Design (E) 314
Technical Report

PV System Efficiency Monitor

Author:
[Hauke Aakmann-Visher]

Student Number:
[24660051]

May 20, 2024

Plagiaatverklaring / Plagiarism Declaration

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Handtekening / *Signature*	**24660051** Studentenommer / *Student number*
H. Aakmann-Visher Voorletters en van / *Initials and surname*	**May 20, 2024** Datum / *Date*

Abstract

This project details the design and implementation of a PV System Efficiency Monitor, which helps maintain the performance of solar photovoltaic (PV) systems. Using an STM32 microcontroller, the system measures key parameters such as PV module voltage, current, ambient temperature, panel temperature, and light intensity. Data is displayed in real-time on an LCD, and the system indicates when the PV modules require cleaning. A calibration feature ensures measurement accuracy. Testing confirmed reliable operation with no delays, although further calibration is needed for more precise ADC measurements. Overall, the monitor is effective in aiding PV system maintenance and optimization.

Contents

List of Figures

List of Tables

List of Acronyms

- LCD: Liquid Crystal Display
- ADC: Analog-to-Digital Converter
- PV: Photovoltaic
- RTC: Real-Time Clock
- PWM: Pulse Width Modulation
- BJT: Bipolar Junction Transistor
- EN: Environmental
- SP: Solar Panel
- CA: Calibration
- MPP: Maximum Power Point
- VOC: Open-Circuit Voltage
- ISC: Short-Circuit Current
- USB: Universal Serial Bus
- LED: Light Emitting Diode

List of Symbols

- V_{in}: Input Voltage
- V_{out}: Output Voltage
- P_{SP}: Solar Panel Power
- V_{SP}: Solar Panel Voltage
- I_{SP}: Solar Panel Current
- R_{sense}: Sense Resistor
- P_{measured}: Measured Power
- $T_{\text{pv\_panel}}$: PV Panel Temperature
- $\text{Lux}_{\text{measured}}$: Measured Lux
- T_{STC}: Standard Testing Conditions Temperature
- β: Temperature Coefficient for PV Module
- $\text{Lux}_{\text{calibrated}}$: Calibrated Lux

- P_{normT}: Temperature-Normalized Power
- $P_{\text{normalised}}$: Normalized Power
- $P_{\text{mpp\_calibrated}}$: Calibrated Maximum Power Point

1 Introduction

The demand for solar photovoltaic (PV) systems as an alternative energy source has significantly increased in South Africa due to the declining costs of PV modules, inverters, and batteries over the past five years. Many households and businesses rely on PV systems to reduce electricity costs and improve energy independence. However, the accumulation of dust and soil on PV modules can drastically reduce their efficiency and power output, leading to financial losses.

Despite the importance of maintaining clean PV modules, many system owners are unaware of the extent to which dirt affects their system's performance. Additionally, the difficulty of visually inspecting roof-mounted PV modules often prevents timely cleaning. To address this issue, this project aims to design and implement a PV System Efficiency Monitor that helps PV system owners determine when cleaning is necessary to optimize power output.

The primary objectives of this project are to:

- Develop a device capable of measuring and reporting the PV module's operating points using both manual and automated adjustable loads.
- Integrate sensors to measure ambient and PV panel temperatures as well as light intensity.
- Implement a feature to indicate whether the PV modules require cleaning based on efficiency measurements.
- Provide a calibration procedure to ensure accurate measurements under clean conditions.
- Display real-time data and measurements on an LCD screen and communicate the results via UART.

By achieving these objectives, the PV System Efficiency Monitor will enable users to maintain their PV systems more effectively, ensuring maximum efficiency and financial returns.

2 System Description

The PV System Efficiency Monitor is engineered to measure, compute, and report the efficiency of a photovoltaic (PV) module. This system is structured from several essential components that collaborate to yield precise measurements and valuable feedback to the user. The core components encompass a power supply, an STM32 microcontroller, sensors for temperature and light measurement, adjustable loads for PV module assessment, and an LCD for displaying results.

2.1 Power Supply

The power supply circuit delivers regulated 5V and 3.3V outputs from a nominal 9V-12V battery or power supply. The 5V supply energizes the STM32 microcontroller, while the 3.3V supply is designated for other components such as sensors and the LCD module.

2.2 PV Module Measurement

The system gauges the PV module's operating points by adjusting the load through either a manual multi-turn potentiometer or an automated active load. The measurements include:

- PV module voltage (V_{PV})
- PV module current (I_{PV})
- Maximum Power Point (MPP) voltage (V_{MPP})
- Maximum Power Point (MPP) current (I_{MPP})

These metrics are vital for determining the maximum power output and efficiency of the PV module.

2.3 Environment Measurement

The device measures ambient temperature using an analog temperature sensor and PV panel temperature using a digital temperature sensor. Additionally, it measures light intensity through a photodiode and an operational amplifier circuit. These measurements are crucial for normalizing the PV module efficiency concerning environmental conditions.

2.4 Indication of Cleaning Requirement

The device indicates whether the PV modules require cleaning based on efficiency measurements. If the efficiency falls below a predetermined threshold, the device signals that cleaning is necessary.

2.5 Calibration Procedure

The system includes a calibration procedure to ensure accurate measurements. This calibration is performed when the PV module is clean, establishing a baseline efficiency measurement. Calibration can be initiated via UART command or a push button on the device.

2.6 Data Display and Communication

The measured data and calculated efficiency values are displayed in real-time on an LCD screen. Additionally, the device communicates the results via UART, enabling users to monitor the PV system's performance remotely.

By integrating these components and functionalities, the PV System Efficiency Monitor offers a comprehensive solution for maintaining and optimizing the efficiency of household PV systems.

Table 1: Hardware Components

Component Description	**Component Name**
Microcontroller	STM32F411RE
Power Supply Circuit	LM7805 and MCP1700
Top Button	Push Button
Bottom Button	Push Button
Middle Button	Push Button
Left Button	Push Button
Right Button	Push Button
Status Indication LEDs	Generic LEDs
Analog Temperature Sensor	LM235
Photodiode for Light Measurement	Photodiode and MCP602 Op-Amp
Digital Temperature Sensor	LMT01
LCD Display	Microrobotics LCD1602-WB-33V
Adjustable Load for PV Measurement	TIP31C BJT

3 Hardware Design and Implementation

This section provides a detailed description of the hardware components and their implementation in the PV System Efficiency Monitor. Each hardware element is designed to fulfill specific functions within the system, ensuring accurate measurement and efficient operation.

3.1 Hardware Block Diagram and Description of Interaction

The hardware block diagram of the system is shown in Figure 1. The power supply provides regulated voltages to the STM32 microcontroller and other components. Sensors for temperature and light measurement, along with adjustable loads, interface with the microcontroller, which processes the data and displays results on the LCD.

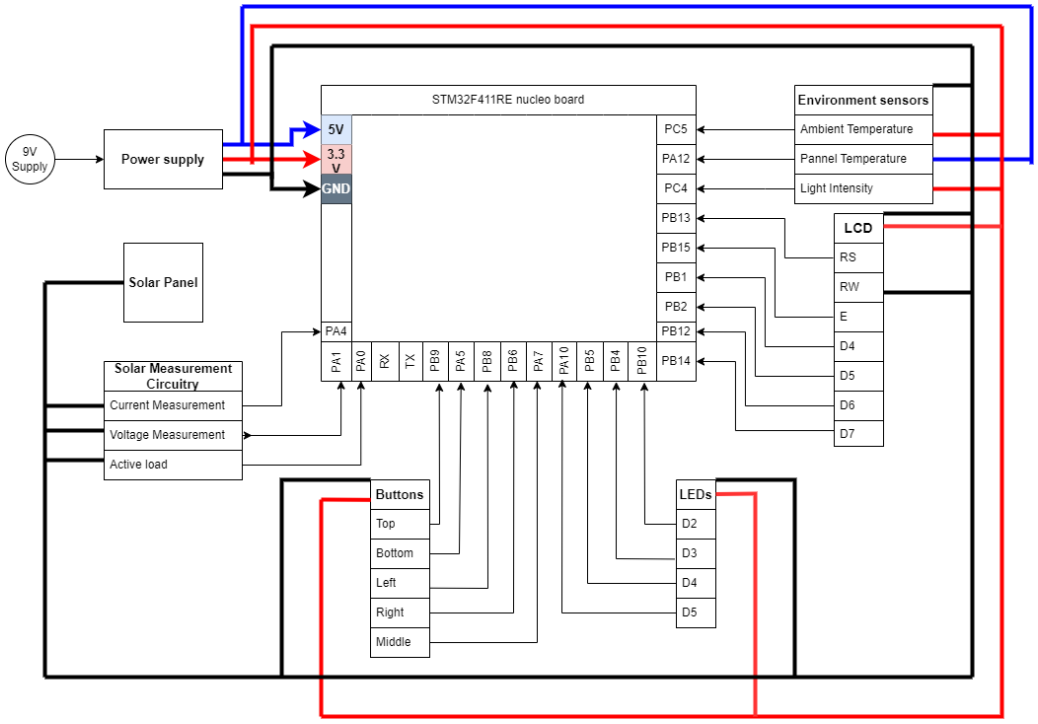

Figure 1: Hardware connections between the components in the PV System Efficiency Monitor.

The pin selection for the PV System Efficiency Monitor was carefully chosen for functionality and design. Buttons were placed on header J11 using PB9, PA5, PA7, PB8, and PA6 to avoid conflicting interrupts. LEDs D2, D3, D4, and D5 were assigned to PB10, PB4, PB5, and PA10 for convenience and conflict avoidance.

The analogue temperature sensor (LM235) connects to ADC Channel 15 on PC5, minimizing signal degradation. The photodiode sensor uses ADC Channel 14 on PC4 for accurate light measurements. The digital temperature sensor (LMT01) uses PA12 for its 5V functionality.

The LCD uses PB pins (e.g., PB13 for RS, PB15 for E, PB1 for D4, PB2 for D5, PB12 for D6, PB14 for D7) due to LCD code requirements. PV module measurements utilize PA4 for current sensing (ADC Channel 4) and PA1 for voltage measurement (ADC Channel 1), both selected for 3.3V compatibility. The active load requires PWM control, with PA0 selected for Timer 5 Channel 1.

Table 2: Hardware Components

Component	3.3V	5V	GND	GPIO Pin
STM32F411re	X	X	X	
Power Supply	X	X	X	
Buttons: Top			X	PB9
Buttons: Bottom			X	PA5
Buttons: Middle			X	PA7
Buttons: Left	X		X	PB8
Buttons: Right		X	X	PA6
LED: D2			X	PB10
LED: D3			X	PB4
LED: D4			X	PB5
LED: D5			X	PA10
Analog Temperature sensor (LM235)	X		X	PC5
Photodiode	X		X	PC4
Digital Temperature sensor (LMT01)		X	X	PA12
LCD Pin: RS				PB13
LCD Pin: RW			X	
LCD Pin: E				PB15
LCD Pin: D4				PB1
LCD Pin: D5				PB2
LCD Pin: D6				PB12
LCD Pin: D7				PB14
PV: Current			X	PA4
PV: Voltage			X	PA1
Active load PWM			X	PA0

3.2 Power Supply

3.2.1 5V voltage regulator

The 5V power supply for the PV System Efficiency Monitor uses the LM7805 voltage regulator, requiring 9 to 12V input to provide a stable 5V output. The circuit, shown in Figure 2, includes capacitors C1 to C3 for voltage stabilization and diode D1 to prevent reverse current. A resistor and LED (D6) indicate power status. The 5V output connects to the microcontroller and components via pin headers J13 and J14, with fuse F1 included for safety. This setup ensures a reliable 5V power supply, crucial for the system's operation.

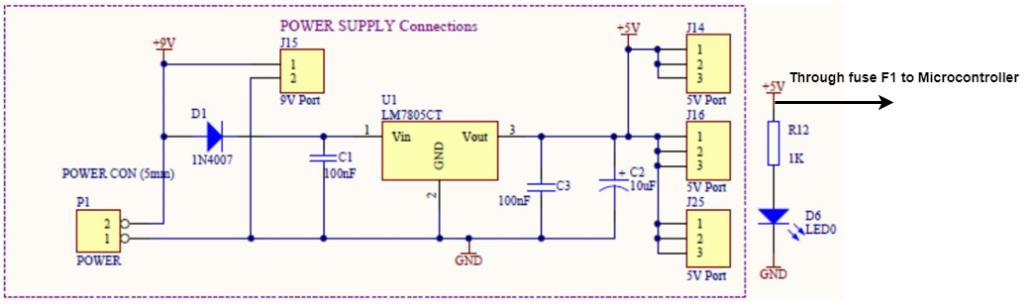

Figure 2: 5V power supply circuit diagram for the PV System Efficiency Monitor.

3.2.2 3.3V Regulator

The 3.3V power supply for the PV System Efficiency Monitor is based on the MCP1700 voltage regulator. This regulator ensures a constant 3.3V output, allowing for a 5 percent variation. The power supply circuit, as shown in Figure 3, includes input and output capacitors (C_{IN} = 100nF and C_{OUT} = 1000nF) to ensure stable DC voltage. The 5V input from the LM7805 regulator is within the input operating voltage range of the MCP1700, which is 2.3V to 6.0V.

The MCP1700 provides a steady 3.3V output essential for powering the sensors and other components though J17,18 and 26 requiring this voltage. The inclusion of the capacitors helps to filter out any noise and maintain a smooth voltage supply to the system components.

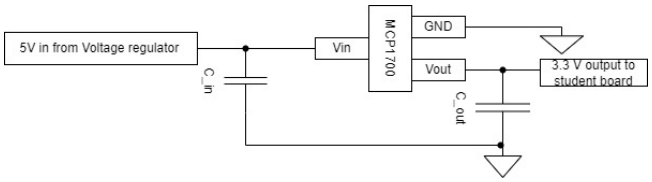

Figure 3: 3.3V power supply circuit diagram for the PV System Efficiency Monitor.

3.3 LEDs (Debug)

Four debug LEDs were incorporated into the design to display the status of the PV System Efficiency Monitor. Table 3 outlines the specific functions assigned to each LED.

Table 3: Debug LED Functions and Pin Connections

Debug LED	State Indication	Selected Output Pin
D2	Blinks when the Solar Pannel power measurement (SP) is in progress and stays on once it is concluded	PB10
D3	Blinks when environmental measurement (EN) is in Progess and stays on when it is concluded	PB4
D4	Blinks when the Calibration Process (CA) is in progress and stays on Once it is concluded	PB5
D5	Turns on when the Solar Pannel Efficiency is above 80 percent. If not, it blinks	PA10

Given that the forward voltage of the LEDs is approximately 1.9V, a 3.3V output was determined to be adequate. To prevent the LEDs from drawing more than 8mA, which is the maximum current that the GPIOs can handle, current-limiting resistors were added. The following equations were utilized to calculate the appropriate resistor value:

$$I = \frac{V_{source} - V_{forward}}{R} \quad \text{(eqn 1)}$$

$$R = \frac{V_{source} - V_{forward}}{I} = \frac{3.3 - 1.9}{0.008} = 175\Omega$$

To ensure the LEDs operate within safe parameters and to reduce power consumption, a 1000 Ω resistor was selected, resulting in a current of 1.4mA through each LED. After constructing the circuit, the LEDs' brightness was visually inspected and verified to be appropriate. Figure 4 illustrates the LED circuit configuration, including the current-limiting resistors.

3.4 Buttons

The PV System Efficiency Monitor includes five push buttons configured for active-low operation. These buttons allow user interaction for various functions such as starting measurements, selecting

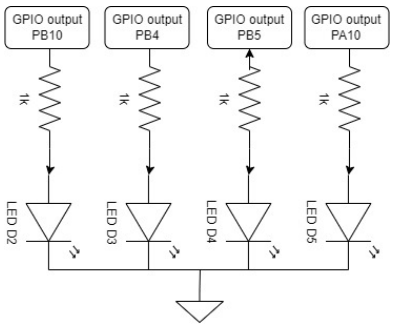

Figure 4: Debug LED circuit setup

modes, setting the time and initiating calibration. The buttons are connected to specific GPIO pins on the STM32 microcontroller and are debounced using software, and configured as active low using software eliminating the need for external pull-up resistors. Table 4 lists the buttons, their respective GPIO pins, and their assigned functions.

Table 4: Push Button Functions and Pin Connections

Button	**GPIO Pin**	**Function**
Top	PB9	Start/Stop environmental measurement
Bottom	PA5	Start/Stop Solar Pannel measurement
Middle	PA7	Initiate RTC setup
Left	PB8	Switch display mode
Right	PA6	Start Calibration Process

Figure 5 shows the wiring diagram of the push buttons connected to the GPIO pins of the STM32 microcontroller. Each button is connected to ground, and pressing a button pulls the corresponding pin low, triggering the associated function in the software.

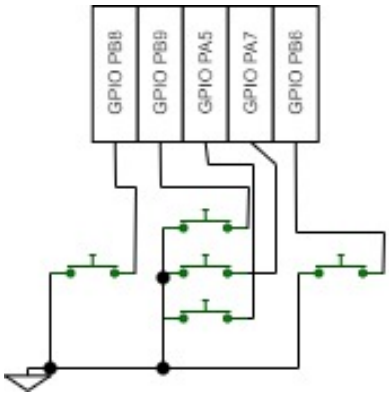

Figure 5: Push buttons circuit diagram.

3.5 Temperature Sensing

3.5.1 Digital Temperature Sensor

The LMT01 digital temperature sensor measures the solar panel temperature and is connected to pin PA12, which supports 5V. The LMT01 is chosen for its high accuracy and resistance to noise, making it suitable for PV system performance optimization. The design uses pin PA12 for the LMT01 sensor due to its interrupt capabilities as it doesnt conflict with the button Interrupts and it was chosen for its Proximity to the Chosen location of the Sensor. A 30kΩ pull-up resistor is added in series with the output to maintain the integrity of the digital signal and protect the input from potential damage.

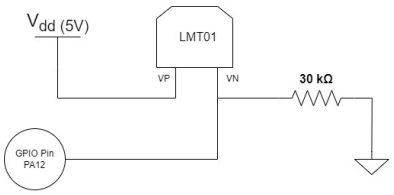

Figure 6: LMT01 Circuit Diagram

The resistor value is chosen based on the LMT01 datasheet. Given $2.15 \leq V_P - V_L < 5.5$, the maximum voltage is calculated as $V_{\max} = 0.7 \times V_{\text{dd}} = 3.5\,V$ (assuming $V_{\text{dd}} = 5\,V$) and the minimum voltage as $V_{\min} = 0.3 \times V_{\text{dd}} = 1.5\,V$. The output high current $I_{\text{OH}} = 125\,\mu A$ and output low current $I_{\text{OL}} = 34\,\mu A$.

Using $R_{\min} = \frac{V_{\max}}{I_{\text{OH}}}$, $R_{\min} = 28\,k\Omega$. Using $R_{\max} = \frac{V_{\min}}{I_{\text{OL}}}$, $R_{\max} = 44.12\,k\Omega$.

A 30kΩ resistor is chosen to ensure a reliable pull-up current.

3.5.2 Analog Temperature Sensor

The analog temperature sensor, LM235, is used for measuring ambient temperature. It is connected to pin PC5 due to its ADC functionality. A 180Ω resistor is selected to prevent the sensor from saturating at high temperatures. The LM235 sensor operates as a precision temperature sensor with a linear output. The sensor's output voltage is directly proportional to the absolute temperature at 10mV/°K. It operates within a current range from $450\mu A$ to 5mA. The selected 180Ω resistor ensures proper operation without saturating the sensor.

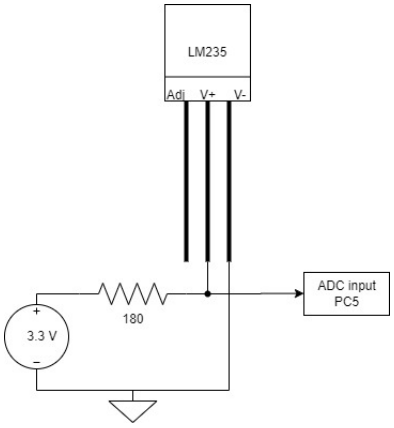

Figure 7: LM235 Circuit Diagram

For $V_{min} = 2.23\,V$ (for minimum temperature at -50°C which is 223 K) and $I_{max} = 5mA$, using $R_{max} = \frac{3.3-2.23}{5mA}$, $R_{max} = 214\,\Omega$.

For $V_{max} = 3.28\,V$ (for maximum temperature at +55°C which is 328 K) and $I_{min} = 450\,\mu A$, using $R_{min} = \frac{3.3-3.28}{450\mu A}$, $R_{min} = 155.56\,\Omega$.

