Final Project

Andrew Truong and Michael Mott

12/10/2025

1. General Description and Purpose

The **Portable Digital Lab Tool** is a compact, microcontroller-based instrument that combines several common bench functions into a single device. The system implements three fully working modes:

- A **digitally adjustable DC power supply** (0–3.3 V)
- A **digital ohmmeter** using a 10 kΩ reference divider
- A **programmable function generator** producing square waves from 10 Hz to 50 kHz

All three modes share a common user interface: a 4×4 keypad for input and two dual 7-segment displays for output. The displays are driven by a pair of MCP23017 I/O expanders over an I²C bus, allowing the STM32G031 microcontroller to control four digits and decimal points using only two signal lines. Two LEDs provide simple range and unit indication (ohms vs kilohms, hertz vs kilohertz).

The purpose of the project is to demonstrate how digital control logic, timers, ADCs, and serial peripherals can be combined to replace traditional stand-alone lab equipment in a compact mixed-signal design. Instead of using separate bench instruments for power, resistance measurement, and signal generation, a student can use this single portable tool to generate a DC voltage, measure resistors, and synthesize test signals.

A fourth feature—frequency measurement—was originally planned using a Schmitt-trigger input stage and timer-based counting. That mode was partially designed and explored in software, but was not robust enough by the final deadline. Since the three completed modes already provide **4.5 project points**, meeting the design requirements, the frequency meter was removed from the final feature set. The rest of the system is fully functional and was demonstrated successfully in lab.

2. Directions for Use

This section serves as the user manual for the Portable Digital Lab Tool. All interaction is performed through a 4×4 keypad, two indicator LEDs, and two dual 7-segment displays. The device powers on in **Power Supply Mode** by default.

2.1 Selecting a Mode

The system has three main operating modes, each selected directly from the keypad:

- **A** – Power Supply Mode
- **B** – Ohmmeter Mode
- **C** – Function Generator Mode

When a mode key is pressed:

- The displays blink briefly to confirm the change.
- The LEDs update to reflect the unit or range used by that mode.
- The device immediately begins operating in the selected function.

2.2 Power Supply Mode

Power Supply Mode generates an adjustable DC output between **0.00 V and 3.30 V**. The output is produced using a PWM signal filtered into a stable DC level.

Display format:
`Vx.xx`
Example: `V1.65`

Controls:

- **1** – Increase output by **0.05 V**
- **2** – Decrease output by **0.05 V**
- **4** – Set output to **0.00 V**
- **5** – Set output to **1.65 V**
- **6** – Set output to **3.30 V**

The left indicator LED ("Base") turns on in this mode to show that the displayed value is in **volts**.

2.3 Ohmmeter Mode

Ohmmeter Mode measures an unknown resistor using a 10 kΩ internal reference resistor and the microcontroller's ADC.

How it works for the user:

1. Connect a resistor to the input terminals.
2. The device automatically measures and updates the display.
3. No button presses are needed while in this mode.

Display behavior:

- **Under 1 kΩ:** Shown in ohms (e.g., `O123`)
- **1 kΩ or higher:** Shown in kilohms with decimal (e.g., `O1.23`)
- **Short circuit:** Display shows a short indicator (`O 0`)
- **Open circuit or no resistor:** Display shows `OPEn`

LED indicators:

- **Base LED on:** Value displayed in ohms
- **Kilo LED on:** Value displayed in kilohms

2.4 Function Generator Mode

Function Generator Mode outputs a square wave from **10 Hz** to **50 kHz** using the microcontroller's hardware timers.

Entering a frequency:

1. Press **C** to enter this mode.
2. Use **0–9** to type frequency digits (up to five digits total).
3. Press ***** to clear the input.
4. Press **#** to confirm and apply the frequency.

If the input is valid:

- The output updates instantly.
- The display reformats the frequency using three significant digits.
- The LEDs show whether the reading is in **Hz** (Base LED) or **kHz** (Kilo LED).

If the input is invalid (outside 10–50,000 Hz):

- An error message appears briefly.
- The system returns to the previously valid frequency.

2.5 Returning to Idle State

When no keypad input is detected for a short period, the device continues updating measurements or output depending on the current mode. Displays and LEDs always reflect the live operating state until the user selects a new mode.

3. Design Analysis

The Portable Digital Lab Tool integrates three independent electronic functions—power supply, ohmmeter, and function generator—into a single microcontroller-based platform. The design combines digital control (keypad interface, PWM generation, I²C display driving) with analog components (RC filtering, resistor networks) to create a compact and versatile lab instrument. This section summarizes the major design choices and the reasoning behind each subsystem.

3.1 System Architecture Overview

At the center of the design is the **STM32G031 microcontroller**, chosen for its onboard 12-bit ADC, hardware timers with PWM capability, and two I²C-capable GPIO pins. The microcontroller communicates with two **MCP23017 I/O expanders** over the I²C bus to drive the four 7-segment displays. A 4×4 matrix keypad provides all user input, and two LEDs on Port B indicate measurement units or mode information.

The final system consists of four major hardware blocks:

1. **Power Supply Generator** – adjustable DC output created from PWM plus a smoothing filter.
2. **Ohmmeter** – a resistor divider into the ADC pin.
3. **Function Generator** – timer-based square-wave output.
4. **User Interface** – keypad input, LED indicators, and dual I²C-driven displays.

Each subsystem operates independently but is coordinated through a simple state machine within the software.

3.2 Power Supply Subsystem

The power supply output is generated digitally using:

- TIM1 Channel 1 running PWM
- A low-pass RC filter to convert PWM into a stable DC level
- The display subsystem for voltage readout

Why PWM?
PWM allows precise analog control using only a digital output pin. By varying the duty cycle from 0–100%, the microcontroller produces output voltages ranging from approximately 0–3.3 V after filtering.

Component Selection:

- The filter resistor and capacitor were selected to smooth the PWM frequency while keeping the output responsive.
- Quick-select presets (0 V, 1.65 V, and 3.30 V) were added for usability.

This design trades absolute precision for simplicity and ease of integration but provides stable and consistent voltage output appropriate for a student-level digital design project.

3.3 Ohmmeter Subsystem

The ohmmeter uses a **voltage divider** between:

- A known reference resistor (**10 kΩ**)
- An unknown external resistor
- The ADC input pin (PA0)

The ADC measures the divider voltage, and software computes:

$$R_{\text{unknown}} = R_{\text{ref}} \cdot \frac{V_{\text{ADC}}}{3.3 - V_{\text{ADC}}}$$

Design considerations:

- A 10 kΩ reference was chosen as a balance between measurable low values (tens of ohms) and reasonably stable readings at higher resistances (tens of kilohms).
- Additional logic handles edge cases such as:
  - Very low voltage → short circuit
  - Very high voltage → open circuit
- Values under 1 kΩ display in ohms; above 1 kΩ display in kilohms.

The LED indicators help clarify which range is active.

3.4 Function Generator Subsystem

The function generator relies on TIM1 Channel 2 to produce square waves from **10 Hz to 50 kHz**. The code dynamically adjusts:

- The **prescaler** (PSC)
- The **auto-reload register** (ARR)

This ensures accurate frequency generation over a wide range while maintaining a 50% duty cycle.

Design goals:

- Allow users to type in any frequency up to five digits
- Use three significant figures in the display, automatically formatting Hz vs. kHz
- Maintain stable output even during rapid keypad entry

This subsystem demonstrates timer configuration and digital waveform synthesis using microcontroller hardware.

3.5 Display and I/O Expansion

Two **MCP23017** expanders multiplex the 7-segment displays:

- Each expander drives two digits (A/B and C/D).
- The STM32 communicates via I²C, sending segment patterns as needed.
- Because the project uses common-anode 7-segment displays, the software inverts segment bits when writing.

Why MCP23017
The MCU does not have enough available GPIO pins to drive four full 7-segment digits directly. The expanders provide 32 additional I/O lines while only using two MCU pins (SCL/SDA).

