Software Description Document
Advanced Multi-Cell Lithium Battery Load Analyzer
2CE1EE

Jake Lin (CE)
Omar Mohamed (EE)
Tyler Shaw (CE)
Faculty Advisor: Scott Tierno

Contents

Section 1 Initialization and Main Program Code:

The initialization code first initializes some of the variables and FSMs to their default values. Some of these values are constants like battery_voltage_divider_ratios, while others change value throughout the program like testing_mode. This initialization is needed so that variables and FSMs have the correct value or are in the correct state when the program begins. This is shown below in figure 1.1 (TS).

```
//set variables to default values
testing_mode = 0x01; //automated test
current_setting = 30; //30 A current
current_setting_100_dig = 0;
current_setting_10_dig = 3;
current_setting_1_dig = 0;
voltage_precision = 0x00; //high voltage precision
min_battery_voltage = 3.0; //min unloaded voltage of 3 V

adc_mode = 0x00; //single ended ADC mode
adc_value = 0;
adc_vref = 2.048; //2.048 voltage reference
battery_voltage_divider_ratios = 5.3;
current_sensing_voltage_divider_ratios = 1;
shunt_resistance_ohms = 0.000145;
OPAMP_gain = 20;
cursor = 1;
quad_pack_entry = 0;
load_current_amps = 0;

LOCAL_INTERFACE_CURRENT_STATE = MAIN_MENU_STATE;
TEST_CURRENT_STATE = ERROR;
VIEW_HISTORY_CURRENT_STATE = SCROLL_PREVIOUS_RESULTS;
PB_PRESS = NONE;
```

Figure 1.1: Variable Instantiation Code (TS)

Next, the different modules must be instantiated so they are prepared to process the data later in the program. These include the ADC for reading voltages, LCD for displaying data, push button Input/Output (I/O) ports for user input, stepper motor for automatically adjusting the load current, and USART3 module for transferring data to a personal computer (PC). The actual initialization of these modules are described in their respective sections. Pin PD5 is made an input so it floats and prevents noise on PD6, which is used for the ADC. Finally, interrupts are enabled. This code is provided below in figure 1.2 (TS).

```
//initialize modules
init_lcd();
ADC_init(0x00);
PB_init();
LOCAL_INTERFACE_FSM();
A4988_init();
USART3_setup();
display_main_menu();

VPORTD.DIR &= ~(PIN5_bm); //make PD5 an input so it floats to prevent noise

sei(); // enable interrupts
```

Figure 1.2: Module Instantiation Code (TS)

The actual main program is an infinite while loop that displays the main menu when in the main menu state. The program only runs based on user input. Therefore, interrupt service routines (ISRs) for the push buttons and UART are used, which will interrupt program flow and carry out local or remote interface functions respectively. The ISRs are discussed more in depth in the local and remote interface sections. The main program code just needs to stop the program from terminating, hence the infinite while loop. This is shown below in figure 1.3 (TS).

```
while(1)
{
    if (LOCAL_INTERFACE_CURRENT_STATE == MAIN_MENU_STATE) //display main menu in main menu state
        {
            display_main_menu();
            _delay_ms(1000);
        }

    asm volatile("nop");
}
```

Figure 1.3: Main Program Code (TS)

Section 2 Background Functions:

Section 2.1 Analog-to-Digital Converter (ADC):

Section 2.1.1 ADC Initialization:

The analog-to-digital-converter (ADC) of the microcontroller is configured to make differential voltage measurements on the battery cells and single-ended measurements on the output of the current-sense amplifier. The ADC mode (differential or single-ended) must be selected and the initialization function must be called before making a measurement. Additionally, the reference voltage is set to 2.048V to increase the voltage resolution when measuring low voltages at the cost of reducing the input range. The code for this is shown below in figure 2.1 (OM).

```
//.....................................................................
void ADC_init(uint8_t mode)
{
    adc_mode = mode;    // single-ended or differential mode

    // Use 2.048V as reference
    VREF.ADC0REF = VREF_REFSEL_2V048_gc;

    // 12-bit resolution, Free-Run mode, differential/single-ended, Right adjusted, Enable
    ADC0.CTRLA = (ADC_RESSEL_12BIT_gc | (ADC_FREERUN_bm) | (adc_mode << 5) | ADC_ENABLE_bm);

    //enables interrupt
    ADC0.INTCTRL |= ADC_RESRDY_bm;

    // Set to accumulate 64 samples
    ADC0.CTRLB = ADC_SAMPNUM_ACC64_gc;

    // Divided CLK_PER by 16
    ADC0.CTRLC = ADC_PRESC_DIV16_gc;
}
```

Figure 2.1: ADC Initialization Code (OM)

Additionally, the correct ADC channel must be selected. The ADC channel corresponds to the microcontroller pin connected to the signal being measured. The code is shown below in figure 2.2 (OM).

```
void ADC_channelSEL(uint8_t AIN_POS, uint8_t AIN_NEG)
{
    /* In differential mode, write both MUXPOS and MUXNEG registers */
    if (adc_mode == 0x01)
    {
        ADC0.MUXPOS = AIN_POS;
        ADC0.MUXNEG = AIN_NEG;
    }
    /* In single-ended mode, only write MUXPOS register */
    else
        ADC0.MUXPOS = AIN_POS;
}
```

Figure 2.2: ADC Channel Selection Code (OM)

Section 2.1.2 Reading ADC Input Voltage:

The result of the ADC measurement has 12-bit resolution and is stored in the 12 most-significant-bits (MSB) of a 16-bit register. Therefore, the register must be shifted right by 4 places so that the result is right adjusted. Additionally, the integer result must be converted to a floating-point by multiplying the integer value by one -least-significant bit (LSB). An LSB is equal to the ADC reference voltage divided by the number of quantization levels. Finally, for differential measurements, there is one signed bit, therefore, the unsigned resolution is 11 bits, therefore, there are half as many positive quantization levels. This is shown below in figure 2.3 (OM).

```
float ADC_read(void)
{
    /* Perform ADC conversion */
    ADC_startConversion();
    while(ADC_isConversionDone() != 0x01);  // wait for conversion to finish
    ADC_stopConversion();

    /* 12 bit result was left adjusted, so shift right 4 places to fix, reading ADC.RES clears interrupt flag */
    if (adc_mode == 0x00)
        return (float)( adc_vref * (ADC0.RES >> 4) / 4096); // single-ended resolution is 12 bits -> 4096 values
    else
        return (float)( adc_vref * (ADC0.RES >> 4) / 2048); // differential resolution is 11 bits -> 2048 values
}
```

Figure 2.3: Reading from the ADC Code (OM)

Section 2.1.3 Measuring Battery Cell Voltage:

In order to read the differential voltage across a battery cell, the ADC is configured in differential mode using the *ADC_init()* function. The ADC channels corresponding to the positive and negative terminals of the battery cell are then selected and the voltage is read from the ADC. High precision uses the 2.048 voltage reference, while low precision uses the 3.3 voltage reference. In order to account for the attenuation introduced by the voltage dividers, the measured ADC input voltage is multiplied by the voltage divider ratios to obtain the actual voltage across the battery cell. This is shown below in figure 2.4 (OM).

```
float batteryCell_read(uint8_t BAT_POS, uint8_t BAT_NEG)
{
    /* Differential measurement */
    ADC_init(0x01);

    if (voltage_precision == 0x01) //Low precision = 3.3 V voltage reference
    {
        VREF.ADC0REF = VREF_REFSEL_VDD_gc;
        adc_vref = 3.3;
    }
    else //high precision = 2.048 V voltage reference
    {
        VREF.ADC0REF = VREF_REFSEL_2V048_gc;
        adc_vref = 2.048;
    }
    /* Select ADC channel and wait for it to settle*/
    ADC_channelSEL(BAT_POS, BAT_NEG);
    _delay_ms(10);
    adc_value = ADC_read();

    /* Multiply by voltage divider ratio to undo attenuation */
    return (float) (adc_value * battery_voltage_divider_ratios);
}
```

Figure 2.4: Reading the Voltage on One Battery Cell Code (OM)

Section 2.1.4 Measuring Loaded and Unloaded Cell Voltages:

During a test, the loaded and unloaded voltages are read and stored in the appropriate memory locations. This is shown below in figures 2.5 and 2.6 (OM).

```
void read_UNLOADED_battery_voltages(void)
{
    /* Read voltage of each cell and store in array when unloaded */
    current_test_result.UNLOADED_battery_voltages[0] = batteryCell_read(B1_ADC_CHANNEL, GND_ADC_CHANNEL);   // B1_POS - GND
    current_test_result.UNLOADED_battery_voltages[1] = batteryCell_read(B2_ADC_CHANNEL, B1_ADC_CHANNEL);    // B2_POS - B1_POS
    current_test_result.UNLOADED_battery_voltages[2] = batteryCell_read(B3_ADC_CHANNEL, B2_ADC_CHANNEL);    // B3_POS - B2_POS
    current_test_result.UNLOADED_battery_voltages[3] = batteryCell_read(B4_ADC_CHANNEL, B3_ADC_CHANNEL);    // B4_POS - B3_POS
}
```

Figure 2.5: Reading Unloaded Voltages Code (OM)

```
void read_LOADED_battery_voltages(void)
{
    /* Read voltage of each cell and store in array once load current reaches 500A */
    current_test_result.LOADED_battery_voltages[0] = batteryCell_read(B1_ADC_CHANNEL, GND_ADC_CHANNEL); // B1_POS - GND
    current_test_result.LOADED_battery_voltages[1] = batteryCell_read(B2_ADC_CHANNEL, B1_ADC_CHANNEL);  // B2_POS - B1_POS
    current_test_result.LOADED_battery_voltages[2] = batteryCell_read(B3_ADC_CHANNEL, B2_ADC_CHANNEL);  // B3_POS - B2_POS
    current_test_result.LOADED_battery_voltages[3] = batteryCell_read(B4_ADC_CHANNEL, B3_ADC_CHANNEL);  // B4_POS - B3_POS
}
```

Figure 2.6: Reading Loaded Voltages Code (OM)

Section 2.1.5 Measuring Load Current:

In order to measure the load current, the ADC is configured in single-ended mode and the ADC channel connected to the op amp output is selected. High precision uses the 2.048 voltage reference, while low precision uses the 3.3 voltage reference. To account for the gain of the differential amplifier, the measured ADC voltage is divided by the gain to calculate the actual voltage across the shunt. The voltage is then converted to a current by dividing it by the known shunt resistance. This is shown below in figure 2.7 (OM).

```
float load_current_Read(void)
{
    /* Put ADC in single-ended mode */
    ADC_init(0x00);
    adc_vref = 2.048; //use 2.048 voltage reference

    ADC_channelSEL(OPAMP_ADC_CHANNEL, GND_ADC_CHANNEL); //select OPAMP ADC Channel and GND ADC Channel
    adc_value = ADC_read();

    /* Multiply by divider ratio to undo attenuation, divide by gain to undo gain, divide by resistance to convert to amps */
    load_current_amps = ((adc_value - 0.096) / (OPAMP_gain*shunt_resistance_ohms));

    if (voltage_precision == 0x01) //Low precision = 3.3 V voltage reference
    {
        VREF.ADC0REF = VREF_REFSEL_VDD_gc;
        adc_vref = 3.3;
    }


    if(load_current_amps < 0.2) //if load current is less than 0.2, return 0
        return 0;
    else //else return load_current_amps
        return load_current_amps;
}
```

Figure 2.7: Reading Load Current Code (OM)

Section 2.2 Liquid Crystal Display (LCD):

The LCD is used to display data to the user and communicates using Serial Peripheral Interface (SPI). The AVR128DB48's SPI1 module is used to communicate with the LCD.