Choosing a resistor value of 180Ω:

$$3.3 - X - 5mA \times 180 = 0 \implies X = 2.4 \implies 240\ ^\circ\text{K} = -33^\circ\text{C}$$

$$3.3 - X - 450\mu A \times 180 = 0 \implies X = 3.219 \implies 322\ ^\circ\text{K} = 48.9^\circ\text{C}$$

These readings and resolution are sufficient for the project's requirements, justifying the choice of a 180Ω resistor. The STM32 microcontroller's ADC operates in 12-bit mode, providing 4096 discrete levels (2^{12}). Given the reference voltage $V_{ref} = 3.3V$:

$$\text{Resolution} = \frac{3.3V}{4095} \approx 0.00080586\,V\ (\,805.86\,\mu V\,)$$

Since the LM235 sensor outputs 10mV/°K, the smallest possible temperature change that this circuit can measure is approximately 0.081 K:

3.6 LCD Display

The LCD1602 display is connected to the microcontroller as shown in Figure 8. DB0-3 are grounded because the display is operated in 4-bit mode to reduce the number of GPIO pins used. The RW pin is grounded since we are only writing to the display, eliminating the need for read operations.

To connect the display to 3.3V instead of 5V, the 5V electric path of the PCB between J16 and J25 was severed, and J25 was bridged to J26, supplying the LCD with the required 3.3V.

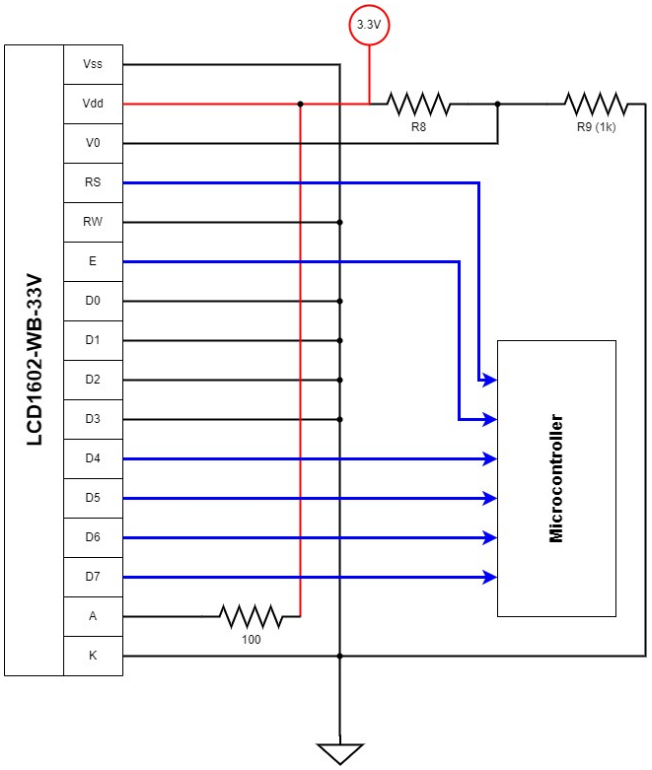

Figure 8: LCD1602 Circuit Diagram

Contrast Resistor Calculation: The contrast of the LCD is adjusted using resistor R8. The optimal contrast voltage for the LCD is typically around 0.5V. Using a voltage divider with R8 and R9, the value of R8 can be calculated as follows: Given $V_{in} = 3.3V$ and $V_{out} = 0.5V$, and given that $R9 = 1k\Omega$, we get $0.5V = 3.3V \times \frac{1k\Omega}{R8+1k\Omega}$. Solving for R8, we find $R8 = \frac{3.3V}{0.5V} \times 1k\Omega - 1k\Omega \approx 5.6k\Omega$.

Backlight Resistor Calculation: The backlight resistor is connected between the Voltage supply and the Anode "A pin" of the LCD. Its value is chosen to limit the current through the LED backlight to a safe level. The forward voltage of the LED backlight is 3.0V, and the desired current is 30mA. Using $R = \frac{V_{supply}-V_{forward}}{I}$, given $V_{supply} = 3.3V$ and $V_{forward} = 3.0V$, we get $R = \frac{3.3V-3.0V}{30mA} = \frac{0.3V}{0.03A} = 10\Omega$. Therefore, a 100Ω resistor is chosen to provide a conservative current limit, ensuring the longevity of the backlight and reducing power consumption. These considerations ensure the LCD operates correctly at 3.3V, with appropriate contrast and backlight settings.

3.7 V/I Sensing Circuit

The PV (Photovoltaic) Panel circuit was designed to measure the open-circuit voltage (VOC) and short-circuit current (ISC) of the PV panel. The circuit is depicted in Figure 9. Assuming 99% of the current flows through the R_{Sense} and $R_{Variable}$ resistors and only 1% through the $R_2 + R_3$ branch, the $R_2 + R_3$ resistance should be 100 times the maximum value of the variable resistor. Therefore, $R_2 + R_3 = 100 \times 1000\Omega = 100k\Omega$. The same logic applies to $R_4 + R_5$, making $R_4 + R_5 = 100k\Omega$.

According to the PV module datasheet, the peak open-circuit voltage can reach 8.2V. When the ADC pin reaches saturation, $V_{ADC} = 3.3V$. Using voltage division, we have $3.3 = 8.2 \times (\frac{R_5}{R_4+R_5})$,

leading to $0.4024 = \frac{R_5}{R_4+R_5}$. Given $R_4 + R_5 = 100k\Omega$, we solve for R_4 and R_5, obtaining $R_4 = 59.75k\Omega \implies 56k\Omega$ and $R_5 = 40.24k\Omega \implies 39k\Omega$. The same values apply to R_2 and R_3 since their relationship and total resistance are the same.

To find the Solar Pannel Voltage the Voltage Across Resistor R5 is calculated as $V_{SP} = V_{ADC(R5)} \times \frac{R_4+R_5}{R_5}$.

The Solar pannel's current is determined by the voltage drop across R_{Sense}, given by $I_{SP} = \frac{V_{SP}-V_{meas}}{R_sense}$ Where $V_{meas} = V_{ADC(R3)} \times \frac{R_3+R_2}{R_3}$. Then the only thing left to find is The Power which is simply The voltage Across R5 multiplied by the current $P_{SP} = V_{SP} \times I_{SP}$.

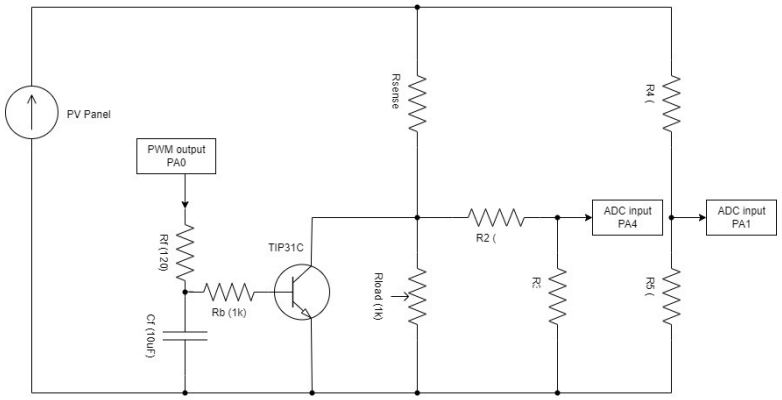

Figure 9: Circuit diagram for the PV Panel current and voltage measurement

Pins PA1 and PA4 are used for ADC measurements. PA1 measures the voltage across $R5$ to determine the Solar Pannel voltage (V_{SP}), while PA4 measures the voltage across R_3 to determine the current (I_{SP}). These pins were chosen because they are ADC-compatible and provide accurate voltage measurements necessary for the calculations.

3.8 LUX Sensing Circuit

The design of the photodiode circuit involves using the MCP602 Op-Amp. According to the datasheet, V_{DD} can range from 2.7V to 6.0V. Here, V_{DD} is set to 3.3V, with V_{SS} connected to ground, ensuring $V_{DD} - V_{SS}$ stays within the 7V limit. The circuit is shown in Figure 10.

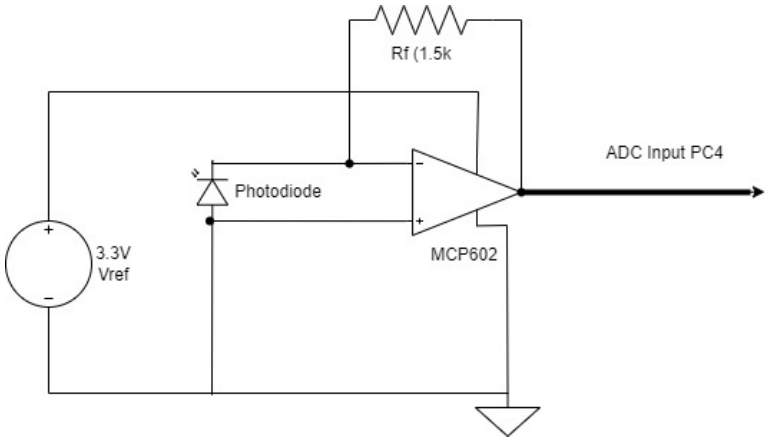

Figure 10: Photodiode and Op-Amp Circuit Diagram

Assuming infinite gain and input impedance for the amplifier, $V_P = V_N = 0V$ and $I_N = I_P = 0A$, simplifies analysis to a single loop. Applying KCL at the photodiode's anode gives $-I_{PD} - \frac{V_N-V_{out}}{R_f} + I_N = 0$. With $V_N = 0V$ and $I_N = 0A$, the current through the photodiode is $I_{PD} = \frac{V_{out}}{R_f}$.

The photodiode saturates at 30,000 lux, ensuring significant readings under bright light and moderate sunlight. From the datasheet's, the relationship between photodiode current and light intensity

is about $80\mu A$ per 1000 lux . Thus we find that at 30,000 lux, the current through the photodiode is $I_{PD} = 2.4mA$.

Given $V_{out} = 3.3V$, the feedback resistor R_f is $R_f = \frac{3.3V}{2.4mA} = 1375\Omega$. A 1500 Ω resistor was chosen. The output voltage connects to pin PC4, configured as an ADC input.

The STM32 ADC operates in 12-bit mode, providing 4096 levels. With $V_{ref} = 3.3V$:

$$\text{Resolution} = \frac{V_{ref}}{2^{12}} = \frac{3.3V}{4096} \approx 0.00080586\,V\,(\,805.86\,\mu V\,)$$

Since the photodiode outputs $10mV/1000\,lux$:

$$\text{Smallest lux change} = \frac{0.00080586\,V}{\frac{3.3V}{1500\,\Omega}} = 366.211\,lux$$

Thus, the smallest detectable change is approximately 366.2 lux.

3.9 Active Load

The active load in the circuit is implemented using a BJT (TIP31C) that receives a voltage signal from the board to its base, with the emitter connected to ground and the collector connected across the load resistor of the PV circuit. Since the STM32 board cannot produce a constant voltage, pin PA0 was chosen to output a 20kHz PWM signal. This signal is then passed through an RC filter to the BJT base.

The RC filter was designed with a cutoff frequency between 100-200Hz, suitable for smoothing the 20kHz PWM signal. Based on the lecture slide's information, a capacitor value of $10\mu F$ and a resistor value of 120Ω were selected for the filter, resulting in a cutoff frequency f_c calculated as follows:

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi \times 120\Omega \times 10\mu F} \approx 132.63\,\text{Hz}$$

This cutoff frequency effectively smooths the PWM signal, ensuring a steady voltage is applied to the BJT base. Additionally, an R_b of $1k\Omega$ was chosen to ensure proper biasing of the BJT.

The voltage difference across the load resistor can be controlled by varying the duty cycle of the PWM signal. Increasing the duty cycle increases the average voltage at the BJT base, while decreasing the duty cycle reduces it. This allows for dynamic adjustment of the load on the PV panel, simulating different load conditions.

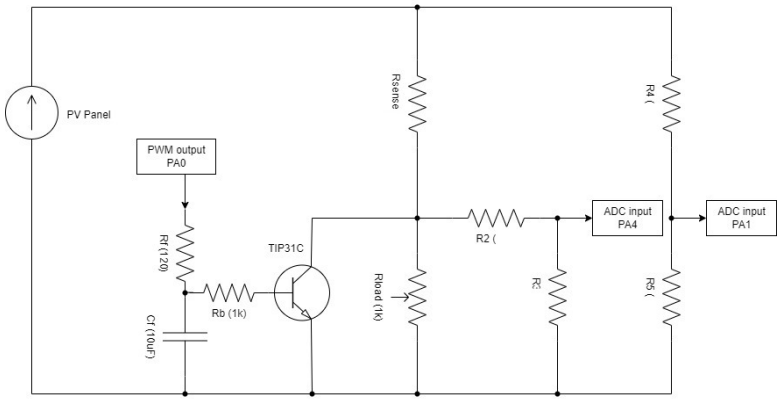

Figure 11: Circuit diagram for the active load implementation

4 Software Design and Implementation

4.1 Top-Level Software Design

The top-level software design is shown in the flowchart in Figure 12. The software is organized into five main sections: Environmental Condition Measurement (EN measurement), Solar Panel Power Measurement (SP measurement), Calibration Process (CA process), Display Mode Switching, and RTC Setting.

The EN measurement section, initiated by pressing the top button or sending the EN UART Command, measures ambient temperature, panel temperature, and light intensity, with LED D3 indicating progress. This concludes when the top button is pressed again or the EN UART Command is sent again, displaying results on the LCD.

The SP measurement section, triggered by pressing the bottom button or sending the SP UART Command, measures the solar panel's voltage, current, and power, with LED D3 indicating progress. This concludes when the bottom button is pressed again or the SP UART Command is sent again. The system calculates efficiency and updates the LCD with measurement data.

The CA process, initiated by pressing the right button or sending the CA UART Command, calibrates the solar panel readings to a baseline, with LEDs D2, D3, and D4 showing progress.

Display Mode Switching, activated by pressing the left button, cycles through display modes on the LCD, showing solar panel power data, environmental conditions, and the current date and time.

RTC Setting, accessed by pressing the middle button, allows setting the date and time on the RTC using the top and bottom buttons to increment and decrement values, with the middle button confirming the settings.

The software ensures smooth interaction between these sections, enabling effective measurement, calibration, and display of relevant data.

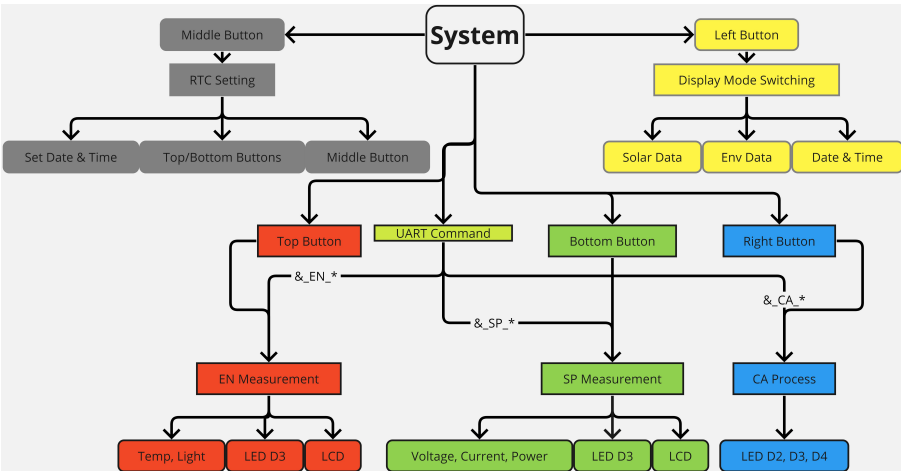

Figure 12: Top-Level Software Design Flowchart

4.2 System Interactions Overview

The diagram in Figure 39 shows the interactions between the project's subsystems. Buttons (PB9, PA5, PB8, PB6, PA7) trigger measurements (EN, SP), calibration (CA), display switching, and RTC setting, with debouncing for accuracy. GPIO interrupts and ADCs (PA1, PA4, PC4, PC5) handle readings for voltage, current, light intensity, and temperature. Timers (TIM1, TIM2, TIM3, TIM4, TIM5) manage PWM generation, LED control, power measurement, calibration, and RTC updates, ensuring precise timing. UART communication allows remote control and data exchange, while global flags track processes to maintain stability. This structure highlights the need for careful coordination and timing in software design, ensuring accurate data processing and functionality.

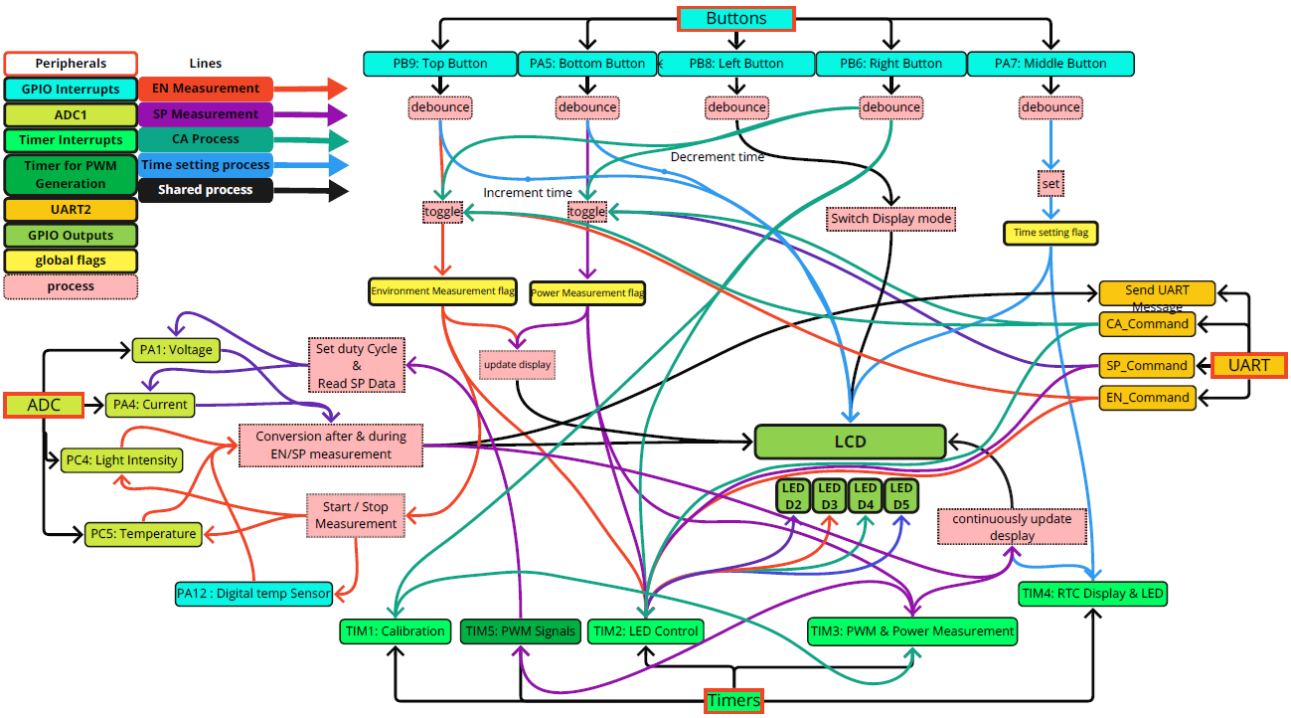

Figure 13: System Interactions and Data Flow Diagram

4.3 Detailed System Processes

4.3.1 Environmental Conditions measurement

The EN measurement starts with pressing the top button or receiving the `&_EN_\n` UART command, toggling the `E_measuring` flag to 1. The LCD updates to display mode 2, and LED D3 blinks 50ms on/off to indicate measurement is in progress.

First, light intensity is measured using the photodiode on ADC channel 14 at PC4, calculated as: $\text{Lux} = \left(\frac{\text{adc\_value}}{4095.0}\right) \times 33.0$.

Next, ambient temperature is measured using the analog sensor on ADC channel 15 at PC5, calculated as: $\text{Amb\_temp} = \left(\frac{\text{adc\_value}}{4095.0}\right) \times 330.0 - 273$.

In parallel, panel temperature is measured by counting pulses from the digital sensor at PA12, using: $\text{Pann\_temp} = \frac{\text{total\_pulse\_count}}{\text{total\_nr\_of\_pulse\_trains}}$.

These measurements are assigned to global variables. The process concludes when the top button is pressed again or a new UART command is received, toggling the `E_measuring` flag to 0. LED D3 stops blinking, the LCD updates with new measurements, and the data is sent via UART.

This sequence ensures continuous and accurate monitoring of environmental conditions, with real-time updates provided through the LCD and LEDs. The flowchart in Figure 14 represents this process.

4.3.2 Solar Panel Power Measurement (SP Measurement)

The SP measurement subsystem begins when the bottom button is pressed and debounced or the `&_SP_\n` UART command is received. This toggles the `P_measuring` flag to 1, initiating the process. The LCD updates to display mode 1, and LED D2 starts blinking 100ms on/off to indicate the process has started.

First, the system configures peripherals and starts Timer 5 for PWM generation on pin PA0. The current duty cycle is initialized to 0, and the Timer 3 interrupt is set to run the measurement loop. The system then enters a loop where the duty cycle is incremented in steps until it reaches 100. In each iteration of the loop, the counter is incremented, and the PWM duty cycle is updated.

