PROJECT 4: "ADDING FUNCTIONALITY"

WORKING WITH DYNAMIC TASK QUEUES FOR TIME-VARYING TASKS, REMOTE SYSTEM COMMUNICATION, EEPROM MEMORY MANAGEMENT, AND SOC CALCULATION TO IMPLEMENT THE THIRD PHASE OF A BATTERY MANAGEMENT SYSTEM

J. Vining
ECE/CSE 474, Embedded Systems
University of Washington – Dept. of Electrical and Computer Engineering

REV: 10 Feb 2021

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Consider that your team has been contracted to implement a basic battery management system. You've been given this document as a basic framework / specification for what the customer wants. The labs in this course break down the development process as 'milestones' for the customer.

1.1 Development Phases

This project is the **second phase** in the development of a simple battery management system for high voltage electric transportation applications. The current phase focuses on "fleshing out the system": adding functionality to tasks that have been developed as part of the basic system architecture, modeling more of the instrumentation in hardware, implementing the high-level data/control flow, managing a dynamic scheduler (linked list), and adding further functional refinement to the display driver and alarm annunciation functions.

2.0 REVISIONS

Table 1 – Project Revision Table (Version Control)

Version	Date	Author	Description
A	11 Jan 2021	J.Vining	**First release of initial architecture specification:** Working with system architecture, a round robin scheduler, task control blocks, and general I/O to implement the first phase of a battery management system
B	19 Jan 2021	J.Vining	**Deletions:** 1. Project 2 guidelines sections: • Development Phases • Layout of the Project/Guidelines for Success • Recommended Design Approach 2. Appendix sections: • Implementing the TCB • Tips for Building Circuits

			Additions: 1. Appendix • Working with Interrupts (Section 9.1) 2. System controller tasks • ***Hardware timer interrupt*** (Section 7.3)*:* Hardware timer to create real-time time base • ***HVIL interrupt routine*** (Sections 7.10)*:* Introduced to set flag and open contactors in the event HVIL transitions to open **Modifications:** 1. Background (Section 3.0) 2. Project Objectives (Section 4.0) 3. System controller tasks • ***Scheduler task*** (Section 7.2): Uses linked list instead of round robin array for calling tasks • ***Touch screen task*** (Section 7.4)*:* i. Display has new state flow for transition between screens: If an alarm is active but not acknowledged, the display will only show the alarm screen until the alarm is acknowledged. ii. The alarm screen has a new button in the event that an alarm is active and needs to be acknowledged. iii. The alarm and measurement screens show data from real-world measurements. • ***Measurement task*** (Section 7.5)*:* Takes analog measurements: temperature, HV current and HV voltage • ***SOC task*** (Section 7.8)*:* Constant value of SOC=0 • ***Contactor task*** (Section 7.9)*:* Contactor state diagram updated to use HVIL alarm flag as an input for state transition criteria • ***Alarm task*** (Section 7.11)*:* Calculates & reports alarms based on measurement data
C	10 Feb 2021	J.Vining	**Deletions:** 1. Scheduler section:

			• Associating Real Time with the Scheduler 2. Appendix section: • Working with Interrupts **Additions:** 1. Appendix • Doubly Linked Lists: Insert and Delete Functions (Section 9.1) 2. System controller tasks • ***Data logging task*** (Section 7.6)***:*** Write / read measurement history values to / from EEPROM • ***Remote terminal task*** (Section 7.7)***:*** Implement a remote user terminal to read or clear measurement history data **Modifications:** 2. Background (Section 3.0) 3. Project Objectives (Section 4.0) 4. System controller tasks • ***Startup task*** (Section 7.1)*:* Added functionality: Reads measurement history values from EEPROM at startup • ***Scheduler task*** (Section 7.2)*:* Time varying tasks now supported. i. Implement functions for adding and removing tasks dynamically from doubly linked list task queue to support time varying tasks. ii. Support more than one periodic task execution rate via counters • ***ALL TASKS:*** Each task will have a unique periodic execution rate associated with it • ***Measurement task*** (Section 7.5)*:* Now tracking measurement history for EEPROM data logging and remote terminal display • ***SOC task*** (Section 7.8)*:* Now calculating SOC based on physical current, voltage, and temperature measurements • ***Contactor task*** (Section 7.9)*:* Contactor state diagram updated to use ALL alarm flags as an input for state transitions

3.0 BACKGROUND

Did well on Project 3, exhausted but excited that the quarter is almost over!...

Oh yes, and **patience**… Hopefully you're learning to practice patience in developing your hardware, software, writing and teamwork skills… and you've come to appreciate your lab partner(s)!

3.1 Cautions and Warnings

Sometimes – but only in the most dire of situations – sacrificing small animals to the code gremlin living in your ATMega chip *could* make your code work. However, **these critters are not included in your kit** and must be purchased separately. Also, be aware that most of the time, code gremlins are not affected by such sacrifices. They simply laugh in your face…bwa ha ha…
Alternately, blaming your partners can work for a short time… until everyone finds out that you are really to blame.

