  case 1:  // Right IR (A3)
44                      ADC_Right_Detect = ADCMEM0;   // Store right IR reading
45                      ADCMCTL0 &= ~ADCINCH_3;       // Clear previous channel
46                      ADCMCTL0 |= ADCINCH_5;        // Set to Thumbwheel (A5)
47                      break;
48
49                  case 2:  // Thumbwheel (A5)
50                      ADC_Thumb = ADCMEM0;          // Store thumbwheel reading
51                      ADCMCTL0 &= ~ADCINCH_5;       // Clear previous channel
52                      ADCMCTL0 |= ADCINCH_2;        // Back to Left IR (A2)
53                      ADC_Channel = RESET;      // Reset channel counter
54                      break;
55
56                  default:
57                      ADC_Channel = RESET;              // Safety reset
58                      break;
59              }
60
61              ADCCTL0 |= ADCENC;       // Re-enable ADC conversions
62              break;
63
64          default:
65              break;
66      }
67  }
```

9.9. UART_interrupts.c

```
//-------------------------------------------------------------------------
//  File:           UART_interrupts.c
//
//  Description:    UART interrupt service routines for data transmission
//                  and reception with circular buffer management
//
//  Name:           Josiah Szobody
//  Date Edited:    November 2025
//  Version:        CCS_20.2.0.556_mac_x86
//-------------------------------------------------------------------------

#include "msp430.h"
#include "Include/macros.h"
#include "Include/ports.h"
#include "Include/functions.h"
#include "Include/globals.h"

//-------------------------------------------------------------------------
// Interrupt:       eUSCI_A0_ISR
//
// Description:     UART A0 interrupt service routine (ESP32 communication)
//-------------------------------------------------------------------------
#pragma vector = EUSCI_A0_VECTOR
__interrupt void eUSCI_A0_ISR(void) {
    char received_char;
    // Handle UART interrupts
    // IV values: 0x02=receive, 0x04=transmit, 0x08=start bit
    switch(__even_in_range(UCA0IV, 0x08)) {
        case 0x02:  // UCRXIFG - Receive interrupt (IOT data received)
            // Store received data in circular buffer
            received_char = UCA0RXBUF;

            // Circular buffer write with wrap-around
            A0_rx_buffer[A0_rx_write++] = received_char;
            if (A0_rx_write >= RX_BUFFER_SIZE) {
                A0_rx_write = ZERO;  // Wrap to beginning of buffer
            }
            // If debug mode active, echo received data to PC via UCA1
            if(debug_mode_active) {
                UCA1IE |= UCTXIE;          // Enable UCA1 transmit interrupts
                UCA1TXBUF = received_char; // Echo character to PC
```

```
            }

            break;
        case 0x04:  // UCTXIFG - Transmit interrupt (send data to IOT)
            // Check if more data waiting in transmit buffer
            if (A0_tx_write != A0_tx_read) {
                // Send next byte from circular buffer
                UCA0TXBUF = A0_tx_buffer[A0_tx_read++];

                // Circular buffer read with wrap-around
                if (A0_tx_read >= TX_BUFFER_SIZE) {
                    A0_tx_read = ZERO;  // Wrap to beginning of buffer
                }
            } else {
                // No more data - disable TX interrupt to prevent spinning
                UCA0IE &= ~UCTXIE;
                A0_tx_busy = FALSE;  // Signal transmission complete
            }
            break;
        default:
            break;  // Other UART interrupts not used
    }
}

//-------------------------------------------------------------------------
// Interrupt:       eUSCI_A1_ISR
//
// Description:     UART A1 interrupt service routine (PC debug)
//-------------------------------------------------------------------------
#pragma vector = EUSCI_A1_VECTOR
__interrupt void eUSCI_A1_ISR(void) {
    char received_char;

    // Handle UART interrupts - IV values: 0x02=receive, 0x04=transmit,
0x08=start bit
    switch(__even_in_range(UCA1IV, 0x08)) {
        case 0x02:  // UCRXIFG - Receive interrupt (data received from PC)
            // Store received data in circular buffer
            received_char = UCA1RXBUF;

            // Circular buffer write with wrap-around
            A1_rx_buffer[A1_rx_write++] = received_char;
```

