DEPARTMENT OF ELECTRONIC & TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

Battery Profiler

Design Documentation

Name	**Index Number**
Bandara M.A.G.S.S.	220064U
Kulasinghe H.P.G.N.A.	220334A

EN2160 Electronic Design Realization

Date: July 11, 2025

Contents

1 Introduction

This report covers the detailed documentation with regard to the battery profiler designed in fulfillment of the Electronic Design Realization module. This documentation covers the design process, conceptual designs, finalized designs and the progress with regard to the product.

2 Schematic Explanation

2.1 Microcontroller

The STM32F103CBT6 was selected due to its optimal balance between performance and peripheral support. Its 72MHz processing speed, built-in Floating Point Unit (FPU), and ample I/O pins made it an excellent choice for real-time data acquisition and control. It also supports various communication protocols (SPI, I2C, UART), which were essential for interfacing with peripherals like DACs, sense amplifiers, and display units.

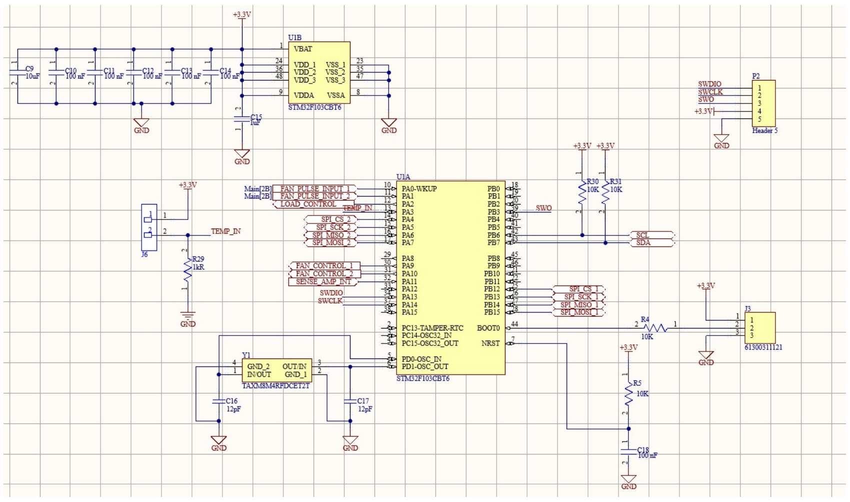

Figure 1: STM32F103CBT6 microcontroller

2.2 Overview of Circuit Design

The system revolves around a central microcontroller that governs battery discharge, monitors voltage, and controls thermal regulation. An external DAC supplies a programmable analog reference voltage, which drives a feedback-controlled analog path. The sense amplifier reads current flow through the system, allowing precise regulation of discharge behavior. A fan controller adjusts cooling mechanisms based on temperature feedback.

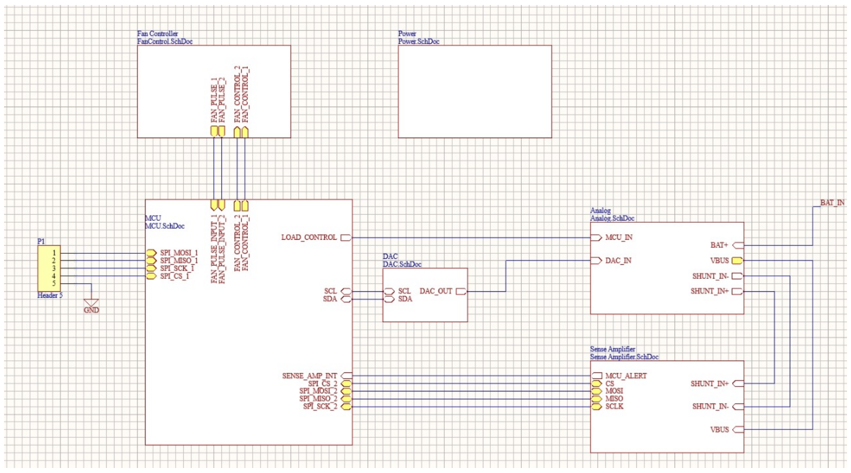

Figure 2: System block diagram

2.3 Analog Section

The analog interface ensures safe and controlled operation during discharge cycles. Protective components like TVS diodes mitigate transient voltage spikes, while reverse polarity protection circuits using OP-AMPs and MOSFETs prevent circuit damage. Decoupling capacitors help maintain voltage stability throughout the power rails.

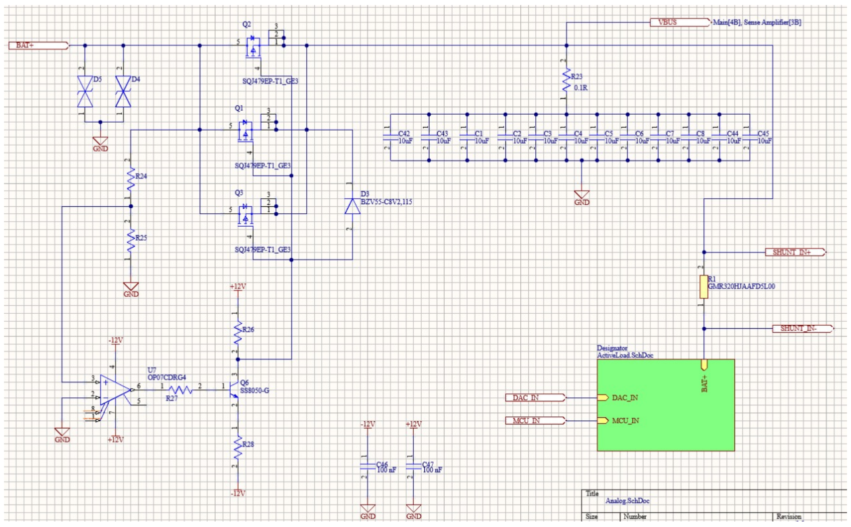

Figure 3: Analog interface circuit

2.4 Active Load Circuit

Power from the battery is dissipated through multiple parallel MOSFET-based circuits acting as active loads. These distribute heat and reduce individual stress on components. The current flow is controlled by adjusting the gate voltage via an OP-AMP feedback loop, based on a precision sense resistor.