4.0 PROJECT OBJECTIVES

In this lab, we will continue working with the Arduino Mega (the "System Controller"). This work has the following goals:

- Develop drivers for more ***peripheral devices*** :: Remote System & EEPROM
- Upgrade the hardware-based system time base for ***time varying tasks***
- Develop a ***formal communication protocol*** for a ***remote system***
    - Incorporate a remote communications system and network interface into the system.
    - Implement a command-and-control interface
- Implement ***data logging*** thru ***EEPROM***
- Implement and test the ***new features and capabilities*** of the system:
    - Amend the formal specifications to reflect the new features.
    - Amend existing UML diagrams to reflect the new features model some of the dynamic aspects of the system.
    - Amend the requirements and design specifications to reflect the new features
    - Incorporate several additional simple tasks to our system.

This project, project report, and program are to be done as a team – play nice, share the equipment, keep any viruses (software or otherwise) to yourself, and no fighting.

5.0 SYSTEM ARCHITECTURE

General System Description:
Battery management systems (BMS) are required to ensure the **safety**, **proper operation** and **long-term reliability** of high voltage batteries in electric transportation applications. The system architecture presented is a **simplified but representative** approach to battery management.

6.0 HARDWARE ARCHITECTURE

The hardware architecture described in this subsection presents the system hardware inputs and outputs as well as the overall layout of the system.

We will be simulating a majority of the following inputs with the exception of the touch screen and accelerometer.

6.1 Inputs

For this stage of the project, the **analog and digital inputs highlighted in grey** will be implemented in hardware and the remaining analog input sensor will be implemented in the final assignment.

Digital:
- High voltage interlock loop (HVIL) signal – digital input actuated via switch
- Touch screen feedback

Analog:
- HV terminal voltage (**0-450V** sensor range scaled 0-5V)
- HV terminal current (**-25A – 25A** sensor range scaled 0-5V)
- Temperature (**-10°C – 45°C** sensor range scaled 0-5V)
- Accelerometer (±1.5g sensor range scaled 0-3.3V)

6.2 Outputs

Digital:

- Contactor on/off (shown via LEDs)
- Touch screen display

6.3 I/O Interface Circuits

The circuit diagrams used to **model the hardware I/O** are provided in Section 7.0 Software Architecture, under the task that accesses the circuit. You will use these as a guide to model/simulate/create the hardware interfaces for your project.

6.4 Block Diagram

The block diagram in Figure 1 provides a high-level *partially complete* block diagram for the system, including all major functional blocks. **WE WILL ONLY BE IMPLEMENTING ITEMS IN LIGHT GREY** in this project.

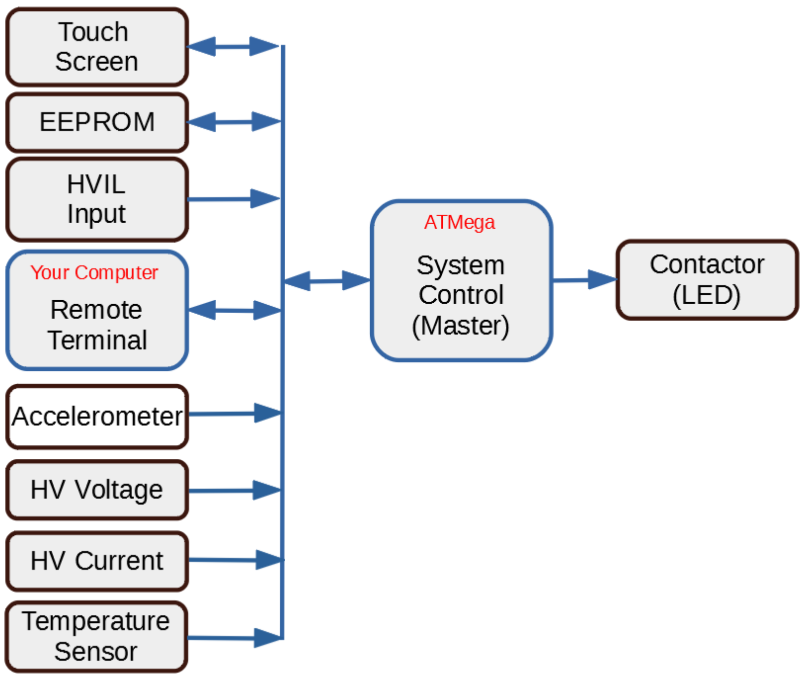

Figure 1 – High-Level *Partially Complete* Block Diagram of the Battery Management System (BMS). *NOTE: Items in light grey are implemented in this project*

7.0 Software Architecture

Your code will include the following tasks:
The tasks highlighted in grey will be implemented at this stage of the project

- Startup
- Scheduler
- Measurement
- Measurement History (Data Logging)
- Touch Screen Task: Display & Touch Input
- Remote Terminal
- SOC (State of Charge Calculation)
- Contactor
- Alarm
- HVIL Interrupt
- Hardware Timer Interrupt
- Accelerometer

The following subsections describe each of the major functional tasks, as they pertain to this stage of the project. Functionality of each of these tasks shall be modified and expanded as the project moves forward.

7.1 Startup Task: setup()

The *Startup Task* is the first task to execute and runs only once when the embedded controller wakes up or resets. The startup task shall reside in the Arduino language's **setup()** function which is configured to run once during startup.

This task initializes all:

- Hardware
  - System time base (timers, etc.)
  - GPIO (general purpose input/output)
  - Communication protocols (UART serial bus, etc)
  - Interrupts, etc.
- Software
  - **Measurement history shared global variables** (initializes by reading values from **EEPROM**). See the *Data Logging Task*, Section 7.6, for information on the measurement history variables.

