PROJECT 3: "FLESHING OUT THE STUBS"

WORKING WITH DYNAMIC TASK QUEUES, INTERRUPT SERVICE ROUTINES, INTRA-SYSTEM COMMUNICATION, AND ANALOG & DIGITAL I/O TO IMPLEMENT THE SECOND PHASE OF A BATTERY MANAGEMENT SYSTEM

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

REV: 2 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 1 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 7.8)*:* 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 7.6)*:* Constant value of SOC=0 • ***Contactor task*** (Section 7.7)*:* Contactor state diagram updated to use HVIL alarm flag as an input for state transition criteria • ***Alarm task*** (Section 7.9)*:* Calculates & reports alarms based on measurement data

3.0 BACKGROUND

Did well on Project 2 and excited to design phase two!

3.1 Cautions and Warnings

Always practice safe software engineering… don't leave unused variables laying around.

Throwing your completed but malfunctioning design on the floor, stomping on it, and screaming **"work you stupid fool, work"** is typically **not the most effective debugging technique**… although it is perhaps one of the more satisfying. When you are debugging your code, writing it, throwing it away, and rewriting it again several dozen times, this does little to fix what is most likely a design error. Such an approach is not highly recommended, but can keep you entertained for hours… particularly if you can convince your lab partner to do it.

4.0 PROJECT OBJECTIVES

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

- Introduce a **dynamic task queue**: Design, implement, and manage a dynamic task queue
- Introduce **hardware timing functions**: Design, implement, and utilize a hardware-based system time base
- Introduce **interrupts and ISRs**: Begin to work with internal and external interrupts and interrupt service routines
- Introduce several **peripheral devices** and work with basic input and output operations
    - Design and incorporate a **console keypad** and a **local display console** into the system.
    - **Implement I/O:** Interface with external circuits to measure data using analog and digital signals
    - Design and incorporate **drivers** for the measurement subsystem
- Implement features to enhance **safety**
- **Add features and capabilities** to an existing product:
    - Implement and test the new features and capabilities of the system,
    - Amend existing UML diagrams to reflect the new features: utilize UML diagrams to model new, dynamic aspects of the system

- Continue to improve skills with **pointers**, the passing of pointers to subroutines, and manipulating them in subroutines.

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 an upcoming 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**.**

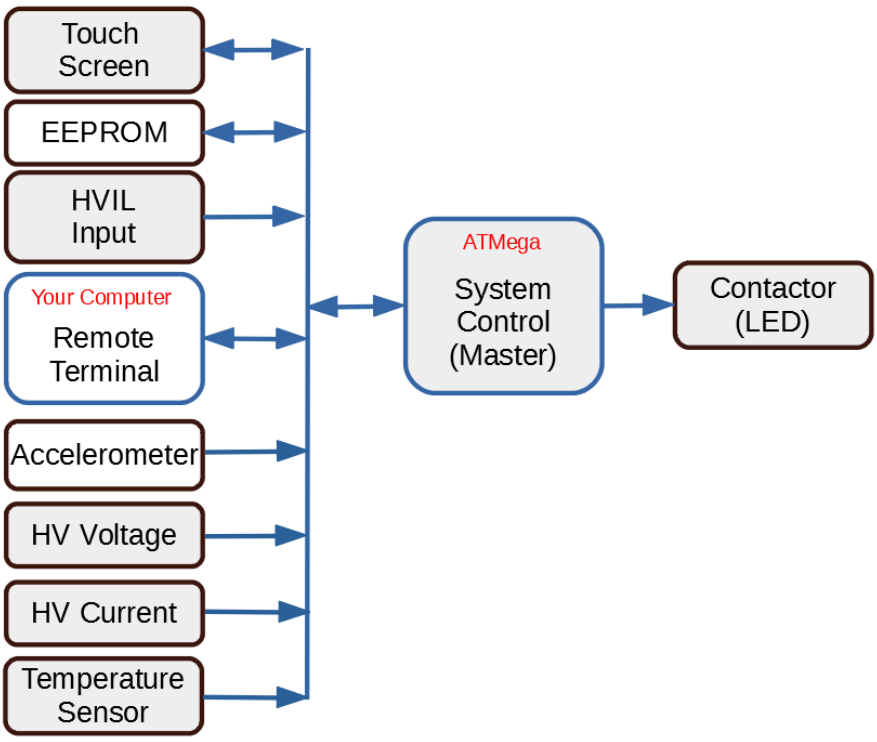

Figure 1 – High-Level *Partially Complete* Block Diagram of the Battery Management System (BMS). *NOTE: Items in light grey are implemented in this project*

7.0 SOFTWARE ARCHITECTURE

Your code will include the following tasks:
Only 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

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
  - 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.

There are several methods for implementing the task queue and scheduling. We will transition from the **Round Robin Scheduler** to a **Dynamic Scheduler** in this project.

7.2.1 Dynamic Scheduler

The dynamic scheduler in this project will be much like the round robin scheduler in that it runs each task from the task queue in succession; however, the task queue will now be implemented as a **doubly linked list** of TCBs rather than an array of pointers to type TCB.

- If a task is not to be run during the current round (i.e. the flag for executing that task is not set), the scheduler skips execution of that task.
- Tasks shall be pre-emptable (each task may be interrupted).
- If a task has nothing to do, it shall exit immediately.

In this lab, the ATMega **Timer1 interrupt** shall pace execution of all tasks within the main loop – more on this to follow.