Section 2.2.1 Initializing the LCD:

To initialize the LCD, first the SPI interface must be initialized. SPI mode 0 is used. SPI1 is used so pins PC0, PC2, and PC3 must be set as outputs for MOSI, SCK, and /SS. MISO is not used because the LCD does not output data. PC3 must be set to disable the LCD slave. This is shown in figure 2.8 (TS).

```
,,
void init_spi_lcd (void)
{
    VPORTC_DIR |= (PIN0_bm | PIN2_bm | PIN3_bm); //set pa4, pa6, pa7 as outputs for mosi, sck, and /ss
    VPORTC_OUT |= PIN3_bm; //ss set high initially, disable LCD slave
    SPI1_CTRLA |= (SPI_MASTER_bm | SPI_ENABLE_bm); //enable spi, and make master mode
    SPI1_CTRLB |=  SPI_MODE_0_gc; //set spi mode to 0
}
```

Figure 2.8: Initializing SPI1 Module Code (TS)

After the SPI module has been initialized, communication with the LCD is possible. The '|' is sent to the LCD meaning enter the settings mode followed by - to clear the display and reset the cursor to the beginning. Next ' ' is written to each character of the LCD's display buffer. This is done to prevent dark boxes appearing on the LCD from no character being used there. The code is provided below in figure 2.9.

```
,,
void init_lcd (void)
{
    init_spi_lcd();     //Initialize mcu for LCD SPI
    _delay_ms(10); //delay 10 ms
    lcd_spi_transmit('|'); //Enter settings mode
    lcd_spi_transmit('-'); //clear display and reset cursor

    /* Outer loop for each line */
    for (uint8_t i = 0; i < 4; i++)
    {
        /* Inner loop for each character */
        for (uint8_t j = 0; j < 20; j++)
        {
            dsp_buff[i][j] = ' ';
        }
    }
}
```

Figure 2.9: Initializing LCD Code (TS)

Section 2.2.2 Updating the LCD:

To update the LCD, each string to be displayed must be placed in the correct array of the variable dsp_buff, which has 4 rows and 21 columns. Each row represents one line of the LCD and each column represents one character. The last character is reserved for the '\0' used in C to terminate strings. Then, the function update_lcd() can be called, which sends each character stored in the display buffer to the LCD in order. This is shown below in figure 2.10 (TS).

```
void update_lcd(void)
{
    /* Outer loop transmits all 4 lines of LCD */
    for (uint8_t i = 0; i < 4; i++)
    {
        /* Inner loop transmit each character of line i */
        for (uint8_t j = 0; j < 20; j++)
        {
            lcd_spi_transmit(dsp_buff[i][j]);
        }
    }
}
```

Figure 2.10: Updating the LCD's Display Buffers Code (TS)

To send characters to the LCD, they must be transmitted using SPI. The lcd_spi_transmit(char cmd) function is used to do this. The function clears PC3 to enable the slave and then places the character cmd on the SPI1 data register to be sent to the LCD. It then waits for the character to be transmitted to the LCD before disabling the slave. There is a 100 us delay to allow the LCD to receive the character before sending another one. This is shown below in figure 2.11 (TS).

```
//
void lcd_spi_transmit (char cmd)
{
    VPORTC_OUT &= ~PIN3_bm; //set PA7 to 0 to enable LCD Slave
    SPI1_DATA = cmd;        //send command
    while(!(SPI1_INTFLAGS & SPI_IF_bm)) {}    // wait until Tx complete
    VPORTC_OUT |= PIN3_bm; //set PA7 to 1 to disable LCD Slave
    _delay_us(100); //delay for command to be processed
}
```

Figure 2.11: Transmitting Characters Using SPI Code (TS)

Section 2.3 Stepper Motor:

Section 2.3.1 Sending a STEP Pulse:

In order to control the stepper motor, the microcontroller simply controls the STEP and DIR pins of the A4988. In order to make the motor complete a step, a 50% duty cycle pulse is generated by toggling the pin HIGH and then LOW with a delay variable used to determine the pulse width. This is shown below in figure 2.12 (OM).

```
void A4988_step(void)
{
    PORTC.OUT |= PIN4_bm;           // Rising edge on STEP pin
    _delay_us(0.5*STEP_PERIOD_US);  // delay for half PERIOD
    PORTC.OUT &= ~PIN4_bm;          // Falling edge on STEP pin
    _delay_us(0.5*STEP_PERIOD_US);  // delay for half PERIOD
}
```

Figure 2.12: Sending a STEP Pulse Code (OM)

Section 2.3.2 Programming the Load Current:

To program the load current via software, feedback from a shunt resistor is used. A while loop repeatedly sends a STEP pulse to the A4988 driver, ensuring the DIR pin is set correctly before each step. In each iteration, the system reads the load current and calculates the error by subtracting it from the user-defined target current. If the error is positive, the current exceeds the target, so the stepper motor turns the carbon pile knob counterclockwise to decrease the current. If the error is negative, the current is below the target, and the motor turns the knob clockwise to increase it. This process is repeated indefinitely until the absolute value of the error between the target and actual load current falls to within a tolerance value of +/- 5A. The feedback algorithm is summarized by the function shown below in figure 2.13 (OM). Note that the *A4988_dir_LOW*() and *A4988_dir_HIGH*() functions simply set the logic level of the GPIO pin associated with DIR.

```
void set_load_current(float target_current_amps)
{
    load_current_amps = load_current_Read();
    volatile float error = load_current_amps - target_current_amps; // error between measured current and target current

    /* Remain in while loop until load current = target current +/- 10 amps */
    while(fabs(error) > 5)
    {
        /* Poll the load current reading from the shunt */
        load_current_amps = load_current_Read();
        error = load_current_amps - target_current_amps;

        /* Turn knob CLOCK-WISE if load current is LESS than target value*/
        if (error <= 0)
        {
            A4988_dir_HIGH();
        }
        /* Turn knob COUNTER-CLOCK-WISE if load current is MORE than target value*/
        else if (error >= 0)
        {
            A4988_dir_LOW();
        }

        /* Rotate the knob by one step of the NEMA-17 on each iteration */
        A4988_step();
    }
}
```

Figure 2.13: Setting the Load Current Code (OM)

Section 2.3.3 Setting the Load Current to 0 A:

An additional function is available to set the load current to 0 A. The same algorithm is used with the load current set to 0 A. However, the stepper motor is instructed to complete one additional rotation to make sure that there is no current being drawn. This is because the control loop ends after the current falls within a tolerance range of +/- 5 A, therefore, an additional counterclockwise rotation is performed to ensure the current is 0 A. The function to set the load current to 0 A is shown below in figure 2.14 (OM).

```
void open_circuit_load(void)
{
    A4988_dir_LOW();    // rotate knob COUNTER-CLOCK-WISE
    load_current_amps = load_current_Read();

    /* Rotate knob until current is at minimum measurable value */
    while(load_current_amps > 5)
    {
        /* Poll the load current reading from the shunt */
        load_current_amps = load_current_Read();
        /* Rotate the knob by one step of the NEMA-17 on each iteration */
        A4988_step();
    }

    /* Complete one more rotation to ensure carbon pile is completely OFF */
    for(uint8_t i = 0; i < 200; i++) { A4988_step(); }
}
```

Figure 2.14: Setting the Load Current to 0 A Code (OM)

Section 3: Local Interface:

Section 3.1 Program Structure:

The local interface program is built around a top-level finite-state machine (FSM) that acts as a wrapper for the core FSMs managing the interface's functionality. The interface operates in four states:

- **Main Menu:** Initial state, Allows users to navigate through different options.
- **Testing:** Where loaded and unloaded tests are performed.
- **View History:** Where users can review results from previous tests.
- **Settings:** Where users can configure device settings. (OM)

The following code snippet in figure 3.1 shows how an enumerated type is used to represent the state of the top-level local interface FSM (OM).

```
/* Program states for the local interface fsm*/
typedef enum {
    MAIN_MENU_STATE,
    TEST_STATE,
    VIEW_HISTORY_STATE,
    SETTINGS_STATE
}  LOCAL_INTERFACE_FSM_STATES;
```

Figure 3.1: Local Interface FSM Type Enumeration (OM)

The following diagram in figure 3.2 serves to further illustrate the overall structure of the local interface software program.

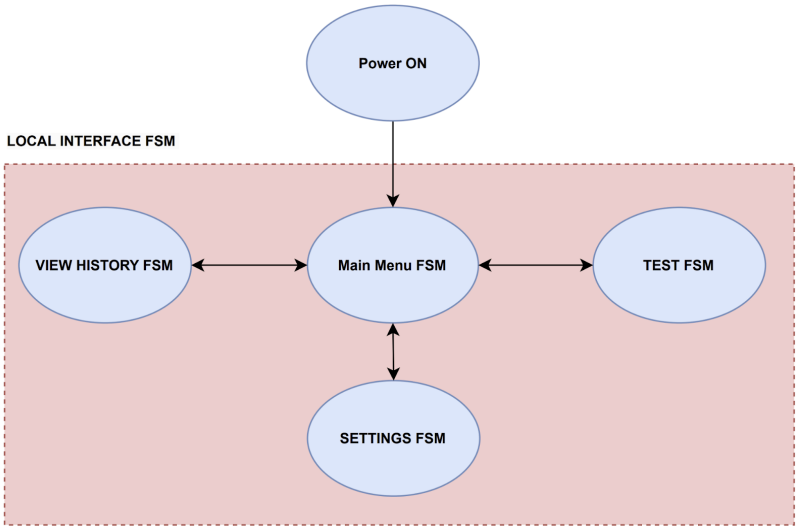

Figure 3.2: Program Overview (OM)

Each state has its own FSM to handle push button inputs and manage state transitions. The local interface sub-system is interrupt-driven, meaning its subroutines are executed only in response to a push-button press, leaving the CPU free to perform other tasks. The program's response to user input depends on both the wrapper FSM's state and the state of the lower-level FSM associated

with the current state of the local interface. The following switch statement in figure 3.3 summarizes how the wrapper FSM determines which FSM to use (OM).

```
switch (LOCAL_INTERFACE_CURRENT_STATE)
{
    case MAIN_MENU_STATE:
        main_menu_fsm(); // Enter main menu fsm
        break;
    case TEST_STATE:
        test_fsm(); // Enter test fsm
        break;
    case VIEW_HISTORY_STATE:
        view_history_fsm(); // Enter view history fsm
        break;
    /* Settings fsm handles pushbutton press */
    case SETTINGS_STATE:
        settings_fsm();
        break;
    default:
        // return to main menu state
        LOCAL_INTERFACE_CURRENT_STATE = MAIN_MENU_STATE;
        main_menu_fsm();
        break;
}
```

Figure 3.3: Local Interface Switch Statement Code (OM)

The modular and abstract design makes it easy to expand the system with new menu options and functions. We can simply add a new state to the enumerated state variable, program the corresponding FSM subroutine, and update the switch statement with a new case to call that subroutine (OM).

Section 3.2 Interrupt Driven State Transitions:

An enumerated type shown in figure 3.4 is used to create an identifier for which push button was pressed (OM).

```
/* Push Button Input Types */
typedef enum {
    OK,     // User pressed OK pushbutton
    BACK,   // User pressed BACK pushbutton
    UP,     // User pressed UP pushbutton
    DOWN,   // User pressed DOWN pushbutton
    NONE    // Neutral PB state to prevent FSM functions from acting on wrong PB press
}  PB_INPUT_TYPE;
```

Figure 3.4: Push Button Enumerated Type (OM)

The interrupt-service-routine (ISR) for each push-button GPIO pin passes the push button identifier to the top-level local interface wrapper FSM which then passes the identifier to the core FSM corresponding to the current state of the local interface. The core FSM can then call its own subroutines to handle the pushbutton input and initiate a state transition if required. Note that all FSMs except for the main menu FSM are Mealy state machines, therefore, the response depends on both the current state and which push button was pressed. This is shown in figure 3.5 (OM).

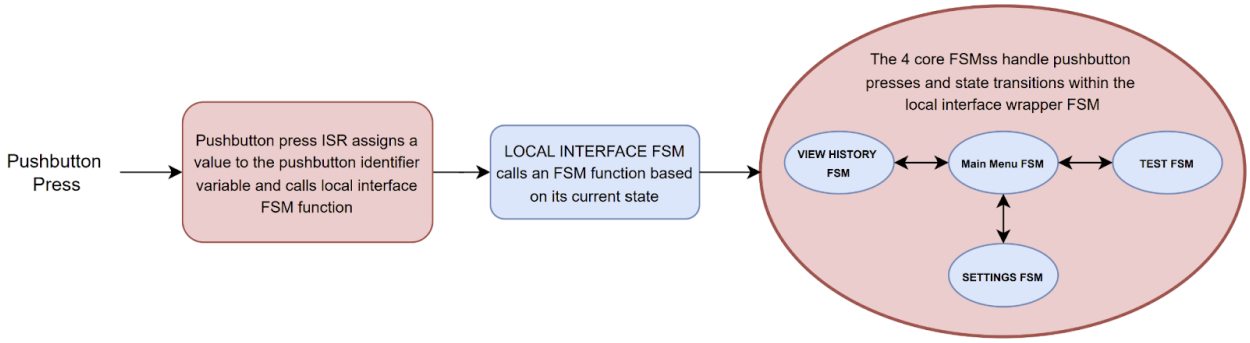

Figure 3.5: Interrupt Driven State Transitions (OM)

Section 3.3 Main Menu FSM:

The flowchart below in figure 3.6 shows the operation of the main menu FSM. It only has a single state that updates the cursor position or initiates a state transition based on the pushbutton input. The subroutine for the main menu FSM uses a switch statement to implement this flowchart in software (OM).