- Task data structures
- Task control blocks (TCB) – see next section on how to initialize these as a doubly linked list

7.2 Scheduling Task

The *Scheduling Task* is executed in the main loop - for the Arduino language, this is the **loop()** function. This task takes care of scheduling and executing the system tasks that make the BMS function.

We will transition from a **Scheduler** with a **single task execution rate** (10Hz) to one that supports **multiple task execution rates** (1Hz, 10Hz, etc.).

7.2.1 Dynamic Scheduler

As before, the dynamic scheduler will:

- Run each task from the task queue in succession.
- Pace execution of the task queue using a **hardware interrupt timer** that sets the **System Time Base (10Hz** as implemented in Project 3**).**
- Use a task queue composed of a **doubly linked list** of TCBs.

The same basic rules for task execution apply:

- Tasks deemed non-atomic shall not be pre-emptable, i.e. no interrupts shall occur during critical code sections.
- If a task has nothing to do, it shall exit immediately.

In this project, the task queue will support **time varying task rates –** for example, some tasks will run once per second, 1Hz.

The scheduler shall execute at the 10Hz rate set in Project 3; however, some of the tasks will not run each time step – they will execute at an **integer multiple** of the **global system time base.** The scheduler shall determine which tasks need to run during each time step. Tasks not running at each time step shall be dynamically added and removed from the task queue using an ***insert function*** and ***delete function*** as described in Section 7.2.2.

Once the scheduler has determined which tasks are to execute during the current time step, it shall execute each TCB in the linked list sequentially without delay and then return to the main loop where it shall be called again after the hardware interrupt timer sets the flag to allow the main loop to cycle again.

7.2.2 Supporting Time Varying Tasks: Adding & Removing Tasks Dynamically from the Task Queue

Tasks that run at a lower rate than the system time base shall be added and removed from the task queue dynamically. A task shall be added to the queue using an **insert function** and removed using a **delete function** as described in Appendix Section 9.1.

To determine if a dynamic task is to run or not, create a **task counter** to track the duration between execution periods. If a task's counter indicates it's time to run, it shall be added to the queue. Once the task has run, it shall be removed from the queue.

Example:
Take a system time base of 10Hz (0.1s period). There are two tasks running at different rates: Task1 at 10Hz, Task2 at 1Hz. Task1 will run every time step (the global time base). Task2 will run every 10 time steps (10*global time base). Task2 will be added and removed from the dynamic task queue such that it runs once every 10 time steps.

7.2.3 Dynamic Task Queue (TCB Linked List) Implementation

Each element of the task queue doubly linked list shall be a TCB, representing one of the tasks identified in Section 7.0.

Memory for tasks shall NOT be created dynamically – in other words, all tasks shall be declared and initialized during startup (all tasks exist in memory indefinitely until power-down). The dynamic scheduler algorithm simply adds or removes tasks that already exist in memory based on the execution rate of the task.

7.3 Hardware Timer Interrupt: timerISR()

A hardware time base is implemented to set the execution rate for the scheduler. This execution rate is known as the **system time base**, aka the **system time step**, and is initialized in the *Startup Task* by attaching an interrupt with a fixed period. The interrupt task for the hardware timer shall set a flag that signals the main loop to cycle again.

The **System Time Base** shall be set to **10Hz.**

7.4 Touch Screen Task: Display & Touch Input

The touch screen display task shall include display and touch input functionality.

Tips:

- You will find it advantageous to create **separate functions** for **touch input** and **display.**
- For the display: Only **update values** on the screen that are **changing** so your code executes faster.

This task shall run at a **1s rate** with ***OPTION*** to run the touch sensing portion at a faster rate for better touch response.

7.4.1 Touch Input

The display shall provide user input buttons to scroll through the following screens:

- **Measurement Screen:** Scroll thru measurements
- **Alarm Screen:** Scroll thru / acknowledge alarms
- **Battery ON/OFF Screen:** Option to turn ON / OFF battery

You may use "next" and "previous" buttons or a single button for each screen.

7.4.1.1 Screen-Specific Input: **Battery ON/OFF Screen**

The Battery ON/OFF screen has the same scroll buttons as the other screens PLUS an additional input: an **ON / OFF toggle switch**. The toggle switch provides user input to turn ON or OFF the battery. See the next section, "Display", for more details.

7.4.1.2 Screen-Specific Input: **Alarm Screen**

The Alarm Screen has the same scroll buttons as the other screens PLUS one additional input: an **alarm acknowledgement button**. In the event that an alarm is active and not acknowledged, the alarm screen will display a single button to allow the user to ACKNOWLEDGE all non-acknowledged active alarms. The next section, "Display", contains more details

7.4.2 Display

The display consists of three screens, described below. The display task shall access shared variables from the following tasks to populate the measurement and alarm screens: *Measurement, SOC,* and *Alarm.*

7.4.2.1 Measurement Screen

The **Measurement Screen** shall display the following sensor data:

- State of Charge: <value>
- Temperature: <value>
- HV Current: <value>
- HV Voltage: <value>
- HVIL (HV Interlock Loop) Status: <value>

7.4.2.2 Alarm Screen

The **Alarm Screen** shall display the value of each of the alarms as listed in Section 7.11.