Solar data is read from ADC channels 1 and 4. Voltage is converted using the formula: $\text{Voltage4} = \left(\frac{\text{ADC\_Value}}{4095}\right) \times 3000$ Current is converted using: $\text{Current} = \frac{\text{Voltage}}{R_{\text{sense}}}$ Power is calculated as: $\text{Power} = \text{Voltage} \times \text{Current}$

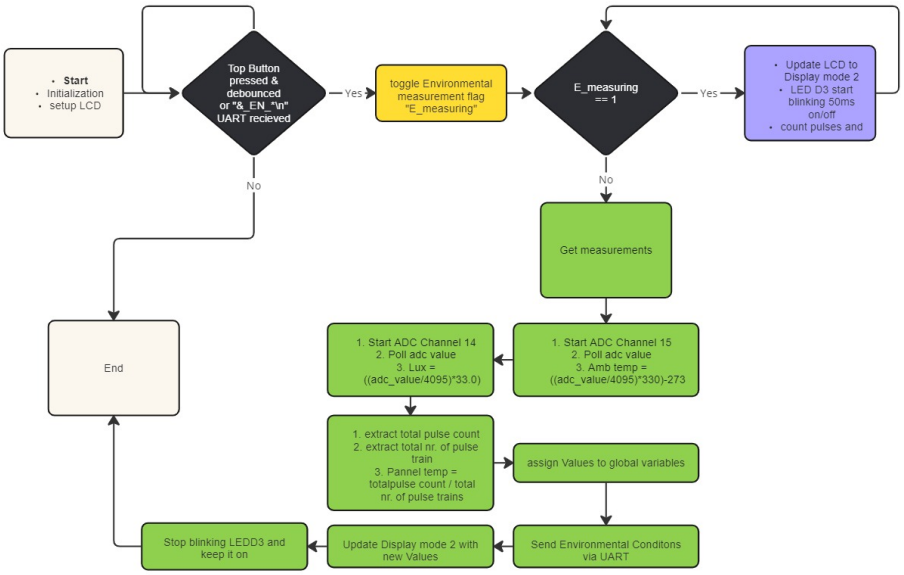

Figure 14: Flowchart of Environmental Condition Measurement (EN Measurement) Process

For more detail on the calculation refer to section 3.7. And for the calculation code refer to the Appendix to the "ReadSolarPanelData" function.

The maximum power point is tracked by comparing the current power with the previous maximum power. If the current power exceeds the previous maximum, the new maximum power, voltage, and current are stored.

Once the duty cycle reaches 100 it is set to 0 again and the system sweeps through the duty cycle again. Once the bottom button is pressed and debounced again or the `&_SP_\n` UART command is received again and the `P_measuring` flag is 0, the process stops. Timer 3 is stopped, and the PWM duty cycle is set to 0. The final values of voltage, current, and power at the maximum power point are displayed on the LCD and sent via UART. LED D2 stops blinking and remains on, indicating the end of the measurement process. The flowchart in Figure 15 visually represents this process.

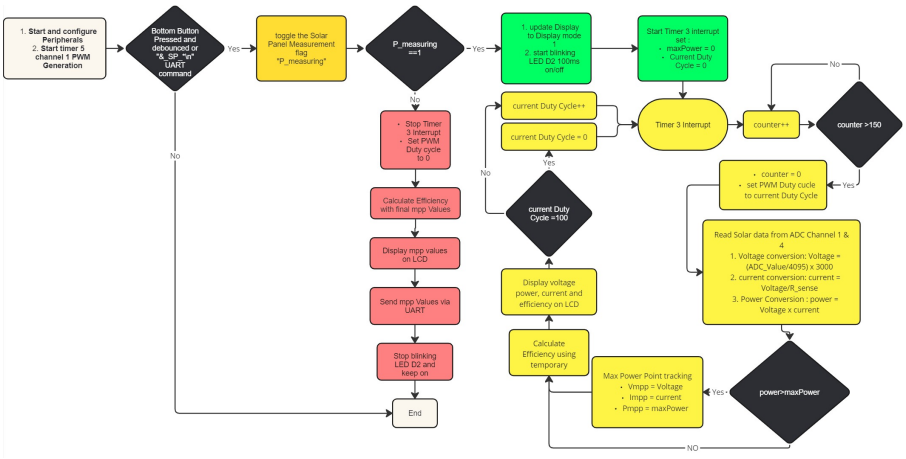

Figure 15: Flowchart of Solar Panel Power Measurement (SP Measurement) Process

4.3.3 Calibration Process (CA Process)

The calibration process begins when the right button is pressed or the `&_CA_\n` UART command is received, toggling the `calibration` flag to 1. The system configures peripherals and starts Timer 5 for PWM generation on pin PA0. Timers 1 and 3 are started, with LED D3 blinking at 50ms intervals

and LED D4 at 200ms intervals to indicate the calibration process is active.

First, the environmental measurement (EN Measurement) is conducted for 2 seconds to record ambient light conditions. After completing this, the system updates the LCD to display mode 2, stops blinking LED D3, and sends the environmental data via UART. The ambient light value is stored in `Lux_calibrated`.

Next, the solar panel measurement (SP Measurement) starts by toggling the `P_measuring` flag. The system reads solar data from ADC channels 1 and 4, converts voltage, current, and power, and updates the maximum power point (MPP) tracking values. The PWM duty cycle increments until it reaches 100, updating voltage, current, and power values.

Once the duty cycle reaches 100, Timer 1 and 3 stop, the PWM duty cycle is set to 0, and the calibrated values are stored and displayed on the LCD and sent via UART. LED D2 and D4 stop blinking and remain on, indicating the end of the calibration process. The flowchart in Figure 16 illustrates this process.

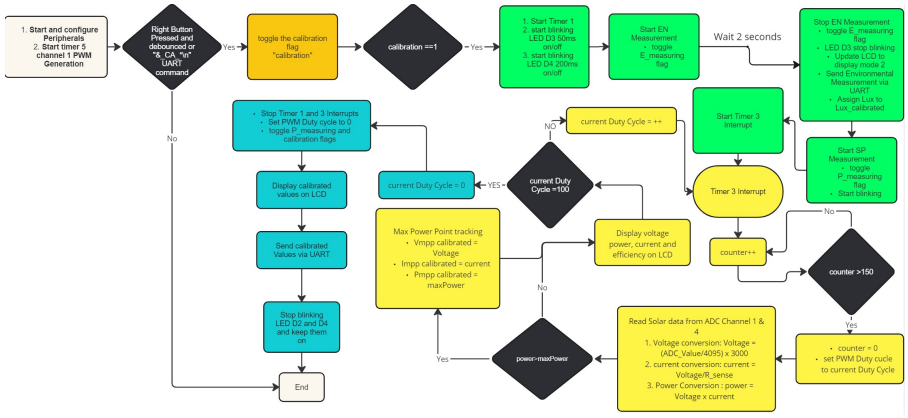

Figure 16: Flowchart of the Calibration Process (CA Process)

4.3.4 Display Mode Switching

The Display Mode Switching section cycles through different display modes when the left button is pressed or the corresponding UART command is received. This allows the user to view various data sets: Mode 1 displays solar panel data (voltage, current, power, efficiency), Mode 2 shows environmental data (ambient temperature, panel temperature, light intensity), and Mode 3 shows the current date and time using the RTC module. Mode 4 automatically cycles through the first three modes. When the left button (GPIO_PIN_8) is pressed, the ‘displayMode‘ variable increments and wraps around after reaching 4. The ‘updateDisplay‘ function refreshes the LCD based on the current ‘displayMode‘. In Mode 4, the system checks elapsed time and cycles through sub-modes 1 to 3.

4.3.5 RTC Setting Process

The RTC Setting Process in the software is initiated when the middle button (GPIO_PIN_7) is pressed. This toggles the ‘settingTime‘ flag and sets the ‘rtcState‘ to ‘STATE_SET_DAY‘. The top button (GPIO_PIN_9) is used to increment the current setting, while the bottom button (GPIO_PIN_5) decrements it. Each press updates the RTC display using ‘updateRTCDisplay‘, which shows the current setting being adjusted. The process cycles through setting the day, month, year, hours, minutes, and seconds with each press of the middle button, advancing the ‘rtcState‘. Once all fields are set, pressing the middle button applies the changes by calling ‘completeRTCSetting‘, which uses ‘HAL_RTC_SetTime‘ and ‘HAL_RTC_SetDate‘ to update the hardware RTC. The display mode is then set to show the updated RTC data, and Timer 4 is started for periodic RTC updates.

4.4 Debug LED and System State Indication Logic

The system uses LEDs paired with buttons to indicate the system state. Each `ButtonLedPair` struct defines a button-LED pair, specifying GPIO pins, ports, and blink durations (e.g., the top button's LED blinks with a 50ms duration).

The timer interrupt function (`HAL_TIM_PeriodElapsedCallback`) manages the blinking. When the timer (TIM2) triggers, LED counters increment, toggling the LED state once the blink duration is reached.

The `toggleLedBlinking` function starts and stops LED blinking. When a button is pressed, it toggles the LED state and manages the timer. If the LED is active, the timer starts; if inactive, the timer stops and the LED turns off. This ensures LEDs provide real-time feedback on button presses and system states.

4.5 UART Communications (Protocol and Timing)

UART communication is essential for the system to describe its current state, report measurements, and receive configuration commands. The communication was established using USART2, with pin PA3 configured as `USART_Rx` and PA2 as `USART_Tx`.

The USART2 channel was set to asynchronous mode with a baud rate of 115200, a 9-bit word length (including parity), and odd parity with 1 stop bit. The mode was configured for both transmission and reception (`TX_RX`), without hardware flow control, and with 16x oversampling. This setup ensures reliable data transmission and reception between the board and external devices.

4.6 Timers

The timers were configured based on the system clock frequency of 32 MHz. The timer period in milliseconds is calculated using the formula:

$$\text{Timer Period (ms)} = \left(\frac{\text{Prescaler} + 1}{32 \times 10^6}\right) \times (\text{Period} + 1) \times 1000$$

Using this formula, the timers were configured as follows:

TIM1: Prescaler 31, period 999, timer period 1 ms. Handles calibration process timing for environmental and power measurement sequences.

TIM2: Prescaler 31, period 999, timer period 1 ms. Manages LED blinking logic for indicating different states such as measurement and calibration.

TIM3: Prescaler 31, period 99, timer period 0.1 ms. Manages PWM duty cycle adjustments and solar panel data measurement, providing high-resolution timing.

TIM4: Prescaler 31999, period 999, timer period 1 second. Handles RTC display updates, ensuring accurate date and time display every second on the LCD.

TIM5: Prescaler 0, period 1599, timer period 50 microseconds. Generates PWM signals for controlling the active load in the power measurement circuit, allowing precise duty cycle control.

4.7 PWM Duty Cycle Adjustment for Active load

The PWM duty cycle was set and adjusted using TIM5. The function `SetPWM_DutyCycle(uint32_t dutyCyclePercentage)` was used to set the duty cycle. The desired duty cycle percentage was passed to this function, which then calculated the corresponding compare value based on the timer's period. This value was then set using the `__HAL_TIM_SET_COMPARE` macro, adjusting the PWM signal's duty cycle to the specified percentage.

4.8 ADC Configuration and Management

In the project, the Analog-to-Digital Converter (ADC) was configured to operate with specific settings to ensure accurate and efficient data acquisition. The ADC was set up with a clock prescaler of `ADC_CLOCK_SYNC_PCLK_DIV4`, a resolution of 12 bits, and continuous conversion mode. To allow for

custom channel selection, the auto-generated channel configuration code was commented out. Instead, custom functions were implemented to dynamically configure the ADC channels as needed.

Four functions were created to select specific ADC channels dynamically, each setting the channel, rank, and sampling time for the desired channel. For instance, the `ADC_Select_CH14` function configures Channel 14 with a sampling time of 480 cycles.

The LightSensing function exemplifies this approach. It selects ADC Channel 14, starts the ADC conversion, waits for the conversion to complete, retrieves the ADC value, and converts it to light intensity using a predefined formula. This method allows flexible and dynamic selection of ADC channels, enabling accurate measurements for various sensors in the system.

4.9 LMT01 Interface, Timing, and Synchronization

The LMT01 digital temperature sensor provides precise temperature readings through a pulse train, with the pulse count indicating the temperature. The sensor connects to a GPIO pin, which triggers an interrupt service routine (ISR) for each detected pulse. In the ISR, the system increments the pulse count and records the timestamp for the start of the pulse train.

Upon pressing the top button, the environment measurement state toggles, initiating pulse counting from the LMT01. During this state, the ISR continuously updates the pulse count until the measurement cycle ends. The pulse trains are counted by comparing the start and stop times of the pulse trains, checking if the interval exceeds 50 ms.

The `DigitalTemperatureSensing` function calculates the average number of pulses per train and converts this average to temperature using the formula:

$P\_temp = \left(\frac{\text{avg\_pulse\_per\_train} \times 256}{4096}\right) - 50$

The environment measurement process involves collecting pulses over a set period, toggling the LED to indicate activity, and updating the display with ambient and panel temperatures, along with light intensity.

Synchronization is ensured by resetting the pulse counts and timestamps at the end of each measurement cycle, maintaining accuracy. The HAL library functions are utilized to manage precise timing and synchronization for the LMT01 interface, ensuring reliable temperature data acquisition.

4.10 Button Debounce

Button debounce is handled in software to ensure accurate input detection. The buttons, configured as active low, trigger interrupts on both rising and falling edges. Upon detecting a press, an interrupt records the current time using `HAL_GetTick()`. This time is compared to the last recorded press time; if the interval exceeds a predefined debounce delay (e.g., 20 ms), the press is considered valid. This filters out noise and rapid unintended presses, ensuring only intentional presses are registered.

4.11 Efficiency Calculation

The efficiency calculation for the solar panel system accounts for variations in temperature and light intensity. Constants defined include the Standard Testing Conditions temperature ($T_{\text{STC}} = 25.0°C$) and the temperature coefficient for the PV module ($\beta = -0.004$). The measured values are the maximum power point (P_{measured}), the PV panel temperature ($T_{\text{pv\_panel}}$), and the light intensity ($\text{Lux}_{\text{measured}}$).

First, the power output is normalized for temperature using: $P_{\text{normT}} = \frac{P_{\text{measured}}}{1+\beta\cdot(T_{\text{pv\_panel}}-T_{\text{STC}})}$

Next, the power is normalized for light intensity with: $P_{\text{normalised}} = P_{\text{normT}} \cdot \left(\frac{\text{Lux}_{\text{calibrated}}}{\text{Lux}_{\text{measured}}}\right)$

Finally, the efficiency is calculated as a percentage of the normalized power relative to the calibrated maximum power point ($P_{\text{mpp\_calibrated}}$): $\text{Efficiency} = \left(\frac{P_{\text{normalised}}}{P_{\text{mpp\_calibrated}}}\right) \cdot 100$

4.12 LCD Interface: Control and Data Conversion

The LCD operates in 4-bit mode to optimize GPIO usage. Key signals include RS for command/data selection and E for latching data, with D4 to D7 for 4-bit data chunks.

Initialization starts by setting the LCD to 8-bit mode with `0x03` sent three times, then switching to 4-bit mode with `0x02`. Commands `0x28` (4-bit, 2-line), `0x0C` (display on, cursor off), `0x06` (increment cursor), and `0x01` (clear display) configure the display.

In 4-bit mode, bytes are split into two nibbles. GPIO pins are set for each nibble, and the Enable line toggles to latch data. Commands and data are sent via dedicated functions: `lcd16x2_sendCommand` sets RS low, sends nibbles, and toggles Enable. `lcd16x2_sendData` sets RS high and sends nibbles. `lcd16x2_init_4bits` handles initialization.

To display data, functions like `lcd16x2_setCursor` and `lcd16x2_printf` are used. `lcd16x2_setCursor` positions the cursor using row and column parameters, calculates the DDRAM address, and sends a command to set it. `lcd16x2_printf` formats a string and sends each character using `lcd16x2_sendData`. `updateDisplay` manages which data to display, updating the LCD with information like voltage, current, power, efficiency, temperature, and light intensity. This ensures accurate data display by calling `lcd16x2_setCursor` and `lcd16x2_printf`.

5 Measurements and Results

5.1 Power Supply

Requirement: The power supply should maintain stable output voltages of 3.3V and 5V within a specified tolerance, across an input voltage range of 9V to 12V.
Test Procedure: The input voltage to the power supply was incrementally adjusted from 8V to 12V. At each step, the output voltages were measured between TIC Pins: 4 (5V), 6 (3.3V), and ground.
Expected Results: The 3.3V output should remain close to 3.3V, and the 5V output should remain stable around 5V, demonstrating effective voltage regulation despite changes in the input voltage.
Test Results: The test confirmed the power supply's ability to maintain regulated outputs within the expected ranges. The 3.3V output was measured between 3.357V and 3.358V, while the 5V output was consistently between 5.007V and 5.009V across the entire range of input voltages.

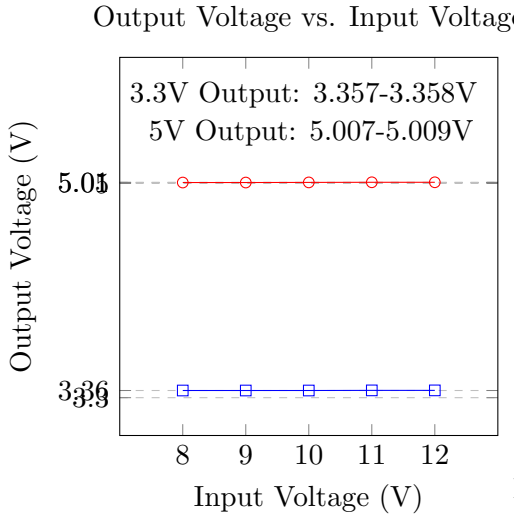

Figure 17: 3.3V and 5V output voltages vs. input voltage .

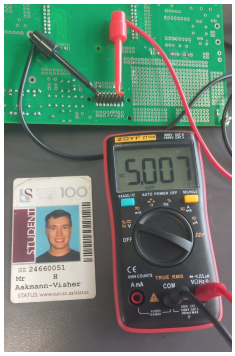

Figure 18: Multimeter measurements

5.2 UART Message Response Time

Requirement: The system should start sending UART messages within 100ms but no later than 500ms after power-up, ensuring timely communication initialization.
Test Procedure: After powering up the system, an oscilloscope was used to monitor the UART line for the first message transmission. The time from power-up to the first message was precisely

measured.
Expected Results: The first UART message should be observed on the oscilloscope trace between 100ms and 500ms after the system is powered up, indicating compliance with the communication startup time requirement.
Test Results: The oscilloscope captured the timing of the first UART message transmission, confirming that the system initiates communication within the specified time frame after power-up. The precise response time measured is 320ms.
Additional Verification: An additional oscilloscope capture shows the EN UART message, validating the system's ability to send specific commands accurately.

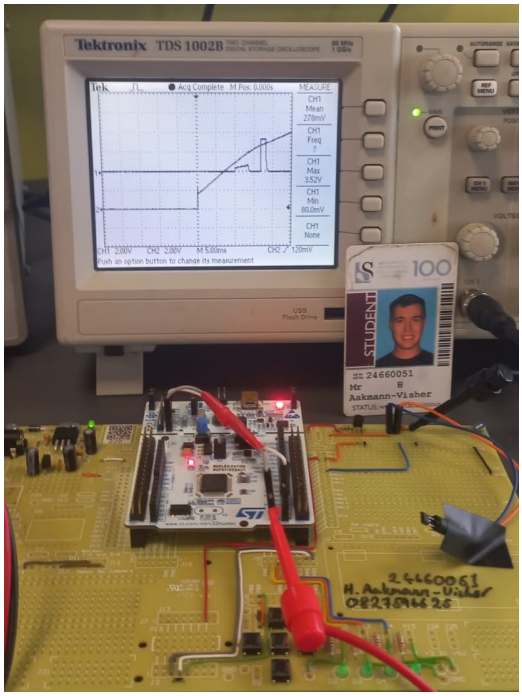

Figure 19: UART message after system powerup response time.

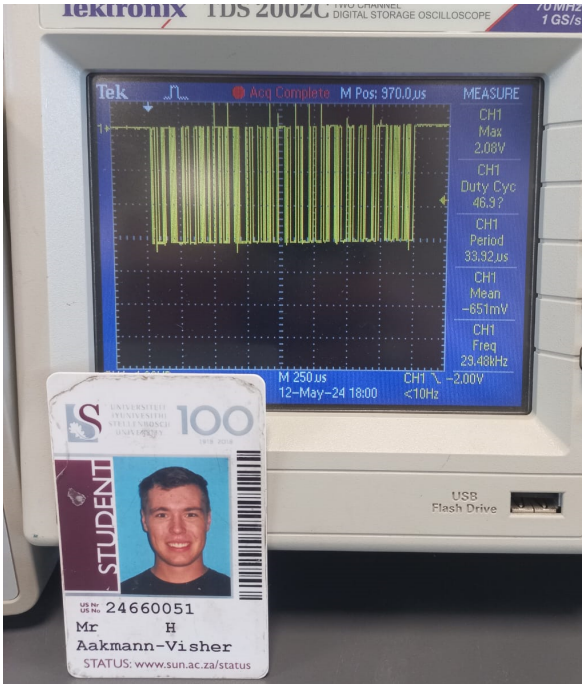

Figure 20: Oscilloscope capture of the EN UART message transmission.