This drastically simplified routing and kept the board organized.

3.6 Keypad Interface

The keypad uses:

- **Four column pins** configured as outputs
- **Four row pins** configured as inputs with pull-ups

The scanning algorithm:

1. Sets one column low at a time
2. Reads row pins
3. Determines which key is pressed based on coordinates

This provides reliable detection with minimal hardware.

3.7 Discrete Component Choices

Your report will include a component-value table, but the reasoning is:

- **Resistors:**
  - 10 kΩ reference resistor → balance of ADC resolution and measurable range
  - RC filter resistors → chosen to stabilize PWM without excessive delay
  - Current-limiting resistors for LEDs → chosen to keep brightness moderate and protect GPIO pins
- **Capacitors:**
  - Filter capacitor selected to produce a smooth DC level at the PWM frequency
  - Local decoupling capacitors used for MCP23017 stability (standard practice)

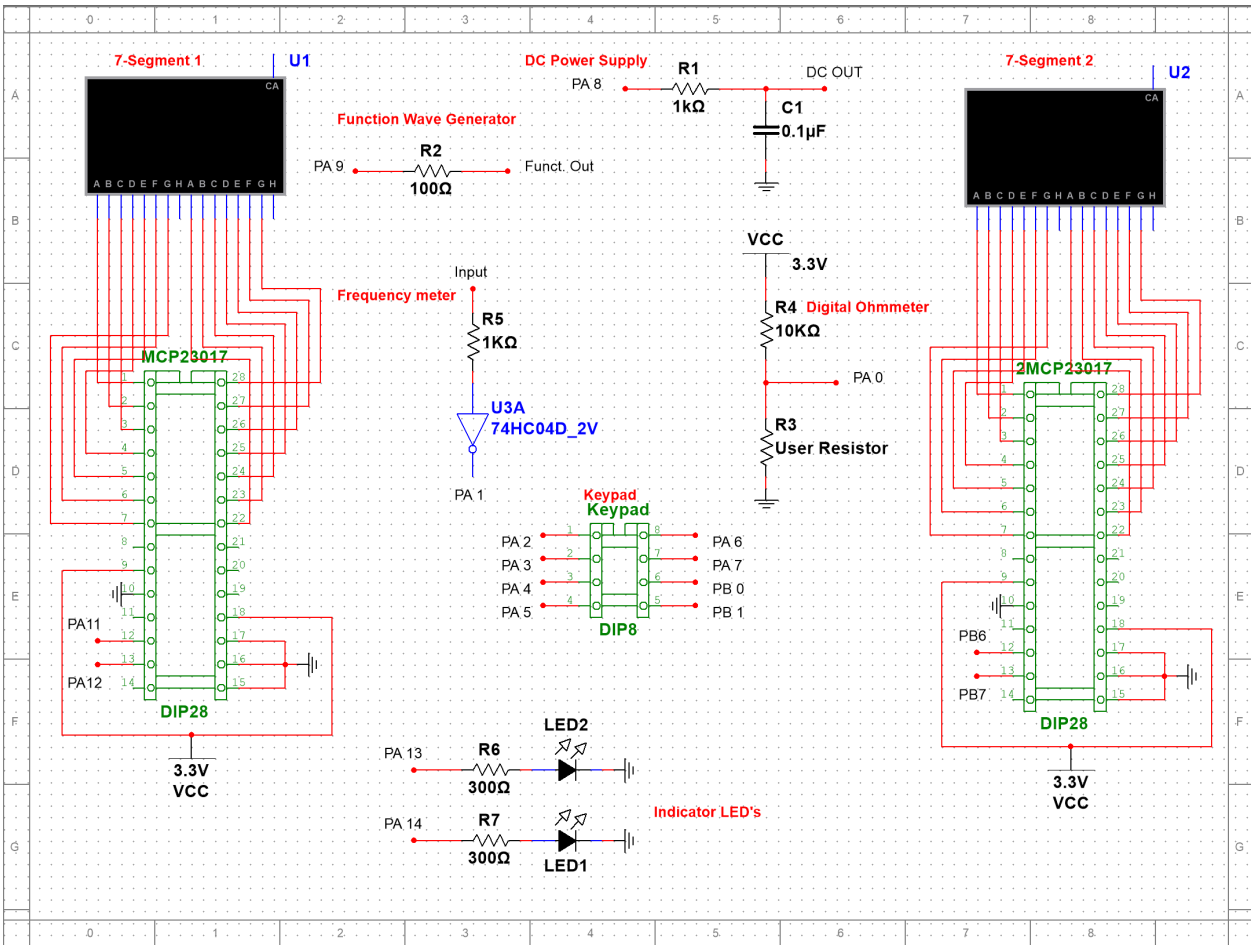
These component values worked reliably in testing and required only minor adjustments during debugging.

4. Final Schematic and Design Changes

4.1 Initial Schematic (Figure 1)

The preliminary schematic laid out the basic structure of the project: two MCP23017 expanders driving a dual seven-segment display, the keypad interface, the PWM power supply, and the resistor divider for the ohmmeter. At this stage, everything was connected in theory, and the diagram represented what we expected to build. What it *didn't* capture yet were some of the real-

world electrical constraints that only became obvious once we started wiring and testing the hardware.



4.2 Final Schematic (Figure 2) and Key Differences

The final schematic reflects the actual working circuit after several rounds of debugging and learning. The biggest changes compared to the preliminary design came from lessons learned the hard way while working with the dual seven-segment displays and the MCP expanders.

Seven-Segment Current Limiting

In the original schematic, the seven-segment displays were connected directly to the MCP23017 outputs **without any current-limiting resistors**. On paper this looked fine — the MCP outputs were just driving LED segments — but in practice it caused two major problems:

1. **The segments drew far more current than the MCP pins were designed to handle.**
2. **We accidentally burned out four seven-segment digits during early testing.**

This was a turning point for the project. After watching multiple digits fail, we realized we needed to rethink how much current each MCP pin was actually sourcing/sinking. That led to the corrected final design, which adds **a 150 Ω resistor on every segment line**.

Once the resistors were added, the segments lit evenly, the MCPs stayed cool, and we finally had reliable displays. This was probably the biggest real-world engineering lesson we took away from the project.