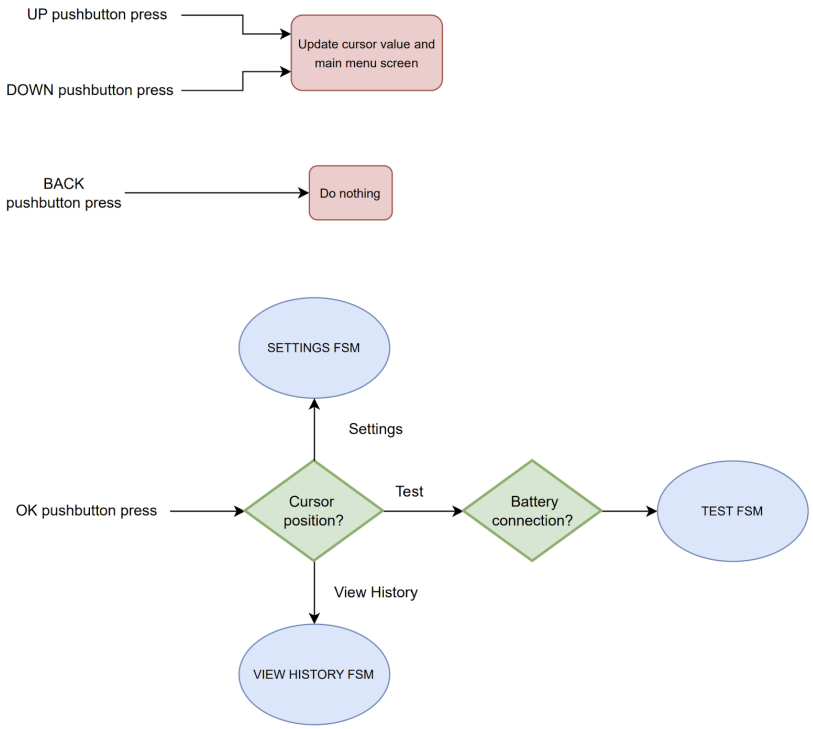

Figure 3.6: Main Menu FSM Flowchart (OM)

Additionally, the main menu FSM was edited to check if any of the battery cell voltages are below 3V while unloaded. The following code snippet in figure 3.7 summarizes the two classes of safety errors that can occur when a test is initiated (OM).

```
/* Error code types */
typedef enum {
    CONNECTION_ERROR,   // There are no battery cells connected
    SAFETY_ERROR        // A battery cell is below the minimum safety threshold
} ERROR_CODE_TYPES;
```

Figure 3.7: Safety Errors Code Snippet (OM)

If the total voltage across the quad-pack is measured to be below 0.1V, then a connection error is reported. If the voltage across any one of the four battery cells is below the minimum safety threshold voltage for testing, then a safety error is reported. The following code snippet in figure 3.8 summarizes the error checking procedure (OM).

```
void test_error_check(void)
{
    uint8_t error_flag = 0x00;  // Error flag, 0x01 -> At least one battery cell is below threshold
    /* Read unloaded battery pack voltages */
    float voltage = batteryCell_read(B4_ADC_CHANNEL, GND_ADC_CHANNEL);       // Total quad-pack voltage
    quad_pack_buffer[0] = batteryCell_read(B1_ADC_CHANNEL, GND_ADC_CHANNEL); // B1_POS - GND
    quad_pack_buffer[1] = batteryCell_read(B2_ADC_CHANNEL, B1_ADC_CHANNEL);  // B2_POS - B1_POS
    quad_pack_buffer[2] = batteryCell_read(B3_ADC_CHANNEL, B2_ADC_CHANNEL);  // B3_POS - B2_POS
    quad_pack_buffer[3] = batteryCell_read(B4_ADC_CHANNEL, B3_ADC_CHANNEL);  // B4_POS - B3_POS

    /* Check if any battery cells are unsafe to test */
    for (uint8_t i = 0; i < 4; i++)
    {
        if (quad_pack_buffer[i] < min_battery_voltage)
            error_flag = 0x01;  // At least one battery cell is below safety threshold
    }

    /* If voltage < 0.1V, no battery connection -> move to ERROR state */
    if (voltage > 20)   // For differential measurements MSB is '1' when negative voltage, therefore it reads as 20 when there is 0V across
    {
        TEST_CURRENT_STATE = ERROR;     // Move to ERROR state
        ERROR_CODE = CONNECTION_ERROR;  // Error code identifier
    }
    /* If any battery cells are below minimum safety threshold -> move to ERROR state */
    else if (error_flag == 0x01)
    {
        TEST_CURRENT_STATE = ERROR; // Move to ERROR state
        ERROR_CODE = SAFETY_ERROR;  // Error code identifier
    }
    /* Otherwise proceed with test */
    else
    {
        TEST_CURRENT_STATE = TESTING;   // Proceed with TEST
    }

    return;
}
```

Figure 3.8: Error Check Code Snippet (OM)

If the test enters the ERROR state then it displays the corresponding error message until the user presses the OK or BACK pushbuttons. The code snippet in figure 3.9 shows this (OM).

```
void display_error_message (PB_INPUT_TYPE pb_type)
{
    /* Return to main menu if OK or BACK PB is pressed */
    if (pb_type == OK || pb_type == BACK)
    {
        cursor = 1;     // Initialize cursor to line 1
        quad_pack_entry = 0;    // Initialize quad pack entry to quad pack 1
        LOCAL_INTERFACE_CURRENT_STATE = MAIN_MENU_STATE;
        display_main_menu();
    }
    /* Display Error message otherwise */
    else
    {
        switch (ERROR_CODE)
        {
            case CONNECTION_ERROR :
                display_connection_error();
                break;
            case SAFETY_ERROR :
                display_safety_error();
                break;
            default :
                asm volatile("nop");
                break;
        }
    }
}
```

Figure 3.9: Display Error Message Code Snippet (OM)

Section 3.4 View History FSM:

An enumerated type is used to represent all of the possible states that can occur within the view history FSM. The following code snippet in figure 3.10 summarizes the various states within the view history FSM as well as the action that is to be performed while in that state (OM).

```
/* States for the fsm that views previous results */
typedef enum {
    SCROLL_PREVIOUS_RESULTS,    // Scroll through quad pack entries where previous test results are saved
    SCROLL_TEST_RESULT_MENU_H,  // Menu to scroll through the data recorded during the test
    VOLTAGE_READINGS_H,         // Display LOADED (left) voltages and UNLOADED(right) voltages
    HEALTH_RATINGS_H,           // Display health ratings of battery cells
    TEST_CONDITIONS_H,          // Display conditions that the quad-pack was tested under
    DISCARD_RESULTS_H,          // Confirm that user would like to discard test results without saving
}  VIEW_HISTORY_FSM_STATES;
```

Figure 3.10: View History FSM Type Enumeration (OM)

The flowchart below in figure 3.11 shows the operation of the view history FSM. A switch statement is used to implement this flowchart in software (OM).

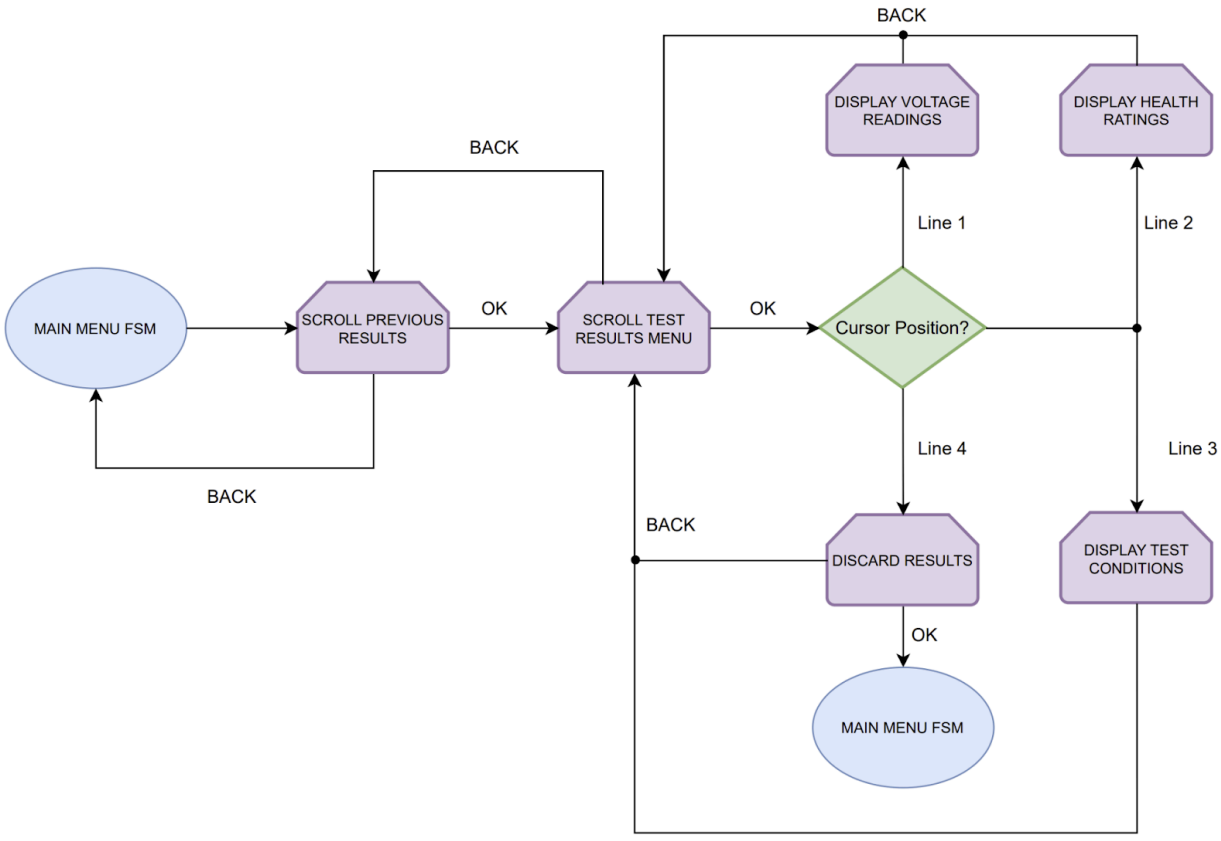

Figure 3.11: View History FSM Flowchart (OM)

Note that this flowchart is merely a state transition diagram for the view history FSM. Each state in this FSM calls various subroutines to perform a given task (OM).

Section 3.5 Test FSM:

An enumerated type is used to represent all of the possible states that can occur within the test FSM. The following code snippet in figure 3.12 summarizes the various states within the test FSM as well as the action that is to be performed while in that state (OM).

```
/* States for the fsm that performs the test procedure */
typedef enum {
    ERROR,                      // Test attempted while no battery pack was connected
    TESTING,                    // Perform LOADED and UNLOADED tests
    SCROLL_TEST_RESULT_MENU_T,  // Menu to scroll through the data recorded during the test
    VOLTAGE_READINGS_T,         // Display LOADED (left) voltages and UNLOADED(right) voltages
    HEALTH_RATINGS_T,           // Display health ratings of battery cells
    TEST_CONDITIONS_T,          // Display conditions that the quad-pack was tested under
    DISCARD_RESULTS_T,          // Confirm that user would like to discard test results without saving
    SAVE_CURRENT_RESULTS,       // Confirm that user would like to save current test results
    SCROLL_SAVE_ENTRIES,        // Scroll through quad pack entries to save current test results
    OVERWRITE_RESULTS           // Confirm that user would like to overwrite previous test results
}  TEST_FSM_STATES;
```

Figure 3.12: Test FSM Type Enumeration (OM)

The flowchart below in figure 3.13 shows the operation of the view history FSM. A switch statement is used to implement this flowchart in software (OM).

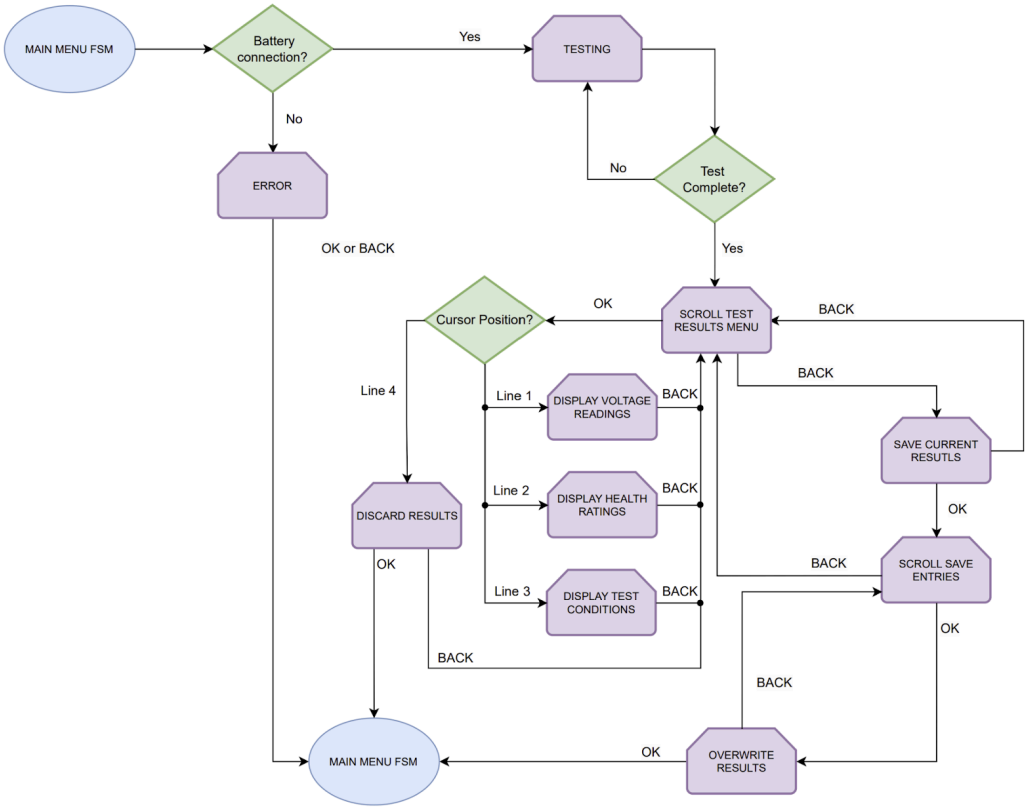

Figure 3.13: Test FSM Flowchart (OM)

Note that this flowchart is merely a state transition diagram for the test FSM. Each state in this FSM calls various subroutines to perform a given task (OM).