5.3 Temperature Sensors

5.3.1 Analog Temperature Sensor (LM235)

Requirement: The LM235 sensor should provide a voltage output proportional to the temperature, with a sensitivity of 10mV/°K.
Test Procedure: The output voltage of the LM235 sensor was measured at 22°C and 35°C using an oscilloscope.
Expected Results: The output voltage should match the expected values calculated using the formula $V_{out} = 10mV/°K \times T(K)$.
Test Results: The oscilloscope measurements were as follows: at 22°C, the measured voltage was 2.96V, and the expected voltage was 2.9515V. At 35°C, the measured voltage was 3.12V, and the expected voltage was 3.0815V. The measured voltages closely match the expected values, confirming the sensor's accuracy.

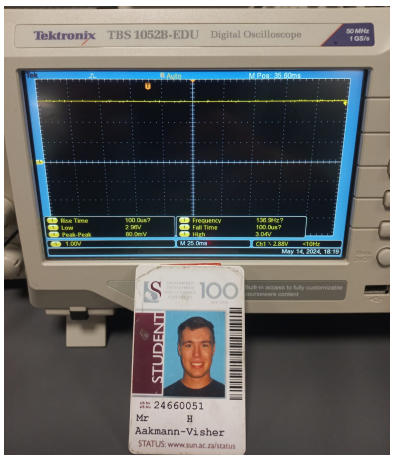

Figure 21: Oscilloscope measurement of LM235 output at 22°C.

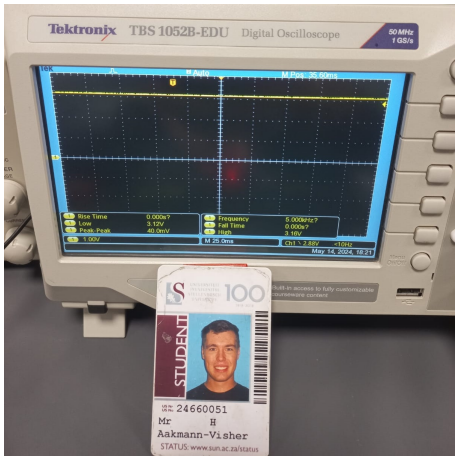

Figure 22: Oscilloscope measurement of LM235 output at 35°C.

5.3.2 Digital Temperature Sensor (LMT01)

Requirement: The LMT01 sensor should accurately measure temperature via a pulse train duration, with sufficient amplitude for the microcontroller to detect HIGH and LOW signals. According to the STM32 datasheet, the voltage difference between HIGH and LOW for a GPIO pin to recognize the signal correctly should be at least 1.2V . The LMT01 sensor datasheet specifies the start-to-start time to be a maximum of 104 ms and the start-to-stop time for the pulse train to be a maximum of 50 ms .

Test Procedure: An oscilloscope was used to capture the pulse train duration, start-to-start time, and signal amplitude.

Expected Results: The pulse count should be proportional to the temperature, the start-to-start time should align with the expected period, and the amplitude should be within the recognizable range for the microcontroller.

Test Results: Oscilloscope measurements confirmed the following: - Pulse train duration corresponds to the calculated temperature using the formula Temperature (°C) $= \frac{\text{Pulse Count} \times 256}{4096} - 50$. - Start-to-start time is 95 ms, which is within the 104 ms maximum specified by the sensor datasheet. - Start-to-stop time is 900 µs, well within the maximum 50 ms specified. - Signal amplitude difference is 1.16V, which is less than sufficient for the STM32 microcontroller to detect HIGH and LOW states.

These measurements validate the LMT01 sensor's accuracy and reliability in measuring temperature.

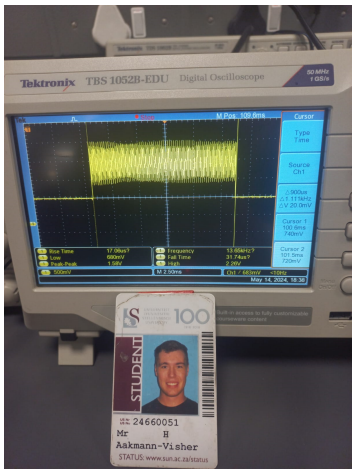

Figure 23: Pulse train duration measurement.

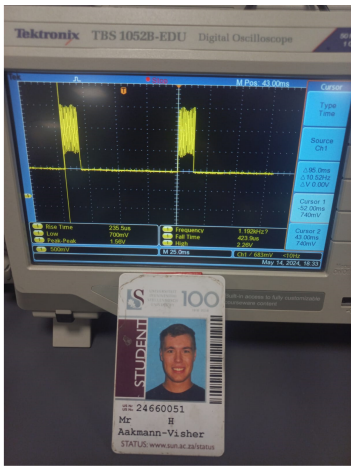

Figure 24: Start to start time measurement.

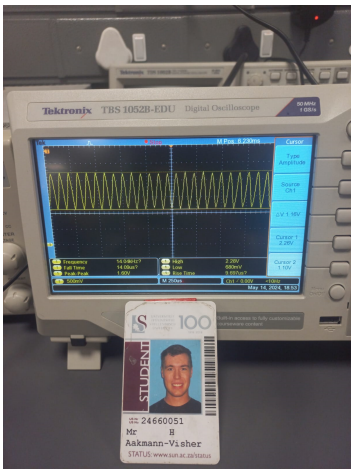

Figure 25: Signal amplitude measurement.

5.4 Light Measurement

Requirement: The photodiode should accurately measure light intensity, with a voltage range recognizable by the ADC for high and low light conditions.

Test Procedure: The voltage was measured at the cathode and ADC pin under various light conditions using an oscilloscope.

Expected Results: The voltage should vary significantly between high and low light conditions, ensuring the ADC can accurately convert the analog signal to a digital value. The calculated expected lux values for the measurements are as follows: - For 292 µV, the expected lux is 0.00291 lux. - For 7.20 mV, the expected lux is 0.072 lux. - For 2.00 mV, the expected lux is 0.020 lux. - For 136 mV, the expected lux is 1.361 lux.

Test Results: - At 586 lux (low light), the voltage at the photodiode cathode was 292µV. - At 21758 lux (high light), the voltage at the photodiode cathode was 7.20mV. - At 879 lux (low light), the voltage at the ADC pin was 2.00mV. - At 12673 lux (high light), the voltage at the ADC pin was 136mV.

These results indicate that the photodiode produces completely different measurements than expected thus it can be assumed that the software that was used is wrong, either due to samle cycles or due to the conversion.

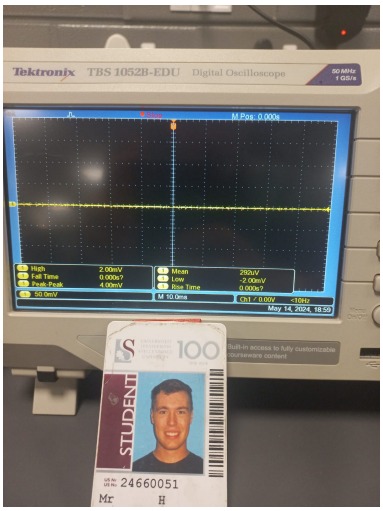

Figure 26: Photodiode cathode at 586 lux.

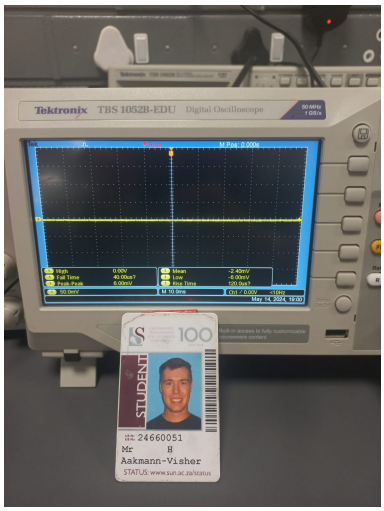

Figure 27: Photodiode cathode at 21758 lux.

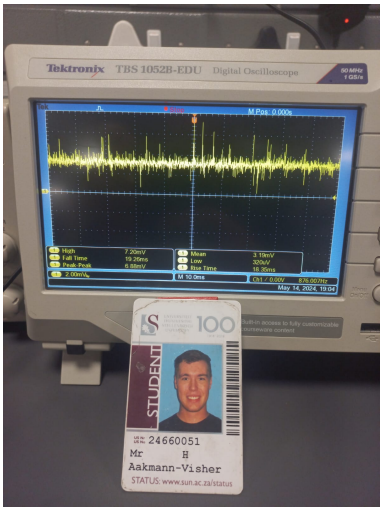

Figure 28: Photodiode pin at 879 lux.

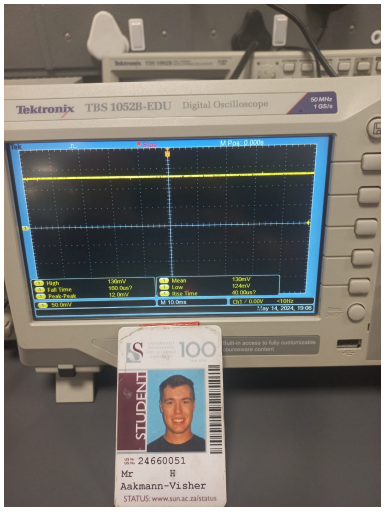

Figure 29: Photodiode pin at 12673 lux.

5.5 LCD Backlight Forward Voltage

Requirement: The LCD backlight should operate within the expected forward voltage range to ensure proper illumination.

Test Procedure: The forward voltage of the LCD backlight was measured at pin A using an oscilloscope, with a backlight resistor connected. The expected forward voltage is 3V without the resistor connected. However, a 100 ohm resistor was used instead of the calculated 10 ohm resistor.

Expected Results: With the calculated 10 ohm resistor, the expected forward voltage is 3V.

Test Results: The measured forward voltage was 2.88V with the 100 ohm resistor connected. The discrepancy is likely due to the higher resistor value used.

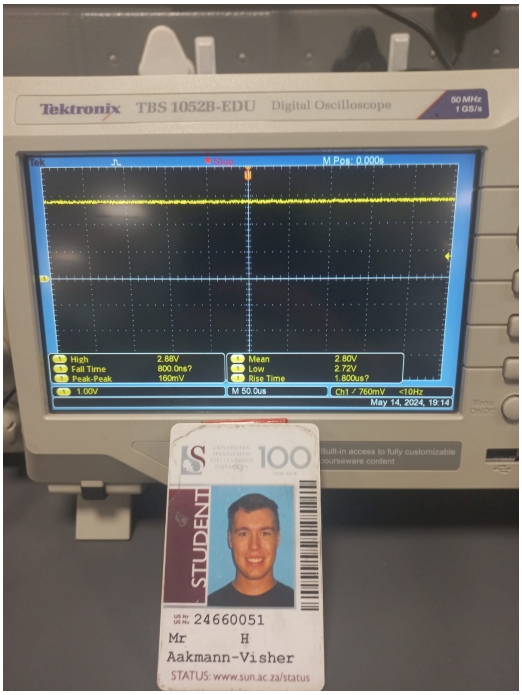

Figure 30: Oscilloscope measurement of LCD backlight forward voltage.

5.6 Voltage, Current, Power, and Efficiency Measurement

Requirement: The system should measure voltage, current, power, and efficiency accurately. The expected values at 5V input should align with the system specifications and calibration settings.

Test Procedure: A power supply was set to 5V with a maximum current of 0.1A to simulate the PV panel. Two measurements were taken: one before calibration and one after calibration, to observe any changes in efficiency.

Expected Results: At a 5V input and assuming a maximum current of 0.1A, the theoretical values are:

- Voltage: 5V
- Current: 0.1A
- Power: $P = V \times I = 5V \times 0.1A = 0.5W$
- Efficiency: This depends on the actual power conversion and load characteristics, but ideally close to 100%.

Test Results: The oscilloscope and LCD measurements are as follows:

- Before Calibration:
  - Voltage: 4.306V

  - Current: 26mA
  - Power: 114mW
  - Efficiency: 0%

- After Calibration:
  - Voltage: 4.245V
  - Current: 32mA
  - Power: 139mW
  - Efficiency: 100%

Analysis and Comparison: The measured values significantly deviate from the expected results. Specifically, the voltage is much lower than the expected 5V, and the current is also much lower than the maximum 0.1A set on the power supply. This leads to much lower power values than the theoretical 500mW. The efficiency reading improved from 0% to 100% after calibration, but the power values remain lower than expected.

Possible Reasons for Discrepancies:

- **Calibration Issues:** The initial calibration might have been off, causing incorrect initial readings. The post-calibration values improved but still show discrepancies, indicating potential issues with the calibration process or the sensors.
- **Power Supply Limitations:** The power supply might not be delivering a stable 5V, or there could be noise affecting the measurements.
- **Measurement Accuracy:** The accuracy of the measurement system (ADC resolution, sensor accuracy) could also contribute to the observed discrepancies.

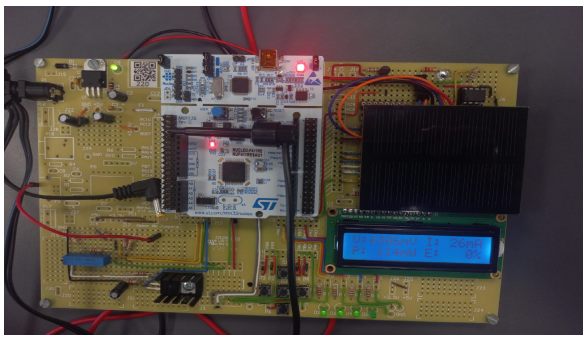

Figure 31: Measurement before calibration.

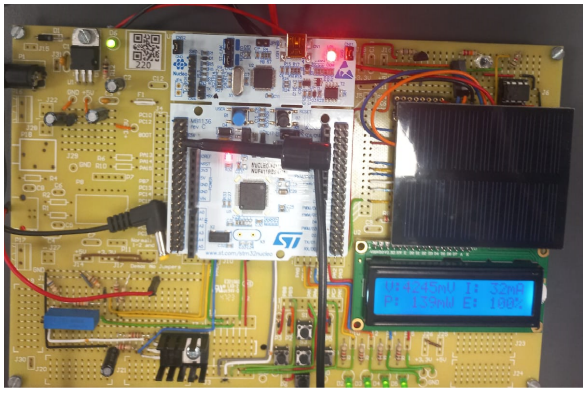

Figure 32: Measurement after calibration.

These observations suggest that further investigation and refinement are needed in the calibration process and measurement system to achieve more accurate results.

Active Load Measurement

The measurements for the active load were taken by analyzing the PWM output at different duty cycles and the corresponding base voltages at the BJT. The following duty cycles and base voltages were observed:

- **50% Duty Cycle:** PWM output (yellow trace), Base voltage (blue trace): 2.00V
- **93% Duty Cycle:** PWM output (yellow trace), Base voltage (blue trace): 2.87V
- **100% Duty Cycle:** PWM output (yellow trace), Base voltage (blue trace): 3.07V

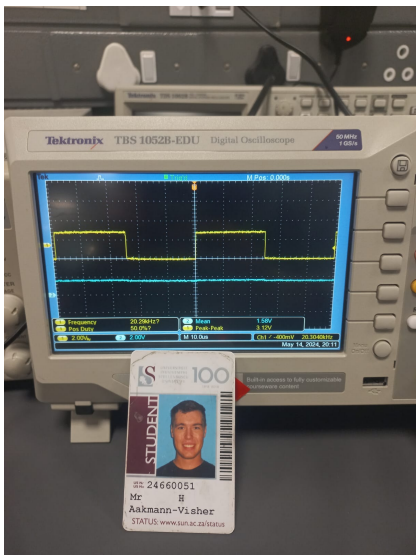

Figure 33: PWM output at 50% duty cycle and corresponding base voltage (2.00V).

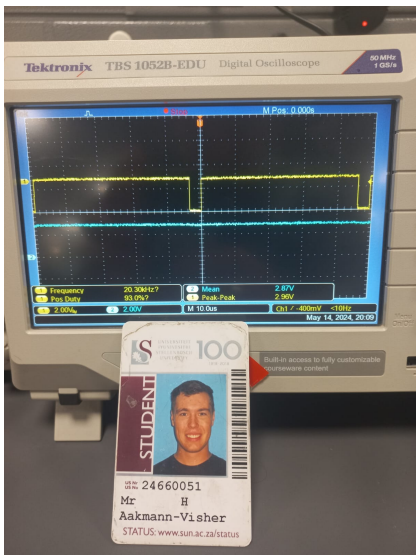

Figure 34: PWM output at 93% duty cycle and corresponding base voltage (2.87V).

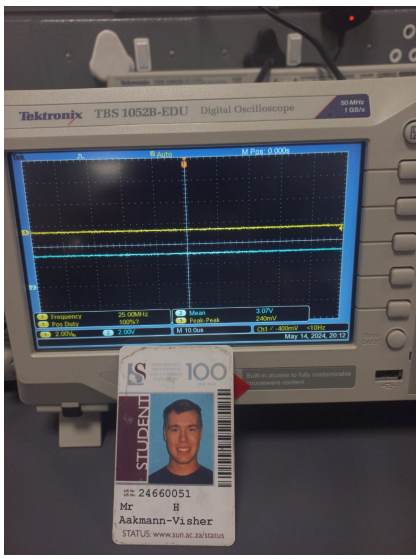

Figure 35: PWM output at 100% duty cycle and corresponding base voltage (3.07V).

The graph in Figure 36 illustrates how the base voltage varies with changes in the PWM duty cycle. The results indicate a nearly linear relationship between the duty cycle and the base voltage, confirming that the PWM signal effectively controls the BJT's base voltage.

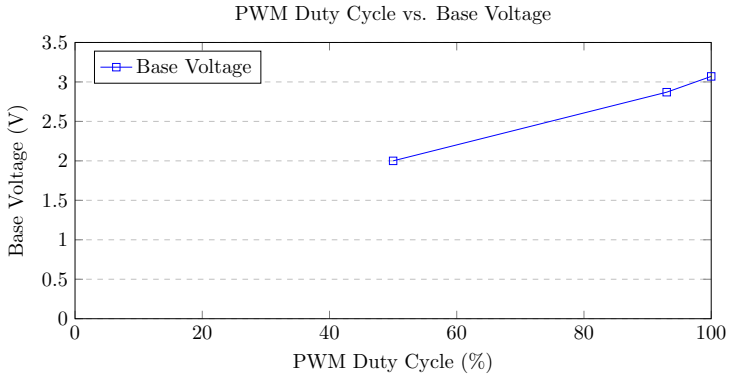

Figure 36: Graph showing the relationship between PWM duty cycle and base voltage.

6 Conclusion

The system was successfully implemented, with all components and functionalities operating as intended. The software code executes smoothly, without any noticeable delays, which indicates robust performance and reliable integration of the various modules. Overall, the system meets the design specifications and performs well under test conditions.

However, it was observed that the ADC measurements were not as accurate as desired. This limitation suggests that additional testing and calibration are necessary to enhance measurement precision. Accurate ADC measurements are crucial for the system's overall accuracy, especially in applications requiring precise data acquisition.

For future improvements, it is recommended to conduct more comprehensive testing and calibration procedures. This will help identify any potential sources of error and allow for fine-tuning of the ADC and related components. Ensuring precise and reliable ADC measurements will significantly improve the system's performance and reliability.

In conclusion, while the current system is functional and performs satisfactorily, there is room for improvement in the accuracy of the ADC measurements. Addressing this issue will contribute to the development of an even more robust and reliable system in the future.