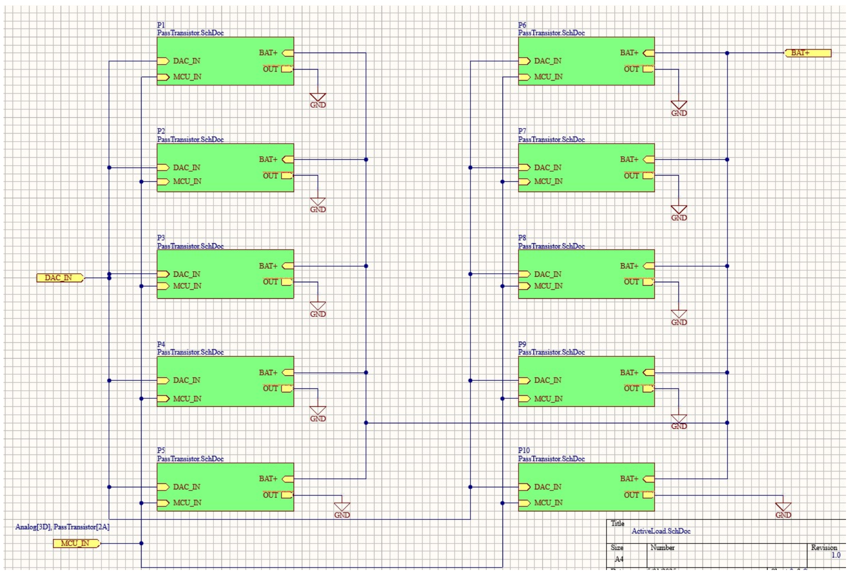

Figure 4: Active load circuit

2.5 Pass Transistor

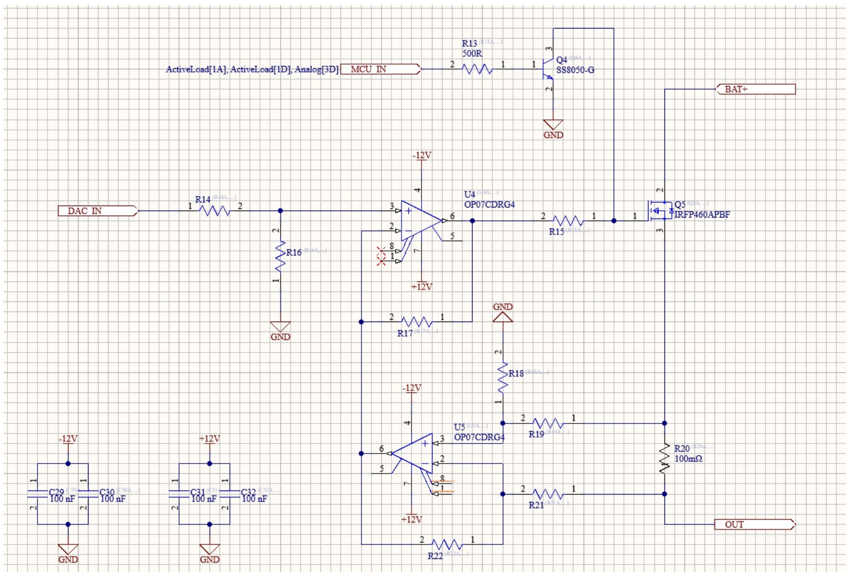

Figure 5: Pass transistor circuit

This is the internal circuit of an active load element. The power is dissipated in the MOSFET labeled as Q5 in the schematic. The specifications are given below:

- **Maximum continuous drain current at 25°C**: 20A
- **Maximum continuous drain current at 100°C**: 13A
- **Maximum power dissipation**: 280W
- **Maximum drain-source voltage**: 500V

Our product requires a maximum continuous current of just 3A per transistor and a maximum power dissipation of 30W per transistor. Therefore, this component was suitable for our needs. The circuit operates based on the feedback principle. The required current is set by the microcontroller and that is converted to an analog voltage level by the DAC. The feedback loop of the U4 OP-AMP adjusts the gate voltage of the MOSFET such that both inputs are at an equal voltage level. Resistor R20 is a current sense resistor that is used as part of the differential amplifier which has been set up with the U5 OP-AMP.