Section 3.6 Settings FSM:

An enumerated type is used to represent all of the possible states that can occur within the settings FSM. The following code snippet in figure 3.14 summarizes the various states within the settings FSM as well as the action that is to be performed while in that state (OM).

```
/* States for the fsm that displays the settings menu */
typedef enum {
    SCROLL_SETTINGS,                // Scroll through the settings menu
    LOAD_CURRENT_SETTINGS_SCREEN,   // Adjust the load current value used for automated tests
    VOLTAGE_PRECISION_SETTINGS_SCREEN   // Adjust the number of decimal points for voltage measurements
}  SETTINGS_FSM_STATES;
```

Figure 3.14: Settings FSM Type Enumeration (OM)

The flowchart below in figure 3.15 shows the operation of the settings FSM. A switch statement is used to implement this flowchart in software (OM).

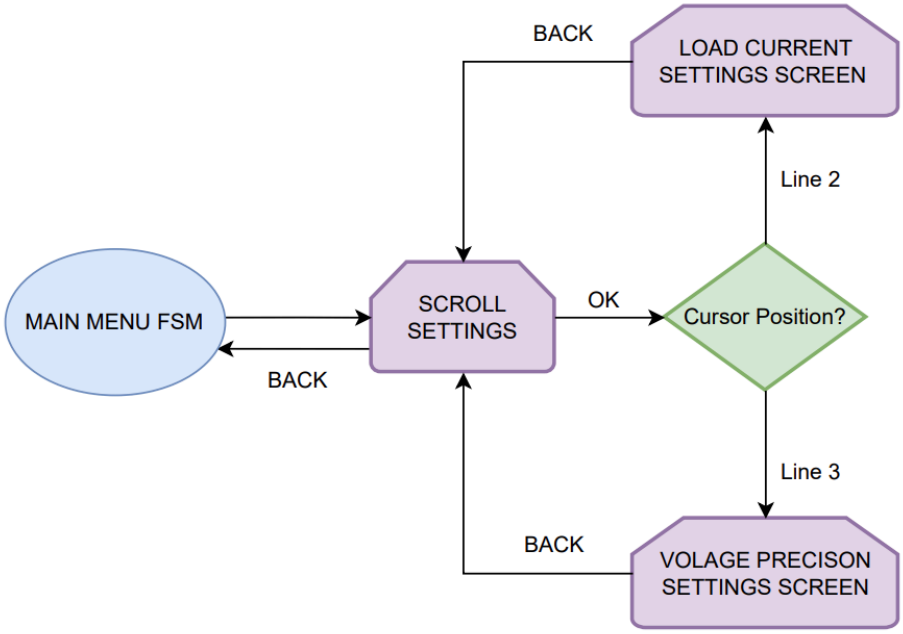

Figure 3.15: Settings FSM Flowchart (OM)

Note that this flowchart is merely a state transition diagram for the view history FSM. Each state in this FSM calls various subroutines to perform a given task (OM).

Section 3.7 Cursor Position:

The cursor's position is stored as an integer variable that can take values from 1 to 4. Moving the cursor UP decrements this variable, while moving it DOWN increments it. The cursor cannot be incremented beyond 4 because line 4 is the bottom line of the display, and it cannot be decremented below 1 because line 1 is the topmost line (OM).

Section 3.8 Storing Test Results:

The test data for a completed quad pack test is stored in a struct containing the loaded voltages, unloaded voltages, load current, test mode (manual or automated), ambient temperature, and the date that the test was performed. The struct is shown below in figure 3.16 (OM).

```
/* Structure for storing all of the data associated with a completed quad pack test. */
typedef struct {
    float UNLOADED_battery_voltages[4];     // UNLOADED Battery cell voltages         : 4 floats = 16 bytes
    float LOADED_battery_voltages[4];       // LOADED Battery cell voltages           : 4 floats = 16 bytes
    uint16_t max_load_current;  // Max load current used to test battery              : 2 bytes
    uint8_t test_mode;          // 0x00 -> Manual test, 0x01 -> Automated test        : 1 byte
    uint8_t ampient_temp;       // Ambient temperature during test in degrees Celsius : 1 bytes
    uint8_t year, month, day;   // 20xx, 0-12, 0-31                                   : 3 bytes
    // SIZE = 16 + 16 + 2 + 1 + 1 + 3 = 39 bytes                              Total : 39 bytes
} test_result;
```

Figure 3.16: Storing Test Results Struct (OM)

The history of the last 13 quad pack tests is stored in a 13 element array of test_result structs that is stored in the microcontroller's internal EEPROM storage. This is shown below in figure 3.17 (OM).

```
/* Data log of 13 previous quad-pack tests, stored in MCU's internal EEPROM storage */
extern test_result EEMEM test_results_history_eeprom[13];   // 507/512 bytes of available EEPROM
volatile test_result current_test_result;   // data from most recent quad-pack test
```

Figure 3.17: Storing Test Results Variable Declarations (OM)

The quad_pack_entry variable holds the array index of the test results that will be displayed when the user presses OK. Pressing the DOWN button increments this variable, while pressing the UP button decrements it. If quad_pack_entry is incremented beyond 12 (the 13th element), it wraps around to 0 (the 1st element). Similarly, if decremented beyond 0, it wraps around to 12. This behavior makes it function as a circular counter. When the user presses OK, the row indexed by this variable in both 2D arrays is displayed on the LCD (OM).

Section 3.9 Scrolling Through Quad Pack Entries:

To enable the cursor to scroll through quad packs, the current quad pack entry must first be displayed on the cursor's current line. This value serves as the base quad pack entry, from which the number of entries above and below the cursor is determined. A while loop then displays the value of quad_pack_entries plus its line offset for rows below the cursor, while another while loop displays the value of quad_pack_entries minus its line offset for rows above the cursor. The code for this is shown below in figure 3.18 (OM).

```
void display_quad_pack_entries(void)
{
    clear_lcd();

    /* display cursor on same line as quad pack entry */
    if(quad_pack_entry + 1 >= 10)
        sprintf(dsp_buff[cursor - 1], "Quad pack %d   <-     ", quad_pack_entry + 1);
    else
        sprintf(dsp_buff[cursor - 1], "Quad pack %d   <-     ", quad_pack_entry + 1);

    uint8_t entries_above_cursor = cursor - 1;  // number of quad pack entries to be displayed above the cursor line
    uint8_t entries_below_cursor = 4 - cursor;  // number of quad pack entries to be displayed below the cursor line
    int8_t quad_pack_display;   // quad_pack_entry number that will appear on the current line of the display

    /* Start on line 4 and display all entries BELOW the entry pointed to by the cursor */
    while (entries_below_cursor != 0)
    {
        quad_pack_display = quad_pack_entry + entries_below_cursor + 1; // quad pack entry number on each line BELOW cursor

        if (quad_pack_display < 1)
            quad_pack_display += 13;    // If value is <= 0, add 13 to ensure that display number 'rolls over' circularly
        if (quad_pack_display > 13)
            quad_pack_display -= 13;    // If value is >= 14, subtract 13 to ensure that display number 'rolls over' circularly

        if(quad_pack_display >= 10)
            sprintf(dsp_buff[cursor + entries_below_cursor - 1], "Quad pack %d        ", quad_pack_display);
        else
            sprintf(dsp_buff[cursor + entries_below_cursor - 1], "Quad pack %d         ", quad_pack_display);
        entries_below_cursor--; // move up 1 line until the cursor line is reached
    }

    /* Start on line 1 and display all entries ABOVE the entry pointed to by the cursor */
    while (entries_above_cursor != 0)
    {
        quad_pack_display = quad_pack_entry - entries_above_cursor + 1; // quad pack entry number on each line ABOVE cursor

        if (quad_pack_display < 1)
            quad_pack_display += 13;    // If value is <= 0, add 13 to ensure that display number 'rolls over' circularly
        if (quad_pack_display > 13)
            quad_pack_display -= 13;    // If value is >= 14, subtract 13 to ensure that display number 'rolls over' circularly

        if(quad_pack_display >= 10)
            sprintf(dsp_buff[cursor - entries_above_cursor - 1], "Quad pack %d        ", quad_pack_display);
        else
            sprintf(dsp_buff[cursor - entries_above_cursor - 1], "Quad pack %d         ", quad_pack_display);
        entries_above_cursor--; // move down 1 line until the cursor line is reached
    }

    update_lcd();
}
```

Figure 3.18: Scrolling Through Quad Pack Entries Code (OM)

Section 3.10 Assigning Health Ratings

Each battery health rating is represented by a two-character string stored in a look-up table (LUT) in memory. The array indexes are arranged in descending order, with the highest ratings at the lower indexes and the lowest ratings at the higher indexes. This is shown below in figure 3.19 (OM).

```
volatile char health_rating_lut[13][2] = {
    "A+", "A ", "A-",       // 0x00, 0x01
    "B+", "B ", "B-",       // 0x02, 0x03, 0x04
    "C+", "C ", "C-",       // 0x05, 0x06, 0x07, 0x08
    "D+", "D ", "D-",       // 0x09, 0x0A, 0x0B
    "F "                    // 0x0C
};
```

Figure 3.19: Health Ratings Lookup Table (OM)

To calculate the health rating of each battery cell in the quad pack and determine the corresponding look up table index, the following algorithm is used:

A temporary threshold variable is initialized to the voltage corresponding to the highest possible health rating. Additionally a look-up table (lut) index variable is initialized to the index of the highest possible health rating. threshold and LUT index to the highest possible rating. A while loop then continuously decrements the threshold variable and increments the LUT index until it is less than the loaded voltage or the minimum possible health rating. Increasing the LUT index decreases the health rating because the health rating characters are stored in descending order with the highest health rating at the first index and the lowest health rating at the last index. Once the health rating is determined, the string corresponding to the LUT index is copied into a buffer array before being moved to the display buffer array. The process is repeated four times, once to determine the health rating of each battery cell. The code for this is shown below as figure 3.20 (OM).

```
void decode_health_rating(test_result result)
{
    uint8_t lut_idx = 0;    // index to lut containing health rating strings
    float rating_threshold = 2.9;   // minimum threshold for A = 2.9

    /* Determine health rating of all 4 battery cells in the quad pack */
    for (uint8_t i = 0; i < 4; i++)     // outer for loop, 4 battery cells
    {
        /* Initialize threshold and lut index for each iteration */
        rating_threshold = 2.9;
        lut_idx = 0;
        /* Loop until threshold falls below 1.69, F, or loaded voltage no longer meets minimum threshold */
        while ( (rating_threshold > 1.69) && (rating_threshold >= result.LOADED_battery_voltages[i]) )
        {
            /* Compare loaded voltage with threshold to determine health rating */
            if (result.LOADED_battery_voltages[i] < rating_threshold)
            {
                lut_idx++;  // Incrementing lut index lowers rating
                rating_threshold -= 0.1;    // lower threshold to compare with a lower health rating
            }
        }

        /* Copy health rating strings from look-up table into buffer array */
        for (uint8_t j = 0; j < 2; j++) // inner for loop, 3 characters per health rating
        {
            health_rating_characters[(2*i) + j] = health_rating_lut[lut_idx][j];
        }
    }
}
```

Figure 3.20: Determine Health Rating Code (OM)

Section 3.11 Load Current BCD Conversion:

The load current is programmed in the settings menu by choosing the binary code decimal (BCD) digits individually, therefore, only integer values of current can be programmed. Once the user programs the load current digits the weighted BCD values are summed into the actual integer current value used during testing. Additionally, if statements were added to ensure that the current limit for manual testing does not exceed 500 A and the current limit for automated testing does not exceed 200 A. The code for this is shown below in figure 3.21 (OM).

```
/* bcd conversion to save current setting before returning to settings menu */
current_setting = (100*current_setting_100_dig) + (10*current_setting_10_dig) + (1*current_setting_1_dig);
```

Figure 3.21: Load Current BCD Conversion Code (OM)

Section 4 Remote Interface:

The remote interface consists of two separate software components. The PC-Side software (written in Python) creates and manages the GUI and facilitates data transfer between the microcontroller and PC. All PC-Side software is contained in the file bla_remote_interface.py. The Embedded software (written in C) does the actual data processing based on commands from the PC-side software and then transfers data back to the GUI. Most of the embedded software is contained in the file remote_interface.c with some references to functions in the adc.c, lcd.c, and stepper_motor.c files to perform various background functions. The GUI and microcontroller communicate using UART. A USB port connects the PC to the CP2102 UART bridge chip, which then connects to the AVR128DB48's USART3 module. A block diagram showing the various components of the remote interface is shown below in figure 4.1. More details about the circuitry of each component can be found in the Hardware Description Document (TS).