A Complete `main.c`

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
#include "lcd16x2.h"

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
//-----------------------------------------------------------Button and LED pairs
typedef struct {
 uint16_t buttonPin;          // Button GPIO Pin
 GPIO_TypeDef* buttonPort;    // Button GPIO Port
 uint16_t ledPin;             // LED GPIO Pin
 GPIO_TypeDef* ledPort;       // LED GPIO Port
 uint8_t State;               // LED State
 uint32_t blinkDuration;      // Blink duration in terms of the number of timer ticks (e.g.,
} ButtonLedPair;

typedef enum {
 CALIBRATION_STATE_IDLE,
 CALIBRATION_STATE_EN_START,
 CALIBRATION_STATE_EN_WAIT,
 CALIBRATION_STATE_EN_STOP,
 CALIBRATION_STATE_SP_START,
 CALIBRATION_STATE_SP_MEASURE,
 CALIBRATION_STATE_COMPLETE
} CalibrationState;

typedef struct {
```

```
 CalibrationState state;
 uint32_t timestamp;
 uint32_t CalibrationCounter;
 uint32_t readIntervalCounter;
} CalibrationContext;

typedef enum {
 STATE_IDLE,
 STATE_SET_HOURS,
 STATE_SET_MINUTES,
 STATE_SET_SECONDS,
 STATE_SET_DAY,
 STATE_SET_MONTH,
 STATE_SET_YEAR,
 STATE_COMPLETED
} RTC_SetState;

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

RTC_HandleTypeDef hrtc;

TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;
TIM_HandleTypeDef htim5;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

// Calibration Context
CalibrationContext calibrationContext = {CALIBRATION_STATE_IDLE, 0, 0, 0};

// RTC State and Data
RTC_SetState rtcState = STATE_IDLE;
RTC_TimeTypeDef rtcTime = {0};
RTC_DateTypeDef rtcDate = {0};

// UART-related variables
```

```
uint8_t rxDataByte[1];                  // Buffer for receiving one byte of data via UART
char stdnum[13] = "&_24660051_*\n";     // Standard number string to transmit via UART
#define UART_BUFFER_SIZE 12             // Buffer size for the command
char uartBuffer[UART_BUFFER_SIZE];      // Buffer to store received characters
int bufferIndex = 0;                    // Index for the next character in the buffer

// Timer-related variables
#define DEBOUNCE_DELAY 40               // Debounce delay in ms
#define EN_MEASURE_TIME 1000            // 1 second for environment measurement
#define SP_MEASURE_TIME 6000            // 8 seconds total for the duty cycle sweep

// Global variable to track button press
volatile uint32_t buttonPressed = 0;

// Environment Sensing-related variables
volatile uint32_t E_measuring = 0;      // Flag for measuring state: 0 - Not measuring, 1 - M
volatile uint32_t P_measuring = 0;      // Flag for Power measurement
volatile uint32_t calibration = 0;      // Flag for Calibration
volatile uint8_t settingTime = 0;       // Flag for setting RTC time

// Pulse counting and temperature variables for digital temperature measurement
volatile uint32_t last_pulse_train_start = 0; // Timestamp for the start of the last pulse t
volatile uint32_t pulse_count = 0;            // Count of pulses in the current pulse train
volatile uint32_t total_pulse_count = 0;      // Total pulses counted across all trains
volatile uint32_t pulse_train_count = 0;      // Count of pulse trains
volatile float Amb_temp = 0.0f;               // Ambient temperature
volatile float Pann_temp = 0.0f;              // Panel temperature
volatile float lightIntensity = 0.0f;         // Light intensity

// Solar Panel Data
volatile int Vpv = 0.0f;                      // PV voltage
volatile int Ipv = 0.0f;                      // PV current
volatile int Ppv = 0.0f;                      // PV power
volatile int Eff = 0.0f;                      // Efficiency
volatile int Vmpp = 0.0f;                     // Maximum power point voltage
volatile int Impp = 0.0f;                     // Maximum power point current
volatile int Pmpp = 0.0f;                     // Maximum power point power
volatile int Pmpp_calibrated = 0;             // Calibrated MPP power
volatile int Vmpp_calibrated = 0;             // Calibrated MPP voltage
volatile int Impp_calibrated = 0;             // Calibrated MPP current
volatile int currentDutyCycle = 0;            // Current PWM duty cycle
volatile int maxPower = 0.0f;                // Maximum power
volatile float Lux_calibrated= 0.0f;          // Calibrated light intensity

// Button-LED configurations
ButtonLedPair buttonLedPairs[] = {
 {GPIO_PIN_9, GPIOB, GPIO_PIN_4, GPIOB, 0, 50}, // Pair 0: Top Button on GPIO_PIN_9, LED on
 {GPIO_PIN_5, GPIOA, GPIO_PIN_10, GPIOB, 0, 100}, // Pair 1: Bottom Button & LED D2
 {GPIO_PIN_8, GPIOB, GPIO_PIN_10, GPIOA, 0, 500}, // Pair 2: Left Button & LED D5
 {GPIO_PIN_6, GPIOB, GPIO_PIN_5, GPIOB, 0, 200}, // Pair 3: Right Button & LED D4
 // Additional pairs can be added here
};
```

```
// Global LCD handle
volatile uint32_t displayMode = 1;  // Flag to track display mode, 'volatile' if updated in
// Declare a variable to keep track of the last update time
static uint32_t lastUpdateTime = 0;

// Declare a variable to keep track of the current sub-mode
static uint8_t currentSubMode = 1;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_RTC_Init(void);
static void MX_TIM5_Init(void);
static void MX_TIM4_Init(void);
static void MX_NVIC_Init(void);
/* USER CODE BEGIN PFP */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim); // Callback function for timer
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart); // Callback function for UART recei

// UART Communication Functions
void EN_Command(const char *commandBuffer); // Handles the "&_EN_*" UART command
void SP_Command(const char *commandBuffer); // Handles the "&_SP_*" UART command
void CA_Command(const char *commandBuffer); // Handles the "&_CA_*" UART command
void PowerConditions(int Vmpp, int Impp, int Pmpp, int Eff); // Formats and sends power cond
void EnvironmentConditions(float Amb_temp, float Pann_temp, float lightintensity); // Format

// PV unit Data Functions
void ReadSolarPanelData(void); // Reads solar panel data
void ProcessCalibration(void); // Processes the calibration procedure
int CalculateEfficiency(); // Calculates the efficiency of the solar panel

// Environment Sensing Functions
float LightSensing(); // Reads and converts ADC value to light intensity
float AnalogTemperatureSensing(); // Reads and converts ADC value to ambient temperature
float DigitalTemperatureSensing(); // Handles temperature measurement and processing for dig

// RTC Setting Functions
void decrementRTCSetting(RTC_SetState state); // Decrements the RTC setting based on the cur
void incrementRTCSetting(RTC_SetState state); // Increments the RTC setting based on the cur
void updateRTCDisplay(RTC_SetState state); // Updates the RTC display based on the current s
void completeRTCSetting(void); // Completes the RTC setting process and applies changes

// LCD Functions
void setup(void); // Initial setup function
void updateDisplay(uint32_t displayMode, float Amb_temp, float Pann_temp, float lightIntensi
```

```
void lcdLiveMeasuring(int Vpv, int Ipv, int Ppv, int Eff); // Updates the LCD with live meas

// LED Control Functions
void toggleLedBlinking(ButtonLedPair *pair); // Toggles LED blinking based on button press
void ControlLED4BasedOnEfficiency(int Eff); // Controls LED4 based on efficiency value

// PWM Control Functions
void SetPWM_DutyCycle(uint32_t dutyCyclePercentage); // Sets the PWM duty cycle

// ADC Channel Selection Functions
void ADC_Select_CH1(void); // Selects ADC Channel 1
void ADC_Select_CH4(void); // Selects ADC Channel 4
void ADC_Select_CH14(void); // Selects ADC Channel 14
void ADC_Select_CH15(void); // Selects ADC Channel 15


/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();
```

```
  MX_TIM1_Init();
  MX_TIM2_Init();
  MX_TIM3_Init();
  MX_RTC_Init();
  MX_TIM5_Init();
  MX_TIM4_Init();

  /* Initialize interrupts */
  MX_NVIC_Init();
  /* USER CODE BEGIN 2 */
  setup();
  updateDisplay(displayMode,Amb_temp, Pann_temp, lightIntensity,Vmpp,Impp,Pmpp,Eff);
  /* Send the standard number message only once after setup */
  HAL_UART_Transmit(&huart2, (uint8_t*) stdnum, 13, 200);
  HAL_TIM_PWM_Start(&htim5, TIM_CHANNEL_1);
  SetPWM_DutyCycle(0);
  /* Re-enable UART receive interrupt */
  HAL_UART_Receive_IT(&huart2, rxDataByte, 1);
  HAL_TIM_Base_Start_IT(&htim4);

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

 /* USER CODE END WHILE */

 /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI|RCC_OSCILLATORTYPE_LSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
```

```
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.LSIState = RCC_LSI_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 64;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 4;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
 Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
         |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV2;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
 Error_Handler();
  }
}

/**
  * @brief NVIC Configuration.
  * @retval None
  */
static void MX_NVIC_Init(void)
{
  /* ADC_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(ADC_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(ADC_IRQn);
  /* TIM2_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(TIM2_IRQn);
  /* TIM3_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(TIM3_IRQn);
  /* USART2_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(USART2_IRQn);
  /* EXTI9_5_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
  /* EXTI15_10_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
}
```

```
/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

//  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = ENABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
 Error_Handler();
  }
//
//  /** Configure for the selected ADC regular channel its corresponding rank in the sequenc
//  */
//  sConfig.Channel = ADC_CHANNEL_14;
//  sConfig.Rank = 1;
//  sConfig.SamplingTime = ADC_SAMPLETIME_15CYCLES;
//  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
//  {
//    Error_Handler();
//  }
//
//  /** Configure for the selected ADC regular channel its corresponding rank in the sequenc
//  */
//  sConfig.Channel = ADC_CHANNEL_15;
//  sConfig.Rank = 2;
//  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
```

```
//  {
//    Error_Handler();
//  }
//
//  /** Configure for the selected ADC regular channel its corresponding rank in the sequenc
//  */
//  sConfig.Channel = ADC_CHANNEL_1;
//  sConfig.Rank = 3;
//  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
//  {
//    Error_Handler();
//  }
//
//  /** Configure for the selected ADC regular channel its corresponding rank in the sequenc
//  */
//  sConfig.Channel = ADC_CHANNEL_4;
//  sConfig.Rank = 4;
//  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
//  {
//    Error_Handler();
//  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief RTC Initialization Function
  * @param None
  * @retval None
  */
static void MX_RTC_Init(void)
{

  /* USER CODE BEGIN RTC_Init 0 */

  /* USER CODE END RTC_Init 0 */

  RTC_TimeTypeDef sTime = {0};
  RTC_DateTypeDef sDate = {0};

  /* USER CODE BEGIN RTC_Init 1 */

  /* USER CODE END RTC_Init 1 */

  /** Initialize RTC Only
  */
  hrtc.Instance = RTC;
  hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
  hrtc.Init.AsynchPrediv = 127;
  hrtc.Init.SynchPrediv = 255;
  hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
```

```
  hrtc.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
  hrtc.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
  if (HAL_RTC_Init(&hrtc) != HAL_OK)
  {
 Error_Handler();
  }

  /* USER CODE BEGIN Check_RTC_BKUP */

  /* USER CODE END Check_RTC_BKUP */

  /** Initialize RTC and set the Time and Date
  */
  sTime.Hours = 0xA;
  sTime.Minutes = 0x0;
  sTime.Seconds = 0x0;
  sTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
  sTime.StoreOperation = RTC_STOREOPERATION_RESET;
  if (HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BCD) != HAL_OK)
  {
 Error_Handler();
  }
  sDate.WeekDay = RTC_WEEKDAY_MONDAY;
  sDate.Month = RTC_MONTH_MAY;
  sDate.Date = 0x10;
  sDate.Year = 0x24;

  if (HAL_RTC_SetDate(&hrtc, &sDate, RTC_FORMAT_BCD) != HAL_OK)
  {
 Error_Handler();
  }
  /* USER CODE BEGIN RTC_Init 2 */

  /* USER CODE END RTC_Init 2 */

}

/**
  * @brief TIM1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM1_Init(void)
{

  /* USER CODE BEGIN TIM1_Init 0 */

  /* USER CODE END TIM1_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM1_Init 1 */
```

```
  /* USER CODE END TIM1_Init 1 */
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 31;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 999;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
 Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
  {
 Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
  {
 Error_Handler();
  }
  /* USER CODE BEGIN TIM1_Init 2 */

  /* USER CODE END TIM1_Init 2 */

}

/**
  * @brief TIM2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 31;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 999;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
```

```
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
 Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
 Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
 Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */

}

/**
  * @brief TIM3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 31;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 99;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
 Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
```

```
 Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
 Error_Handler();
  }
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */

}

/**
  * @brief TIM4 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM4_Init(void)
{

  /* USER CODE BEGIN TIM4_Init 0 */

  /* USER CODE END TIM4_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM4_Init 1 */

  /* USER CODE END TIM4_Init 1 */
  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 31999;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 999;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
 Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
 Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
  {
 Error_Handler();
  }
```

```
  /* USER CODE BEGIN TIM4_Init 2 */

  /* USER CODE END TIM4_Init 2 */

}

/**
  * @brief TIM5 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM5_Init(void)
{

  /* USER CODE BEGIN TIM5_Init 0 */

  /* USER CODE END TIM5_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};

  /* USER CODE BEGIN TIM5_Init 1 */

  /* USER CODE END TIM5_Init 1 */
  htim5.Instance = TIM5;
  htim5.Init.Prescaler = 0;
  htim5.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim5.Init.Period = 1599;
  htim5.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim5.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim5) != HAL_OK)
  {
 Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim5, &sClockSourceConfig) != HAL_OK)
  {
 Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim5) != HAL_OK)
  {
 Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim5, &sMasterConfig) != HAL_OK)
  {
 Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM2;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_LOW;
```

```
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim5, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
 Error_Handler();
  }
  __HAL_TIM_DISABLE_OCxPRELOAD(&htim5, TIM_CHANNEL_1);
  /* USER CODE BEGIN TIM5_Init 2 */

  /* USER CODE END TIM5_Init 2 */
  HAL_TIM_MspPostInit(&htim5);