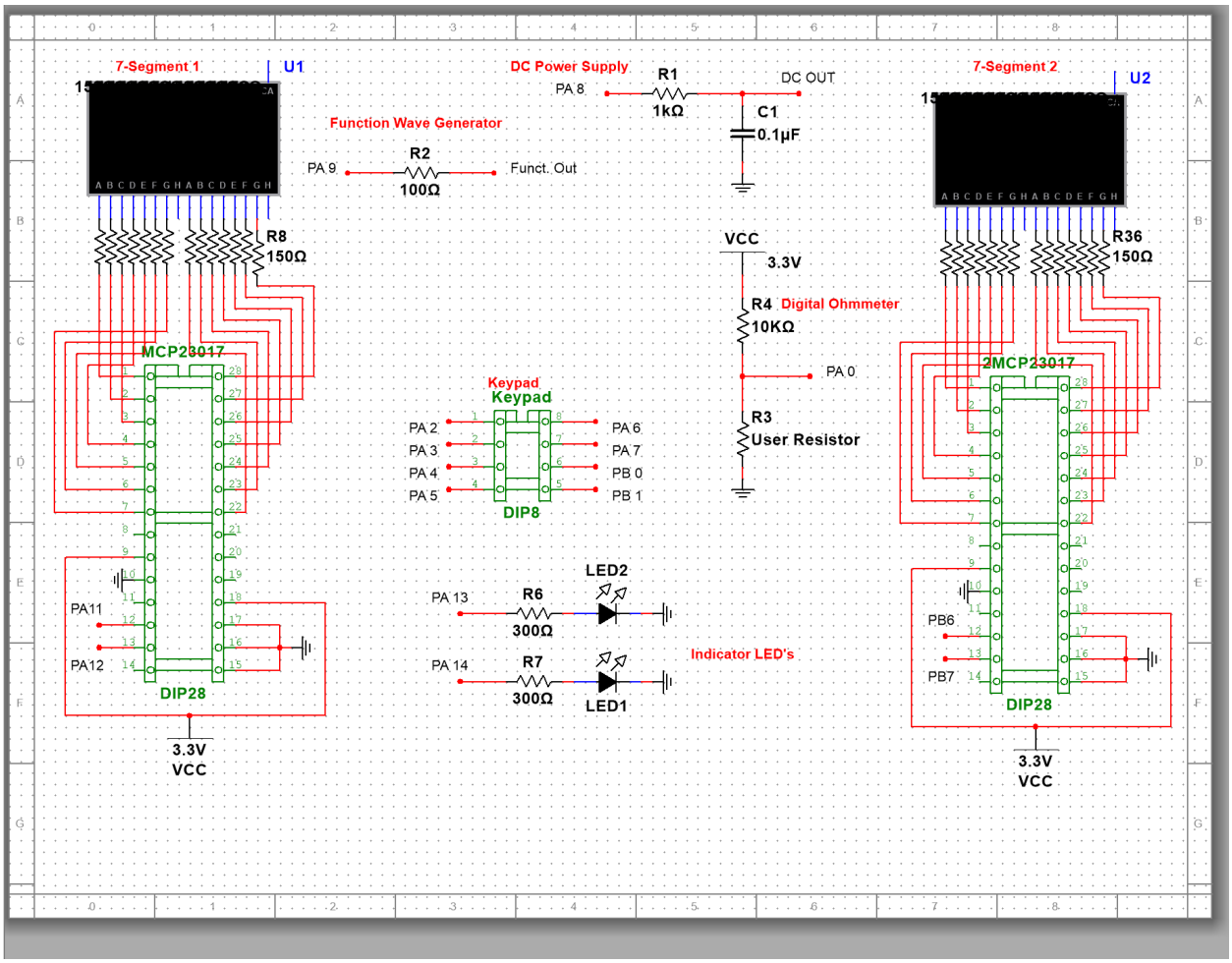
Corrected MCP-to-Display Pin Mapping

The second major improvement was cleaning up the wiring between the MCP23017 outputs and the display pins. In the preliminary schematic the mapping wasn't fully verified, and during testing we saw mismatched segments — for example, asking the software to display a "3" would light up the wrong combination of LEDs.

The final schematic updates the mapping so every MCP output drives the correct segment for both the left and right digits. Once this was corrected, the software could reliably write any number or character to the display.

Removal of the Unused Frequency-Meter Circuit

We originally included a frequency-meter input circuit based on a 74HC04 inverter. Since we weren't able to get that subsystem working consistently before the deadline, we removed it from the final version. Only the three working subsystems — power supply, ohmmeter, and function generator — remain in the final schematic.



5. Component Value Selection

Even though most of the project relied on digital control, a few analog components were critical to making each subsystem work reliably. These values weren't chosen at random — they come from practical limits of the hardware, safety margins, and what we observed during testing.

5.1 Current-Limiting Resistors for the Seven-Segment Displays (150 Ω)

In the early stages we connected the seven-segment digits directly to the MCP23017 pins. That quickly taught us a lesson — without resistors, each LED segment pulled far more current than the I/O expanders were rated for, and we ended up burning out several digits.

To fix this, we added a **150 Ω resistor on every segment line**. This keeps the segment current roughly in the safe range when driven from 5 V and gives a brightness level that's easy to read without pushing the MCPs anywhere near their limits. We tested values higher and lower than 150 Ω, and this one gave the best balance of visibility and safety.

5.2 Ohmmeter Reference Resistor (10 kΩ)

The ohmmeter uses a simple voltage divider between the unknown resistor and a known reference. We chose **10 kΩ** for a few reasons:

- It keeps the current low enough not to heat the resistor or stress the ADC input.
- It places the midpoint voltage in a good range for measuring values between a few ohms and several tens of kilohms.
- It pairs well with the STM32 ADC input impedance and avoids loading issues.

A smaller resistor would have wasted power and skewed readings; a much larger one would have made the ADC readings noisy. Ten kilohms ended up being the most practical middle ground.

5.3 LED Indicator Resistors (300 Ω)

For the "kilo" and "base/Hz" mode LEDs, we used **300 Ω** resistors. These LEDs don't need to be extremely bright — they just indicate whether the device is displaying base units or kilohertz — so we deliberately chose a higher resistor value to keep them softer and avoid driving them too hard from the microcontroller pins.

5.4 RC Filter on the Power-Supply Output

The output of the power supply comes from a PWM signal, so we added a simple RC low-pass filter. The resistor and capacitor we chose (the ones shown in the schematic) were picked to smooth the PWM into a stable DC level while still responding quickly when the user adjusts the voltage. Smaller capacitor values made the output too "ripply," while larger ones made the voltage slow to settle. The final values were selected based on scope measurements during testing.

5.5 Miscellaneous Resistors and Capacitors

Throughout the project we also used a handful of supporting resistors and small capacitors for pull-ups, keypad stabilization, and noise reduction. These were mostly standard values recommended in the STM32 and MCP23017 documentation, and in places where the datasheet gave a range, we chose values based on availability in the lab and what produced the cleanest readings.

6. Final Complete Software

/**

* Digital Design Lab Final Project

* Andrew Truong and Michael Mott

*

*

*

* Power Supply + Ohmmeter + Function Generator

*

* Switch between modes using keypad:

* Button A: Power Supply Mode

* Button B: Ohmmeter Mode

* Button C: Function Generator Mode

*

* LED2 : Left LED

* LED1 : Right LED

*

*

* POWER SUPPLY MODE (Button A):

* Display: "VX.XX" (e.g., "V1.65")

* Button 1: Increment voltage by 0.05V (max 3.3V)

* Button 2: Decrement voltage by 0.05V (min 0V)

* Button 4: Set to 0V (quick preset)

* Button 5: Set to 1.65V (quick preset)

* Button 6: Set to 3.3V (quick preset)

* LEDs: LED2 (Base) on

*

* OHMMETER MODE (Button B):

* Display: "XXX" for ohms, "X.XX" for kilohms, "OPEn" for open

* Continuously measures resistance at PA0

* LEDs: LED2 on for ohms, LED1 on for kilohms

*

* FUNCTION GENERATOR MODE (Button C):

* Display: Shows frequency with 3 sig figs (e.g., "1.00" for 1kHz)

* Keys 0-9: Enter frequency digits (up to 5 digits)

* Key *: Clear input buffer

* Key #: Confirm and set frequency

* Valid range: 10 Hz - 50000 Hz

* LEDs: LED1 (Kilo) on for kHz, LED2 (Base) on for Hz

*

* Hardware:

* PA0 = ADC input (ohmmeter voltage divider)

* PA8 = TIM1_CH1 PWM output (power supply)

* PA9 = TIM1_CH2 PWM output (function generator)

* PA11 = I2C2_SCL (displays)

* PA12 = I2C2_SDA (displays)

* PA6, PA7, PB0, PB1 = Keypad columns (outputs)

* PB2-PB5 = Keypad rows (inputs)