- High Voltage Interlock Alarm: <state>
- Overcurrent: <state>
- High Voltage Out of Range: <state>

If any of the alarm states = "ACTIVE, NOT ACKNOWLEDGED", the display shall automatically navigate to this screen if it is not there already. When this occurs, a button will appear (as described in the Touch Input section above) with the option to acknowledge the alarm. This acknowledgement by the user is passed as a **flag** to the *Alarm Task*, indicating the alarm is acknowledged and may change state to "ACTIVE, ACKNOWLEDGED".

7.4.2.3 Battery ON/OFF Screen

The **Battery ON/OFF Screen** shall display the **current state of the battery contactors** as well as a toggle switch to allow user input to turn ON or OFF the battery. The two toggle switch states yield the following actions:

- Turn ON… (CLOSE contactors by sending **flag** to the *Contactor Task*)
- Turn OFF… (OPEN contactors by sending **flag** to the *Contactor Task*)

Note that the **flag** is a shared variable that is passed to the *Contactor Task.* This tells the *Contactor Task* what state the user wants the battery to be in.

7.5 Measurement Tasks: Sensor Measurements

Measurements shall be taken to provide both the BMS and outside world with the system state according to the block diagram in Figure 1.

The Measurement Task shall run at **10Hz (100ms).**

7.5.1 Temperature, HV Current & Voltage Measurements

At this stage in the project**, we will be implementing temperature, current & voltage measurements** using **analog inputs**. Since we do not have actual temperature, current and voltage sensors, we will be simulating these sensors using test circuits as described in the next section.

7.5.1.1 Test Circuits for Simulated [0-5V] Analog Sensors: ANALOG INPUT

Description of circuit in real-world application:
HV voltage, HV current, and temperature sensors are some of the most important components of the battery management system. These sensors tell the system about the battery's electrical and thermal state.

When designing these sensors, **the full output range of the sensor is scaled to the input range of the microcontroller's analog input pins for maximum resolution**. In the case of the ATMega, the analog input range is [0, 5V].

Temperature sensor analog input scaling:

- Temperature (**-10°C – 45°C** sensor range scaled to analog input 0-5V)

HV voltage and current sensor analog input scaling:

- HV **terminal**** voltage (**0V – 450V sensor range** scaled to analog input 0-5V)
- HV **terminal**** current (**-25A – 25A sensor range** scaled to analog input 0-5V)
  ****Terminal** means that the **measurement** is taken at the **battery's terminals**.

Implementation:
These analog inputs shall be modeled using the circuit in Figure 2:

- The **potentiometer** allows you to produce a variable voltage in the range [0, 5V], which covers the range of each of the sensors.
- Note that the circuit requires a **software-enabled pullup resistor** within the ATMega. This is configured in the *Startup Task* when initializing the input pin. If the pullup resistor is not enabled, you should only see 0V at the input no matter the resistance value of the potentiometer.

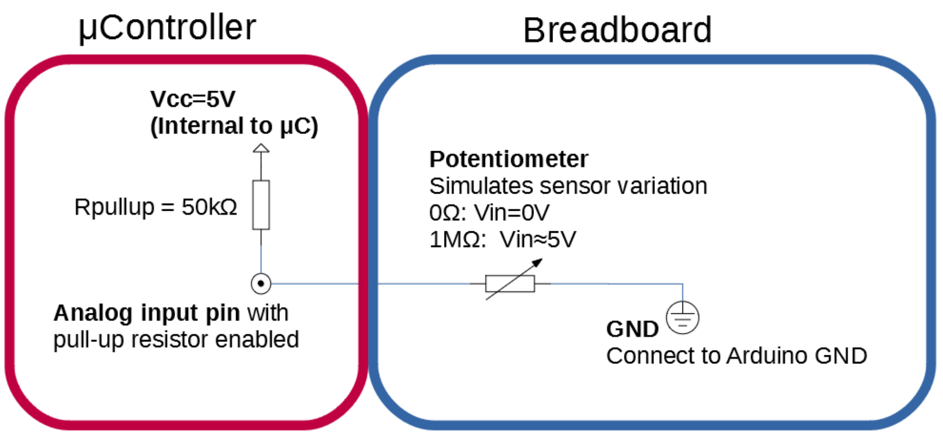

Figure 2 – ANALOG INPUT: Test Circuits for Sensors in Range [0, 5V]

7.5.2 High Voltage Interlock Loop (HVIL)

The HV Interlock Loop circuit will remain the same as implemented in the previous lab. As before, **this measurement shall be stored in a shared variable for inter-task data exchange with the display**, similar to the other measurement values.

7.5.2.1 Test Circuit for HVIL: DIGITAL INPUT

Description of circuit in real-world application:
The high voltage interlock loop is a circuit that detects whether or not all high voltage connectors are connected since its circuit runs alongside the high voltage cabling. The circuit provides a safety check for the battery management system to ensure that no exposed high voltage cabling is present under operating conditions.

There are many means to implement one of these loops and the detection circuit that reads whether the HVIL is OPEN or CLOSED. For this project, we will simulate a HVIL detection circuit that provides a digital reading to the microcontroller of whether the loop is OPEN or CLOSED.