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_9B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_ODD;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
 Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
```

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOH_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOB, D4_Pin|D5_Pin|LED1_Pin|D6_Pin
      |RS_Pin|D7_Pin|E_Pin|LED2_Pin
      |LED3_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(LED4_GPIO_Port, LED4_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin : B1_Pin */
GPIO_InitStruct.Pin = B1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pins : B_Button_Pin M_Button_Pin */
GPIO_InitStruct.Pin = B_Button_Pin|M_Button_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING_FALLING;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/*Configure GPIO pins : D4_Pin D5_Pin LED1_Pin D6_Pin
       RS_Pin D7_Pin E_Pin LED2_Pin
       LED3_Pin */
GPIO_InitStruct.Pin = D4_Pin|D5_Pin|LED1_Pin|D6_Pin
      |RS_Pin|D7_Pin|E_Pin|LED2_Pin
      |LED3_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

/*Configure GPIO pin : LED4_Pin */
GPIO_InitStruct.Pin = LED4_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(LED4_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : Digital_temp_Pin */
GPIO_InitStruct.Pin = Digital_temp_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(Digital_temp_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pins : R_Button_Pin L_Button_Pin T_Button_Pin */
```

```
  GPIO_InitStruct.Pin = R_Button_Pin|L_Button_Pin|T_Button_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* Callback function for timer period elapsed interrupt */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
 /**
  * Static counters for LED blinking, display updates, and tick counter
  */
 static uint32_t blinkCounters[sizeof(buttonLedPairs) / sizeof(buttonLedPairs[0])] = {0};
 static uint32_t displayCounters = 0;
 static uint32_t tickCounter = 0;

 /**
  * Handle calibration process if TIM1 interrupt is triggered and calibration is enabled
  */
 if (htim->Instance == TIM1 && calibration) {
  ProcessCalibration();
 }

 /**
  * Handle LED blinking if TIM2 interrupt is triggered
  */
 if (htim->Instance == TIM2) {
  // Iterate through each button-LED pair
  for (int i = 0; i < sizeof(buttonLedPairs) / sizeof(buttonLedPairs[0]); i++) {
   if (buttonLedPairs[i].State == 1) {
    // Increment blink counter for each LED
    blinkCounters[i]++;
    if (blinkCounters[i] >= buttonLedPairs[i].blinkDuration) {
     // Toggle LED state and reset blink counter
     HAL_GPIO_TogglePin(buttonLedPairs[i].ledPort, buttonLedPairs[i].ledPin);
     blinkCounters[i] = 0;
    }
   } else {
    // Reset blink counter if LED is not enabled
    blinkCounters[i] = 0;
   }
  }
 }

 /**
  * Handle PWM duty cycle updates and solar panel data measurement if TIM3 interrupt is trig
  */
 if (htim->Instance == TIM3 && P_measuring) {
  // Increment tick counter
  tickCounter++;
```

```
  // Check if it's time to increment the duty cycle
  if (tickCounter >= 140) {
   tickCounter = 0;  // Reset the tick counter
   if (currentDutyCycle < 100) {
    currentDutyCycle++;  // Increment the duty cycle
   } else {
    currentDutyCycle = 0;  // Reset the duty cycle after reaching 100%
   }
   SetPWM_DutyCycle(currentDutyCycle);  // Update the PWM duty cycle
  }

  // Measure solar panel data and update display if enabled
  if (P_measuring && displayMode == 1) {
   ReadSolarPanelData();
   displayCounters++;
   if (displayCounters >= 2000) {
    Eff = CalculateEfficiency();
    lcdLiveMeasuring(Vpv, Ipv, Ppv, Eff);
    displayCounters = 0;
   }
  }
 }

 /**
  * Handle RTC time and date updates if TIM4 interrupt is triggered
  */
 if (htim->Instance == TIM4) {
  if (displayMode == 3) {
   RTC_TimeTypeDef sTime;
   RTC_DateTypeDef sDate;

   // Get the current time and date from the RTC
   HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
   HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);

   // Update the display
   lcd16x2_1stLine();
   lcd16x2_printf("%02d/%02d/%04d          ", sDate.Date, sDate.Month, 2000 + sDate.Year);
   lcd16x2_2ndLine();
   lcd16x2_printf("%02d:%02d:%02d        ", sTime.Hours, sTime.Minutes, sTime.Seconds);
  }
  ControlLED4BasedOnEfficiency(Eff);
 }
}

/* Callback function for GPIO external interrupt */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
 static uint32_t lastTimeButtonPressed = 0;  // Timestamp of the last button press
 uint32_t currentTime;

 if (GPIO_Pin == GPIO_PIN_7) {  // Middle button (RTC setting)
  currentTime = HAL_GetTick();
  if ((currentTime - lastTimeButtonPressed) > 20) {  // Debounce check
```

```
  lastTimeButtonPressed = currentTime;
  if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_7) == GPIO_PIN_SET) {  // Active low check
   if (!settingTime) {
    settingTime = 1;
    rtcState = STATE_SET_DAY;  // Start RTC setting mode
   } else {
    // Cycle through RTC setting states
    switch (rtcState) {
     case STATE_SET_DAY:
      rtcState = STATE_SET_MONTH;
      break;
     case STATE_SET_MONTH:
      rtcState = STATE_SET_YEAR;
      break;
     case STATE_SET_YEAR:
      rtcState = STATE_SET_HOURS;
      break;
     case STATE_SET_HOURS:
      rtcState = STATE_SET_MINUTES;
      break;
     case STATE_SET_MINUTES:
      rtcState = STATE_SET_SECONDS;
      break;
     case STATE_SET_SECONDS:
      rtcState = STATE_COMPLETED;
      completeRTCSetting();  // Apply the RTC settings
      settingTime = 0;  // Exit setting mode
      rtcState = STATE_IDLE;  // Reset state to idle
      break;
     default:
      break;
    }
   }
   updateRTCDisplay(rtcState);  // Update the display with the new state
  }
 }
}

if (GPIO_Pin == GPIO_PIN_8) {  // Left button (Display mode switch)
 currentTime = HAL_GetTick();
 if ((currentTime - lastTimeButtonPressed) > 150) {  // Debounce check
  lastTimeButtonPressed = currentTime;
  if (HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_8) == GPIO_PIN_RESET) {  // Active low check
   displayMode++;
   if (displayMode > 4) {
    displayMode = 1;  // Wrap around if mode exceeds 3
   }
   updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);
  }
 }
}

if (GPIO_Pin == GPIO_PIN_6) {  // Right button (Calibration)
```

```
 currentTime = HAL_GetTick();
 if ((currentTime - lastTimeButtonPressed) > 100) {  // Debounce check
  lastTimeButtonPressed = currentTime;
  if (HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_6) == GPIO_PIN_RESET) {  // Active low check
   CA_Command("&_CA_*\n");  // Start calibration
  }
 }
}

if (GPIO_Pin == GPIO_PIN_9) {  // Top button (Environment measurement)
 currentTime = HAL_GetTick();
 if ((currentTime - lastTimeButtonPressed) > 75) {  // Debounce check
  lastTimeButtonPressed = currentTime;
  if (HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_9) == GPIO_PIN_RESET) {  // Active low check
   if (settingTime) {
    incrementRTCSetting(rtcState);  // Increment RTC setting
    updateRTCDisplay(rtcState);  // Update the display
   } else {
    E_measuring = !E_measuring;
    displayMode = 2;  // Switch to environment display mode
    updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);
    toggleLedBlinking(&buttonLedPairs[0]);  // Toggle LED for environment measurement
    if (!E_measuring) {
     lightIntensity = LightSensing();
     Amb_temp = AnalogTemperatureSensing();
     Pann_temp = DigitalTemperatureSensing();
     EnvironmentConditions(Amb_temp, Pann_temp, lightIntensity);
     updateDisplay(2, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);
    }
   }
  }
 }
}

if (GPIO_Pin == GPIO_PIN_5) {  // Bottom button (Power measurement)
 currentTime = HAL_GetTick();
 if ((currentTime - lastTimeButtonPressed) > 25) {  // Debounce check
  lastTimeButtonPressed = currentTime;
  if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_5) == GPIO_PIN_RESET) {  // Active low check
   if (settingTime) {
    decrementRTCSetting(rtcState);  // Decrement RTC setting
    updateRTCDisplay(rtcState);  // Update the display
   } else {
    P_measuring = !P_measuring;
    if (P_measuring) {
     maxPower = 0;
     displayMode = 1;  // Switch to power display mode
      toggleLedBlinking(&buttonLedPairs[1]);  // Toggle LED for power measurement
     HAL_TIM_Base_Start_IT(&htim3);  // Start Timer 3
    } else {
     HAL_TIM_Base_Stop_IT(&htim3);  // Stop Timer 3
     Eff = CalculateEfficiency();  // Calculate efficiency
     updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff)
```

```
      PowerConditions(Vmpp, Impp, Pmpp, Eff);  // Update power conditions
      SetPWM_DutyCycle(0);  // Reset PWM duty cycle
      toggleLedBlinking(&buttonLedPairs[1]);  // Toggle LED for power measurement
     }

    }
   }
  }
 }

 if (GPIO_Pin == Digital_temp_Pin) {  // Digital temperature sensor pulse counting
  uint32_t current_time = HAL_GetTick();
  if ((current_time - last_pulse_train_start) > 50) {
   last_pulse_train_start = current_time;
   if (E_measuring) {
    total_pulse_count += pulse_count;
    pulse_train_count++;
   }
   pulse_count = 0;
  }
  pulse_count++;
 }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
 UNUSED(huart);

 // Append received character to the buffer and ensure null-termination
 if (bufferIndex < UART_BUFFER_SIZE - 1) {
  uartBuffer[bufferIndex++] = rxDataByte[0];
  uartBuffer[bufferIndex] = '\0';  // Null-terminate the string
 }

 // Check if the command is complete (ends with '\n')
 if (rxDataByte[0] == '\n') {

  // Process commands based on the received buffer content
  if (strcmp(uartBuffer, "&_EN_*\n") == 0) {
   EN_Command(uartBuffer);
  } else if (strcmp(uartBuffer, "&_SP_*\n") == 0) {
   SP_Command(uartBuffer);
  } else if (strcmp(uartBuffer, "&_CA_*\n") == 0) {
   CA_Command(uartBuffer);
  }

  // Reset the buffer for the next command
  memset(uartBuffer, 0, UART_BUFFER_SIZE);
  bufferIndex = 0;
 }

 // Re-enable UART receive interrupt for the next byte
 HAL_UART_Receive_IT(&huart2, rxDataByte, 1);
}
```

```
/* Handles the "&_EN_*" UART command to start or stop environment measurement */
void EN_Command(const char *commandBuffer) {
 if (strcmp(commandBuffer, "&_EN_*\n") == 0) {  // Check for the "&_EN_*" command
  E_measuring = !E_measuring;  // Toggle environment measuring state
  displayMode = 2;  // Set display mode to environment mode
  updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);  /

  toggleLedBlinking(&buttonLedPairs[0]);  // Toggle the LED for environment measurement

  if (!E_measuring) {  // If stopping the measurement
   lightIntensity = LightSensing();
   Amb_temp = AnalogTemperatureSensing();
   Pann_temp = DigitalTemperatureSensing();
   EnvironmentConditions(Amb_temp, Pann_temp, lightIntensity);  // Send environment data ove
   updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);
  }
 }
}

/* Handles the "&_SP_*" UART command to start or stop power measurement */
void SP_Command(const char *commandBuffer) {
 if (strcmp(commandBuffer, "&_SP_*\n") == 0) {  // Check for the "&_SP_*" command
  // Toggle the power measuring state
  P_measuring = !P_measuring;
  toggleLedBlinking(&buttonLedPairs[1]);  // Start blinking LED for power measurement
  if (P_measuring) {
   maxPower = 0;
   displayMode = 1;  // Switch to power display mode
   HAL_TIM_Base_Start_IT(&htim3);  // Start Timer 3

  } else {
   HAL_TIM_Base_Stop_IT(&htim3);  // Stop Timer 3
   Eff = CalculateEfficiency();  // Calculate efficiency
   updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);
   PowerConditions(Vmpp, Impp, Pmpp, Eff);  // Update power conditions
   SetPWM_DutyCycle(0);  // Reset PWM duty cycle
   }

 }
}

/* Handles the "&_CA_*" UART command to start the calibration process */
void CA_Command(const char *commandBuffer) {
 if (strcmp(commandBuffer, "&_CA_*\n") == 0) {  // Check for the "&_CA_*" command
  calibration = !calibration;
  if(calibration){
  calibrationContext.state = CALIBRATION_STATE_EN_START;  // Start calibration state
  calibrationContext.timestamp = HAL_GetTick();  // Record the start time
  HAL_TIM_Base_Start_IT(&htim1);  // Start Timer 1
  toggleLedBlinking(&buttonLedPairs[3]);  // Start blinking LED pair 3
  toggleLedBlinking(&buttonLedPairs[0]);  // Start blinking LED pair 0
  ProcessCalibration();  // Start the calibration process
```

```
 }
 }
}

/* Formats and sends power conditions data over UART */
void PowerConditions(int Vmpp, int Impp, int Pmpp, int Eff) {
 char msg[100];

 // Format the message according to the provided example
 sprintf(msg, "&_%04d_%03d_%03d_%03d_*\n", Vmpp, Impp, Pmpp, Eff);

 // Send the message over UART
 HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 100);
}

/* Formats and sends environmental conditions data over UART */
void EnvironmentConditions(float Amb_temp, float Pann_temp, float lightintensity) {
 char msg[100];
 int A_temp_int = (int)Amb_temp;  // Integer part of Ambient temperature
 int P_temp_int = (int)Pann_temp;  // Integer part of Panel temperature
 int lightintensity_int = (int)(lightintensity * 10000);  // Adjust multiplier if needed to

 // Ensure only 3 digits for temperatures
 A_temp_int = A_temp_int % 1000;
 P_temp_int = P_temp_int % 1000;

 // Convert light intensity to a 5-digit string with leading zeros
 char light_str[6];
 sprintf(light_str, "%05d", lightintensity_int);  // Left pads with zeros to make 5 digits

 // Format the message as specified
 sprintf(msg, "&_%03d_%03d_%s_*\n", A_temp_int, P_temp_int, light_str);

 // Send the message over UART
 HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 100);
}

void ReadSolarPanelData(void) {
 float voltage, current, power;

 // Start ADC Sol1 conversion (Channel 1)
 ADC_Select_CH1();
 HAL_ADC_Start(&hadc1);
 HAL_ADC_PollForConversion(&hadc1, 100);
 uint32_t adcValue_C1 = HAL_ADC_GetValue(&hadc1);  // Voltage at R4-R5 divider
 HAL_ADC_Stop(&hadc1);  // Stop ADC after the last measurement

 // Start ADC Sol2 conversion (Channel 4)
 ADC_Select_CH4();
 HAL_ADC_Start(&hadc1);
 HAL_ADC_PollForConversion(&hadc1, 100);
 uint32_t adcValue_C2 = HAL_ADC_GetValue(&hadc1);  // Voltage at R2-R3 divider
 HAL_ADC_Stop(&hadc1);  // Stop ADC after the last measurement
```

```
 // Calculate voltages based on ADC readings
 float voltageAcrossR4_R5 = ((float)adcValue_C1 / 4095.0) * 3000.0;  // in millivolts
 float voltageAcrossR2_R3 = ((float)adcValue_C2 / 4095.0) * 3000.0;  // in millivolts

 float Voltage4 = voltageAcrossR4_R5 * (30.0 / 18.0);  // Adjusted to real voltage
 float Voltage2 = voltageAcrossR2_R3 * (30.0 / 18.0);  // Adjusted to real voltage

 // Calculate voltage, current, and power if Voltage4 is greater than Voltage2
 if (Voltage4 > Voltage2) {
  voltage = Voltage4 - Voltage2;  // Voltage difference across the solar panel
  current = voltage / 32.0;  // Current calculation based on shunt resistor (in milliamperes
  power = Voltage4 * (current / 1000.0);  // Power in milliwatts


  // Record maximum power
  if (maxPower < (int) power) {
   maxPower = (int) power;
   Pmpp = maxPower;
   Vmpp = (int) Voltage4;
   Impp = (int) current;
  }
 }

 // Store the latest readings
 Vpv = (int)(Voltage4);  // Voltage in millivolts
 Ipv = (int)(current);  // Current in milliamperes
 Ppv = (int)(power);  // Power in milliwatts
 Eff = 0;  // Placeholder for efficiency calculation
}


/* Processes the calibration procedure */
void ProcessCalibration(void) {
 switch (calibrationContext.state) {
  case CALIBRATION_STATE_EN_START:
   // Start environmental measurement and update display
   E_measuring = 1;
   displayMode = 2;  // Set display to environmental data mode
   updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp, Impp, Pmpp, Eff);
   calibrationContext.state = CALIBRATION_STATE_EN_WAIT;
   calibrationContext.timestamp = HAL_GetTick();
   break;

  case CALIBRATION_STATE_EN_WAIT:
   // Wait for 3 seconds
   if (HAL_GetTick() - calibrationContext.timestamp >= 3000) {
    calibrationContext.state = CALIBRATION_STATE_EN_STOP;
   }
   break;

  case CALIBRATION_STATE_EN_STOP:
   // Stop environmental measurement, record data, and toggle LEDs
```

```
 E_measuring = 0;
 Lux_calibrated = LightSensing();
 Amb_temp = AnalogTemperatureSensing();
 Pann_temp = DigitalTemperatureSensing();
 EnvironmentConditions(Amb_temp, Pann_temp, Lux_calibrated);
 updateDisplay(displayMode, Amb_temp, Pann_temp, Lux_calibrated, Vmpp, Impp, Pmpp, Eff);
 toggleLedBlinking(&buttonLedPairs[0]);  // Stop blinking LED pair 0
 toggleLedBlinking(&buttonLedPairs[1]);  // Start blinking LED pair 1
 calibrationContext.state = CALIBRATION_STATE_SP_START;
 break;

case CALIBRATION_STATE_SP_START:
 // Start the PWM sweep for solar panel measurements
 P_measuring = 1;
 displayMode = 1;  // Set display to solar panel data mode
 HAL_TIM_Base_Start_IT(&htim3);
 calibrationContext.state = CALIBRATION_STATE_SP_MEASURE;
 calibrationContext.CalibrationCounter = 0;
 calibrationContext.readIntervalCounter = 0;
 break;

case CALIBRATION_STATE_SP_MEASURE:
 // Perform normal SP measurement and assign to calibrated variables
 if (P_measuring && displayMode == 1) {
  ReadSolarPanelData();
  Pmpp_calibrated = Pmpp;
  Vmpp_calibrated = Vmpp;
  Impp_calibrated = Impp;
 }
 calibrationContext.CalibrationCounter++;
 if (calibrationContext.CalibrationCounter >= SP_MEASURE_TIME) {
  calibrationContext.state = CALIBRATION_STATE_COMPLETE;
 }
 break;

case CALIBRATION_STATE_COMPLETE:
 // Complete the calibration process
 HAL_TIM_Base_Stop_IT(&htim1);
 P_measuring = 0;
 toggleLedBlinking(&buttonLedPairs[1]);  // Stop blinking LED pair 1
 toggleLedBlinking(&buttonLedPairs[3]);  // Stop blinking LED pair 3
 calibration = 0;  // End calibration
 SetPWM_DutyCycle(0);  // Reset the duty cycle
 PowerConditions(Vmpp_calibrated, Impp_calibrated, Pmpp_calibrated, Eff);
 displayMode = 1;
 updateDisplay(displayMode, Amb_temp, Pann_temp, lightIntensity, Vmpp_calibrated, Impp_cal
 calibrationContext.CalibrationCounter = 0;  // Reset the counter for future calibrations
 calibrationContext.state = CALIBRATION_STATE_IDLE;
 break;

case CALIBRATION_STATE_IDLE:
default:
 // Do nothing in idle state
```

```
   break;
 }
}

/* Calculates the efficiency of the solar panel */
int CalculateEfficiency() {
 // Constants
 const float T_STC = 25.0f;  // Standard Testing Conditions temperature in °C
 const float beta = -0.004;  // Temperature coefficient for the PV module

 // Measured values
 float Pmeasured = Pmpp;  // Measured power in mW
 float Tpv_panel = Pann_temp;  // Measured PV panel temperature in °C
 float Lux_measured = lightIntensity;  // Measured LUX value


 // Normalize the measured power with respect to temperature
 float PnormT = Pmeasured / (1 + beta * (Tpv_panel - T_STC));  // Power normalization for te

 // Normalize the measured power with respect to LUX
 float Pnormalised = PnormT * (Lux_calibrated / Lux_measured);  // Power normalization for l

 // Calculate efficiency
 float efficiency = (Pnormalised / Pmpp_calibrated) * 100.0f;

 // Ensure efficiency does not exceed 100%
 if (efficiency > 100.0f) {
  efficiency = 100.0f;
 }

 return (int)efficiency;
}

/* Reads and converts ADC value to light intensity */
float LightSensing() {
 uint32_t adcValue = 0;
 float light = 0.0f;

 // Start ADC conversion for Channel 14
 ADC_Select_CH14();
 HAL_ADC_Start(&hadc1);
 if (HAL_ADC_PollForConversion(&hadc1, 1000000) == HAL_OK) {
  adcValue = HAL_ADC_GetValue(&hadc1);  // Get ADC value for channel 14
 }
 HAL_ADC_Stop(&hadc1);  // Stop ADC conversion

 // Convert ADC value to light intensity
 light = (((float)adcValue / 4095.0f) * 33.0);  // Example conversion formula
 return light;
}

/* Reads and converts ADC value to ambient temperature */
float AnalogTemperatureSensing() {
```

```
 uint32_t adcValue = 0;

 // Start ADC conversion for Channel 15
 ADC_Select_CH15();
 HAL_ADC_Start(&hadc1);
 if (HAL_ADC_PollForConversion(&hadc1, 1000000) == HAL_OK) {
  adcValue = HAL_ADC_GetValue(&hadc1);  // Get ADC value for channel 15
 }
 HAL_ADC_Stop(&hadc1);  // Stop ADC conversion

 // Convert ADC value to ambient temperature
 float Amb_temp = (((float)adcValue / 4095.0f) * 330.0f) - 273;
 return Amb_temp;
}

/* Handles digital temperature measurement and processing */
float DigitalTemperatureSensing() {
 float P_temp = 0.0f;
 if (!E_measuring) {  // Check if the measurement has just stopped
  float avg_pulse_per_train = (pulse_train_count > 0) ? (float)total_pulse_count / pulse_tra
  P_temp = ((avg_pulse_per_train * 256) / 4096) - 50;  // Adjust formula as needed

  // Reset measurement variables for the next cycle
  total_pulse_count = 0;
  pulse_train_count = 0;
 }
 return P_temp;  // Return the computed temperature
}

/* Decrements the RTC setting based on the current state */
void decrementRTCSetting(RTC_SetState state) {
 switch (state) {
  case STATE_SET_DAY:
   rtcDate.Date = (rtcDate.Date == 1) ? 31 : rtcDate.Date - 1;  // Decrement day, wrap to 31
   break;
  case STATE_SET_MONTH:
   rtcDate.Month = (rtcDate.Month == 1) ? 12 : rtcDate.Month - 1;  // Decrement month, wrap
   break;
  case STATE_SET_YEAR:
   rtcDate.Year = (rtcDate.Year == 0) ? 99 : rtcDate.Year - 1;  // Decrement year, wrap to 9
   break;
  case STATE_SET_HOURS:
   rtcTime.Hours = (rtcTime.Hours == 0) ? 23 : rtcTime.Hours - 1;  // Decrement hours, wrap
   break;
  case STATE_SET_MINUTES:
   rtcTime.Minutes = (rtcTime.Minutes == 0) ? 59 : rtcTime.Minutes - 1;  // Decrement minute
   break;
  case STATE_SET_SECONDS:
   rtcTime.Seconds = (rtcTime.Seconds == 0) ? 59 : rtcTime.Seconds - 1;  // Decrement second
   break;
  default:
   break;
 }
```

```
}

/* Increments the RTC setting based on the current state */
void incrementRTCSetting(RTC_SetState state) {
 switch (state) {
  case STATE_SET_DAY:
   rtcDate.Date = (rtcDate.Date % 31) + 1;  // Increment day, wrap to 1 if at 31
   break;
  case STATE_SET_MONTH:
   rtcDate.Month = (rtcDate.Month % 12) + 1;  // Increment month, wrap to 1 if at 12
   break;
  case STATE_SET_YEAR:
   rtcDate.Year = (rtcDate.Year + 1) % 100;  // Increment year, wrap to 0 if at 99
   break;
  case STATE_SET_HOURS:
   rtcTime.Hours = (rtcTime.Hours + 1) % 24;  // Increment hours, wrap to 0 if at 23
   break;
  case STATE_SET_MINUTES:
   rtcTime.Minutes = (rtcTime.Minutes + 1) % 60;  // Increment minutes, wrap to 0 if at 59
   break;
  case STATE_SET_SECONDS:
   rtcTime.Seconds = (rtcTime.Seconds + 1) % 60;  // Increment seconds, wrap to 0 if at 59
   break;
  default:
   break;
 }
}

/* Updates the RTC display based on the current setting state */
void updateRTCDisplay(RTC_SetState state) {
 // Print the current date and time
 lcd16x2_setCursor(0, 0);
 lcd16x2_printf("%02d/%02d/%04d         ", rtcDate.Date, rtcDate.Month, 2000 + rtcDate.Year);
 lcd16x2_setCursor(1, 0);
 lcd16x2_printf("%02d:%02d:%02d         ", rtcTime.Hours, rtcTime.Minutes, rtcTime.Seconds);

 // Position and blink cursor based on the current setting state
 switch (state) {
  case STATE_SET_DAY:
   lcd16x2_enable_cursor_blink(0, 1);  // Enable blinking cursor at day
   break;
  case STATE_SET_MONTH:
   lcd16x2_enable_cursor_blink(0, 4);  // Enable blinking cursor at month
   break;
  case STATE_SET_YEAR:
   lcd16x2_enable_cursor_blink(0, 9);  // Enable blinking cursor at year
   break;
  case STATE_SET_HOURS:
   lcd16x2_enable_cursor_blink(1, 1);  // Enable blinking cursor at hours
   break;
  case STATE_SET_MINUTES:
   lcd16x2_enable_cursor_blink(1, 4);  // Enable blinking cursor at minutes
   break;
```

```
  case STATE_SET_SECONDS:
   lcd16x2_enable_cursor_blink(1, 7);  // Enable blinking cursor at seconds
   break;
  case STATE_COMPLETED:
   lcd16x2_cursorShow(false);  // Turn off cursor blinking
  case STATE_IDLE:
   // Disable cursor blinking when not setting
   lcd16x2_cursorShow(false);  // This assumes lcd16x2_cursorShow turns off the cursor entir
   break;
 }
}

/* Completes the RTC setting process and applies changes */
void completeRTCSetting() {
 // Set time and date on the hardware RTC
 HAL_RTC_SetTime(&hrtc, &rtcTime, RTC_FORMAT_BIN);
 HAL_RTC_SetDate(&hrtc, &rtcDate, RTC_FORMAT_BIN);
 displayMode = 3;  // Set display mode to show RTC data
 HAL_TIM_Base_Start_IT(&htim4);  // Start Timer 4 for periodic RTC updates
}

void setup(void) {
 lcd16x2_init_4bits(RS_GPIO_Port, RS_Pin, E_Pin,
  // D0_GPIO_Port, D0_Pin, D1_Pin, D2_Pin, D3_Pin,
  D4_GPIO_Port, D4_Pin, D5_Pin, D6_Pin, D7_Pin);
}

/* Updates the LCD display based on the current display mode */
void updateDisplay(uint32_t displayMode, float Amb_temp, float Pann_temp, float lightIntensi
 // Get the current time in milliseconds
 uint32_t currentTime = HAL_GetTick();

 // Check if the display mode is 4 (cycling through modes 1 to 3)
 if (displayMode == 4) {
  // Check if it's time to switch to the next sub-mode
  if (currentTime - lastUpdateTime >= DISPLAY_DURATION) {
   // Update the last update time
   lastUpdateTime = currentTime;

   // Switch to the next sub-mode
   currentSubMode++;
   if (currentSubMode > 3) {
    currentSubMode = 1;  // Wrap around to sub-mode 1
   }
  }

  // Set the display mode to the current sub-mode
  displayMode = currentSubMode;
 }

 // Update the display based on the current display mode
 if (displayMode == 1) {
  // Display voltage, current, power, and efficiency
```

```
  lcd16x2_1stLine();  // Clear the first line of the display
  lcd16x2_printf("V:%4dmV I:%3dmA", Vmpp, Impp);  // Display voltage and current

  lcd16x2_2ndLine();  // Move to the second line
  lcd16x2_printf("P: %3dmW E: %3d%%", Pmpp, Eff);  // Display power and efficiency

 } else if (displayMode == 2) {
  // Display ambient and panel temperatures, and light intensity
  int Ta = (int)Amb_temp;  // Integer part of Ambient temperature
  int Tp = (int)Pann_temp;  // Integer part of Panel temperature
  int lux = (int)(lightIntensity * 10000);  // Convert light intensity to integer

  lcd16x2_1stLine();  // Clear the display
  lcd16x2_printf("AMB:%3dC SP:%3dC", Ta, Tp);  // Display temperatures

  lcd16x2_2ndLine();  // Move to the second line
  lcd16x2_printf("LUX:%5d         ", lux);  // Display light sensor value

 } else if (displayMode == 3) {
  // Display current date and time
  RTC_TimeTypeDef sTime;
  RTC_DateTypeDef sDate;

  // Get the current time and date from the RTC
  HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
  HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);

  // Display date in DD-MM-YYYY format
  lcd16x2_1stLine();
  lcd16x2_printf("%02d/%02d/%04d          ", sDate.Date, sDate.Month, 2000 + sDate.Year);

  // Display time in HH:MM:SS format
  lcd16x2_2ndLine();
  lcd16x2_printf("%02d:%02d:%02d          ", sTime.Hours, sTime.Minutes, sTime.Seconds);
 }
}

/* Updates the LCD with live measuring data */
void lcdLiveMeasuring(int Vpv, int Ipv, int Ppv, int Eff) {
 // Clear the first line of the display
 lcd16x2_1stLine();
 lcd16x2_printf("V:%4dmV I:%3dmA", Vpv, Ipv);  // Display voltage and current

 // Move to the second line
 lcd16x2_2ndLine();
 lcd16x2_printf("P: %3dmW E: %3d%%", Ppv, Eff);  // Display power and efficiency
}


void toggleLedBlinking(ButtonLedPair *pair) {
 // Calculate the index of the pair for specific behavior if needed
 int index = pair - buttonLedPairs; // This assumes buttonLedPairs is an array
```

```
 // Toggle the LED state
 pair->State = !(pair->State);

 // Depending on the pair, handle the LED behavior
 switch (index) {
  case 0:
   // For pair 0, manage the timer and direct control of LED
   if (pair->State == 1) {
    // Start the timer to begin blinking
    HAL_TIM_Base_Start_IT(&htim2);
   } else {
    // Stop the timer and turn off the LED
    HAL_TIM_Base_Stop_IT(&htim2);
    HAL_GPIO_WritePin(pair->ledPort, pair->ledPin, GPIO_PIN_SET); // Ensure LED is off when
   }
   break;
  case 1:
   // For pair 1, let's assume we also want to manage blinking with the timer
   if (pair->State == 1) {
    HAL_TIM_Base_Start_IT(&htim2); // Maybe use a different timer or setting if needed
   } else {
    HAL_TIM_Base_Stop_IT(&htim2);
    HAL_GPIO_WritePin(pair->ledPort, pair->ledPin, GPIO_PIN_SET); // Ensure LED is off when
   }
   break;
  case 2:
   // For pair 1, let's assume we also want to manage blinking with the timer
   if (pair->State == 1) {
    HAL_TIM_Base_Start_IT(&htim2); // Maybe use a different timer or setting if needed
   } else {
    HAL_TIM_Base_Stop_IT(&htim2);
    HAL_GPIO_WritePin(pair->ledPort, pair->ledPin, GPIO_PIN_SET); // Ensure LED is off when
   }
   break;
  case 3:
   // For pair 1, let's assume we also want to manage blinking with the timer
   if (pair->State == 1) {
    HAL_TIM_Base_Start_IT(&htim2); // Maybe use a different timer or setting if needed
   } else {
    HAL_TIM_Base_Stop_IT(&htim2);
    HAL_GPIO_WritePin(pair->ledPort, pair->ledPin, GPIO_PIN_SET); // Ensure LED is off when
   }
   break;
  default:
   // For all other pairs, just toggle the state of the LED without using a timer
   if (pair->State == 1) {
    HAL_GPIO_WritePin(pair->ledPort, pair->ledPin, GPIO_PIN_SET); // Turn LED on
   } else {
    HAL_GPIO_WritePin(pair->ledPort, pair->ledPin, GPIO_PIN_RESET); // Turn LED off
   }
   break;
 }
}
```

```
/* Controls LED4 based on the efficiency value */
void ControlLED4BasedOnEfficiency(int Eff) {
 static uint32_t lastBlinkTime = 0;  // Last time the LED was toggled
 uint32_t currentTime = HAL_GetTick();
 const uint32_t blinkPeriod = 500;  // Blink period in milliseconds

 if (Eff < 80) {
  // Blink LED4 if efficiency is below 80%
  if (currentTime - lastBlinkTime >= blinkPeriod) {
   HAL_GPIO_TogglePin(LED4_GPIO_Port, LED4_Pin);
   lastBlinkTime = currentTime;
  }
 } else {
  // Ensure LED4 is on if efficiency is 80% or above
  HAL_GPIO_WritePin(LED4_GPIO_Port, LED4_Pin, GPIO_PIN_SET);
 }
}

/* Sets the PWM duty cycle */
void SetPWM_DutyCycle(uint32_t dutyCyclePercentage) {
 if (dutyCyclePercentage > 100) {
  dutyCyclePercentage = 100;  // Cap the duty cycle at 100%
 }

 // Calculate the new compare value based on the percentage
 uint32_t pulse = (htim5.Init.Period + 1) * dutyCyclePercentage / 100;

 // Set the new compare value for TIM5 Channel 1
 __HAL_TIM_SET_COMPARE(&htim5, TIM_CHANNEL_1, pulse);
}

/* Selects ADC Channel 1 */
void ADC_Select_CH1(void) {
 ADC_ChannelConfTypeDef sConfig = {0};
 sConfig.Channel = ADC_CHANNEL_1;
 sConfig.Rank = 1;
 sConfig.SamplingTime = ADC_SAMPLETIME_112CYCLES;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) {
  Error_Handler();
 }
}

/* Selects ADC Channel 4 */
void ADC_Select_CH4(void) {
 ADC_ChannelConfTypeDef sConfig = {0};
 sConfig.Channel = ADC_CHANNEL_4;
 sConfig.Rank = 1;
 sConfig.SamplingTime = ADC_SAMPLETIME_112CYCLES;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) {
  Error_Handler();
 }
}
```

```
/* Selects ADC Channel 14 */
void ADC_Select_CH14(void) {
 ADC_ChannelConfTypeDef sConfig = {0};
 sConfig.Channel = ADC_CHANNEL_14;
 sConfig.Rank = 1;
 sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) {
  Error_Handler();
 }
}

/* Selects ADC Channel 15 */
void ADC_Select_CH15(void) {
 ADC_ChannelConfTypeDef sConfig = {0};
 sConfig.Channel = ADC_CHANNEL_15;
 sConfig.Rank = 1;
 sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) {
  Error_Handler();
 }
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
  ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
```

```
 #endif /* USE_FULL_ASSERT */
```