* PB6 = Kilo LED, PB7 = Base LED

*

*

```
*/

#include <stdint.h>
#include <stdbool.h>

// RCC Register Definitions
#define RCC_CR (*(volatile uint32_t *)0x40021000)
#define RCC_IOPENR (*(volatile uint32_t *)0x40021034)
#define RCC_APBENR1 (*(volatile uint32_t *)0x4002103C)
#define RCC_APBENR2 (*(volatile uint32_t *)0x40021040)

// GPIO Port A Register Definitions
#define GPIOA_MODER (*(volatile uint32_t *)0x50000000)
#define GPIOA_OTYPER (*(volatile uint32_t *)0x50000004)
#define GPIOA_OSPEEDR (*(volatile uint32_t *)0x50000008)
#define GPIOA_PUPDR (*(volatile uint32_t *)0x5000000C)
#define GPIOA_IDR (*(volatile uint32_t *)0x50000010)
#define GPIOA_ODR (*(volatile uint32_t *)0x50000014)
#define GPIOA_AFRL (*(volatile uint32_t *)0x50000020)
#define GPIOA_AFRH (*(volatile uint32_t *)0x50000024)

// GPIO Port B Register Definitions
#define GPIOB_MODER (*(volatile uint32_t *)0x50000400)
```

```
#define GPIOB_PUPDR (*(volatile uint32_t *)0x5000040C)
#define GPIOB_IDR (*(volatile uint32_t *)0x50000410)
#define GPIOB_ODR (*(volatile uint32_t *)0x50000414)

// I2C2 Register Definitions
#define I2C2_CR1 (*(volatile uint32_t *)0x40005800)
#define I2C2_CR2 (*(volatile uint32_t *)0x40005804)
#define I2C2_TIMINGR (*(volatile uint32_t *)0x40005810)
#define I2C2_ISR (*(volatile uint32_t *)0x40005818)
#define I2C2_ICR (*(volatile uint32_t *)0x4000581C)
#define I2C2_TXDR (*(volatile uint32_t *)0x40005828)

// ADC Register Definitions
#define ADC_ISR (*(volatile uint32_t *)0x40012400)
#define ADC_CR (*(volatile uint32_t *)0x40012408)
#define ADC_CFGR1 (*(volatile uint32_t *)0x4001240C)
#define ADC_SMPR (*(volatile uint32_t *)0x40012414)
#define ADC_CHSELR (*(volatile uint32_t *)0x40012428)
#define ADC_DR (*(volatile uint32_t *)0x40012440)

// TIM1 Register Definitions (for PWM on PA8 and PA9)
#define TIM1_CR1 (*(volatile uint32_t *)0x40012C00)
#define TIM1_EGR (*(volatile uint32_t *)0x40012C14)
```

```
#define TIM1_CCMR1 (*(volatile uint32_t *)0x40012C18)
#define TIM1_CCER (*(volatile uint32_t *)0x40012C20)
#define TIM1_CNT (*(volatile uint32_t *)0x40012C24)
#define TIM1_PSC (*(volatile uint32_t *)0x40012C28)
#define TIM1_ARR (*(volatile uint32_t *)0x40012C2C)
#define TIM1_CCR1 (*(volatile uint32_t *)0x40012C34)
#define TIM1_CCR2 (*(volatile uint32_t *)0x40012C38)
#define TIM1_BDTR (*(volatile uint32_t *)0x40012C44)

// MCP23017 Definitions
#define MCP23017_ADDR1 0x20
#define MCP23017_ADDR2 0x21

#define MCP_IODIRA 0x00
#define MCP_IODIRB 0x01
#define MCP_GPPUA 0x0C
#define MCP_GPPUB 0x0D
#define MCP_GPIOA 0x12
#define MCP_GPIOB 0x13

// Pin Definitions
#define ROW1_PIN 2 // PB2
#define ROW2_PIN 3 // PB3
```

```
#define ROW3_PIN 4 // PB4
#define ROW4_PIN 5 // PB5

#define COL1_PIN_A 6 // PA6
#define COL2_PIN_A 7 // PA7
#define COL3_PIN_B 0 // PB0
#define COL4_PIN_B 1 // PB1

#define LED1_PIN 6 // PB6 (Kilo)
#define LED2_PIN 7 // PB7 (Base)

#define ADC_PIN 0 // PA0
#define PWM_PS_PIN 8 // PA8 (Power Supply)
#define PWM_FG_PIN 9 // PA9 (Function Generator)

#define RREF 10000 // 10kΩ reference resistor
#define ADC_OFFSET 3 // ADC calibration offset

// 7-Segment Patterns
#define SEG_A 0x01
#define SEG_B 0x02
#define SEG_C 0x04
#define SEG_D 0x08
```

```
#define SEG_E 0x10
#define SEG_F 0x20
#define SEG_G 0x40
#define SEG_DP 0x80

const uint8_t DIGITS[10] = {
SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F, // 0
SEG_B | SEG_C, // 1
SEG_A | SEG_B | SEG_D | SEG_E | SEG_G, // 2
SEG_A | SEG_B | SEG_C | SEG_D | SEG_G, // 3
SEG_B | SEG_C | SEG_F | SEG_G, // 4
SEG_A | SEG_C | SEG_D | SEG_F | SEG_G, // 5
SEG_A | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G, // 6
SEG_A | SEG_B | SEG_C, // 7
SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G, // 8
SEG_A | SEG_B | SEG_C | SEG_D | SEG_F | SEG_G // 9
};

#define CHAR_BLANK 0x00
#define CHAR_V (SEG_B | SEG_C | SEG_D | SEG_E | SEG_F) // V shape
#define CHAR_O (SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F)
#define CHAR_P (SEG_A | SEG_B | SEG_E | SEG_F | SEG_G)
#define CHAR_E (SEG_A | SEG_D | SEG_E | SEG_F | SEG_G)
```

```
#define CHAR_n (SEG_C | SEG_E | SEG_G)

#define CHAR_r (SEG_E | SEG_G)

#define CHAR_F (SEG_A | SEG_E | SEG_F | SEG_G)


// Mode Definitions

#define MODE_POWER_SUPPLY 0

#define MODE_OHMMETER 1

#define MODE_FUNC_GEN 2


// Global Variables

uint8_t mcp1_addr = MCP23017_ADDR1;

uint8_t mcp2_addr = MCP23017_ADDR2;

uint8_t current_mode = MODE_POWER_SUPPLY;

uint16_t voltage_centivolts = 165; // Power supply voltage (0-330)


// Function generator state

uint32_t fg_current_freq = 1000;

uint8_t fg_input_buffer[5] = {0};

uint8_t fg_input_index = 0;

bool fg_input_mode = false; // true = entering digits


// Delay Functions

void delay_ticks(volatile uint32_t count) {
```

```
while (count > 0) count--;
}

void wait_ms(uint32_t ms) {
for (uint32_t i = 0; i < ms; i++) {
delay_ticks(1600);
}
}

// LED Functions
void led1_on(void){
GPIOB_ODR |= (1 << LED1_PIN);
}
void led1_off(void){
GPIOB_ODR &= ~(1 << LED1_PIN);
}

void led2_on(void){
GPIOB_ODR |= (1 << LED2_PIN);
}

void led2_off(void){
GPIOB_ODR &= ~(1 << LED2_PIN);
```

```
}

void leds_on(void){
led1_on();
led2_on();
}

void leds_off(void){
led1_off();
led2_off();
}

void blink(int count) {
for (int i = 0; i < count; i++) {
leds_on();
wait_ms(100);
leds_off();
wait_ms(100);
}
wait_ms(300);
}

// Clock Initialization
```

```
void init_clock(void) {
RCC_CR |= (1 << 8);
while (!(RCC_CR & (1 << 10))) {}
RCC_IOPENR |= (1 << 0) | (1 << 1); // GPIOA, GPIOB
RCC_APBENR1 |= (1 << 22); // I2C2
RCC_APBENR2 |= (1 << 20) | (1 << 11); // ADC, TIM1
}

// LED GPIO Initialization
void init_LED_GPIO(void) {
GPIOB_MODER &= ~((3 << (LED1_PIN * 2)) | (3 << (LED2_PIN * 2)));
GPIOB_MODER |= (1 << (LED1_PIN * 2)) | (1 << (LED2_PIN * 2));
leds_off();
}

// I2C2 GPIO Initialization
void init_I2C2_GPIO(void) {
GPIOA_MODER &= ~((3 << (11 * 2)) | (3 << (12 * 2)));
GPIOA_MODER |= (2 << (11 * 2)) | (2 << (12 * 2));
GPIOA_OTYPER |= (1 << 11) | (1 << 12);
GPIOA_PUPDR &= ~((3 << (11 * 2)) | (3 << (12 * 2)));
GPIOA_PUPDR |= (1 << (11 * 2)) | (1 << (12 * 2));
GPIOA_AFRH &= ~((0xF << ((11 - 8) * 4)) | (0xF << ((12 - 8) * 4)));
```

```
GPIOA_AFRH |= (6 << ((11 - 8) * 4)) | (6 << ((12 - 8) * 4));
}

// I2C Bus Recovery
void i2c_bus_recovery(void) {
GPIOA_MODER &= ~((3 << (11 * 2)) | (3 << (12 * 2)));
GPIOA_MODER |= (1 << (11 * 2)) | (1 << (12 * 2));
GPIOA_ODR |= (1 << 11) | (1 << 12);
for (int i = 0; i < 16; i++) {
GPIOA_ODR &= ~(1 << 11);
for (volatile int j = 0; j < 500; j++);
GPIOA_ODR |= (1 << 11);
for (volatile int j = 0; j < 500; j++);
}
GPIOA_ODR |= (1 << 11);
GPIOA_ODR &= ~(1 << 12);
for (volatile int j = 0; j < 500; j++);
GPIOA_ODR |= (1 << 12);
for (volatile int j = 0; j < 500; j++);
}

// I2C2 Peripheral Initialization
void init_I2C2(void) {
```

```
i2c_bus_recovery();

GPIOA_MODER &= ~((3 << (11 * 2)) | (3 << (12 * 2)));

GPIOA_MODER |= (2 << (11 * 2)) | (2 << (12 * 2));

I2C2_TIMINGR = 0x00303D5B;

I2C2_CR1 |= (1 << 0);

}


// I2C2 Write Register

// Used during MCP23017 initialization and GPIO writes to ensure that the I2C Data is not corrupted.

bool I2C2_write_register(uint8_t address, uint8_t reg, uint8_t value) {

uint32_t timeout;

I2C2_ICR = 0xFFFFFFFF;

I2C2_CR2 = 0;

I2C2_CR2 |= (address << 1);

I2C2_CR2 |= (2 << 16);

I2C2_CR2 |= (1 << 13);

timeout = 100000;

while (timeout--) {

if (I2C2_ISR & (1 << 4)){

I2C2_ICR = (1 << 4);

I2C2_CR2 |= (1 << 14);

return false;

}
```

```