2.6 Fan Regulation Module

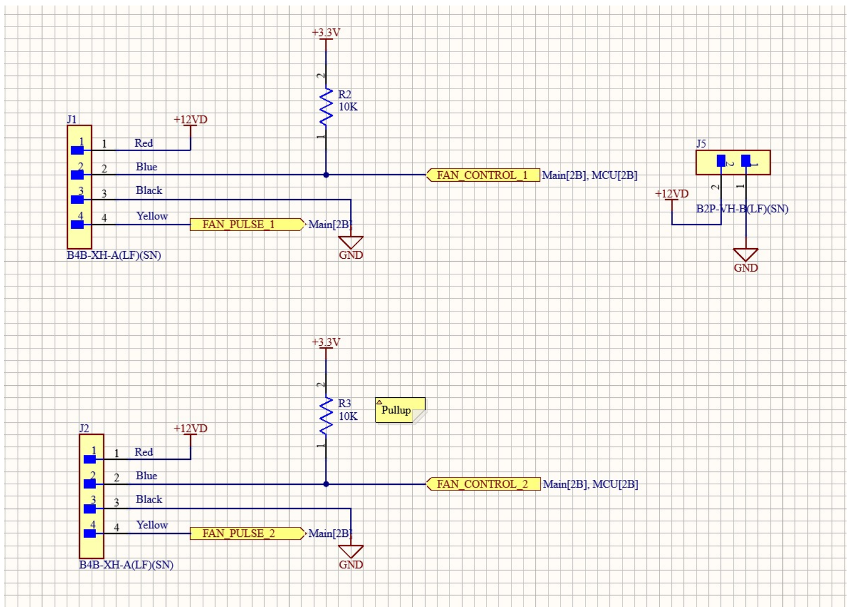

Figure 6: Fan control circuit

Temperature-dependent control is implemented using PWM signals generated by the microcontroller. Feedback from the fan's encoder pins enables accurate speed monitoring. By modulating the duty cycle, the system maintains thermal stability under varying power loads.

2.7 Digital-to-Analog Conversion

To finely control discharge currents, the AD5693RBRMZ 16-bit DAC was used. This DAC provides high-resolution analog voltage outputs with excellent accuracy and low temperature drift. Signal conditioning with operational amplifiers and filters ensures stability and noise reduction.

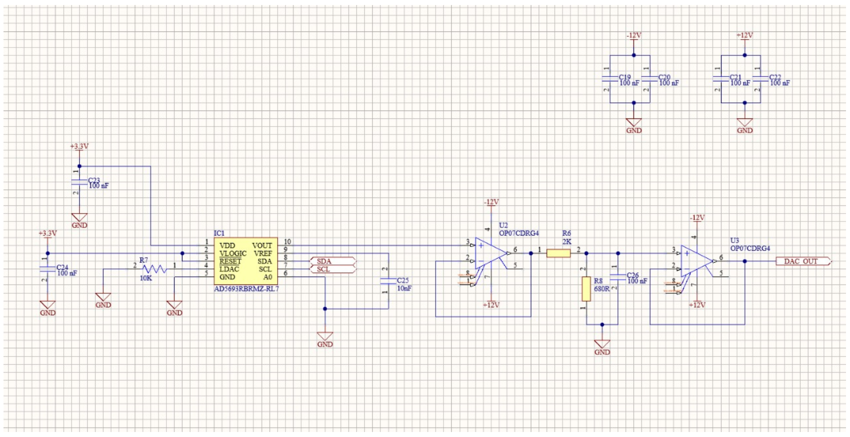

Figure 7: DAC circuit

2.8 Sense Amplifier

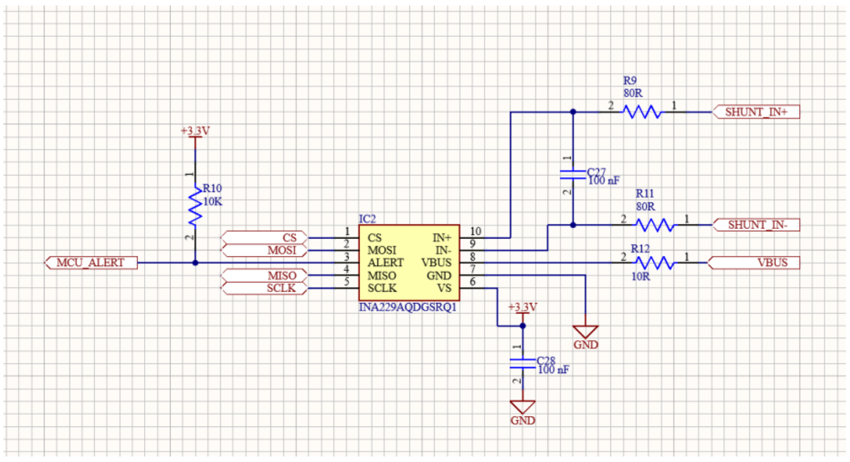

Figure 8: Sense amplifier circuit

This schematic shows a current and voltage sensing circuit using the INA229 digital power monitor. The shunt resistor network allows differential current sensing between SHUNT IN+ and SHUNT IN-, while voltage is sensed via VBUS through R12. The INA229 communicates via SPI and provides an ALERT output pulled up by R10.

2.9 Power Supply Design

The power module converts AC input into ±12V rails for analog circuits, and steps down to 3.3V for powering digital devices. This dual-supply setup supports OP-AMP operation and ensures logic compatibility with the microcontroller and communication components.