Implementation:
The circuit in Figure 3 shows how to simulate the HVIL detection circuit's connection to the microcontroller by using a DIP switch to OPEN and CLOSE the circuit. The expected input at the microcontroller should be (notice these are ***states***):

- OPEN DIP switch:
    - Produces 5V at the digital input pin (reading logic 1)
    - LED will not light up
    - HVIL is OPEN!
- CLOSED DIP switch
    - Produces 0V at the digital input pin (reading logic 0)
    - LED will light up
    - HVIL is CLOSED, yay, no danger!

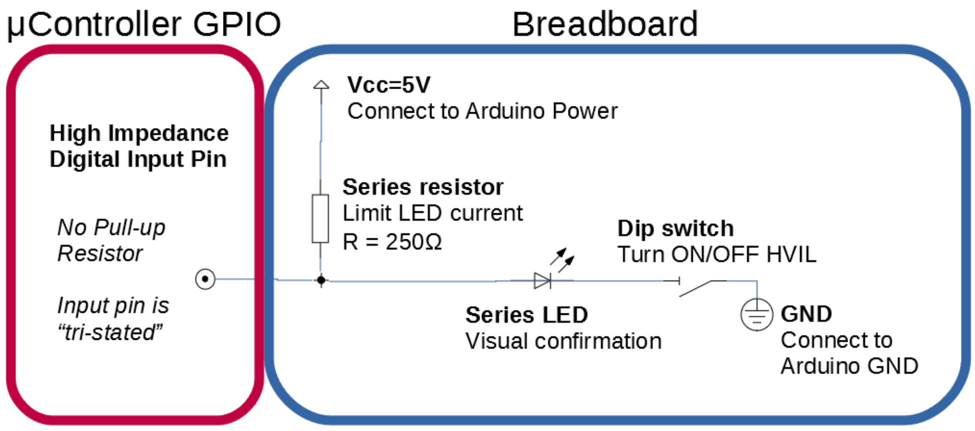

Figure 3 – DIGITAL INPUT: Test Circuit for High Voltage Interlock Loop (HVIL)

7.5.3 Measurement History Tracking

The *Measure Task* shall track whether **measurement history minimum / maximum values** have changed during execution (tracked measurement history values are listed in the Remote Terminal Task, Section 7.6). These measurement history values are stored in shared global variables which are accessed by the following tasks: *Data Logging Task* and *Remote Terminal Task*.

The *Measurement* Task shall update the measurement history values under the following conditions:

1. If **measurement history values have changed**, the *Measure Task* shall store the new measurement history values. A **flag** shall also be set to let the *Data Logging Task*

know which measurement history value has changed, i.e. min or max for a specific measurement has changed.

2. If a **reset flag** is set indicating the user has requested to reset EEPROM in the *Remote Terminal Task*, the following "reset" values shall be written to EEPROM:
    - HV Current *Reset Value* = 0
    - HV Voltage *Reset Value* = -1
    - Temperature *Reset Value* = 0

   NOTE: **Reset value "-1"** is used for measurement values whose **lowest range is 0**

7.6 Data Logging Task: EEPROM Measurement History

The *Data Logging Task* shall store measurement history values in EEPROM (on the ATMega board). Since EEPROM memory persists when power is removed, it is good for storing measurement history values and for tracking historical operating extremes. Measurement history shall include highest and lowest values for:

- HV Current
- HV Voltage
- Temperature

This task shall run at a **5s rate** since the EEPROM has **limited read/write cycles**

Measurement history data shall be stored in shared global variables for use by both the *Remote Terminal Task* and *Measurement Task.*

Values stored in EEPROM shall be updated under the following conditions:

1. If a **measurement history value change flag** has been set in the *Measure Task*, the *Data Logging* task shall update the corresponding value in EEPROM.
   → The EEPROM memory has a specified life of 100,000 write/erase cycles, so values are only written to EEPROM when they change
2. If a **reset flag** is set indicating the user has requested to reset EEPROM in the *Remote Terminal Task*, the following "reset" values shall be written to EEPROM:
    - HV Current *Reset Value* = 0
    - HV Voltage *Reset Value* = -1
    - Temperature *Reset Value* = 0

   NOTE: **Reset value "-1"** is used for measurement values whose **lowest range is 0.**

Once the *Data Logging* task has serviced a flag, it shall reset the flag.

You are permitted to use the Arduino language's **EEPROM.get()** and **EEPROM.put()** functions. These will write any data type or object to EEPROM.
EXTRA CREDIT POINTS (+30): Use the Arduino language's **EEPROM.read()** and **EEPROM.write()** functions instead of get() and put(). These write a single byte-sized integer value to EEPROM.

NOTE: The **first time the program** has run with the *Data Logging Task* added, it is a good idea to **reset** the **measurement history values** via the *Remote Terminal Task*.

7.7 Remote Terminal Task: Formal Communication Protocol

The *Remote Terminal Task* shall present a menu displaying options for requesting data from the System Controller. The menu will prompt for user input and display the desired output. Output values shall be collected from the System Controller using a formal communication protocol developed by you between the remote terminal and System Controller.

This task shall run at a **1s rate.**

It is recommended that you use the Serial0 port and the Arduino IDE Serial Monitor, however you are free to implement the remote terminal menu in another operating environment and over a different communication channel.