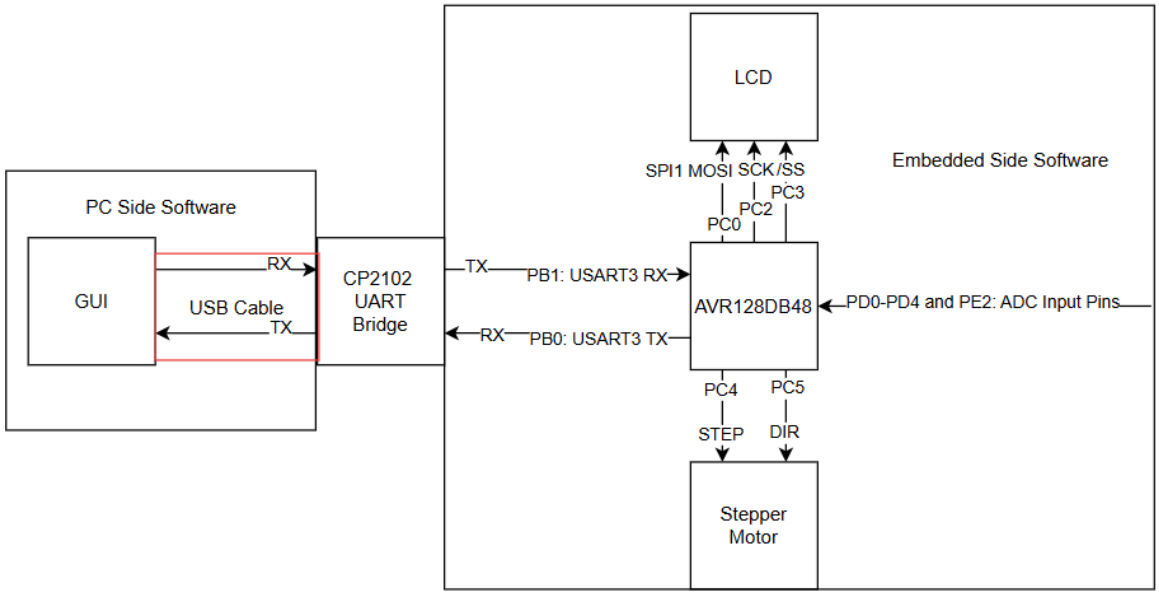

Figure 4.1: Remote Interface Block Diagram (TS)

Section 4.1 PC-Side Software:
The PC-Side software is a mostly sequential program that makes decisions based on user input. The PC-Side software was divided into subcomponents, each of which is described below in detail (TS).

Section 4.1.1 Initialization and Main Program Code:
The initialization code initializes the GUI and serial interface, as described in their respective sections below.

Additionally, several global variables are declared so they are easily accessible throughout the program. One of the variables declared is an IntVar with a default value of 0 used to store the value of radio buttons. Radio buttons are used by the GUI for parameterized user input. An integer variable is used to create multiple threads with different numbers being assigned to each thread. Two arrays of type string are used to store the list of available COM ports on the computer and the list of battery names. The other 4 variables declared are arrays used to store the results received from the microcontroller. The unloaded and loaded voltage arrays are of type float, the current array is of type int, and the health rating arrays are of type string. The global variable declarations are shown below in figure 4.2.

```
v = IntVar(value = 0) #variable to store values from radio buttons
unloaded = [] #arrays to store results from mc
loaded = []
health = []
current = []
ports = [] #array to store COM ports
battery_names = [] #array to store battery names
j = 0 #global variable to store thread numbers
```

Figure 4.2: Global Variable Declarations (TS)

After declaring the global variables the program calls the function main_menu which displays the main menu screen. Finally, root.main_loop() is called, which keeps the program and all GUI functions active until it is closed by the user. This is similar to an infinite while loop in Embedded C (TS).

The first screen displayed to the user is the main menu, which provides options for each of the different functions. Each of these functions is described in the subsequent sections. However, there is also a “Quit” button which calls root.destroy(), which destroys the root and prevents further execution of the program. This is used when the program is closed. A flowchart of the initialization and main program code is shown below in figure 4.3 (TS).

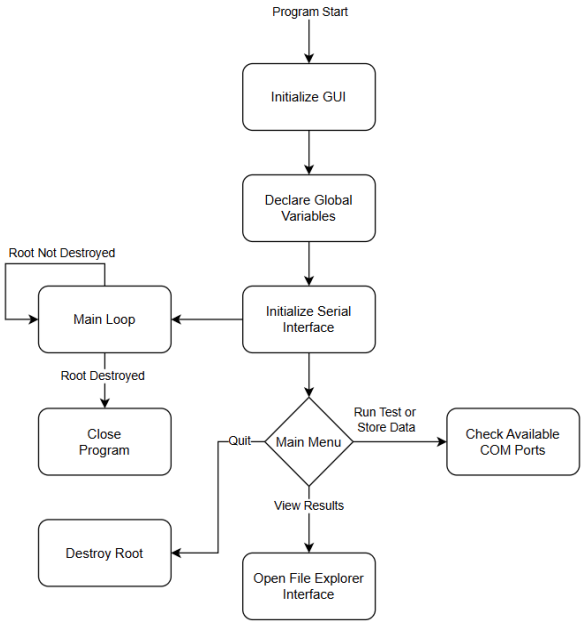

Figure 4.3: Initialization and Main Program Code Flowchart (TS)

Section 4.1.2 Graphical User Interface (GUI):

The GUI is created using Python's tkinter library. To initialize the GUI, a Tk object is instantiated. Then, a window frame is created with a size of 250 pixels. The window is created as a grid, where items are displayed based on their row and column indices. Row 0 and Column 0 is at the center of the window, with row indices increasing downwards and column indices increasing to the right. The initialization code is shown below in figure 4.4.

```
root = Tk() #create GUI window
root.title("Battery Load Analyzer Remote Interface") #name window
frm = ttk.Frame(root, padding=250) #create frame of size 250
frm.grid() #create grid on frame
```

Figure 4.4: GUI Initialization Code (TS)

This program uses four elements in GUI windows: labels, buttons, radio buttons, and entries. Each element is specified as being placed in the previously created window "frm." Labels are lines of text specified as a String. Buttons are used to provide user input, with each button calling a different function specified in the command parameter. These are used when each option will call a different function. Radio buttons are similar to buttons but instead of calling a function, they are used to place the value specified in the value parameter into the variable specified in the variable parameter. These are used to provide a parameter for the next function to be called. Entries are text boxes used to get input from the user, which is saved in a string variable. Each element is placed in a specific location on the grid specified by the row and column index. Every time the window needs to be changed, all of the previous elements (known as widgets in tkinter) need to be individually destroyed before new elements are placed. An example of updating the GUI window is shown below in figure 4.5 (TS).

```
for widget in frm.winfo_children(): #clear GUI window
        widget.destroy()
ttk.Label(frm, text = "Which COM port would you like to connect to?", font = ('Arial', 15)).grid(column = 0, row = 0) #display text
for j in range(len(ports)):
    ttk.Radiobutton(frm, text = ports[j], variable = v, value = j).grid(column = 0, row = j + 1) #display available COM ports as ra
ttk.Button(frm, text = "Continue", command = connect_com_ports).grid(column = 0, row = j + 2) #connect to COM port
ttk.Button(frm, text = "Cancel", command = main_menu).grid(column = 0, row = j + 3) #return to main menu
```

Figure 4.5: Updating the GUI Window Code (TS)

Section 4.1.3 Serial Interface:

The serial interface uses the Pyserial library. First, an instance of a serial port is declared in the main method (TS).

Then, the program must check all available serial ports on the computer for use and display them to the user. A function called get_serial_ports() does this and is shown in section 4.6. The flowchart of get_serial_ports() is shown below in figure 4.7 (TS).

```
def get_serial_ports(): #get list of COM ports that are currently connected
    global ports #use global variables
    if sys.platform.startswith('win'): #get all of PC's COM ports
        ports = ['COM%s' % (i + 1) for i in range(256)]


    result = [] #create array to store COM ports that are currently connected
    for port in ports: #check each of the PC's COM ports
        try: #add ports to array that can be connected to
            s = serial.Serial(port) #open port
            s.close() #close port
            result.append(port) #add port to array
        except (OSError, serial.SerialException): #skip ports that return an error message or are not connected
            pass
    return result #return array of COM ports that are currently connected
```

Figure 4.6: get_serial_ports() function (TS)

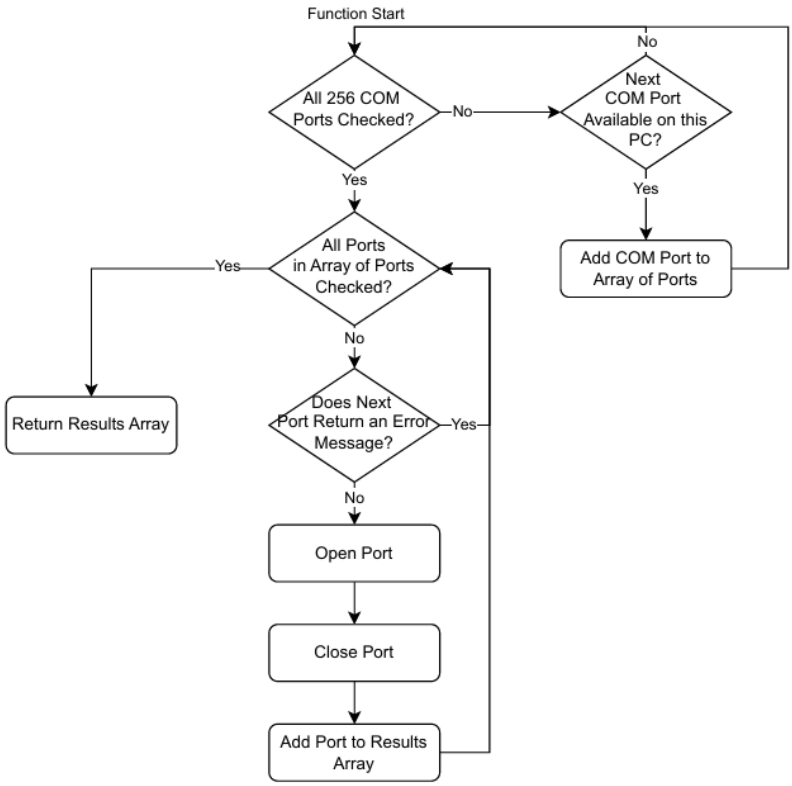

Figure 4.7: Function get_serial_ports() Flowchart (TS)

Once the user selects a COM port, the serial port instance is assigned to that specific COM port. The Baud Rate is set to 9600 and the COM port is opened (TS).

Once the COM port is opened, UART can be used to communicate between the microcontroller and GUI. The read() and write() functions will be used to transmit and receive respectively. Pyserial only supports the reading and writing of bytes, so data must be explicitly written in bytes in the utf-8 format as shown in figure 4.9 or encoded as shown in figure 4.10. Data read must be decoded as shown in figure 4.11 (TS).

```
ser.write(b'1')
```

Figure 4.8: Writing to the Serial Interface Using Explicit Bytes (TS)

```
ser.write(bytes(str(x), "utf-8"))
```

Figure 4.9: Writing to the Serial Interface Using Bytes Encoded from Strings (TS)

```
str(unloaded[0].decode(encoding = 'utf-8'))
```

Figure 4.10: Decoding Bytes Read from the Serial Interface (TS)

The serial port is closed after the program returns to the main menu. This ensures that the serial port is always closed when the user is finished with the program or when they wish to switch to another COM port. Figure 4.11 shows the COM port flowchart (TS).

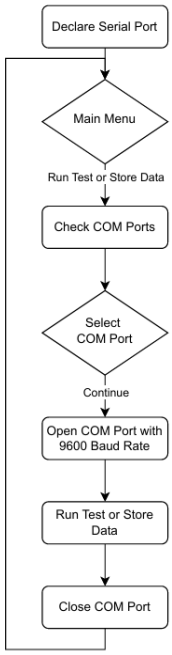

Figure 4.11: COM Port Flowchart (TS)

A character system is used by the GUI to communicate with the microcontroller. Each character received or transmitted has a specific meaning or task associated with it. The system of characters sent by the GUI to the microcontroller is detailed in table 4.1 while the system of characters sent by the microcontroller to the GUI is detailed in table 4.2 (TS).

Table 4.1: Characters Sent From the GUI to the Microcontroller and their Meaning (TS)

Character Sent from PC to MCU (case sensitive)	Meaning
‘u’	Perform an Unloaded Test
‘m’ followed by three numbers ‘0’-’9’ (max 500)	Perform a Loaded Manual Test with the Specified Current (first digit is hundreds place, second is tens place, third is ones place)
‘a’ followed by three numbers ‘0’-’9’ (max 500)	Perform a Loaded Automatic Test with the Specified Current (first digit is hundreds place, second is tens place, third is ones place)
‘c’	Cancel Loaded Manual Test
‘e’	Cancel Loaded Automatic Test
‘r’	Send Results From Just Performed Test to PC
‘1’ followed by a number ‘0’-’3’	Retrieve Data From EEPROM Quad Pack 10-13 (the first number represents the tens place of the quad pack number and the second number represents the ones place of the quad pack number)
‘0’ followed by a number ‘1’-’9’	Retrieve Data From EEPROM Quad Pack 1-9 (the first number represents the tens place of the quad pack number and the second number represents the ones place of the quad pack number)

Table 4.2: Characters Sent From the Microcontroller to the GUI and their Meaning (TS)

Character Sent from MCU to PC (case sensitive)	Meaning
‘u’	Unloaded Voltages are Being Sent Back
‘l’	Loaded Voltages are Being Sent Back
‘h’	Health Ratings are Being Sent Back
‘c’	Current is Being Sent Back
‘d’	Unloaded Test Done
‘i’	Turn Down Current
‘f’	Manual Loaded Test Done or Test Cancelled Successfully
‘a’	Automatic Loaded Test Done or Test Cancelled Successfully
‘v’	Unloaded Voltages are Low and Results will be Sent to the Remote Interface
‘e’	Battery Not Connected

Section 4.1.4 Running Tests:

The flowchart in figure 4.12 is used to run tests. Note that the character system described in Tables 4.1 and 4.2 of Section 4.1.3 is used to communicate between the microcontroller and the GUI.