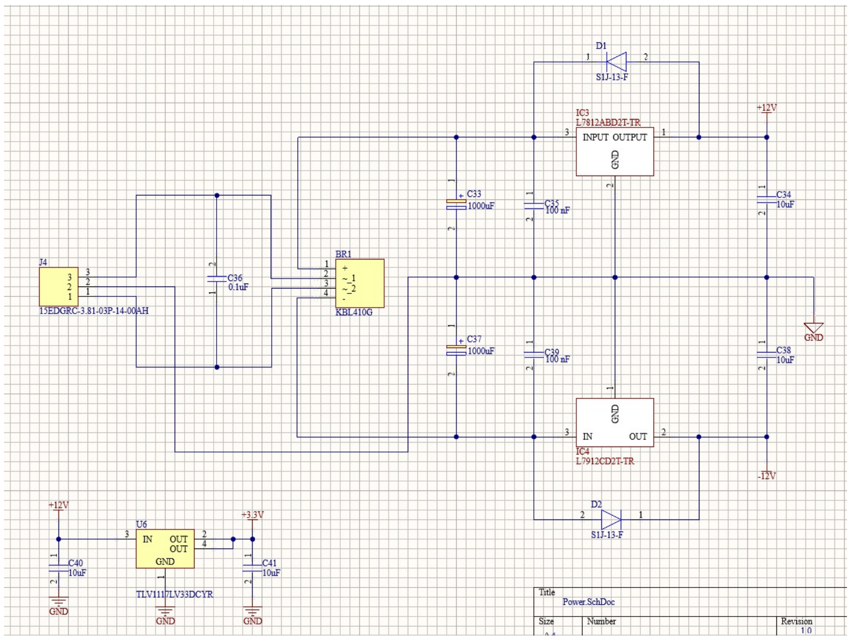

Figure 9: Power supply circuit

3 Printed Circuit Board

A 4-layer PCB was designed for optimal signal routing and power distribution. The layout includes dedicated ground and power planes to minimize interference and enhance thermal dissipation. Component placement was carefully optimized for accessibility and thermal management.

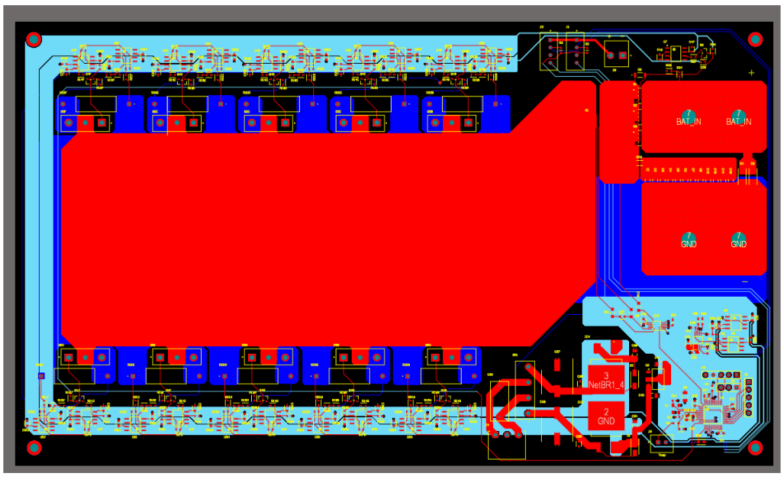

Figure 10: PCB routes

A 4-layer PCB was designed for optimal signal routing and power distribution. The board stackup consists of:

- **Layer 1 (Top)**: Primary component placement and high-speed signal routing
- **Layer 2 (Ground Plane)**: Continuous ground plane for EMI reduction

- **Layer 3 (Power Plane)**: Dedicated power distribution with multiple voltage islands
- **Layer 4 (Bottom)**: Secondary component placement and low-speed signals

3.1 Design Considerations

The PCB layout incorporates several critical design features:

- **Component Placement**:
  - Power MOSFETs positioned near board edges for optimal heat dissipation
  - Sensitive analog components isolated from digital noise sources
  - Decoupling capacitors placed close to IC power pins
- **Routing Strategy**:
  - 45° trace angles to reduce signal reflections
  - Differential pairs for high-precision current sensing
  - Star-point grounding for analog and digital sections
- **Thermal Management**:
  - Thermal vias under power components
  - Copper pours for heat spreading
  - Adequate clearance for heatsink mounting

3.2 Manufacturing Specifications

Parameter	**Value**
PCB Dimensions	160mm × 100mm
Layer Count	4
Material	FR-4
Copper Weight	1oz (outer), 0.5oz (inner)
Minimum Trace Width	0.2mm
Minimum Clearance	0.2mm
Surface Finish	HASL

4 Enclosure Design

These images show the SolidWorks design of the enclosure . The design was done considering the factors strength, robustness, durability, and cost while ensuring the functional requirements of the user.

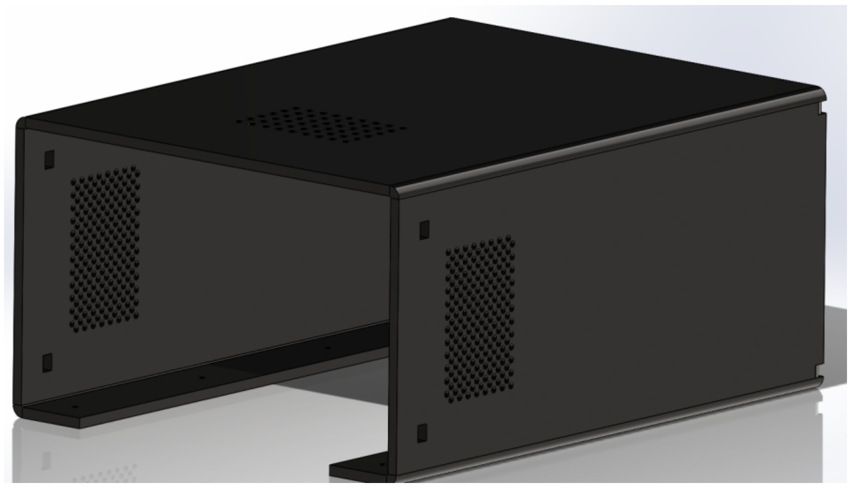
Figure 11: Sheet Metal

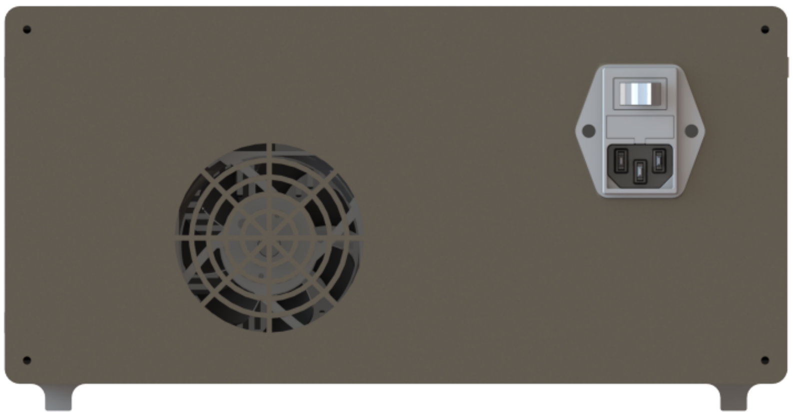
Figure 12: Rear View

5 Code

The code for our project is included here. For the purpose of coding, we used embedded C as the language and the STM32 Cube IDE as the development environment. In order to facilitate modularity, we created libraries for individual functions of the code.