B Complete `lcd.c`

```
/*
* lcd16x2.c
*
*  Created on: Apr 23, 2024
*      Author: hauke
*/
  /*

  #include "lcd16x2.h"
  #include <string.h>
  #include <stdio.h>
  #include <stdarg.h>

  //Milisecond function
  #define LCD_MS_DELAY(X) (HAL_Delay(X))

  /* List of COMMANDS */
  #define LCD_CLEARDISPLAY      0x01
  #define LCD_RETURNHOME        0x02
  #define LCD_ENTRYMODESET      0x04
  #define LCD_DISPLAYCONTROL    0x08
  #define LCD_CURSORSHIFT       0x10
  #define LCD_FUNCTIONSET       0x20
  #define LCD_SETCGRAMADDR      0x40
  #define LCD_SETDDRAMADDR      0x80

  /* List of commands Bitfields */
  //1) Entry mode Bitfields
  #define LCD_ENTRY_SH          0x01
  #define LCD_ENTRY_ID          0x02
  //2) Entry mode Bitfields
  #define LCD_ENTRY_SH          0x01
  #define LCD_ENTRY_ID          0x02
  //3) Display control
  #define LCD_DISPLAY_B         0x01
  #define LCD_DISPLAY_C         0x02
  #define LCD_DISPLAY_D         0x04
  //4) Shift control
  #define LCD_SHIFT_RL          0x04
  #define LCD_SHIFT_SC          0x08
  //5) Function set control
  #define LCD_FUNCTION_F        0x04
  #define LCD_FUNCTION_N        0x08
  #define LCD_FUNCTION_DL       0x10

  /* LCD Library Variables */
  static bool is8BitsMode = true;
```

```
  static GPIO_TypeDef* PORT_RS_and_E;               // RS and E PORT
  static uint16_t PIN_RS, PIN_E;                    // RS and E pins
  static GPIO_TypeDef* PORT_LSB;                    // LSBs D0, D1, D2 and D3 PORT
  static uint16_t D0_PIN, D1_PIN, D2_PIN, D3_PIN;   // LSBs D0, D1, D2 and D3 pins
  static GPIO_TypeDef* PORT_MSB;                    // MSBs D5, D6, D7 and D8 PORT
  static uint16_t D4_PIN, D5_PIN, D6_PIN, D7_PIN;   // MSBs D5, D6, D7 and D8 pins
  #define T_CONST   20
  static uint8_t DisplayControl = 0x0F;
  static uint8_t FunctionSet = 0x38;

  /* private functions prototypes */
  /**
* @brief DWT Cortex Tick counter for Microsecond delay
*/
  static uint32_t DWT_Delay_Init(void) {
 /* Disable TRC */
 CoreDebug->DEMCR &= ~CoreDebug_DEMCR_TRCENA_Msk;
 /* Enable TRC */
 CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
 /* Disable clock cycle counter */
 DWT->CTRL &= ~DWT_CTRL_CYCCNTENA_Msk;
 /* Enable clock cycle counter */
 DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
 /* Reset the clock cycle counter value */
 DWT->CYCCNT = 0;
 /* 3 NO OPERATION instructions */
 __NOP();
 __NOP();
 __NOP();
 /* Check if clock cycle counter has started */
 if(DWT->CYCCNT)
 {
   return 0;
 }
 else
 {
   return 1;
 }
  }

  __STATIC_INLINE void DWT_Delay_us(volatile uint32_t usec)
  {
uint32_t clk_cycle_start = DWT->CYCCNT;
usec *= (HAL_RCC_GetHCLKFreq() / 1000000);
while ((DWT->CYCCNT - clk_cycle_start) < usec);
  }

  /**
* @brief Enable Pulse function
*/
  static void lcd16x2_enablePulse(void)
  {
 HAL_GPIO_WritePin(PORT_RS_and_E, PIN_E, GPIO_PIN_SET);
```

```
 DWT_Delay_us(T_CONST);
 HAL_GPIO_WritePin(PORT_RS_and_E, PIN_E, GPIO_PIN_RESET);
 DWT_Delay_us(60);
  }

  /**
* @brief RS control
*/
  static void lcd16x2_rs(bool state)
  {
 HAL_GPIO_WritePin(PORT_RS_and_E, PIN_RS, (GPIO_PinState)state);
  }

  /**
* @brief Write parallel signal to lcd
*/
  static void lcd16x2_write(uint8_t wbyte)
  {
 uint8_t LSB_nibble = wbyte&0xF, MSB_nibble = (wbyte>>4)&0xF;
 if(is8BitsMode)
 {
   //LSB data
   HAL_GPIO_WritePin(PORT_LSB, D0_PIN, (GPIO_PinState)(LSB_nibble&0x1));
   HAL_GPIO_WritePin(PORT_LSB, D1_PIN, (GPIO_PinState)(LSB_nibble&0x2));
   HAL_GPIO_WritePin(PORT_LSB, D2_PIN, (GPIO_PinState)(LSB_nibble&0x4));
   HAL_GPIO_WritePin(PORT_LSB, D3_PIN, (GPIO_PinState)(LSB_nibble&0x8));
   //MSB data
   HAL_GPIO_WritePin(PORT_MSB, D4_PIN, (GPIO_PinState)(MSB_nibble&0x1));
   HAL_GPIO_WritePin(PORT_MSB, D5_PIN, (GPIO_PinState)(MSB_nibble&0x2));
   HAL_GPIO_WritePin(PORT_MSB, D6_PIN, (GPIO_PinState)(MSB_nibble&0x4));
   HAL_GPIO_WritePin(PORT_MSB, D7_PIN, (GPIO_PinState)(MSB_nibble&0x8));
   lcd16x2_enablePulse();
 }
 else
 {
   //MSB data
   HAL_GPIO_WritePin(PORT_MSB, D4_PIN, (GPIO_PinState)(MSB_nibble&0x1));
   HAL_GPIO_WritePin(PORT_MSB, D5_PIN, (GPIO_PinState)(MSB_nibble&0x2));
   HAL_GPIO_WritePin(PORT_MSB, D6_PIN, (GPIO_PinState)(MSB_nibble&0x4));
   HAL_GPIO_WritePin(PORT_MSB, D7_PIN, (GPIO_PinState)(MSB_nibble&0x8));
   lcd16x2_enablePulse();
   //LSB data
   HAL_GPIO_WritePin(PORT_MSB, D4_PIN, (GPIO_PinState)(LSB_nibble&0x1));
   HAL_GPIO_WritePin(PORT_MSB, D5_PIN, (GPIO_PinState)(LSB_nibble&0x2));
   HAL_GPIO_WritePin(PORT_MSB, D6_PIN, (GPIO_PinState)(LSB_nibble&0x4));
   HAL_GPIO_WritePin(PORT_MSB, D7_PIN, (GPIO_PinState)(LSB_nibble&0x8));
   lcd16x2_enablePulse();
 }
  }

  /**
* @brief Write command
*/
```

```
  static void lcd16x2_writeCommand(uint8_t cmd)
  {
 lcd16x2_rs(false);
 lcd16x2_write(cmd);
  }

  /**
* @brief Write data
*/
  static void lcd16x2_writeData(uint8_t data)
  {
 lcd16x2_rs(true);
 lcd16x2_write(data);
  }

  /**
* @brief 4-bits write
*/
  static void lcd16x2_write4(uint8_t nib)
  {
 nib &= 0xF;
 lcd16x2_rs(false);
 //LSB data
 HAL_GPIO_WritePin(PORT_MSB, D4_PIN, (GPIO_PinState)(nib&0x1));
 HAL_GPIO_WritePin(PORT_MSB, D5_PIN, (GPIO_PinState)(nib&0x2));
 HAL_GPIO_WritePin(PORT_MSB, D6_PIN, (GPIO_PinState)(nib&0x4));
 HAL_GPIO_WritePin(PORT_MSB, D7_PIN, (GPIO_PinState)(nib&0x8));
 lcd16x2_enablePulse();
  }

  /* Public functions definitions */

  /**
* @brief Initialise LCD on 8-bits mode
* @param[in] *port_rs_e RS and EN GPIO Port (e.g. GPIOB)
* @param[in] *port_0_3 D0 to D3 GPIO Port
* @param[in] *port_4_7 D4 to D7 GPIO Port
* @param[in] x_pin GPIO pin (e.g. GPIO_PIN_1)
*/
  void lcd16x2_init_8bits(
   GPIO_TypeDef* port_rs_e, uint16_t rs_pin, uint16_t e_pin,
   GPIO_TypeDef* port_0_3, uint16_t d0_pin, uint16_t d1_pin, uint16_t d2_pin, uint16_t d3_pin
   GPIO_TypeDef* port_4_7, uint16_t d4_pin, uint16_t d5_pin, uint16_t d6_pin, uint16_t d7_pin
  {
 DWT_Delay_Init();
 //Set GPIO Ports and Pins data
 PORT_RS_and_E = port_rs_e;
 PIN_RS = rs_pin;
 PIN_E = e_pin;
 PORT_LSB = port_0_3;
 D0_PIN = d0_pin;
 D1_PIN = d1_pin;
 D2_PIN = d2_pin;
```

```
 D3_PIN = d3_pin;
 PORT_MSB = port_4_7;
 D4_PIN = d4_pin;
 D5_PIN = d5_pin;
 D6_PIN = d6_pin;
 D7_PIN = d7_pin;
 is8BitsMode = true;
 FunctionSet = 0x38;

 //Initialise LCD
 //1. Wait at least 15ms
 LCD_MS_DELAY(20);
 //2. Attentions sequence
 lcd16x2_writeCommand(0x30);
 LCD_MS_DELAY(5);
 lcd16x2_writeCommand(0x30);
 LCD_MS_DELAY(1);
 lcd16x2_writeCommand(0x30);
 LCD_MS_DELAY(1);
 //3. Function set; Enable 2 lines, Data length to 8 bits
 lcd16x2_writeCommand(LCD_FUNCTIONSET | LCD_FUNCTION_N | LCD_FUNCTION_DL);
 //4. Display control (Display ON, Cursor ON, blink cursor)
 lcd16x2_writeCommand(LCD_DISPLAYCONTROL | LCD_DISPLAY_B | LCD_DISPLAY_C | LCD_DISPLAY_D);
 //5. Clear LCD and return home
 lcd16x2_writeCommand(LCD_CLEARDISPLAY);
 LCD_MS_DELAY(2);
  }

  /**
* @brief Initialise LCD on 4-bits mode
* @param[in] *port_4_7 D4 to D7 GPIO Port
* @param[in] x_pin GPIO pin (e.g. GPIO_PIN_1)
*/
  void lcd16x2_init_4bits(
   GPIO_TypeDef* port_rs_e, uint16_t rs_pin, uint16_t e_pin,
   GPIO_TypeDef* port_4_7, uint16_t d4_pin, uint16_t d5_pin, uint16_t d6_pin, uint16_t d7_pin
  {
 DWT_Delay_Init();
 //Set GPIO Ports and Pins data
 PORT_RS_and_E = port_rs_e;
 PIN_RS = rs_pin;
 PIN_E = e_pin;
 PORT_MSB = port_4_7;
 D4_PIN = d4_pin;
 D5_PIN = d5_pin;
 D6_PIN = d6_pin;
 D7_PIN = d7_pin;
 is8BitsMode = false;
 FunctionSet = 0x28;

 //Initialise LCD
 //1. Wait at least 15ms
 LCD_MS_DELAY(20);
```

```
 //2. Attentions sequence
 lcd16x2_write4(0x3);
 LCD_MS_DELAY(5);
 lcd16x2_write4(0x3);
 LCD_MS_DELAY(1);
 lcd16x2_write4(0x3);
 LCD_MS_DELAY(1);
 lcd16x2_write4(0x2);  //4 bit mode
 LCD_MS_DELAY(1);
 //4. Function set; Enable 2 lines, Data length to 4 bits
 lcd16x2_writeCommand(LCD_FUNCTIONSET | LCD_FUNCTION_N);
 //3. Display control (Display ON, Cursor ON, blink cursor)
 lcd16x2_writeCommand(LCD_DISPLAYCONTROL | LCD_DISPLAY_B | LCD_DISPLAY_C | LCD_DISPLAY_D);
 //4. Clear LCD and return home
 lcd16x2_writeCommand(LCD_CLEARDISPLAY);
 LCD_MS_DELAY(3);
  }

  /**
* @brief Set cursor position
* @param[in] row - 0 or 1 for line1 or line2
* @param[in] col - 0 - 15 (16 columns LCD)
*/
  void lcd16x2_setCursor(uint8_t row, uint8_t col)
  {
 uint8_t maskData;
 maskData = (col)&0x0F;
 if(row==0)
 {
   maskData |= (0x80);
   lcd16x2_writeCommand(maskData);
 }
 else
 {
   maskData |= (0xc0);
   lcd16x2_writeCommand(maskData);
 }
  }
  /**
* @brief Move to beginning of 1st line
*/
  void lcd16x2_1stLine(void)
  {
 lcd16x2_setCursor(0,0);
  }
  /**
* @brief Move to beginning of 2nd line
*/
  void lcd16x2_2ndLine(void)
  {
 lcd16x2_setCursor(1,0);
  }
```

```
  /**
* @brief Select LCD Number of lines mode
*/
  void lcd16x2_twoLines(void)
  {
 FunctionSet |= (0x08);
 lcd16x2_writeCommand(FunctionSet);
  }
  void lcd16x2_oneLine(void)
  {
 FunctionSet &= ~(0x08);
 lcd16x2_writeCommand(FunctionSet);
  }

  /**
* @brief Cursor ON/OFF
*/
  void lcd16x2_cursorShow(bool state)
  {
 if(state)
 {
   DisplayControl |= (0x03);
   lcd16x2_writeCommand(DisplayControl);
 }
 else
 {
   DisplayControl &= ~(0x03);
   lcd16x2_writeCommand(DisplayControl);
 }
  }

  /**
* @brief Enable cursor blinking at a specific position on the LCD.
* @param row The row where the cursor should blink (0 for first row, 1 for second row)
* @param col The column where the cursor should blink (0-15)
*/
  void lcd16x2_enable_cursor_blink(uint8_t row, uint8_t col) {
   // Set cursor position
   lcd16x2_setCursor(row, col);

   // Modify DisplayControl to enable cursor and blinking
   DisplayControl |= (LCD_DISPLAY_C | LCD_DISPLAY_B);
   lcd16x2_writeCommand(DisplayControl);
  }

  /**
* @brief Display clear
*/
  void lcd16x2_clear(void)
  {
 lcd16x2_writeCommand(LCD_CLEARDISPLAY);
 LCD_MS_DELAY(3);
  }
```

```
  /**
* @brief Display ON/OFF, to hide all characters, but not clear
*/
  void lcd16x2_display(bool state)
  {
 if(state)
 {
   DisplayControl |= (0x04);
   lcd16x2_writeCommand(DisplayControl);
 }
 else
 {
   DisplayControl &= ~(0x04);
   lcd16x2_writeCommand(DisplayControl);
 }
  }

  /**
* @brief Shift content to right
*/
  void lcd16x2_shiftRight(uint8_t offset)
  {
 for(uint8_t i=0; i<offset;i++)
 {
   lcd16x2_writeCommand(0x1c);
 }
  }

  /**
* @brief Shift content to left
*/
  void lcd16x2_shiftLeft(uint8_t offset)
  {
 for(uint8_t i=0; i<offset;i++)
 {
   lcd16x2_writeCommand(0x18);
 }
  }

  /**
* @brief Print to display any datatype (e.g. lcd16x2_printf("Value1 = %.1f", 123.45))
*/
  void lcd16x2_printf(const char* str, ...)
  {
 char stringArray[20];
 va_list args;
 va_start(args, str);
 vsprintf(stringArray, str, args);
 va_end(args);
 for(uint8_t i=0;  i<strlen(stringArray) && i<16; i++)
 {
   lcd16x2_writeData((uint8_t)stringArray[i]);
```

```
}
 }
```