The remote terminal shall present the user with the following options:

```
[1]  Reset EEPROM
[2]  HV Current Range [Hi, Lo]
[3]  HV Voltage Range [Hi, Lo]
[4]  Temperature Range [Hi, Lo]

Enter your menu choice [1-4]: <char>
```

If option 1 is selected, an EEPROM reset flag shall be set – both the *Measurement Task* and *Data Logging Task* shall monitor this flag. Otherwise, the remote terminal host shall respond with the desired values as accessed via shared global variables.

7.8 SOC Task: Calculating Battery State of Charge (SOC)

The battery management system shall track the state of charge (SOC) of the high voltage battery. There are many methods to calculate state of charge, with more advanced systems using coulomb counting and neural networks to track the charge state of the battery. This system shall track SOC by **interpolating data** within a **2D lookup table** with respect to open circuit voltage and temperature as shown in Table 1.

This task shall run at a **100ms rate**.

To calculate SOC, the program shall use the measured temperature, terminal voltage and terminal current. Finding the correct **temperature entry** in the 2D table is simple; however, the **open circuit voltage** must be back calculated from the *measured terminal voltage and current* using Ohm's Law if the **current is non-zero.** ASSUME: Internal battery resistance is 0.5Ω and the battery can be modeled according to the circuit in Figure 4.

Table 2 – State of Charge w.r.t. Open Circuit Battery Voltage and Temperature

Temperature (°C)	**Open Circuit Voltage (V_{oc}) = ($V_{terminal} + R_{batt}I_{terminal}$)**				
	200V	250V	300V	350V	400V
-10°C	0%	10%	35%	100%	100%
0°C	0%	0%	20%	80%	100%
25°C	0%	0%	10%	60%	100%
45°C	0%	0%	0%	50%	100%

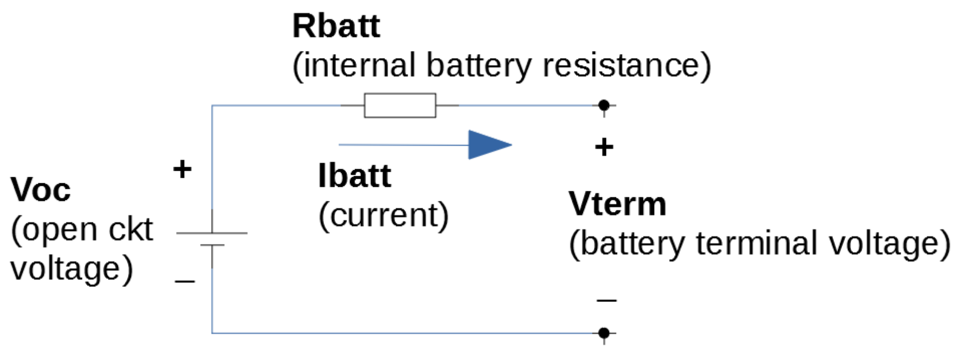

Figure 4 – Battery Internal Circuit Model

7.9 Contactor Task: Setting Contactors (signified by LED output)

As discussed in Section 7.9.1 "Simulated Contactor Output Circuit", contactors are a safety mechanism to protect the external world from the high voltage potential inside the battery. Within the context of this lab, the functionality of the *Contactor Task* is to show that logic exists for actuating the contactors properly so that when a power circuit is made available to actuate an actual contactor solenoid, it would function as specified.

For the purposes of this lab, the *Contactor Task* shall actuate the digital output pin associated with the contactor simulation circuit defined in Section 7.9.1.

This task shall run at a **100ms rate.**

States for contactors are:

1. <state1>: "OPEN" (default/entry state)
2. <state2>: "CLOSED"

The logic for moving between these states is as follows:

- Contactors shall be initially OPEN.
- Contactors shall be set to OPEN if either or both of the following conditions are met:
  - **Any alarm state** is "ACTIVE, NOT ACKNOWLEDGED" or "ACTIVE, ACKNOWLEDGED".
  - User requests TURN OFF BATTERY (i.e. OPEN contactors)
- Contactors shall transition to CLOSED when the user inputs a request to TURN ON BATTERY (i.e. CLOSE contactors) ***and*** **all alarms** are "NOT ACTIVE".

  NOTE: The alarm flags are defined in *Alarm Task*, Section 7.11.
  NOTE: Request to OPEN/CLOSE contactors shall come in the form of a **flag** from the *Display Task's* Battery ON/OFF screen. **The *Contactor Task* shall acknowledge the flag after it has acted upon it.**

In a real-world system, the OPEN/CLOSE command for the contactors will likely come from another embedded controller in charge of components that the high voltage battery rails interfaces with. This exercise puts that command in the user's hands and can be considered a debugging tool.

7.9.1 Simulated Contactor Output Circuit: DIGITAL OUTPUT

Description of circuit in real-world application:
High voltage contactors provide a means to disconnect the battery's high voltage rails from the external world. A typical contactor for this application is actuated by a solenoid, which requires more current to actuate that the microcontroller can source / sink
→ Important point to note! **Microcontrollers are limited in their capability to drive digital outputs over a few watts.** Most microcontrollers require external, board-mounted FETs to drive signals requiring higher power levels.

Implementation:
For the purposes of this exercise, the contactor shall be modeled using an LED in series with a resistor as shown in Figure 5. This circuit allows for software simulation with visual confirmation that the output pin is being actuated.