if (I2C2_ISR & (1 << 1)) break;
}
if (timeout == 0) { I2C2_CR2 |= (1 << 14); return false; }
I2C2_TXDR = reg;
timeout = 100000;
while (timeout--) {
if (I2C2_ISR & (1 << 4)){
I2C2_ICR = (1 << 4);
I2C2_CR2 |= (1 << 14);
return false;
}
if (I2C2_ISR & (1 << 1)) break;
}
if (timeout == 0) { I2C2_CR2 |= (1 << 14); return false; }
I2C2_TXDR = value;
timeout = 100000;
while (timeout--) {
if (I2C2_ISR & (1 << 4)){
I2C2_ICR = (1 << 4);
I2C2_CR2 |= (1 << 14);
return false;
}
if (I2C2_ISR & (1 << 6)) break;
```

```
}
if (timeout == 0){
I2C2_CR2 |= (1 << 14);
return false;
}
I2C2_CR2 |= (1 << 14);
timeout = 100000;
while ((I2C2_CR2 & (1 << 14)) && timeout--);
return true;
}

// MCP23017 Initialization
bool init_MCP23017(uint8_t address) {
wait_ms(100);
if (!I2C2_write_register(address, MCP_IODIRA, 0x00)) return false;
if (!I2C2_write_register(address, MCP_IODIRB, 0x00)) return false;
if (!I2C2_write_register(address, MCP_GPPUA, 0x00)) return false;
if (!I2C2_write_register(address, MCP_GPPUB, 0x00)) return false;
if (!I2C2_write_register(address, MCP_GPIOA, 0xFF)) return false;
if (!I2C2_write_register(address, MCP_GPIOB, 0xFF)) return false;
return true;
}
```

```
// ADC Initialization
void init_ADC(void) {
GPIOA_MODER |= (3 << (ADC_PIN * 2));
GPIOA_PUPDR &= ~(3 << (ADC_PIN * 2));
ADC_CR &= ~(1 << 0);
wait_ms(1);
ADC_CFGR1 = 0;
ADC_SMPR = (7 << 0);
ADC_CHSELR = (1 << 0);
ADC_CR |= (1 << 28);
wait_ms(20);
ADC_CR |= (1 << 31);
while (ADC_CR & (1 << 31)) {}
wait_ms(1);
ADC_CR |= (1 << 0);
while (!(ADC_ISR & (1 << 0))) {}
}

// ADC Read Functions
uint16_t read_adc(void) {
ADC_CR |= (1 << 2);
while (!(ADC_ISR & (1 << 2))) {}
return ADC_DR;
```

```
}

uint16_t read_adc_avg(int samples) {
uint32_t sum = 0;
for (int i = 0; i < samples; i++) {
sum += read_adc();
wait_ms(2);
}
return sum / samples;
}

// Resistance Calculation
int32_t calculate_resistance(uint16_t adc_value) {
if (adc_value <= ADC_OFFSET) adc_value = 0;
else adc_value -= ADC_OFFSET;
uint32_t v_mv = ((uint32_t)adc_value * 3300) / 4095;
if (v_mv < 20) return 0;
if (v_mv > 3280) return -1;
uint32_t denominator = 3300 - v_mv;
if (denominator < 10) return -1;
uint32_t resistance = ((uint32_t)RREF * v_mv) / denominator;
return (int32_t)resistance;
}
```

```
// PWM Initialization (TIM1 CH1 on PA8 for Power Supply, CH2 on PA9 for Func Gen)
void init_PWM(void) {
wait_ms(10);
// PA8 as TIM1_CH1 (AF2) - Power Supply
GPIOA_MODER &= ~(3 << (PWM_PS_PIN * 2));
GPIOA_MODER |= (2 << (PWM_PS_PIN * 2));
GPIOA_AFRH &= ~(0xF << ((PWM_PS_PIN - 8) * 4));
GPIOA_AFRH |= (2 << ((PWM_PS_PIN - 8) * 4));
// PA9 as TIM1_CH2 (AF2) - Function Generator
GPIOA_MODER &= ~(3 << (PWM_FG_PIN * 2));
GPIOA_MODER |= (2 << (PWM_FG_PIN * 2));
GPIOA_AFRH &= ~(0xF << ((PWM_FG_PIN - 8) * 4));
GPIOA_AFRH |= (2 << ((PWM_FG_PIN - 8) * 4));
GPIOA_OSPEEDR |= (3 << (PWM_FG_PIN * 2));
TIM1_CR1 = 0;
TIM1_CCER = 0;
TIM1_CCMR1 = 0;
TIM1_BDTR = 0;
TIM1_PSC = 15;
TIM1_ARR = 999;
TIM1_CCR1 = 0;
TIM1_CCR2 = 500;
```

```
// CH1: PWM mode 1, preload enable (for power supply)
TIM1_CCMR1 |= (6 << 4) | (1 << 3);
// CH2: PWM mode 1, preload enable (for function generator)
TIM1_CCMR1 |= (6 << 12) | (1 << 11);
TIM1_CCER |= (1 << 0) | (1 << 4); // CC1E, CC2E
TIM1_BDTR |= (1 << 15); // MOE
TIM1_CR1 |= (1 << 7); // ARPE
TIM1_EGR = (1 << 0); // UG
TIM1_CR1 |= (1 << 0); // CEN
}

// Set Power Supply Voltage (TIM1_CH1)
void set_voltage(uint16_t voltage_cv) {
if (voltage_cv > 330)
voltage_cv = 330;
uint32_t duty = ((uint32_t)voltage_cv * 999) / 330;
TIM1_CCR1 = duty;
}

// Set Function Generator Frequency (TIM1_CH2)
void set_fg_frequency(uint32_t freq_hz) {
if (freq_hz < 10) freq_hz = 10;
if (freq_hz > 50000) freq_hz = 50000;
```

```
uint32_t psc, arr;
if (freq_hz <= 100) {
psc = 1599; // 16MHz / 1600 = 10kHz
arr = (10000 / freq_hz) - 1;
} else if (freq_hz <= 1000) {
psc = 159; // 16MHz / 160 = 100kHz
arr = (100000 / freq_hz) - 1;
} else if (freq_hz <= 10000) {
psc = 15; // 16MHz / 16 = 1MHz
arr = (1000000 / freq_hz) - 1;
} else {
psc = 0; // 16MHz
arr = (16000000 / freq_hz) - 1;
}
TIM1_CR1 &= ~1;
TIM1_PSC = psc;
TIM1_ARR = arr;
TIM1_CCR2 = arr / 2; // 50% duty
TIM1_CNT = 0;
TIM1_EGR = (1 << 0);
TIM1_CR1 |= 1;
fg_current_freq = freq_hz;
}
```

```
// Keypad GPIO Initialization
void init_keypad_GPIO(void) {
GPIOB_MODER &= ~((3 << (ROW1_PIN * 2)) | (3 << (ROW2_PIN * 2)) |
(3 << (ROW3_PIN * 2)) | (3 << (ROW4_PIN * 2)));
GPIOB_PUPDR &= ~((3 << (ROW1_PIN * 2)) | (3 << (ROW2_PIN * 2)) |
(3 << (ROW3_PIN * 2)) | (3 << (ROW4_PIN * 2)));
GPIOB_PUPDR |= (1 << (ROW1_PIN * 2)) | (1 << (ROW2_PIN * 2)) |
(1 << (ROW3_PIN * 2)) | (1 << (ROW4_PIN * 2));
GPIOA_MODER &= ~((3 << (COL1_PIN_A * 2)) | (3 << (COL2_PIN_A * 2)));
GPIOA_MODER |= (1 << (COL1_PIN_A * 2)) | (1 << (COL2_PIN_A * 2));
GPIOB_MODER &= ~((3 << (COL3_PIN_B * 2)) | (3 << (COL4_PIN_B * 2)));
GPIOB_MODER |= (1 << (COL3_PIN_B * 2)) | (1 << (COL4_PIN_B * 2));
GPIOA_ODR |= (1 << COL1_PIN_A) | (1 << COL2_PIN_A);
GPIOB_ODR |= (1 << COL3_PIN_B) | (1 << COL4_PIN_B);
}

// Display Functions
void write_digit(uint8_t mcp_addr, uint8_t digit, uint8_t pattern) {
uint8_t inverted = ~pattern;
if (digit == 0) {
I2C2_write_register(mcp_addr, MCP_GPIOA, inverted);
} else {
```

```
I2C2_write_register(mcp_addr, MCP_GPIOB, inverted);
}
}

// Power Supply display: "VX.XX"
void display_voltage(uint16_t voltage_cv) {
uint8_t integer_part = voltage_cv / 100;
uint8_t decimal_1 = (voltage_cv / 10) % 10;
uint8_t decimal_2 = voltage_cv % 10;
write_digit(mcp2_addr, 1, CHAR_V);
write_digit(mcp2_addr, 0, DIGITS[integer_part] | SEG_DP);
write_digit(mcp1_addr, 1, DIGITS[decimal_1]);
write_digit(mcp1_addr, 0, DIGITS[decimal_2]);
}

// Ohmmeter displays
void display_open(void) {
write_digit(mcp2_addr, 1, CHAR_O);
write_digit(mcp2_addr, 0, CHAR_P);
write_digit(mcp1_addr, 1, CHAR_E);
write_digit(mcp1_addr, 0, CHAR_n);
}
```

```
void display_short(void) {
write_digit(mcp2_addr, 1, CHAR_O);
write_digit(mcp2_addr, 0, CHAR_BLANK);
write_digit(mcp1_addr, 1, CHAR_BLANK);
write_digit(mcp1_addr, 0, DIGITS[0]);
}

void display_ohms(uint16_t ohms) {
if (ohms > 999) ohms = 999;
uint8_t d2 = ohms / 100;
uint8_t d1 = (ohms / 10) % 10;
uint8_t d0 = ohms % 10;
write_digit(mcp2_addr, 1, CHAR_O);
write_digit(mcp2_addr, 0, DIGITS[d2]);
write_digit(mcp1_addr, 1, DIGITS[d1]);
write_digit(mcp1_addr, 0, DIGITS[d0]);
}

void display_kilohms(uint16_t ohms) {
uint16_t ck = ohms / 10;
if (ck > 999) ck = 999;
uint8_t d2 = ck / 100;
uint8_t d1 = (ck / 10) % 10;
```

```
uint8_t d0 = ck % 10;
write_digit(mcp2_addr, 1, CHAR_O);
write_digit(mcp2_addr, 0, DIGITS[d2] | SEG_DP);
write_digit(mcp1_addr, 1, DIGITS[d1]);
write_digit(mcp1_addr, 0, DIGITS[d0]);
}

void display_resistance(int32_t resistance) {
if (resistance < 0) {
display_open();
led1_off(); led2_off();
} else if (resistance < 10) {
display_short();
led1_on(); led2_on();
} else if (resistance < 1000) {
display_ohms((uint16_t)resistance);
led1_off(); led2_on();
} else {
display_kilohms((uint16_t)resistance);
led1_on(); led2_off();
}
}
```

```
// Function Generator display with 3 significant digits
void display_fg_frequency(uint32_t freq_hz) {
uint8_t d0, d1, d2, d3;
uint8_t dp2 = 0, dp3 = 0;
if (freq_hz >= 10000) {
d3 = (freq_hz / 10000) % 10;
d2 = (freq_hz / 1000) % 10;
d1 = (freq_hz / 100) % 10;
d0 = 0xFF;
dp2 = 1;
} else if (freq_hz >= 1000) {
d3 = (freq_hz / 1000) % 10;
d2 = (freq_hz / 100) % 10;
d1 = (freq_hz / 10) % 10;
d0 = 0xFF;
dp3 = 1;
} else if (freq_hz >= 100) {
d3 = 0xFF;
d2 = (freq_hz / 100) % 10;