C Software Diagrams

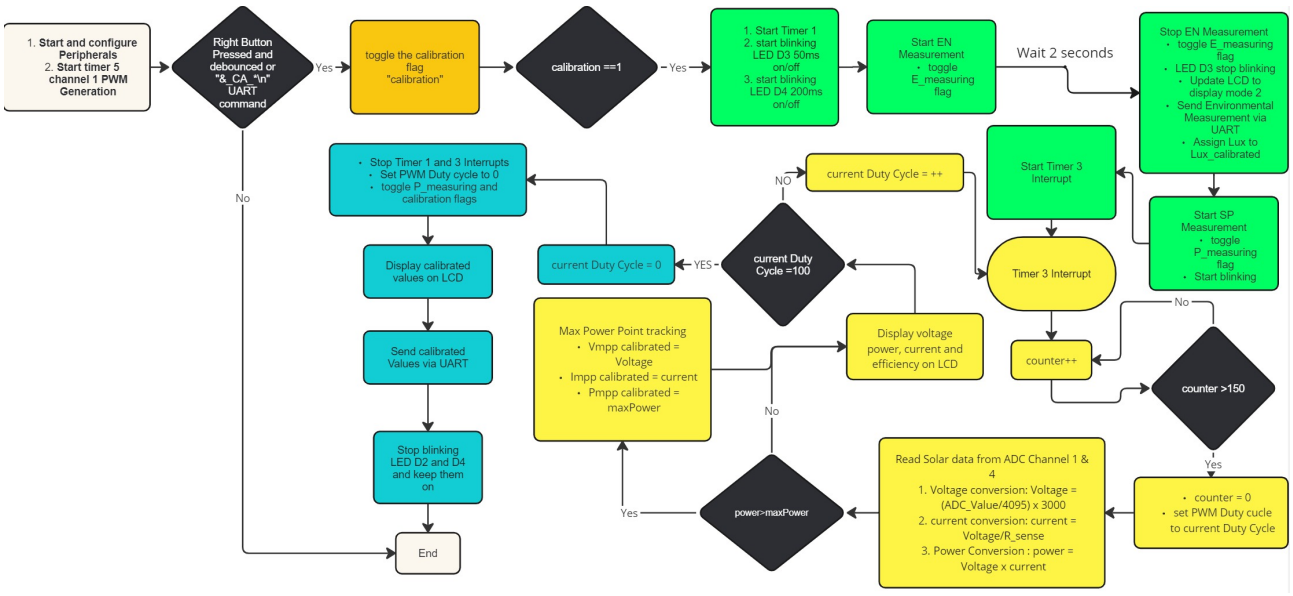

Figure 37: Calibration Process Flowchart

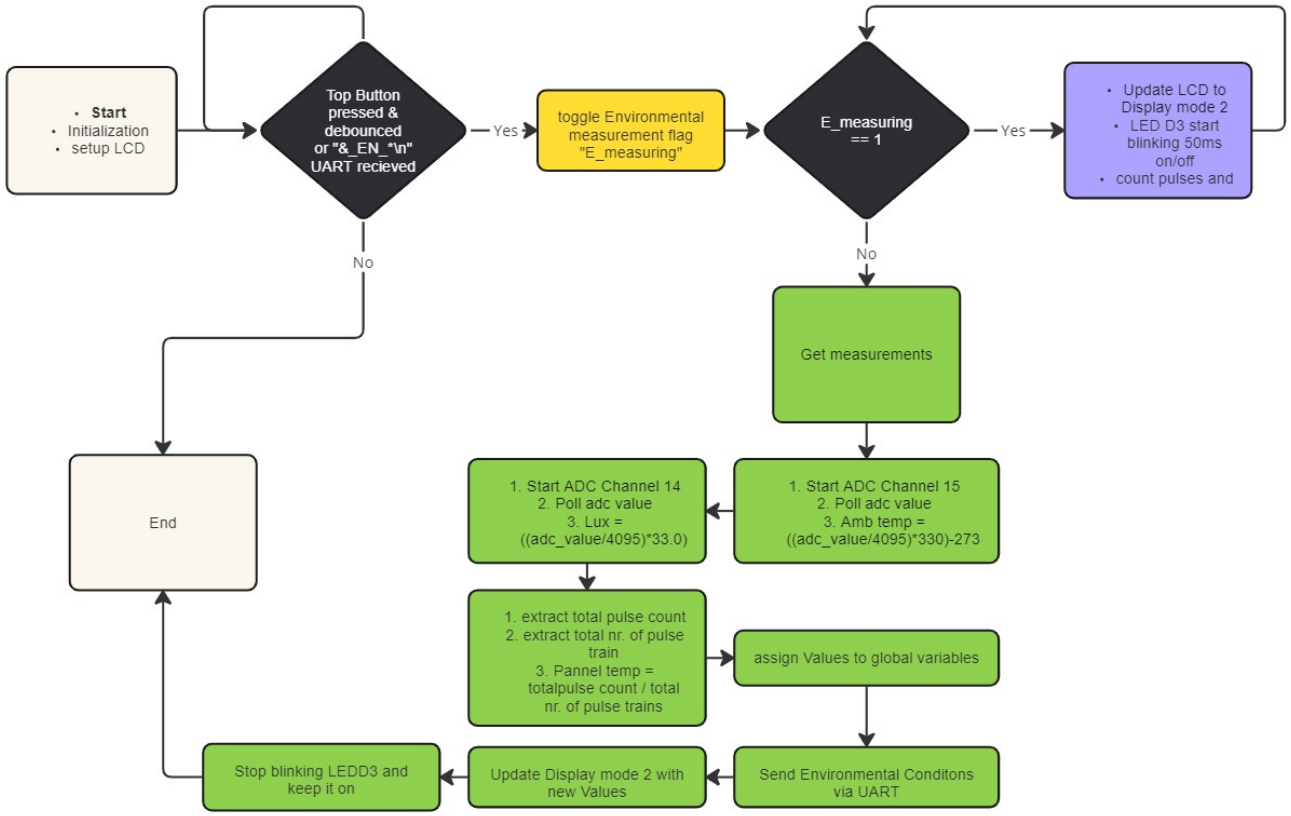

Figure 38: Environmental Condition Measurement Process Flowchart

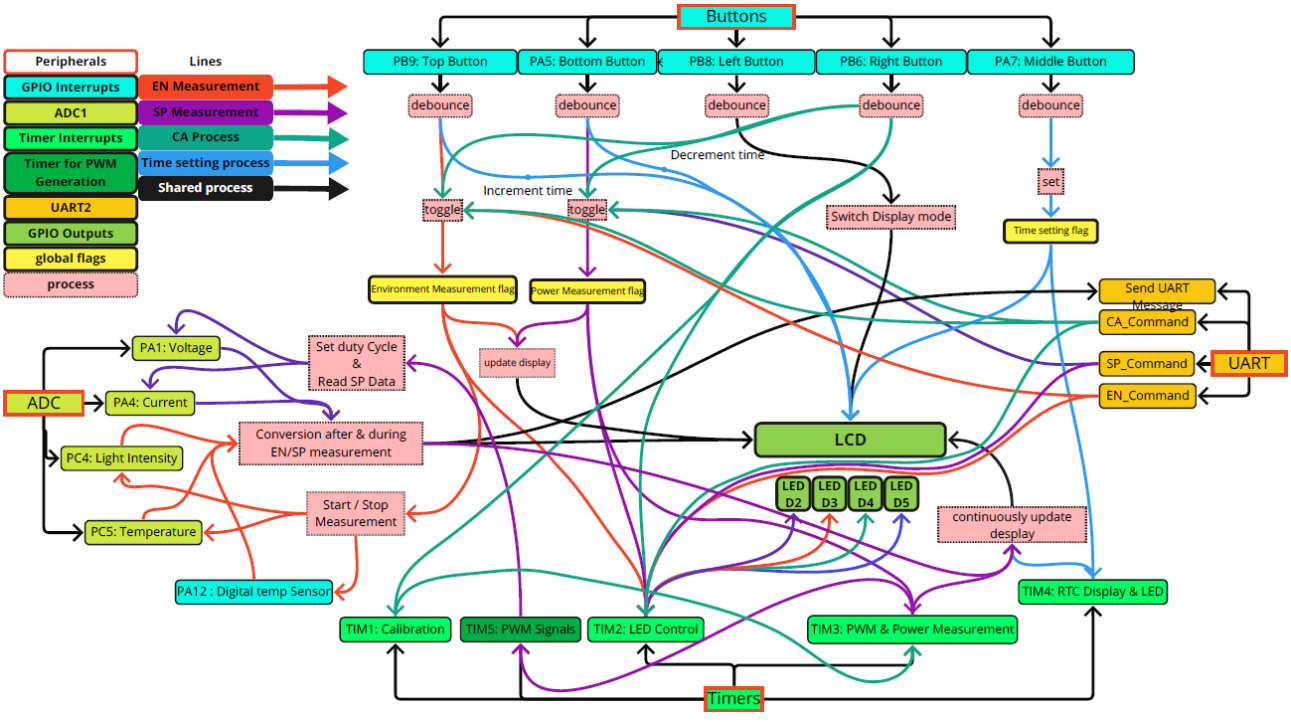

Figure 39: System Interactions and Data Flow Diagram

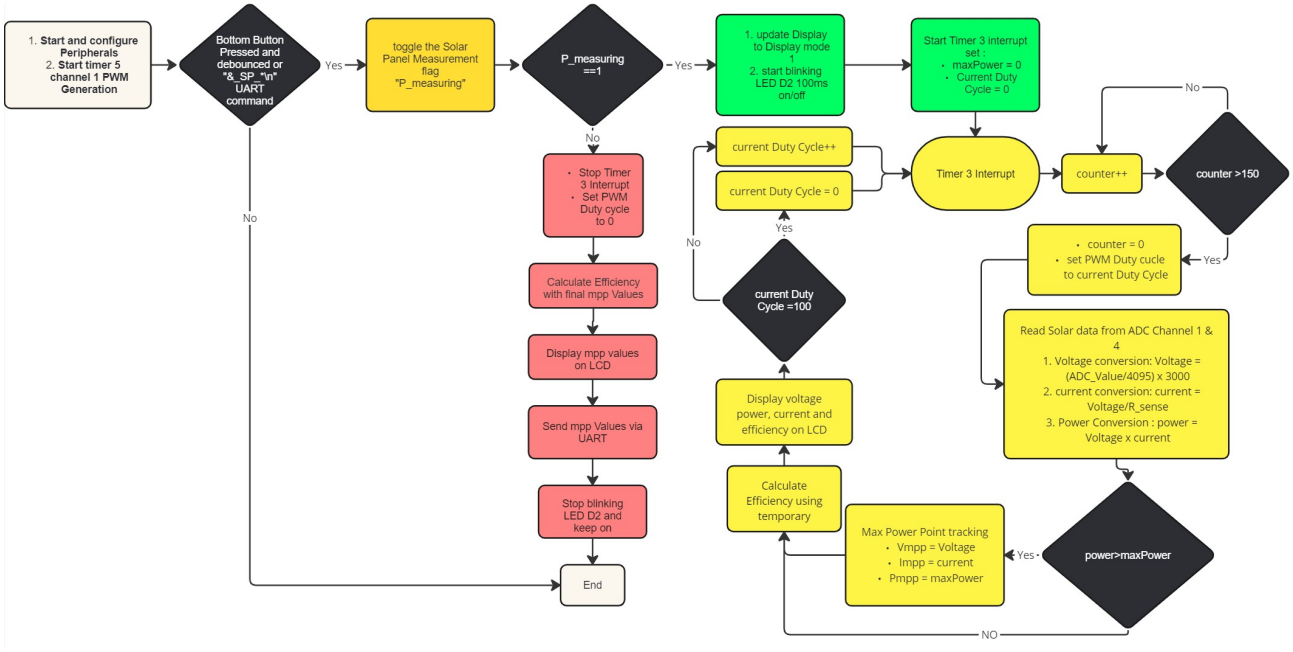

Figure 40: Solar Panel Power Measurement Process Flowchart

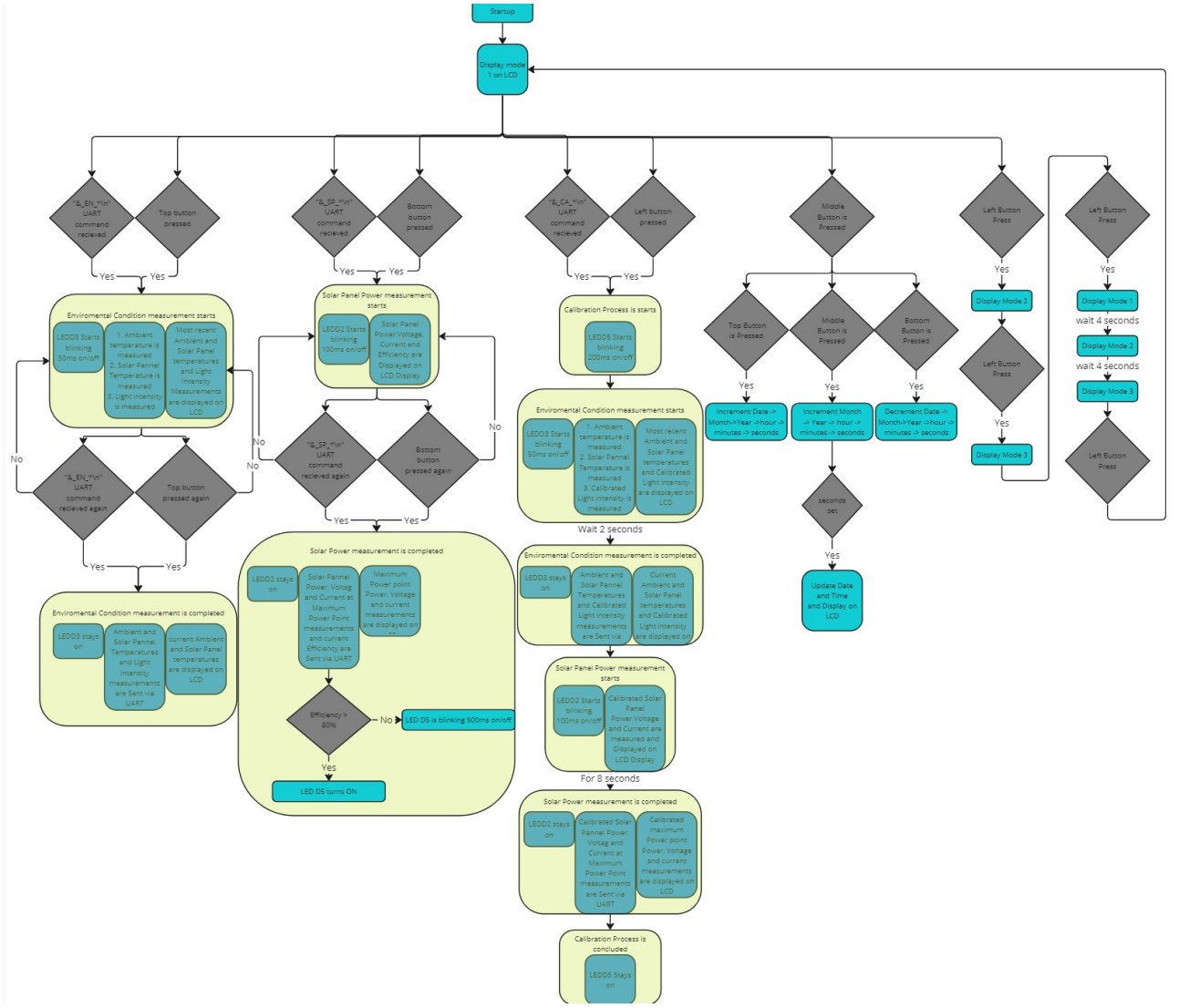

Figure 41: High-Level System Block Diagram

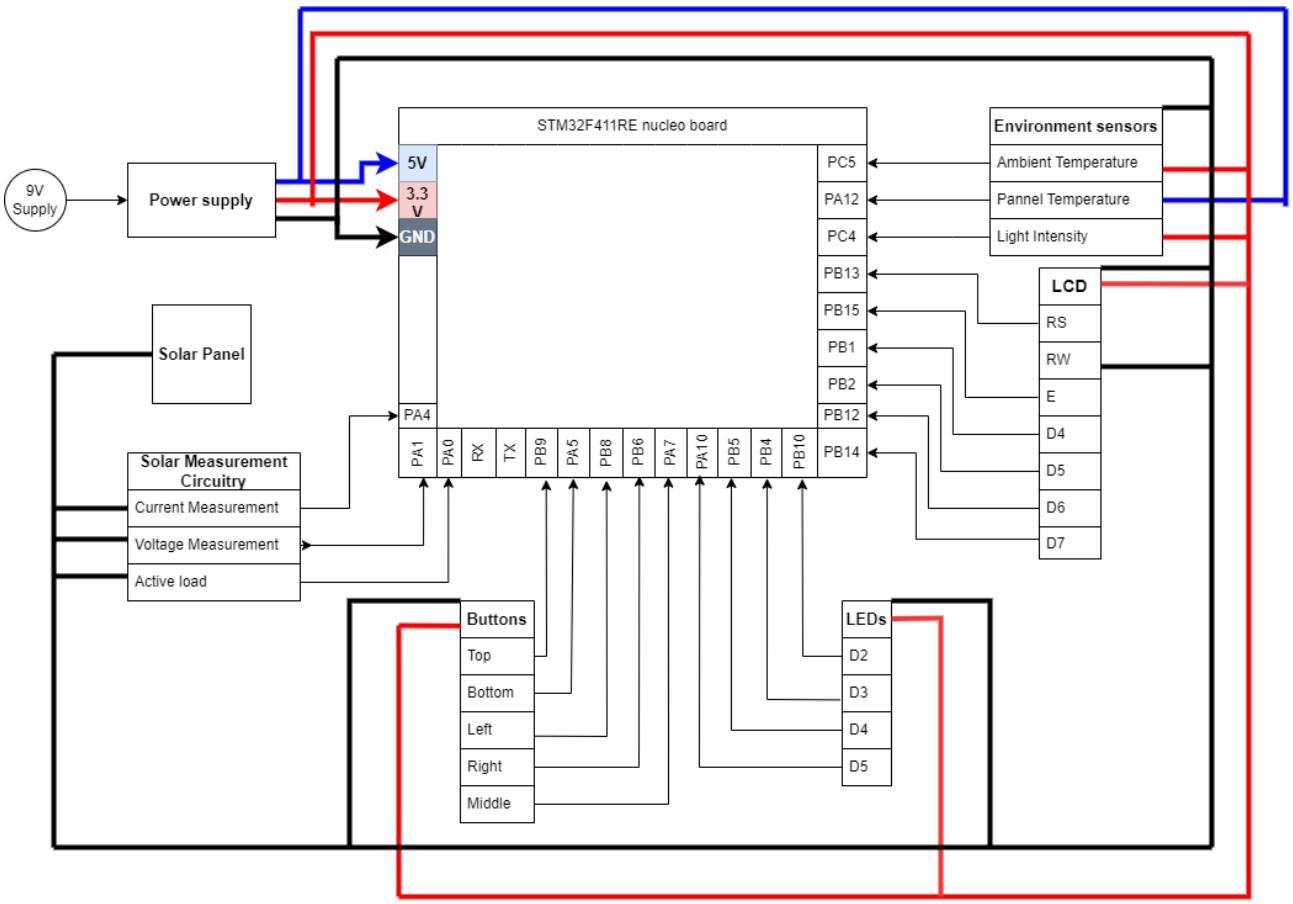

Figure 42: Whole System Hardware Connections

References

- STMicroelectronics, 2024. *DesignE314_2024_PDD_v07B*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Hitachi, 1998. *HD44780 LCD controller data sheet*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Texas Instruments, 2015. *LM235, LM335, LM135 Precision Temperature Sensors*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Texas Instruments, 2017. *LMT01 - Temperature Sensors*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Microchip Technology Inc., 2017. *MCP602 - Low Power, Low Noise, Operational Amplifiers*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Microchip Technology Inc., 2018. *MCP1700 - Low Quiescent Current LDO*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Various Authors, 2000. *Photodiode*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- STMicroelectronics, 2016. *PM0214 Programming Manual - STM32 Cortex-M4*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- Texas Instruments, 1999. *SLOA097 - Designing Gain and Offset in Thirty Seconds*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- STMicroelectronics, 2018. *STM32F411xC STM32F411xE - ARM Cortex-M4 32b MCU*. [pdf] Available at: ¡link¿ [Accessed 20 May 2024].

- STMicroelectronics, 2006. *L7800 Series - Positive Voltage Regulators.* [pdf] Available at: ¡link¿ [Accessed 20 May 2024].
- STMicroelectronics, 2020. *UM1724 User Manual - STM32 Nucleo-64 Boards (MB1136).* [pdf] Available at: ¡link¿ [Accessed 20 May 2024].