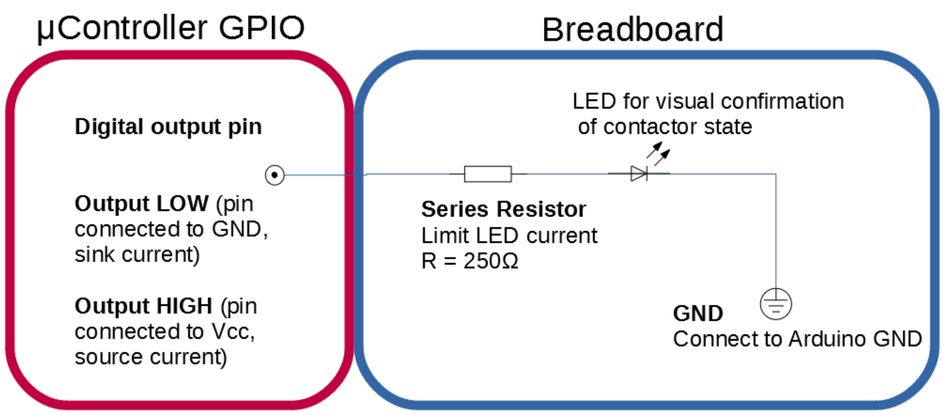
µController GPIO
Breadboard
Digital output pin
Output LOW (pin connected to GND, sink current)
Output HIGH (pin connected to Vcc, source current)
LED for visual confirmation of contactor state
Series Resistor
Limit LED current
R = 250Ω
GND
Connect to Arduino GND

Figure 5 – DIGITAL OUTPUT: Test Circuit for Simulating Contactors

7.10 High Voltage Interlock (HVIL) Interrupt Routine

The HVIL interrupt routine shall **immediately** act upon a **transition of the HVIL loop from CLOSED to OPEN**. This task is associated with safety and, as such, is the quickest software method available to OPEN the contactors should HVIL transition to an unsafe state.

Conditions for triggering interrupt:

1. HVIL transitions from CLOSED to OPEN.

Actions taken during the interrupt:

1. Set the *HVIL Alarm* to "ACTIVE, NOT ACKNOWLEDGED", i.e. <state2> of the alarm states defined in 7.11.
2. OPEN contactors by writing to the output port directly from the interrupt routine.

Interrupt implementation:

1. The interrupt trigger shall be tied to the HVIL input port using conditions described above.
2. The interrupt routine shall be tied to this task.
3. Use the attachInterrupt() function to initialize the interrupt.

7.11 Alarm Task

The following subsections describe the three alarms provided by the battery management system. Each alarm has three states:

1. <state1>: "NOT ACTIVE" (default/entry state)
2. <state2>: ACTIVE, NOT ACKNOWLEDGED
3. <state3>: ACTIVE, ACKNOWLEDGED

This task shall run at a **100ms rate**.

7.11.1 Alarm State Transitions

- Upon first triggering an alarm, the alarm shall transition from "NOT ACTIVE" to "ACTIVE, NOT ACKNOWLEDGED".
- Once an alarm is set to "ACTIVE, NOT ACKNOWLEDGED", the **touch screen** shall transition to the **alarm screen** (unless it is there already) where all alarm statuses are listed. Here the user shall have the option of acknowledging the "ACTIVE, NOT ACKNOWLEDGED" alarms.
- The user must acknowledge all "ACTIVE, NOT ACKNOWLEDGED" alarms in order to navigate away from the alarm screen (*alarms do not need to be acknowledged individually*).
- Once an "ACTIVE, NOT ACKNOWLEDGED" alarm is acknowledged on the alarm screen, the alarm shall transition to "ACTIVE, ACKNOWLEDGED". The *Alarm Task* receives user input of alarm acknowledgement via a **flag** sent by the *Touch Screen Task*.

- Alarms remain in ACTIVE states (<state2> and <state3>) until conditions are met for their dismissal. Conditions for dismissal are described for each alarm in the subsections below.

  NOTE: **No hysteresis shall be placed on the alarms conditions**; however, implementation of hysteresis is typical in a real-world system to avoid repeatedly triggering an alarm when conditions are on the edge.

7.11.2 High Voltage Interlock Alarm

This subtask shall handle transitioning the *HVIL alarm* to the "ACTIVE, ACKNOWLEDGED" and "NOT ACTIVE" states.

- The *HVIL Alarm* shall be set to "NOT ACTIVE" if HVIL is CLOSED.
- The *HVIL Alarm* shall be set to "ACTIVE, ACKNOWLEDGED" in the manner described in Section 7.11.1 Alarm State Transitions.
- *** The interrupt routine defined in Section 7.10 shall handle transitioning the *HVIL Alarm* to the "ACTIVE, NOT ACKNOWLEDGED" state. ***

7.11.3 Over Current Alarm

This subtask shall handle transitioning the *Overcurrent alarm* between the three alarm states:

- The *Overcurrent Alarm* shall be set to "ACTIVE, NOT ACKNOWLEDGED" if the current measurement lies **outside** or equal to the range [-5A, 20A].
- The *Overcurrent Alarm* shall be set to "ACTIVE, ACKNOWLEDGED" in the manner described in Section 7.11.1 Alarm State Transitions.
- The *Overcurrent Alarm* shall be set to "NOT ACTIVE" if the current measurement lies **inside** of the range (-5A, 20A).