```
84              if (A1_rx_write >= RX_BUFFER_SIZE) {
85                  A1_rx_write = ZERO;  // Wrap to beginning of buffer
86              }
87
88              // ONLY pass to ESP32 (UCA0) if debug mode is active
89              if(debug_mode_active) {
90                  // Direct passthrough to ESP32 for debugging
91                  if(UCA0IFG & UCTXIFG) {  // Is UCA0 transmit buffer empty?
92                      UCA0TXBUF = received_char;  // Forward character to ESP32
93                  }
94              }
95              break;
96          case 0x04:  // UCTXIFG - Transmit interrupt (send data to PC)
97              // Check if more data waiting in transmit buffer
98              if (A1_tx_write != A1_tx_read) {
99                  // Send next byte from circular buffer
100                 UCA1TXBUF = A1_tx_buffer[A1_tx_read];
101                 // Circular buffer read with wrap-around
102                 A1_tx_read++;
103                 if (A1_tx_read >= TX_BUFFER_SIZE) {
104                     A1_tx_read = ZERO;  // Wrap to beginning of buffer
105                 }
106             } else {
107                 // No more data - disable TX interrupt to prevent spinning
108                 UCA1IE &= ~UCTXIE;
109                 A0_tx_busy = FALSE;  // Signal transmission complete
110             }
111             break;
112         default:
113             break;  // Other UART interrupts not used
114     }
115 }
```

10. Conclusion

This work successfully designed and implemented the core hardware and software infrastructure for a Wi-Fi controlled autonomous vehicle platform. The system integrates key components including power regulation, sensor input, motor control, and wireless communication using the MSP430FR2355 microcontroller. The development process provided practical experience in troubleshooting and resolving complex hardware-software integration issues, from signal conflicts to communication protocols.

This implementation resulted in a fully characterized system where motor control, sensor data acquisition, and wireless communication were proven to operate cohesively. The path to this result honed critical skills in diagnosing hardware and software faults, turning theoretical design into a practical, working application.

The completed system provides a proven and adaptable foundation for embedded systems. The greatest value of this endeavor lies in its demonstration of a complete, practical workflow for building and troubleshooting complex electronic systems from the ground up.

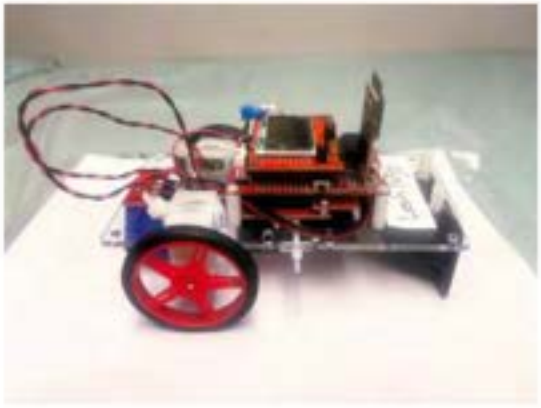

Figure 78: Leah Page's Bot

Figure 79: Josiah Szobody's Bot

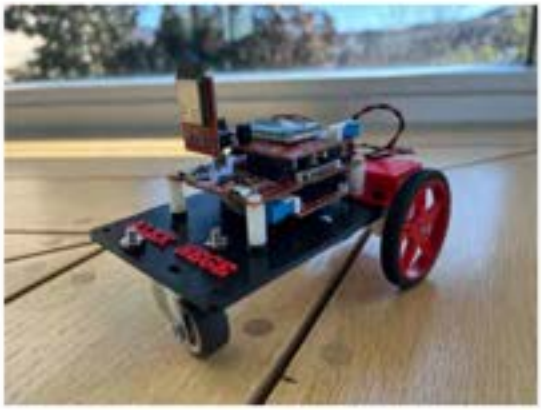

Figure 80: Alex Hege's Bot

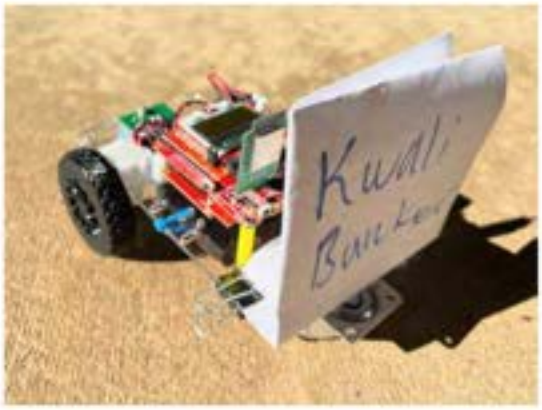

Figure 81: Kwali Bunker's Bot