5.1 Main Function

```c
#include "main.h"
#include "movingaverage.h"
#include "fancontrol.h"
#include "AD5693R.h"
#include "INA229.h"

ADC_HandleTypeDef hadc1;
I2C_HandleTypeDef hi2c1;
SPI_HandleTypeDef hspi1, hspi2;
TIM_HandleTypeDef htim1, htim2;

INA229 adc;
PID_HandleTypeDef hpid;

uint8_t ARRAY_SIZE = 10;
float setpoint = 25.0;
float kp = 1.0, ki = 0.1, kd = 0.01;

MovingAverageArray voltagearray, currentarray, powerarray, energyarray;

int main(void) {
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_I2C1_Init();
    MX_SPI1_Init();
    MX_SPI2_Init();
    MX_TIM1_Init();
    MX_TIM2_Init();
    MX_ADC1_Init();

    INA229_Initialize(&adc, &hspi1);
    INA229_SetRST(&adc);
    INA229_SetRSTACC(&adc);
    INA229_SetTEMPCOMP(&adc);
    INA229_SetADCRANGE(&adc);
    INA229_SetCONVDLY(&adc, 0xFF);

    InitMovingAverageArray(&voltagearray);
    InitMovingAverageArray(&currentarray);
    InitMovingAverageArray(&powerarray);
    InitMovingAverageArray(&energyarray);

    while (1) {
        if (HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY) ==
            HAL_OK) {
```

```c
            float temperature =
                Convert_ADCValue_To_Temperature(HAL_ADC_GetValue(&hadc1));
            float control = PID_Compute(&hpid, setpoint, temperature);

            control = (control < 0) ? 0 : (control > 999 ? 999 :
                control);
            __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, control);
            __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, control);
        }

        float voltage = GetVoltage();
        float current = GetCurrent();
        float power = GetPower();
        float energy = GetEnergy();

        UpdateMovingAverageArray(&voltagearray, voltage);
        UpdateMovingAverageArray(&currentarray, current);
        UpdateMovingAverageArray(&powerarray, power);
        UpdateMovingAverageArray(&energyarray, energy);

        float avgVoltage = ComputeMovingAverage(&voltagearray);
        float avgCurrent = ComputeMovingAverage(&currentarray);
        float avgPower = ComputeMovingAverage(&powerarray);
        float avgEnergy = ComputeMovingAverage(&energyarray);

        HAL_Delay(1000);
    }
}
```

Listing 1: Main STM32 Code

5.2 Moving average header file

```c
uint8_t size = 10; // Here, we use 10 point moving average

typedef struct {
    float array[10];
    uint8_t index;
} MovingAverageArray;

void InitMovingAverageArray(MovingAverageArray *maArray)
{
    for (uint8_t i = 0; i < size; i++) {
        maArray->array[i] = 0.0f;
    }
    maArray->index = 0;
}

void UpdateMovingAverageArray(MovingAverageArray *maArray, float
    newValue)
{
    maArray->array[maArray->index] = newValue;
    maArray->index = (maArray->index + 1) % size;
}

float ComputeMovingAverage(MovingAverageArray *maArray)
{
    float sum = 0;
    for (uint8_t i = 0; i < size; i++) {
        sum += maArray->array[i];
    }
    return sum / size;
}

// Below functions will need to be implemented.
// For now, we just return average readings

float GetVoltage(void)
{
    return (float)(rand() % 100);
}

float GetCurrent(void)
{
    return (float)(rand() % 100);
}

float GetPower(void)
{
    return (float)(rand() % 100);
}

float GetEnergy(void)
{
    return (float)(rand() % 100);
}
```