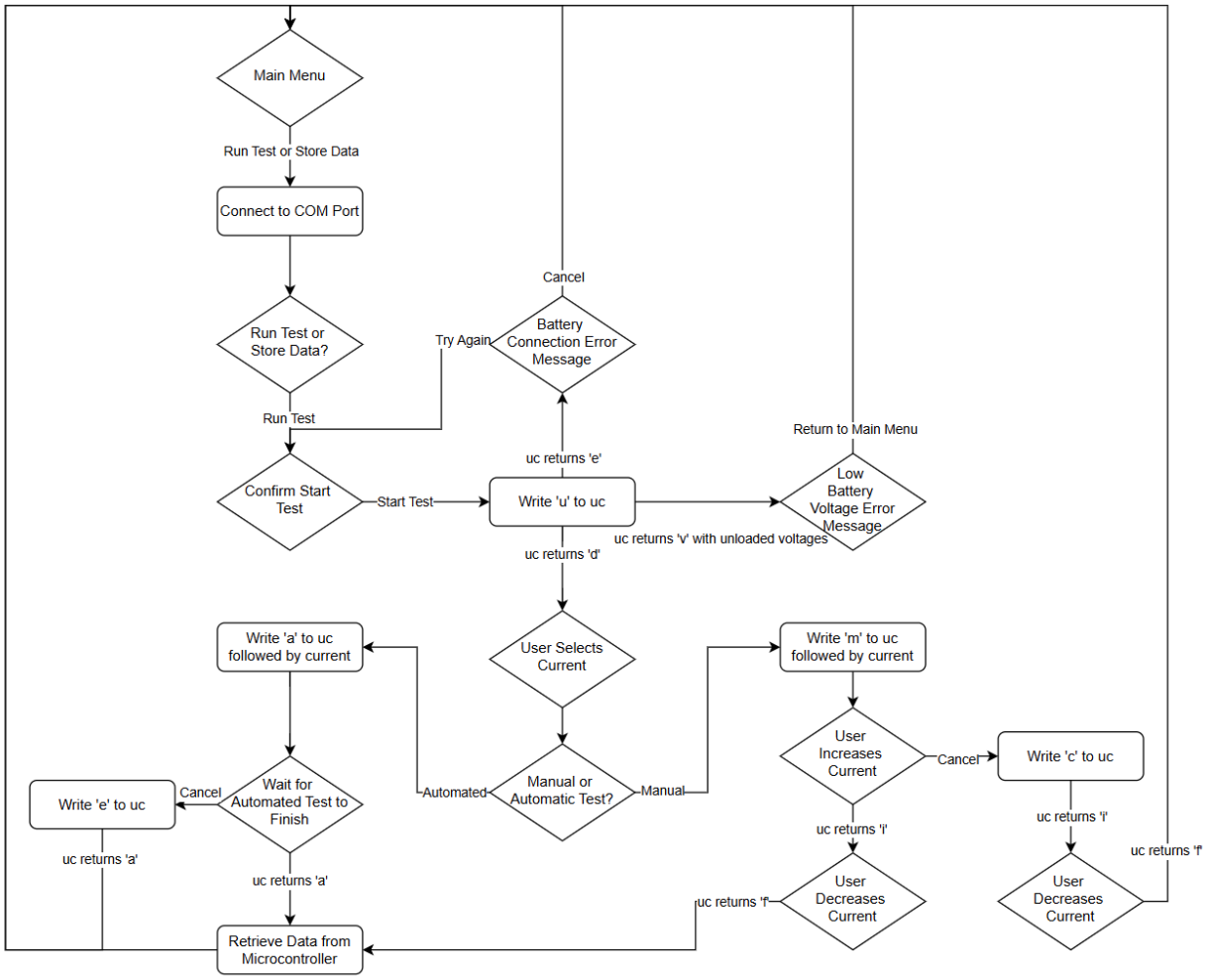

Figure 4.12: Running Tests Flowchart (TS)

The ser.read() function blocks the program from continuing until a character arrives in the serial port from the microcontroller. This is similar to polling on a microcontroller. Threads are used during the automated and manual loading tests so multiple tasks can be performed simultaneously. A distinct integer variable is passed as a parameter to each thread so each thread has a unique number allowing multiple threads to be created while the program is running. The main program code displays the GUI window and checks to see if the user wants to cancel the test in case the system malfunctions. The thread waits for the microcontroller to send the ‘f’ (manual test) or ‘a’ (automated test) indicating the loaded test is complete and then calls another function to continue the execution of the program. The thread code is shown below in figures 4.13 and 4.14 (TS).

```
thread = threading.Thread(target = automated_loaded_test_waiting_thread, args = (str(j))) #start thread to wait for uc to send acknowledgement character
thread.start() #start thread
j = j + 1 #increment thread number
```

Figure 4.13: Automated Test Thread Instantiation Code (TS)

```
def automated_loaded_test_waiting_thread(name): #thread to wait for automated loaded test to finish
    ser.read() #wait for 'a' from uc - automated test cancelled
    confirm_store_data() #go to retrieve data from uc
```

Figure 4.14: Automated Test Thread Code (TS)

Section 4.1.5 Retrieving Past Results from the Local Interface:

The flowchart in figure 4.15 describes the process of retrieving past results from the local interface. The quad pack slot to receive results from must be sent to the microcontroller so it knows which EEPROM slot to read from. The microcontroller and GUI communicate using the system described in Tables 4.1 and 4.2 of Section 4.1.3 (TS).

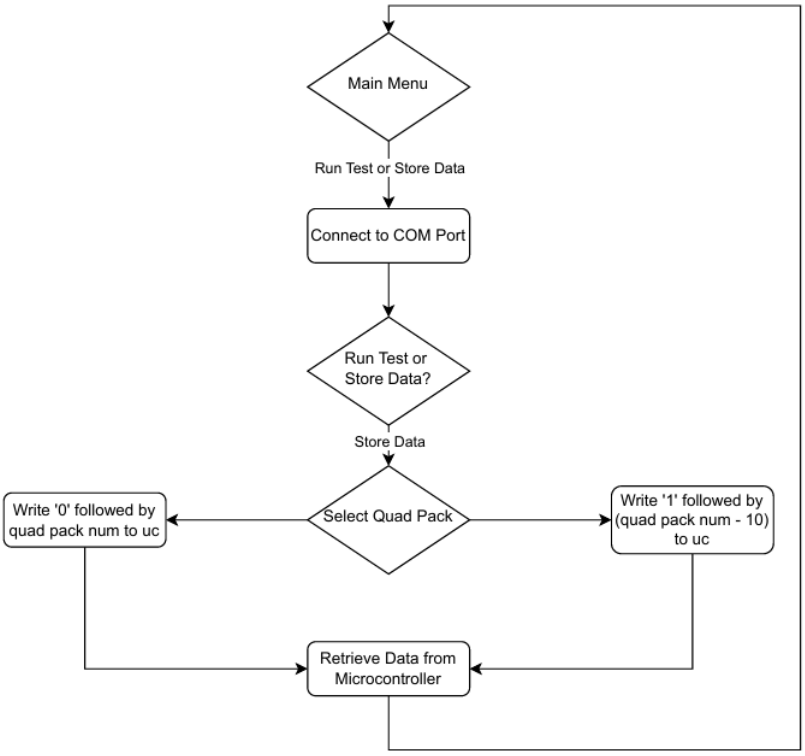

Figure 4.15: Retrieving Past Results Flowchart (TS)

Section 4.1.6 Receiving Results:

The code and flowchart to receive results from the microcontroller is shown below in figures 4.16 and 4.17 respectively. The character system from tables 4.1 and 4.2 of section 4.1.3 is used here. The data is sent in the following order: unloaded voltages, loaded voltages, health ratings, current. The GUI waits until all characters associated with a specific value have arrived, then reads all characters from the UART RX Buffer and adds them to the back of the array associated with that value (TS).

```
def receive_test_results(): #receive all test results from uc
    global unloaded #use global variables
    global loaded
    global health
    global current
    ser.write(b'r') #send 'r' to get results from uc
    ser.read() #get 'u' from uc - indicates that unloaded voltages are being sent from mcu


    while len(unloaded) < 4: #wait until all 4 unloaded voltages have been read
        if ser.in_waiting >= 5:  #wait for 5 characters of each voltage to be available
            unloaded.append(ser.read(5))    #add the next voltage value to the array


    ser.read() #get 'l' from uc - loaded being sent from uc

    while len(loaded) < 4: #wait until all 4 loaded voltages have been read
        if ser.in_waiting >= 5 : #wait for 5 characters of each voltage to be available
            loaded.append(ser.read(5)) #add the next voltage value to the array


    ser.read() #get 'h' from mcu- health rating being sent from mcu


    while len(health) < 4: #wait until all 4 health ratings have been read
        if ser.in_waiting >= 2: #wait for 2 characters of each health rating to be available
            health.append(ser.read(2)) #add the next voltage value to the array

    ser.read() #get 'c' from uc- current being sent

    while len(current) < 1: #wait until current has been read
        if ser.in_waiting >= 3:  #wait for 3 characters of current to be read
            current.append(ser.read(3)) #add the current to the array
```

Figure 4.16: Receiving Results Code (TS)

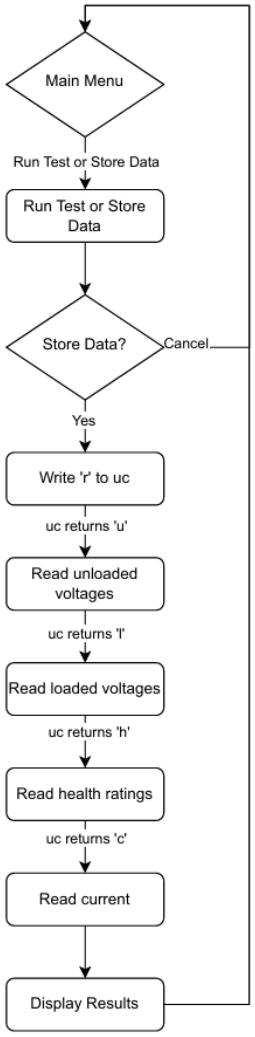

Figure 4.17: Receiving Results Flowchart (TS)

Section 4.1.7 Displaying and Saving Results:

The flowchart that outlines the procedure to display and save results is shown below in figure 4.18. The user is given the option to save or discard results. Results are saved as a .txt file using a Windows File Explorer Interface. To discard the results means not to save them. When the program returns to the main method, all data arrays are cleared, meaning the data is permanently lost (TS).

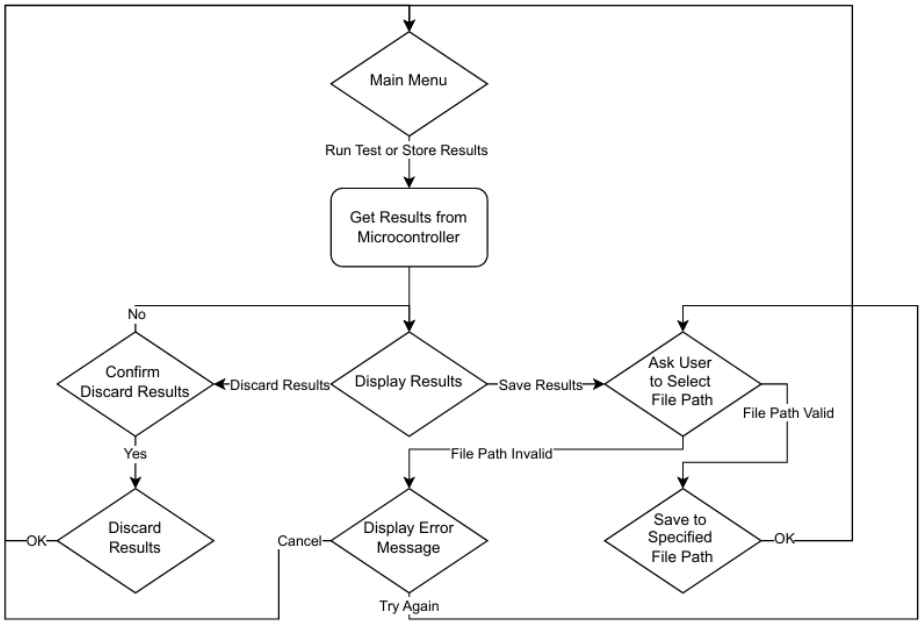

Figure 4.18: Displaying and Saving Results Flowchart (TS)

Section 4.1.8 Viewing Past Results Stored on the Remote Interface:
The flowchart to view past results is shown below in figure 4.19. Similarly to saving results, viewing results uses a Windows File Explorer Interface window but for opening files instead of saving them (TS).