d1 = (freq_hz / 10) % 10;
d0 = freq_hz % 10;
} else {
d3 = 0xFF;
```

```
d2 = 0xFF;
d1 = (freq_hz / 10) % 10;
d0 = freq_hz % 10;
}
// Digit 0 (rightmost on mcp1)
if (d0 == 0xFF) {
write_digit(mcp1_addr, 0, CHAR_BLANK);
} else {
write_digit(mcp1_addr, 0, DIGITS[d0]);
}
// Digit 1 (mcp1)
if (d1 == 0xFF) {
write_digit(mcp1_addr, 1, CHAR_BLANK);
} else {
write_digit(mcp1_addr, 1, DIGITS[d1]);
}
// Digit 2 (mcp2) - may have decimal point
if (d2 == 0xFF) {
write_digit(mcp2_addr, 0, CHAR_BLANK);
} else {
if (dp2) {
write_digit(mcp2_addr, 0, DIGITS[d2] | SEG_DP);
} else {
```

```
write_digit(mcp2_addr, 0, DIGITS[d2]);
}
}
// Digit 3 (leftmost on mcp2) - may have decimal point
if (d3 == 0xFF) {
write_digit(mcp2_addr, 1, CHAR_BLANK);
} else {
if (dp3) {
write_digit(mcp2_addr, 1, DIGITS[d3] | SEG_DP);
} else {
write_digit(mcp2_addr, 1, DIGITS[d3]);
}
}
if (freq_hz >= 1000) {
led1_on();
led2_off();
} else {
led1_off();
led2_on();
}
}

// Function Generator input display
```

```
void display_fg_input(void) {
uint8_t d0, d1, d2, d3;
if (fg_input_index == 0) {
d0 = d1 = d2 = d3 = 0xFF;
} else {
uint32_t val = 0;
for (uint8_t i = 0; i < fg_input_index; i++) {
val = val * 10 + fg_input_buffer[i];
}
d0 = val % 10;
d1 = (val >= 10) ? ((val / 10) % 10) : 0xFF;
d2 = (val >= 100) ? ((val / 100) % 10) : 0xFF;
d3 = (val >= 1000) ? ((val / 1000) % 10) : 0xFF;
if (val >= 10000) {
d0 = (val / 10) % 10;
d1 = (val / 100) % 10;
d2 = (val / 1000) % 10;
d3 = (val / 10000) % 10;
}
}
// Digit 0 (rightmost on mcp1)
if (d0 == 0xFF) {
write_digit(mcp1_addr, 0, CHAR_BLANK);
```

```
} else {
write_digit(mcp1_addr, 0, DIGITS[d0]);
}
// Digit 1 (mcp1)
if (d1 == 0xFF) {
write_digit(mcp1_addr, 1, CHAR_BLANK);
} else {
write_digit(mcp1_addr, 1, DIGITS[d1]);
}
// Digit 2 (mcp2)
if (d2 == 0xFF) {
write_digit(mcp2_addr, 0, CHAR_BLANK);
} else {
write_digit(mcp2_addr, 0, DIGITS[d2]);
}
// Digit 3 (leftmost on mcp2)
if (d3 == 0xFF) {
write_digit(mcp2_addr, 1, CHAR_BLANK);
} else {
write_digit(mcp2_addr, 1, DIGITS[d3]);
}
leds_off();
}
```

```
// Function Generator error display
void display_fg_error(void) {
write_digit(mcp2_addr, 1, CHAR_E);
write_digit(mcp2_addr, 0, CHAR_r);
write_digit(mcp1_addr, 1, CHAR_r);
write_digit(mcp1_addr, 0, CHAR_BLANK);
leds_off();
wait_ms(1000);
}

// Keypad Scanning
const char key_chars[4][4] = {
{'D', 'C', 'B', 'A'},
{'#', '9', '6', '3'},
{'0', '8', '5', '2'},
{'*', '7', '4', '1'}
};

void all_cols_high(void) {
GPIOA_ODR |= (1 << COL1_PIN_A) | (1 << COL2_PIN_A);
GPIOB_ODR |= (1 << COL3_PIN_B) | (1 << COL4_PIN_B);
}
```

```
void set_col_low(int col) {
all_cols_high();
switch(col) {
case 0: GPIOA_ODR &= ~(1 << COL1_PIN_A); break;
case 1: GPIOA_ODR &= ~(1 << COL2_PIN_A); break;
case 2: GPIOB_ODR &= ~(1 << COL3_PIN_B); break;
case 3: GPIOB_ODR &= ~(1 << COL4_PIN_B); break;
}
}

uint8_t read_rows(void) {
uint8_t rows = 0;
if (!(GPIOB_IDR & (1 << ROW1_PIN))) rows |= 0x01;
if (!(GPIOB_IDR & (1 << ROW2_PIN))) rows |= 0x02;
if (!(GPIOB_IDR & (1 << ROW3_PIN))) rows |= 0x04;
if (!(GPIOB_IDR & (1 << ROW4_PIN))) rows |= 0x08;
return rows;
}

char scan_keypad(void) {
for (int col = 0; col < 4; col++) {
set_col_low(col);
```

```
wait_ms(2);
uint8_t rows = read_rows();
if (rows != 0) {
for (int row = 0; row < 4; row++) {
if (rows & (1 << row)) {
all_cols_high();
return key_chars[row][col];
}
}
}
}
all_cols_high();
return 0;
}

// Function Generator Input Helpers
void clear_fg_input(void) {
fg_input_index = 0;
for (uint8_t i = 0; i < 5; i++) {
fg_input_buffer[i] = 0;
}
fg_input_mode = false;
}
```

```
// Mode Switch Handlers
void switch_to_power_supply_mode(void) {
// Set mode and initialize state
current_mode = MODE_POWER_SUPPLY;
clear_fg_input();
set_voltage(voltage_centivolts);
display_voltage(voltage_centivolts);
// Double blink to confirm mode change
leds_on();
wait_ms(100);
leds_off();
wait_ms(100);
leds_on();
wait_ms(100);
leds_off();
// Set mode indicator: LED2 (Base) on for Power Supply mode
led2_on();
}

void switch_to_ohmmeter_mode(void) {
// Set mode and initialize state
current_mode = MODE_OHMMETER;
```

```
clear_fg_input();
set_voltage(0); // Turn off power supply for safety
display_open();
// Double blink to confirm mode change
leds_on();
wait_ms(100);
leds_off();
wait_ms(100);
leds_on();
wait_ms(100);
leds_off();
//LEDs will be set by display_resistance() based on measurement
}

void switch_to_func_gen_mode(void) {
// Set mode and initialize state
current_mode = MODE_FUNC_GEN;
clear_fg_input();
set_fg_frequency(fg_current_freq);
display_fg_frequency(fg_current_freq);
// Double blink to confirm mode change
leds_on();
wait_ms(100);
```

```
leds_off();
wait_ms(100);
leds_on();
wait_ms(100);
leds_off();

if (fg_current_freq >= 1000) {
led1_on();
led2_off();
} else {
led1_off();
led2_on();
}
}

// MAIN
int main(void) {
init_clock();
init_LED_GPIO();
wait_ms(200);
blink(3); // Startup
init_I2C2_GPIO();
init_I2C2();
```

```
wait_ms(100);
bool mcp1_ok = init_MCP23017(MCP23017_ADDR1);
bool mcp2_ok = init_MCP23017(MCP23017_ADDR2);
if (mcp1_ok && mcp2_ok) {
blink(2);
} else {
for (int i = 0; i < 10; i++) {
leds_on(); wait_ms(50); leds_off(); wait_ms(50);
}
}
init_PWM();
blink(1);
init_ADC();
blink(1);
init_keypad_GPIO();
switch_to_power_supply_mode();
char last_key = 0;
uint32_t loop_count = 0;
while (1) {
char key = scan_keypad();
if (key != 0 && key != last_key) {
// Mode switching
if (key == 'A') {
```

```
switch_to_power_supply_mode();
} else if (key == 'B') {
switch_to_ohmmeter_mode();
} else if (key == 'C') {
switch_to_func_gen_mode();
}
// Power Supply mode controls
else if (current_mode == MODE_POWER_SUPPLY) {
bool voltage_changed = false;
switch (key) {
case '1':
if (voltage_centivolts <= 325) {
voltage_centivolts += 5;
voltage_changed = true;
}
break;
case '2':
if (voltage_centivolts >= 5) {
voltage_centivolts -= 5;
voltage_changed = true;
}
break;
case '4':
```

```
voltage_centivolts = 0;
voltage_changed = true;
break;
case '5':
voltage_centivolts = 165;
voltage_changed = true;
break;
case '6':
voltage_centivolts = 330;
voltage_changed = true;
break;
}
if (voltage_changed) {
set_voltage(voltage_centivolts);
display_voltage(voltage_centivolts);
leds_on(); wait_ms(30); leds_off();
led2_on();
}
}
// Function Generator mode controls
else if (current_mode == MODE_FUNC_GEN) {
// Numeric keys 0-9
if (key >= '0' && key <= '9') {
```

```
if (fg_input_index < 5) {
fg_input_buffer[fg_input_index] = key - '0';
fg_input_index++;
fg_input_mode = true;
display_fg_input();
}
}
// * = Clear
else if (key == '*') {
clear_fg_input();
display_fg_input();
fg_input_mode = true;
}
// # = Confirm
else if (key == '#') {
if (fg_input_index > 0) {
uint32_t input_freq = 0;
for (uint8_t i = 0; i < fg_input_index; i++) {
input_freq = input_freq * 10 + fg_input_buffer[i];
}
if (input_freq >= 10 && input_freq <= 50000) {
set_fg_frequency(input_freq);
display_fg_frequency(input_freq);
```

```
} else {
display_fg_error();
display_fg_frequency(fg_current_freq);
}
clear_fg_input();
}
}
}
// Ohmmeter mode: no additional key functions
}
last_key = (key != 0) ? key : 0;
// Mode-specific updates
if (current_mode == MODE_OHMMETER) {
uint16_t adc_value = read_adc_avg(8);
int32_t resistance = calculate_resistance(adc_value);
display_resistance(resistance);
} else if (current_mode == MODE_POWER_SUPPLY) {
loop_count++;
if (loop_count >= 100) {
loop_count = 0;
display_voltage(voltage_centivolts);
}
} else if (current_mode == MODE_FUNC_GEN) {
```