Listing 2: Moving Average and Measurement Functions

5.3 AD5693R.c

```c
#include "AD5693R.h"

uint8_t AD5693R_Initialize(AD5693R *dev, I2C_HandleTypeDef *i2cHandle){
    dev->i2cHandle = i2cHandle;
    HAL_StatusTypeDef status;
    uint8_t errNum = 0;
    uint16_t dacinputValue = 0x0000;
    status = AD5693R_WriteDacIn(dev, &dacinputValue);
    errNum += (status != HAL_OK);
    return errNum;
}

HAL_StatusTypeDef AD5693R_WriteInput(AD5693R *dev, uint16_t *regData){
    HAL_StatusTypeDef status;
    uint8_t data[2] = {((*regData >> 8) & 0x00FF), (*regData &
        0x00FF)};
    status = AD5693R_WriteRegister(dev, AD5693R_WRITE_INPUT, data, 2);
    return status;
}

HAL_StatusTypeDef AD5693R_UpdateDac(AD5693R *dev, uint16_t *regData){
    HAL_StatusTypeDef status;
    uint8_t data[2] = {((*regData >> 8) & 0x00FF), (*regData &
       0x00FF)};
    status = AD5693R_WriteRegister(dev, AD5693R_UPDATE_DAC, data, 2);
    return status;
}

HAL_StatusTypeDef AD5693R_WriteDacIn(AD5693R *dev, uint16_t *regData){
    HAL_StatusTypeDef status;
    uint8_t data[2] = {((*regData >> 8) & 0x00FF), (*regData &
       0x00FF)};
    status = AD5693R_WriteRegister(dev, AD5693R_WRITE_DAC_IN, data, 2);
    return status;
}

HAL_StatusTypeDef AD5693R_ChangeGain(AD5693R *dev, uint8_t gain_val){
    HAL_StatusTypeDef status;
    uint16_t controlReg;
    if(gain_val == 1){
        controlReg = 0x0000;
    } else if(gain_val == 2){
        controlReg = 0x0001;
    }
    uint8_t data[2] = {((controlReg >> 8) & 0x00FF), (controlReg &
       0x00FF)};
    status = AD5693R_WriteRegister(dev, AD5693R_WRITE_CONTROL, data,
       2);
    return status;
}

HAL_StatusTypeDef AD5693R_WriteRegister(AD5693R *dev, uint8_t reg,
    uint8_t *data, uint8_t length){
    return HAL_I2C_Mem_Write(dev->i2cHandle, AD5693R_I2C_ADDR, reg,
                             I2C_MEMADD_SIZE_8BIT, data, length,
                                HAL_MAX_DELAY);
}
```

Listing 3: AD5693R DAC Source File

5.4 AD5693R.h

```c
#ifndef AD5693R_H
#define AD5693R_H

#include "stm32f1xx_hal.h" // Needed for I2C

// I2C address
#define AD5693R_I2C_ADDR (0x68 << 1)

// Commands
#define AD5693R_WRITE_INPUT     0x10 // 0001XXXX
#define AD5693R_UPDATE_DAC      0x20 // 0010XXXX
#define AD5693R_WRITE_DAC_IN    0x30 // 0011XXXX
#define AD5693R_WRITE_CONTROL   0x40 // 0100XXXX

typedef struct {
    I2C_HandleTypeDef *i2cHandle;
} AD5693R;
```

```

uint8_t AD5693R_Initialize(AD5693R *dev, I2C_HandleTypeDef *i2cHandle);
HAL_StatusTypeDef AD5693R_WriteInput(AD5693R *dev, uint16_t *regData);
HAL_StatusTypeDef AD5693R_UpdateDac(AD5693R *dev, uint16_t *data);
HAL_StatusTypeDef AD5693R_WriteDacIn(AD5693R *dev, uint16_t *data);
HAL_StatusTypeDef AD5693R_ChangeGain(AD5693R *dev, uint8_t gain_val);
HAL_StatusTypeDef AD5693R_WriteRegister(AD5693R *dev, uint8_t reg,
                                        uint8_t *data, uint8_t length);

#endif /* AD5693R_H */
```

Listing 4: AD5693R DAC Header File

5.5 Fan controller header file

```
typedef struct {
    float kp;
    float ki;
    float kd;
    float prev_error;
    float integral;
    float dt;
} PID_HandleTypeDef;

// Function prototypes
void PID_Init(PID_HandleTypeDef *pid, float kp, float ki, float kd,
   float dt);
float PID_Compute(PID_HandleTypeDef *pid, float setpoint, float
   measured);
float Convert_ADCValue_To_Temperature(uint32_t adcValue);

ADC_HandleTypeDef hadc1;
TIM_HandleTypeDef htim2;

// Initialize PID
void PID_Init(PID_HandleTypeDef *pid, float kp, float ki, float kd,
   float dt) {
    pid->kp = kp;
    pid->ki = ki;
    pid->kd = kd;
    pid->prev_error = 0;
    pid->integral = 0;
    pid->dt = dt;
}

// Function to calculate PID output
float PID_Compute(PID_HandleTypeDef *pid, float setpoint, float
   measured) {
    float error = setpoint - measured;
    pid->integral += error * pid->dt;
    float derivative = (error - pid->prev_error) / pid->dt;
    pid->prev_error = error;
    float output = (pid->kp * error) + (pid->ki * pid->integral)
                 + (pid->kd * derivative);
    return output;
}

```

```c
// Temperature conversion for ADC
float Convert_ADCValue_To_Temperature(uint32_t adcValue) {
    // Convert the ADC value to temperature
    float voltage = (adcValue / 4095.0) * 3.3;
    float temperature = voltage * 100.0;
    return temperature;
}
```