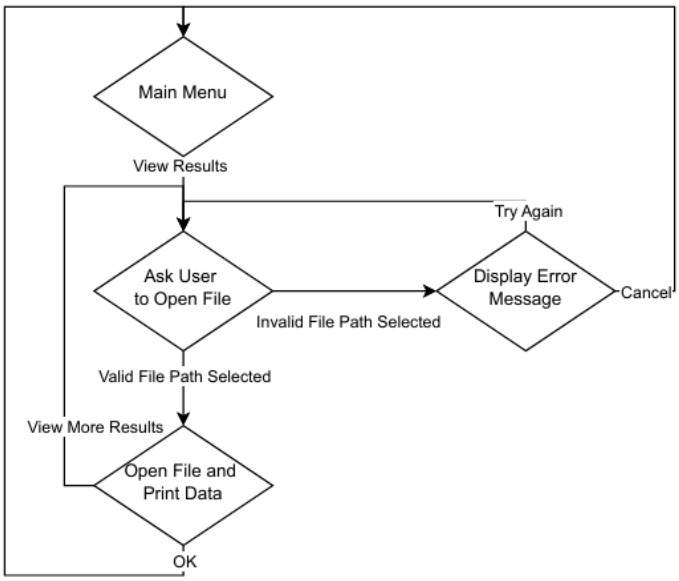

Figure 4.19: Viewing Results Flowchart (TS)

Section 4.2 Embedded Software:

This content pertains to the flowcharts shown in figures 4.20 and 4.21 at the embedded level (JL).

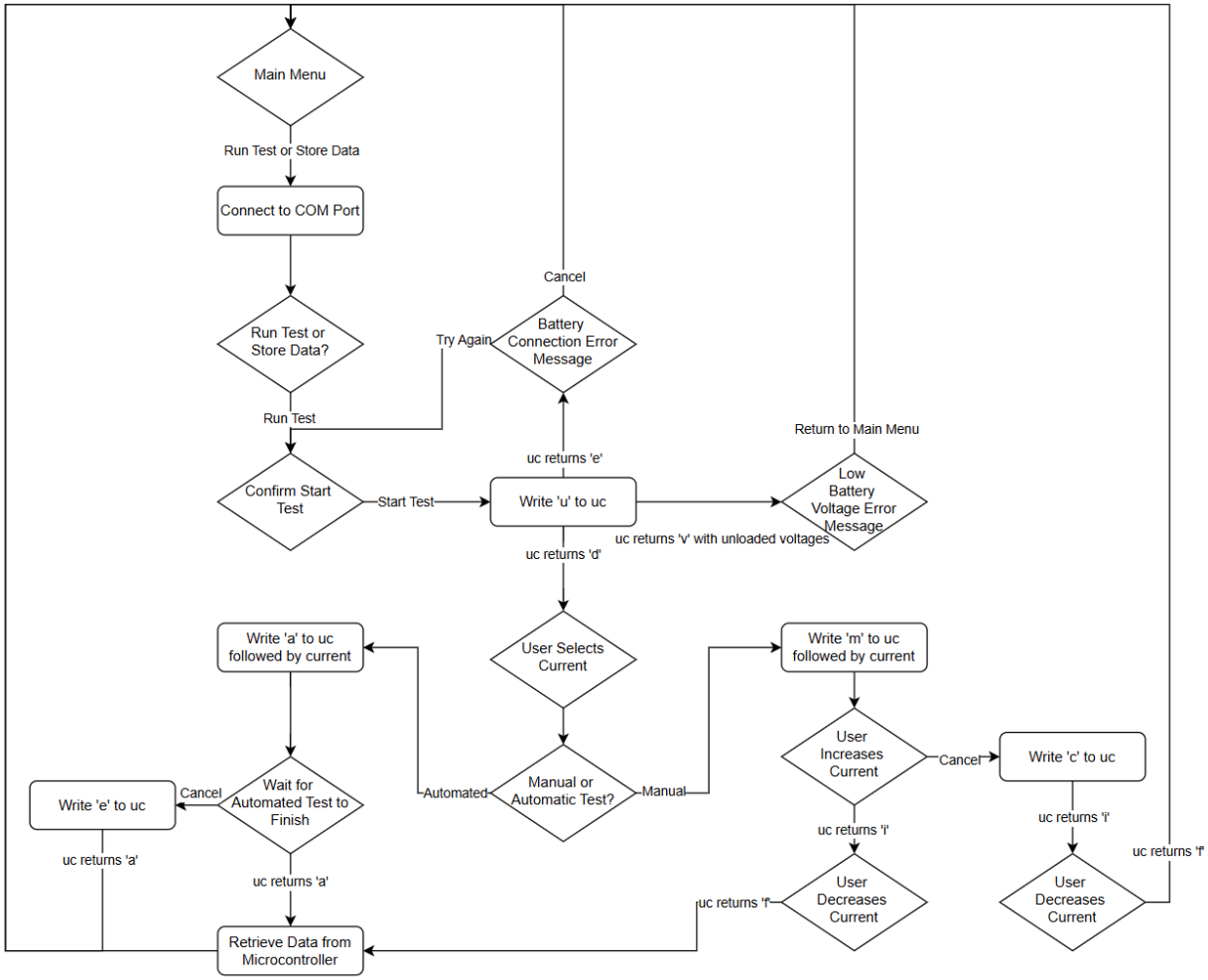

Figure 4.20: Running Tests Flowchart (JL)

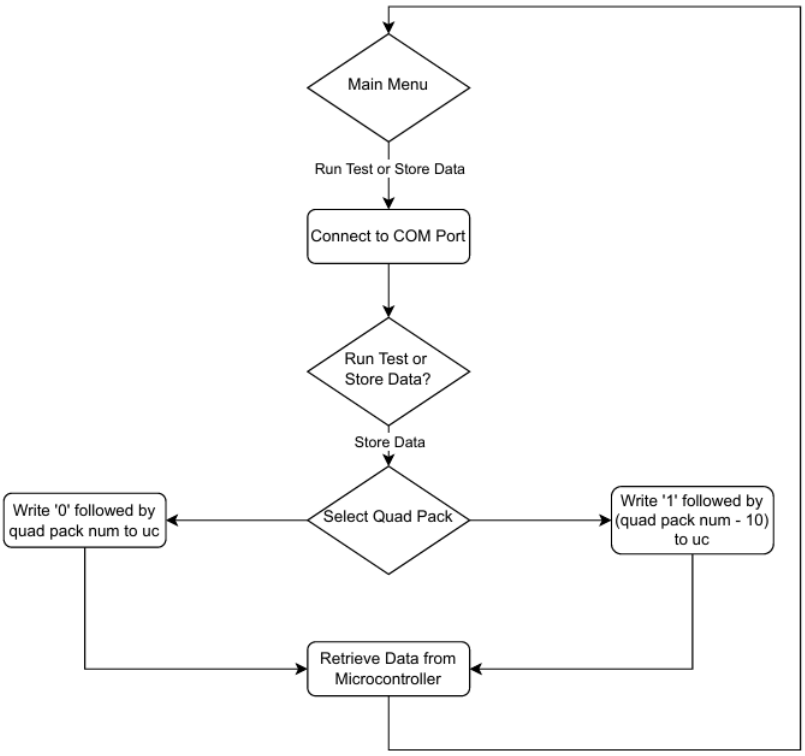

Figure 4.21: Retrieving Past Results Flowchart (TS)

Section 4.2.1 UART Initialization:

Within main.c, USART3_setup() is called to configure USART3 settings for communication between PC and MCU. This function will initialize the baud rate, set the mode of data transfer, configure PB5 as TX and PB4 as RX, and set direction of their GPIO pins. These pins must be connected to the corresponding RX and TX pins of the CP2102 USART bridge. This function is shown below in figure 4.22 (JL).

```
void USART3_setup(void)
{
    USART3.BAUD = baud_rate ;
    USART3.CTRLC= (USART_CMODE_ASYNCHRONOUS_gc | USART_PMODE_DISABLED_gc | USART_CHSIZE_8BIT_gc); //set frame type
    VPORTB_DIR |= 0x01; //make PB0 as output
    VPORTB_DIR &= 0xFD; //make PB1 as input
    USART3.CTRLA |= USART_RXCIE_bm; //enable interrupt for when data has been received
    USART3.CTRLB |= USART_RXEN_bm | USART_TXEN_bm; //enable transmit and receive
}
```

Figure 4.22: UART Initialization (JL)

Section 4.2.2 Sending Commands from the PC:
When commands are sent from the PC via USART bridge, they trigger the interrupt service routine of the remote interface. This interrupt calls specific functions based on the character received from the PC. Table 4.3 shows the characters sent from the PC and their associated functions (JL).

Table 4.3: Characters Sent From the GUI to the Microcontroller and their Meaning (TS)

Character Sent from PC to MCU (case sensitive)	Meaning
‘u’	Perform an Unloaded Test
‘m’ followed by three numbers ‘0’-’9’ (max 500)	Perform a Loaded Manual Test with the Specified Current (first digit is hundreds place, second is tens place, third is ones place)
‘a’ followed by three numbers ‘0’-’9’ (max 500)	Perform a Loaded Automatic Test with the Specified Current (first digit is hundreds place, second is tens place, third is ones place)
‘c’	Cancel Loaded Manual Test
‘e’	Cancel Loaded Automatic Test
‘r’	Send Results From Just Performed Test to PC
‘1’ followed by a number ‘0’-’3’	Retrieve Data From EEPROM Quad Pack 10-13 (the first number represents the tens place of the quad pack number and the second number represents the ones place of the quad pack number)
‘0’ followed by a number ‘1’-’9’	Retrieve Data From EEPROM Quad Pack 1-9 (the first number represents the tens place of the quad pack number and the second number represents the ones place of the quad pack number)

Section 4.2.3: UART Interrupt Service Routine (ISR):

At the start of the ISR, received_char is used to read commands from the PC, transmit_char is used to transmit statuses back to PC. This is shown below in figure 4.23.

```
//
ISR(USART3_RXC_vect)
{
    cli(); //disable interrupts
    char received_char = USART3.RXDATAL; //get received character
    uint8_t quad_pack;
    char transmit_char;
```

Figure 4.23: Start of ISR Code (JL)

The received character (command) is used in a case statement to determine which function to be called. Once that function has finished, it returns a status character back to the PC to acknowledge that it has finished. This is necessary to maintain proper synchrony between the GUI and MCU. The code is provided below in figure 4.24. Refer to table 4.4 for the status characters (JL).

```
case 'u': //unloaded test
    transmit_char = test_unloaded_remote();
    USART3_transmit_character(transmit_char); //send unloaded test result character, d = successful, e = battery not connected, v = low voltages
    if (transmit_char == 'v') //if unloaded voltages are low, send unloaded voltages
        send_unloaded_voltages();
    break;
case 'm': //manual loaded test
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for hundreds digit of current
    received_char = USART3.RXDATAL; //get hundreds digit of current
    current_test_result.max_load_current = (received_char - '0') * 100; //add hundreds digit of current to variable
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for tens digit of current
    received_char = USART3.RXDATAL; //get tens digit of current
    current_test_result.max_load_current += (received_char - '0') * 10; //add tens digit of current to variable
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for ones digit of current
    received_char = USART3.RXDATAL; //get ones digit of current
    current_test_result.max_load_current += (received_char - '0'); //add ones digit of current to variable
    transmit_char = manual_test_loaded_remote(); //perform manual loaded test
    USART3_transmit_character(transmit_char); //transfer 'f', manual loaded test complete
    break;
case 'a': //automated loaded test
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for hundreds digit of current
    received_char = USART3.RXDATAL; //get hundreds digit of current
    current_test_result.max_load_current = (received_char - '0') * 100; //add hundreds digit of current to variable
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for tens digit of current
    received_char = USART3.RXDATAL; //get tens digit of current
    current_test_result.max_load_current += (received_char - '0') * 10; //add tens digit of current to variable
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for ones digit of current
    received_char = USART3.RXDATAL; //get ones digit of current
    current_test_result.max_load_current += (received_char - '0'); //add ones digit of current to variable
    transmit_char= automatic_test_loaded_remote(); //perform automated loaded test
    USART3_transmit_character(transmit_char);//transfer 'a', automated loaded test complete
    break;
case 'r': //get test results
    send_results_pc(); //send results to PC
    break;
```

Figure 4.24: ISR Case Statement (JL)

Table 4.4: Characters Sent From the Microcontroller to the GUI and their Meaning (TS)

Character Sent from MCU to PC (case sensitive)	Meaning
‘u’	Unloaded Voltages are Being Sent Back
‘l’	Loaded Voltages are Being Sent Back
‘h’	Health Ratings are Being Sent Back
‘c’	Current is Being Sent Back
‘d’	Unloaded Test Done
‘i’	Turn Down Current
‘f’	Manual Loaded Test Done or Test Cancelled Successfully
‘a’	Automatic Loaded Test Done or Test Cancelled Successfully
‘v’	Unloaded Voltages are Low and Readings will be Sent to the Remote Interface
‘e’	Battery Not Connected

Section 4.2.4 Reading from EEPROM:

If the user wants to save data from the MCU's EEPROM to the PC's filesystem, a '0' or '1' is sent. A '0' is sent to read measurements from a quad pack among entries 1-9, and '1' is sent to read measurements from a quad pack among entries 10-13. The reason why reading is separated into two cases is because entries 10-13 require two characters to specify while entries 1-9 only require one. Once an entry category is picked, it needs to be followed by another character, (1-9) for case 0 and (0-3) for case 1, to specify a quad pack entry to read within that category. For example, reading entry 9 requires a '0' followed by '9'. The code for this is shown below in figure 4.25 (JL).

```
case '0': //get data from quad pack 1-9
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for next character
    received_char = USART3.RXDATAL; //get ones digit of quad pack
    quad_pack = received_char - '0'; //save quad pack digit to variable
    read_EEPROM(quad_pack - 1); //read from specified EEPROM quad pack
    send_results_pc(); //send results to PC
    break;
case '1': //get data from quad pack 10-13
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for next character
    received_char = USART3.RXDATAL; //get ones digit of quad pack
    quad_pack = received_char - '0'; //save quad pack digit to variable
    quad_pack = quad_pack + 10; //add offset of 10 to variable
    read_EEPROM(quad_pack - 1); //read from specific EEPROM quad pack
    send_results_pc(); //send results to PC
    break;
```

Figure 4.25: EEPROM Reading Code (JL)

Section 4.2.5 Unloaded Test:

When the PC sends a ‘u’, it specifies to run an unloaded test. First, the MCU checks if the battery is connected by performing a single ended measurement from battery cell 4’s positive terminal to ground (should be 13.2 V). If the battery voltage is below 0.1 V, the status ‘e’ will be sent back to the PC to indicate the battery is not connected and the GUI will display this error. Otherwise, unloaded battery measurements will be performed. If any battery has a voltage of less than 3.0 V, a ‘v’ will be sent back to the PC along with the unloaded voltages. Otherwise, a status ‘d’ will be sent back to the PC notifying that unloaded measurements are done. The code is shown below as figure 4.26 (JL).

```
char test_unloaded_remote(void)
{
    /* Read total battery pack voltage with single-ended measurement */
    ADC_init(0x01);
    ADC_channelSEL(B4_ADC_CHANNEL, GND_ADC_CHANNEL);
    float voltage = ADC_read() * battery_voltage_divider_ratios;

    /* If voltage < 0.1V, no battery connection and return 'e' */
    if (voltage < 0.1)
        return 'e';
    else //else read unloaded battery voltages
        read_UNLOADED_battery_voltages();
    for (uint8_t i = 0; i < 4; i++)
    {
        if (current_test_result.UNLOADED_battery_voltages[i] < min_battery_voltage) //if unloaded voltage below min value (3.0), return 'v'
            return 'v';
    }
    return 'd'; //else return 'd' - test successful
}
```

Figure 4.26: Running an Unloaded Test Code (JL)

Section 4.2.6 Manual Loaded Test:

When the PC sends a ‘m’, it specifies to run a manual loaded test. The next three digits sent to the microcontroller are the hundreds, ten, and ones digits of the specified load current. The current will be displayed on the LCD and the GUI will prompt the user to crank the carbon pile until the desired current is met and the buzzer sounds. This is the current that the loaded voltages will be measured and call a function to perform a loaded test. Alternatively, if a ‘c’ is sent, a flag variable will be set to cancel the test and break out of the while loop. In this case, the loaded voltages will not be read. Otherwise, the while loop will continue to update current on the LCD. Code segments are shown below in figures 4.27 and 4.28 (JL).

```
        case 'm': //manual loaded test
            while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for hundreds digit of current
            received_char = USART3.RXDATAL; //get hundreds digit of current
            current_test_result.max_load_current = (received_char - '0') * 100; //add hundreds digit of current to variable
            while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for tens digit of current
            received_char = USART3.RXDATAL; //get tens digit of current
            current_test_result.max_load_current += (received_char - '0') * 10; //add tens digit of current to variable
            while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for ones digit of current
            received_char = USART3.RXDATAL; //get ones digit of current
            current_test_result.max_load_current += (received_char - '0'); //add ones digit of current to variable
            transmit_char = manual_test_loaded_remote(); //perform manual loaded test
            USART3_transmit_character(transmit_char); //transfer 'f', manual loaded test complete
            break;
        case 'a': //automated loaded test
```

Figure 4.27: Manual Loaded Test Code 1 (JL)

```
load_current_amps = load_current_Read(); //read load current
clear_lcd();
sprintf(dsp_buff[0], "Rotate Knob Until   ");
sprintf(dsp_buff[1], "Beeping Sound is    ");
sprintf(dsp_buff[2], "Heard...            ");
sprintf(dsp_buff[3], "Load Current: %.1fA ", load_current_amps);
update_lcd();

while (load_current_amps < current_test_result.max_load_current) //while load current is below specified current
{
    if(USART3.RXDATAL == 'c') //if cancel test is selected
    {
        cancel_test = 0x01; //set flag to cancel test
        break; //exit increase current while loop
    }
    load_current_amps = load_current_Read(); //update current reading

    _delay_ms(50);  // delay to prevent LCD to updating too fast
    clear_lcd();
    sprintf(dsp_buff[0], "Rotate Knob Until   ");
    sprintf(dsp_buff[1], "Beeping Sound is    ");
    sprintf(dsp_buff[2], "Heard...            ");
    if (load_current_amps >= 100)
        sprintf(dsp_buff[3], "Load Current: %.1fA", load_current_amps);
    else if (load_current_amps >= 10)
        sprintf(dsp_buff[3], "Load Current: %.1fA ", load_current_amps);
    else
        sprintf(dsp_buff[3], "Load Current: %.1fA  ", load_current_amps);
    update_lcd();

}
```

Figure 4.28: Manual Loaded Test Code 2 (JL)

Next, the program will first save the current that the user has specified, and then wait 1 second before measuring voltages to ensure stable readings. Reading the loaded battery voltages simply calls the local interface's read_loaded_voltages function. If the cancel test flag was set, the loaded voltages will not be read and the flag will be cleared. Once the voltages have been measured, the status 'i' is sent back to the PC to indicate that the measurements have been made and so that the GUI can prompt the user to lower the current back to 0. While the current is being lowered, its value is displayed on the LCD. The buzzer is also toggled simultaneously. Once the current is below 1 A, the LCD screen will return to the main menu and the status 'f' will be sent to the PC to indicate that the current is below 1A. This is shown in figure 4.29 below (JL).

```
USART3_transmit_character('i'); //transmit 'i' so user knows to turn down current

if(cancel_test = 0x00) //if test was not canceled
{
    current_test_result.max_load_current = load_current_amps; //save max load current
    _delay_ms(100);
    read_LOADED_battery_voltages(); //read loaded voltages
}
cancel_test = 0x00; //reset cancel test flag

while (load_current_amps > 1) //while load current is greater than 1 A
{
    load_current_amps = load_current_Read(); //update current reading

    temp = load_current_amps;
    _delay_ms(200);
    clear_lcd();
    sprintf(dsp_buff[0], "Test complete...    ");
    sprintf(dsp_buff[1], "Rotate Knob until   ");
    sprintf(dsp_buff[2], "beeping stops...    ");
    if (load_current_amps >= 100)
        sprintf(dsp_buff[3], "Load Current: %.1fA", load_current_amps);
    else if (load_current_amps >= 10)
        sprintf(dsp_buff[3], "Load Current: %.1fA ", load_current_amps);
    else
        sprintf(dsp_buff[3], "Load Current: %.1fA  ", load_current_amps);
    update_lcd();

    buzzer_ON();
    _delay_ms(1000);        // wait 1 second
    buzzer_OFF();
    _delay_ms(1000);        // wait 1 second
}

display_main_menu(); //go back to main menu
return 'f'; //'f' means manual loaded test finished
```

Figure 4.29: Manual Loaded Test Code 3 (JL)

Section 4.2.7 Automated Loaded Test:

When the PC sends a ‘a’, it specifies to run an automated loaded test. The next three digits sent to the microcontroller are the hundreds, ten, and ones digits of the specified load current. The stepper motor’s set_load_current function will be called to set the load current to the desired value. If an ‘e’ is sent, a flag variable will be set to cancel the test. In this case, the loaded voltages will not be read. Reading the loaded battery voltages simply calls the local interface’s read_loaded_voltages function. If the cancel test flag was set, the loaded voltages will not be read and the flag will be cleared. Once the voltages have been measured, the open_circuit_load function is called to lower the current back to 0. The buzzer is also toggled simultaneously. Once the current is below 1 A, the LCD screen will return to the main menu and the status ‘a’ will be sent to the PC to indicate that the current is below 1A. This is shown below in figures 4.30 and 4.31 (JL).

```
case 'a': //automated loaded test
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for hundreds digit of current
    received_char = USART3.RXDATAL; //get hundreds digit of current
    current_test_result.max_load_current = (received_char - '0') * 100; //add hundreds digit of current to variable
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for tens digit of current
    received_char = USART3.RXDATAL; //get tens digit of current
    current_test_result.max_load_current += (received_char - '0') * 10; //add tens digit of current to variable
    while(!(USART3_STATUS & USART_RXCIF_bm)) ; //wait for ones digit of current
    received_char = USART3.RXDATAL; //get ones digit of current
    current_test_result.max_load_current += (received_char - '0'); //add ones digit of current to variable
    transmit_char= automatic_test_loaded_remote(); //perform automated loaded test
    USART3_transmit_character(transmit_char);//transfer 'a', automated loaded test complete
    break;
```

Figure 4.30: Automated Loaded Test Code 1 (JL)

```
//.....
char automatic_test_loaded_remote()
{
    set_load_current(current_test_result.max_load_current); //set load current to specified current
    if(cancel_test = 0x00) //if test is not canceled
    {
        read_LOADED_battery_voltages();  //read loaded battery voltages
        buzzer_ON();
        open_circuit_load(); //set load current back to 0
        _delay_ms(1000);
        buzzer_OFF();
    }
    cancel_test = 0x00; //reset cancel test variable
    open_circuit_load(); //set load current back to 0
    return 'a'; //return 'a' to indicate test finished
}
```

Figure 4.31: Automated Loaded Test Code 2 (JL)

Section 4.2.8 Reading Results:

After the measurements have been made and current is below 1 A, the PC will send the ‘r’ command to specify that it wants to read test results including the health rating and current of the loaded test. First, a string array remote_buff is created. The unloaded battery voltages, loaded battery voltages, and current are of floating type and need to be converted to string before they can be transmitted. Using the sprintf function, the values are simultaneously converted to string and stored on a character buffer such as remote_buff. This is shown below in figure 4.32 (JL).

```
for(uint8_t i = 0; i < 4; i++) //add unloaded voltages to buffer array
{
    sprintf(remote_buff[i], "%.3f", current_test_result.UNLOADED_battery_voltages[i]);
}

for(uint8_t i = 0; i < 4; i++) //add loaded voltages to buffer array
{
    sprintf(remote_buff[i + 4], "%.3f", current_test_result.LOADED_battery_voltages[i]);
}

/* Write health ratings into character buffer */
decode_health_rating(current_test_result);

//add current to current buffer variable
if(current_test_result.max_load_current < 10)
{
    sprintf(current_buff, "%d  ", current_test_result.max_load_current);
}
else if(current_test_result.max_load_current < 100)
{
    sprintf(current_buff, "%d ", current_test_result.max_load_current);
}
else
{
    sprintf(current_buff, "%d", current_test_result.max_load_current);
}
```

Figure 4.32: Reading Results Code (JL)

Section 4.2.9 Transferring Data to the PC:

Then, remote_buff is parsed, and each character within each string is transmitted to the PC using the USART3_transmit_character function. Before sending a character, the status character ‘u’ will be sent to the PC to indicate that unloaded measurements are being sent. This way, the PC will know when to read incoming values and store them in its local memory. The same is done for the other results except the status character ‘l’ is used for loaded, ‘h’ for health ratings, and ‘c’ for current. This is shown in figure 4.33 below (JL).

```
//transmit unloaded voltages
USART3_transmit_character('u'); //unloaded voltages are being sent
for(uint8_t i = 0; i < 4; i++)
{
    for(uint8_t j = 0; j < 5; j++)
    {
        USART3_transmit_character(remote_buff[i][j]);
        _delay_ms(10);
    }
}
_delay_ms(10);

//transmit loaded
USART3_transmit_character('l'); //loaded voltages are being sent
for(uint8_t i = 4; i < 8; i++)
{
    for(uint8_t j = 0; j < 5; j++)
    {
        USART3_transmit_character(remote_buff[i][j]);
        _delay_ms(10);
    }
}

_delay_ms(10);

//transmit health ratings
USART3_transmit_character('h'); //health ratings are being sent
for(uint8_t i = 0; i < 8; i++)
{
    USART3_transmit_character(health_rating_characters[i]);
    _delay_ms(10);
}

//transmit current
USART3_transmit_character('c'); //current is being sent
for(uint8_t i = 0; i < 5; i++)
{
    USART3_transmit_character(current_buff[i]);
    _delay_ms(10);
}
```

Figure 4.33: Transferring Data to the PC Code (JL)

Lastly, within the USART3_transmit_character function each character is added to the data register, which is then moved to TX shift register that shifts each bit of the character using USART protocol. The status of both registers need to be checked before loading the next character. This is shown below in figure 4.34 (JL).

```
void USART3_transmit_character(char transmit_char)
{
    while(!(USART3.STATUS & USART_DREIF_bm)){} // Wait for data register to be empty
    USART3.TXDATAL= transmit_char;  // Load character to be sent into data register

    while(!(USART3.STATUS & USART_TXCIF_bm)){} // Check if shift register is empty -> transmit done

}
```

Figure 4.34: UART Transmit Character Code (JL)