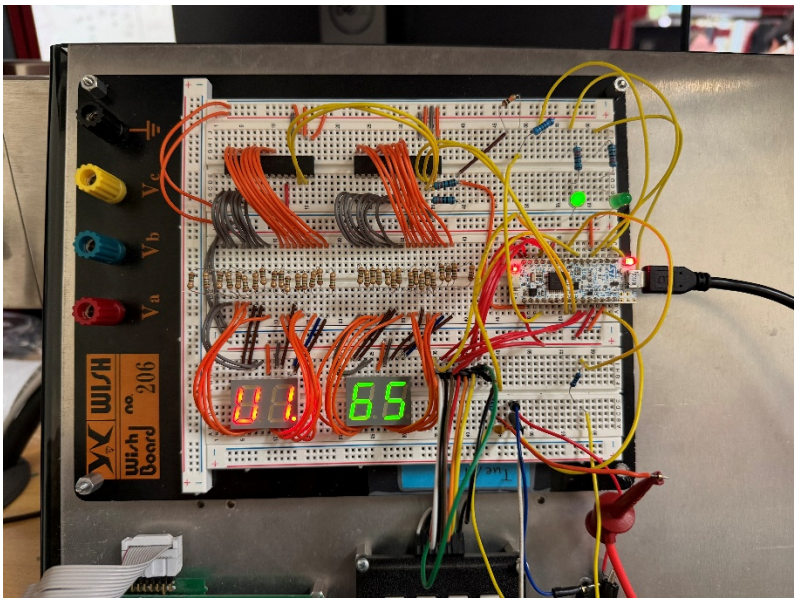
```
// Only refresh if not in input mode

loop_count++;

if (!fg_input_mode && loop_count >= 100) {

loop_count = 0;

display_fg_frequency(fg_current_freq);

}

}

wait_ms(50);

}

return 0;

}
```