Listing 5: PID Controller Implementation in C

5.6 INA229.h

```c
#ifndef INA229_H
#define INA229_H

#include "stm32f1xx_hal.h" /*Needed for I2C*/

// Commands
#define INA229_CONFIG               0x00
#define INA229_ADC_CONFIG           0x01
#define INA229_SHUNT_CAL            0x02
#define INA229_SHUNT_TEMPCO         0x03
#define INA229_VSHUNT               0x04
#define INA229_VBUS                 0x05
#define INA229_DIETEMP              0x06
#define INA229_CURRENT              0x07
#define INA229_POWER                0x08
#define INA229_ENERGY               0x09
#define INA229_CHARGE               0x0A
#define INA229_DIAG_ALRT            0x0B
#define INA229_SOVL                 0x0C
#define INA229_SUVL                 0x0D
#define INA229_BOVL                 0x0E
#define INA229_BUVL                 0x0F
#define INA229_TEMP_LIMIT           0x10
#define INA229_PWR_LIMIT            0x11
#define INA229_MANUFACTURER_ID      0x3E
#define INA229_DEVICE_ID            0x3F

typedef struct {
    SPI_HandleTypeDef *spiHandle;
    uint32_t shunt_voltage;
    uint32_t bus_voltage;
    uint32_t current;
    uint16_t config;
    uint16_t adc_config;
    uint16_t flags;
    uint64_t energy;
    uint32_t power;
    uint16_t device_id;
    uint16_t manufacture_id;
} INA229;

uint8_t INA229_Initialize(INA229 *dev, SPI_HandleTypeDef *spiHandle);
HAL_StatusTypeDef INA229_ReadShuntVoltage(INA229 *dev);
HAL_StatusTypeDef INA229_ReadBusVoltage(INA229 *dev);
HAL_StatusTypeDef INA229_ReadCurrent(INA229 *dev);
HAL_StatusTypeDef INA229_ReadEnergy(INA229 *dev);
HAL_StatusTypeDef INA229_ReadPower(INA229 *dev);
HAL_StatusTypeDef INA229_ReadFlags(INA229 *dev);
HAL_StatusTypeDef INA229_ReadManufactureId(INA229 *dev);
HAL_StatusTypeDef INA229_ReadDeviceId(INA229 *dev);
HAL_StatusTypeDef INA229_ReadDieTemp(INA229 *dev);
HAL_StatusTypeDef INA229_ReadConfigReg(INA229 *dev);
HAL_StatusTypeDef INA229_WriteConfig(INA229 *dev);
HAL_StatusTypeDef INA229_WriteAdcConfigs(INA229 *dev);
HAL_StatusTypeDef INA229_WriteShuntConfig(INA229 *dev, uint16_t *config);
HAL_StatusTypeDef INA229_WriteShuntTempConfig(INA229 *dev, uint16_t *config);
HAL_StatusTypeDef INA229_WriteSOVL(INA229 *dev, uint16_t *config); // Shunt Overvoltage Threshold
HAL_StatusTypeDef INA229_WriteSUVL(INA229 *dev, uint16_t *config); // Shunt Undervoltage Threshold
HAL_StatusTypeDef INA229_WriteBOVL(INA229 *dev, uint16_t *config); // Bus Overvoltage Threshold
HAL_StatusTypeDef INA229_WriteBUVL(INA229 *dev, uint16_t *config); // Bus Undervoltage Threshold
HAL_StatusTypeDef INA229_WriteTempLimit(INA229 *dev, uint16_t *config); // Temperature Over-Limit Threshold
HAL_StatusTypeDef INA229_WritePowerLimit(INA229 *dev, uint16_t *config); // Power Over-Limit Threshold

HAL_StatusTypeDef INA229_ReadRegister(INA229 *dev, uint8_t reg,
                                      uint8_t *rxBuf, uint8_t length);
HAL_StatusTypeDef INA229_WriteRegister(INA229 *dev, uint8_t reg,
                                       uint16_t value, uint8_t length);

void INA229_SetRST(INA229 *dev);
void INA229_ClearRST(INA229 *dev);
void INA229_SetRSTACC(INA229 *dev);
void INA229_ClearRSTACC(INA229 *dev);
void INA229_SetTEMPCOMP(INA229 *dev);
void INA229_ClearTEMPCOMP(INA229 *dev);
void INA229_SetADCRANGE(INA229 *dev);
void INA229_ClearADCRANGE(INA229 *dev);
void INA229_SetCONVDLY(INA229 *dev, uint8_t value);
void INA229_SetMODE(INA229 *dev, uint8_t mode);
uint8_t INA229_GetMODE(uint16_t reg_value);
void INA229_SetVBUSCT(INA229 *dev, uint8_t value);
uint8_t INA229_GetVBUSCT(uint16_t reg_value);
void INA229_SetVSHCT(INA229 *dev, uint8_t value);
uint8_t INA229_GetVSHCT(uint16_t reg_value);
void INA229_SetVTCT(INA229 *dev, uint8_t value);
uint8_t INA229_GetVTCT(uint16_t reg_value);
void INA229_SetAVG(INA229 *dev, uint8_t value);
uint8_t INA229_GetAVG(uint16_t reg_value);

#endif /* INC_TPS55288_H_ */
```

Listing 6: INA229 Driver Header File

6 User Interface

The user interface of the product has been designed to be intuitive and user-friendly.