7.11.4 High Voltage Out of Range Alarm

This subtask shall handle transitioning the *Voltage out of range alarm* between the three alarm states:

- The *Voltage Out of Range Alarm* shall be set to "ACTIVE, NOT ACKNOWLEDGED" if the high voltage measurement lies **outside** or equal to the range [280V, 405V].

- The *Voltage Out of Range Alarm* shall be set to “ACTIVE, ACKNOWLEDGED” in the manner described in Section 7.11.1 Alarm State Transitions.
- The *Voltage Out of Range Alarm* shall be set to “NOT ACTIVE” if the current measurement lies **inside** of the range (280V, 405V).

8.0 DELIVERABLES

Project 4 does not have a written report BUT you must submit updated diagrams in a pdf!

- Use the **report format** and include the following sections: **(1) Title Page, (2) Software Implementation** (include only diagrams in this section), **(3) Appendix** (explain code file names here)
- **Diagrams must be labeled!!** See Software Implementation section below.

We are giving you a break so you can focus on the course exam.

Software Implementation section:

- In addition to updating the **charts** and **diagrams** from **Project 3,** ADD the following TWO DIAGRAMS
- **Use case diagram** for remote terminal
- **Sequence diagram** for remote terminal
- Also note whether or not you completed the EEPROM extra credit.

Questions section:

- What me worry?... you'll do fine, I promise 😊

What to do with your CODE???:

- All code must follow the embedded coding standard.
- Code must be commented, explaining intended functionality.
- Zip all your code files and submit alongside the report.
- **Explain code file names in your Appendix.**

9.0 APPENDIX

9.1 Doubly Linked Lists: Insert and Delete Functions

A dynamic scheduler makes use of a doubly linked list as depicted in Figure 6. The dynamic scheduler works as described in the Scheduling Task in Section 7.2, where the list of tasks can grow or shrink by adding or deleting tasks. The **insert function** and **delete function** are described in the following subsections.

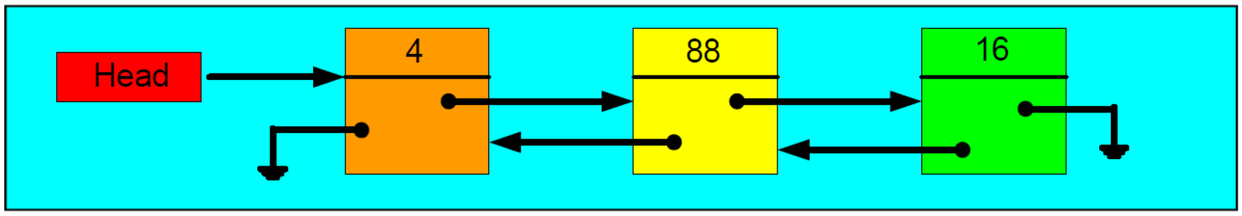

Figure 6 – Doubly Linked List: Graphical Representation

9.1.1 Doubly Linked List Insert Function

Invoking the insert function will add a TCB element to the linked list. This function simply adds a TCB node to the end of the list and updates all necessary pointers. Note that the TCB element to be inserted must be properly declared. See Figure 7 for a sample insert function.

9.1.2 Doubly Linked List Delete Function:

The delete function is a bit more involved but it simply removes a node from the list and updates the necessary pointers. This function should be designed so that a task can delete itself regardless of its position in the list (head, middle, or tail). See Figure 8 for pseudocode for the delete function.

©J.D. Olsen, Innovative Software and TA. Ltd.,

Note: Be sure to initialize all pointers to NULL before working with them.

```
// Insert function
// Arguments: Pointer to TCB node
// Returns: void
// Function: Adds the TCB node to the end of the list and updates
head and tail pointers
```

```
// This function assumes that head and tail pointers have already
been created and are global and that the pointers contained in the
TCB node have already been initialized to NULL
// This function also assumes that the “previous” and “next” pointers
in the TCB node are called “prev” and “next” respectively
*/

void insert(TCB* node)
{
if(NULL = = head) // If the head pointer is pointing to nothing
{
head = node; // set the head and tail pointers to point to this node
tail = node;
}
else // otherwise, head is not NULL, add the node to the end of the
list
{
tail -> next = node;
node -> prev = tail; // note that the tail pointer is still pointing
// to the prior last node at this point
tail = node; // update the tail pointer
}
return;
}
```

Figure 7 – Doubly Linked List: Sample Insert Function

```
// If the head pointer points to NULL, the list is empty and there is
nothing to delete

// Otherwise, if the head pointer and tail pointer are equal, there
is only one node in the list, set the head and tail to NULL

// Otherwise, if the head pointer is equal to the node we want to
delete set the head pointer to the next node in the list (head = head
-> next)

// Otherwise, if the tail pointer is equal to the node we want to
delete, set the tail pointer to the previous node in the list

// Otherwise, we are in the middle of the list update the previous
and next pointers of the surrounding nodes

// NOTE: this is just pseudo code and does not show all the pointer
updates
// Be particularly careful to properly set the previous and next
pointers of deleted nodes to NULL
```

Figure 8 – Doubly Linked List: Pseudocode for the Delete Function