7. Supporting Documentation

This section includes photos demonstrating each working mode of the device and showing the system operating with real hardware during testing.

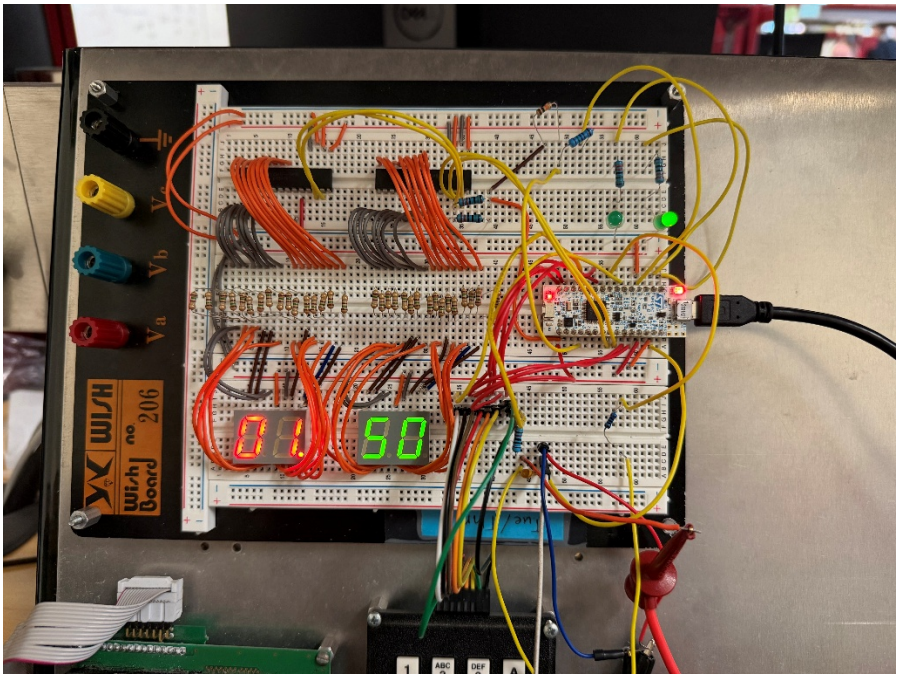
7.1 Power Supply Mode Demonstration

In Power Supply mode, the device generates a DC output by filtering a PWM signal from the microcontroller. The keypad allows the user to adjust the voltage in small steps or jump directly to preset values like 0 V, 1.65 V, or 3.3 V.
The example shown is the system producing **1.65 V**, one of the preset levels, with the display confirming the output voltage.



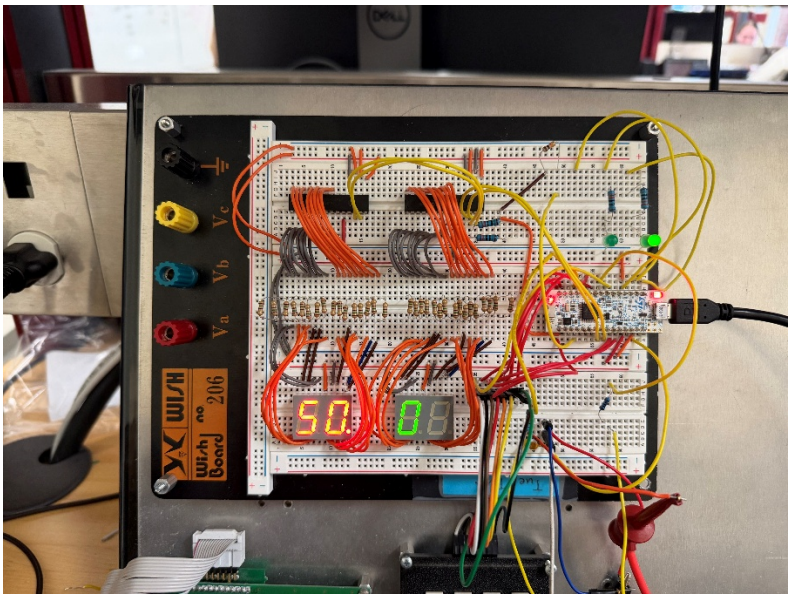
7.2 Ohmmeter Demonstration

The ohmmeter uses a 10 kΩ reference resistor and the STM32's ADC to determine the value of an unknown resistor through a voltage divider. The software automatically switches between displaying ohms and kilohms depending on the reading.
In this example, the device is measuring a **1.5 kΩ resistor**, with the reading shown in kilohms and the correct indicator LED lit.



7.3 Function Generator Demonstration

In Function Generator mode, the user enters a frequency on the keypad, and the system generates a square wave between 10 Hz and 50 kHz.
The example shown is the function generator set to **50 kHz**, the upper limit of the range, with the display and LEDs showing the correct units.



7.4 Project Poster

A project poster was created to give a clear, one-page overview of the system during the demonstration.