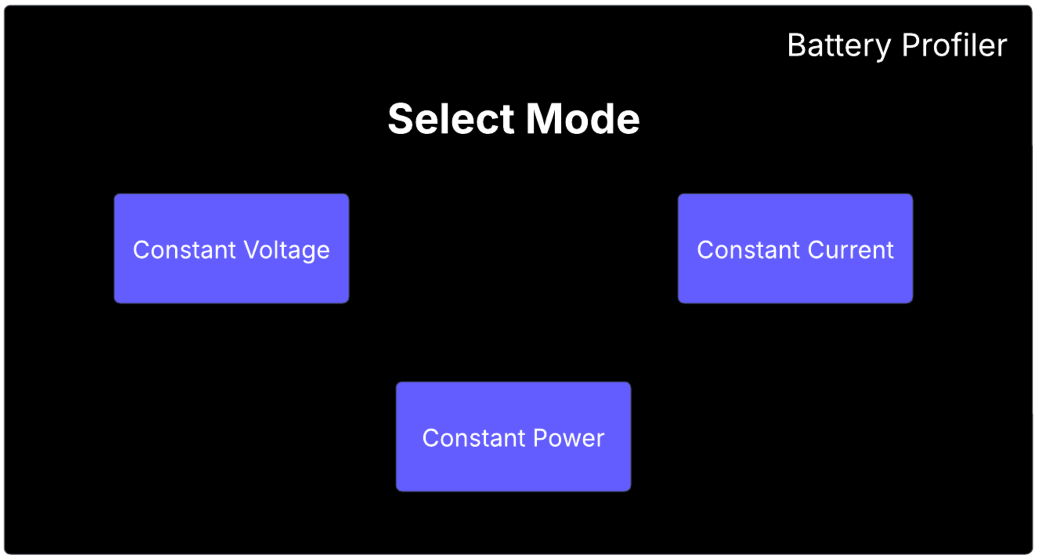

Figure 13: Main Screen

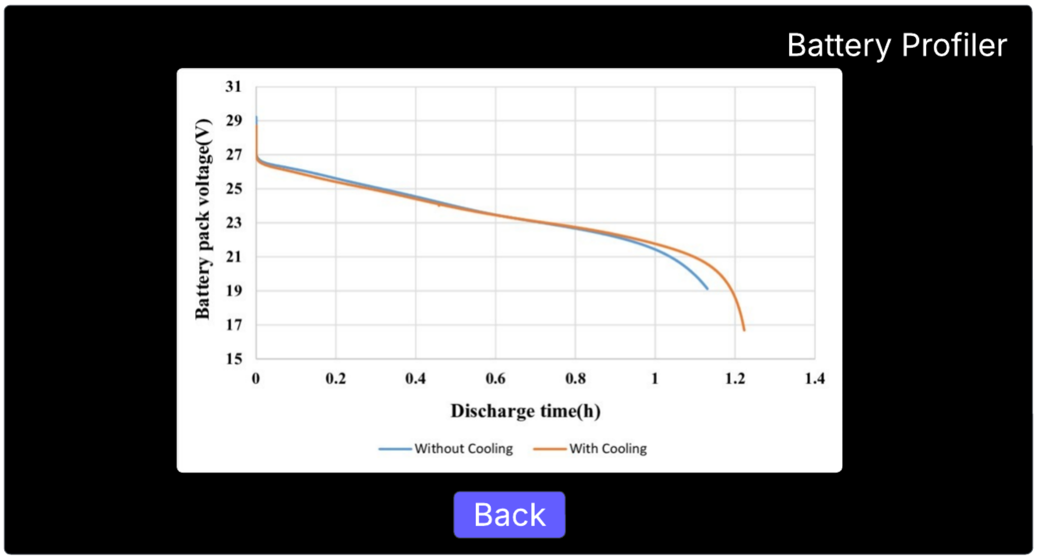

Figure 14: Graph

This is the measurement section. The user should first enter the value of the constant current, power or voltage using the rotary encoder or the number pad and press the start button in the button pad. This will start the discharge process. When discharging, the voltage at the output is measured and plotted on the graph.

7 Design Updates After Mid Evaluation

After the mid-evaluation of our Battery Profiler project, we made significant progress in both hardware integration and software development. The following updates were carried out:

7.1 Hardware Integration

- **PCB Component Soldering:** We successfully soldered all the required components onto the custom-designed PCB. This included the microcontroller, voltage and current sensing components, display module headers, connectors, and other passive components.
- **Microcontroller Replacement:** During initial testing after soldering, the original microcontroller was accidentally damaged due to incorrect power handling. After diagnosing the issue, we carefully desoldered the burnt microcontroller and replaced it with a new one. We double-checked the power lines and ensured proper ESD precautions during the second attempt.

7.2 Software Development and Debugging

- **Code Debugging and Functional Verification:** Once the new microcontroller was in place, we uploaded the firmware and began testing. We faced several issues related to sensor communication, timing mismatches, and calibration inaccuracies. Through systematic debugging and use of serial print statements and oscilloscope monitoring, we resolved these issues and achieved stable readings.
- **Graphical User Interface Implementation:** A major milestone achieved after the mid-evaluation was the integration of a graphical user interface (GUI) on the display screen. The interface now displays real-time voltage, current, power, and battery capacity values in graphical form.
- **Screen Display Enhancements:** We added visualization features that plot graphs dynamically on the screen, improving user experience. These include:
  - Real-time voltage vs time graph
  - Real-time current vs time graph
  - Battery capacity indication
  - User-friendly menu system to start/stop profiling
- **Firmware Optimization:** We also optimized the code for better performance, removing unnecessary delays and ensuring non-blocking UI updates. The final code version is now modular and easier to maintain.

7.3 Enclosure Design and Fabrication

- **Front Face Enclosure Design:** To improve the device's usability and presentation, we designed a custom front face enclosure for the Battery Profiler. This enclosure includes precise cut-outs for the display screen, buttons, and power/battery terminals.

- **Fabrication:** The enclosure was fabricated using laser-cut acrylic sheets and assembled using spacers and screws. The display fits snugly behind a transparent acrylic window, and the design ensures easy access to user controls and ports while offering mechanical protection to internal components.
- **Aesthetic and Functional Considerations:** The design also considered heat dissipation and compactness. The result is a durable and user-friendly casing that enhances the product's final appearance and field usability.

7.4 Summary of Post Mid-Evaluation Achievements

- Successful assembly and testing of the final PCB
- Replacement of damaged microcontroller and power debugging
- Full integration of display interface with real-time graph plots
- Stable operation and verified data readings
- Design and fabrication of the front face enclosure

These updates significantly improved the functionality, reliability, and user experience of our Battery Profiler device, bringing it closer to a finished product.

8 Conclusion

The product has been designed to be able to appropriately satisfy the needs of the users. Thorough research has been done to make sure this product meets the needs of users and functions appropriately as expected. Further progress can be made by obtaining feedback from users after the product has been sold.

9 References

References

[1] Keysight Technologies, *E36731A Battery Emulator and Profiler* :- `https://www.keysight.com/us/en/product/E36731A/battery-emulator-and-profiler.html`