7.2.2 Dynamic Task Queue (TCB Linked List) Implementation

How to build the task queue?
The task queue is a **doubly linked list of TCBs**. The *Schedule Task* executes each TCB in the doubly linked list and then returns to the main loop where it is called again once the timing interrupt flag has been set – more on the timer to follow.

Creating the Doubly Linked List of TCBs:
The TCBs shall be initialized with all task and data pointers at startup as was done in the previous project. To **initialize the doubly linked list of TCBs**:

- The TCB's linked list pointers, "next" and "prev", shall be set to point to the next and previous TCB in the list.
  NOTE: A value of NULL shall be assigned to the "prev" pointer at the head and the "next" pointer at the tail of the linked list.

7.2.3 Associating Real Time with the Scheduler

The *Schedule Task* 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 (the next section provides more detail about implementing the hardware timer). Pseudocode below shows how the system time base flag functions to pace execution within the main loop:

```
while(1)
{
  if( 1 == timeBaseFlag )  // check if timeBaseFlag has been set by timer
  {                        //   interrupt (interrupt runs once per global
                           //   time base period)

    timeBaseFlag = 0;      // clear timeBaseFlag so that the loop will only
                           //   run again after timer interrupt sets flag

    Scheduler();
  }
}
```

Be sure to implement your queue so that when it gets to the last element, it wraps back around to the head of the linked list.

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.

To implement a timer interrupt in the ATMega, use the **Timer1 library** included with the lab assignment. The code snippets below show how to use this library:

```
#include <TimerOne.h>

/****  STARTUP ROUTINE  ***/
   /*  Set the Timer 1 interrupt request (IRQ) frequency  */
   Timer1.initialize( time_in_uS );    // Set the timer period in uS

   /*  Set the timer 1 IRQ function to be called  */
   Timer1.attachInterrupt(timerISR);   // Attach the interrupt service
                                       //    routine (ISR)
```

```
   /*  Start the timer  */
   Timer1.start();                    // Start the timer


/****  ISR Function  ****/
void timerISR()
{
    **  set global time base flag here  **
}
```

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.

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>

Note that HVIL Status, HV Current & Voltage, and temperature (highlighted in grey) will be implemented in hardware at this phase. State of charge data will be software generated according to specifications in 7.6.

7.4.2.2 Alarm Screen

The **Alarm Screen** shall display the value of each of the alarms as listed in Section 7.9.

- 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.

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.

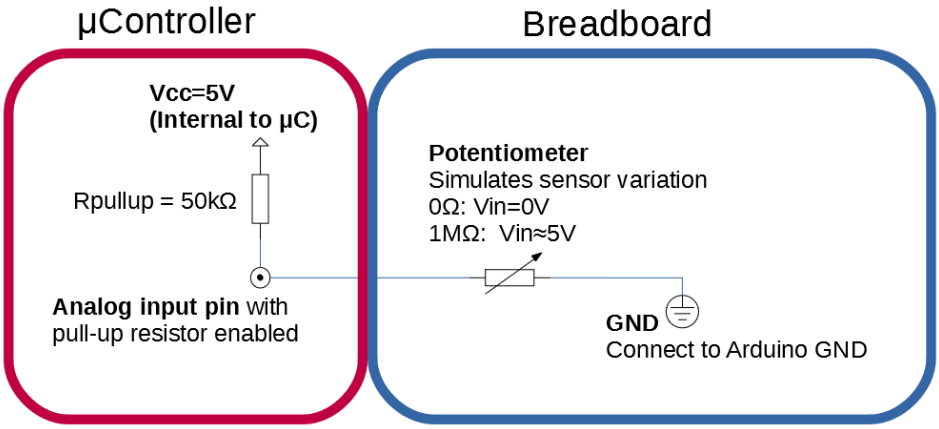

Figure 2 – ANALOG INPUT: Test Circuits for Sensors in Range [0, 5V]

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!

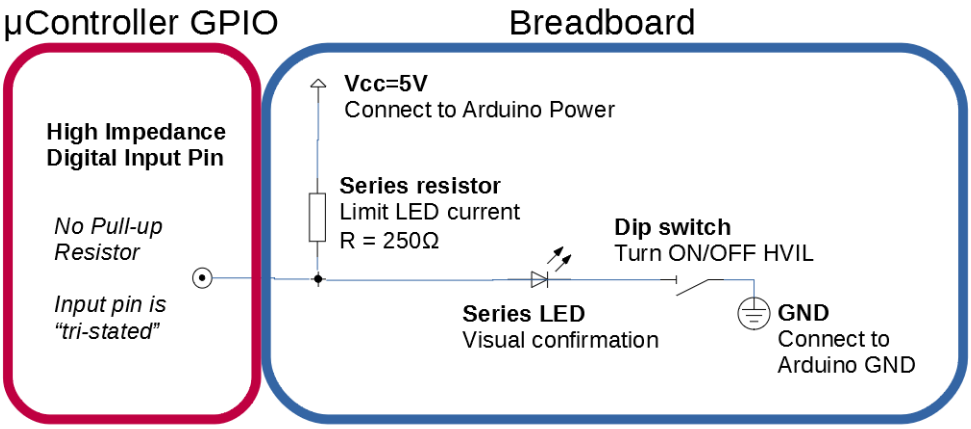

Figure 3 – DIGITAL INPUT: Test Circuit for High Voltage Interlock Loop (HVIL)

7.6 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.

For this phase of the project, the following value shall be used for the SOC. As mentioned in Section 7.4.2, the touch screen display task will share access to the SOC variable.

- State of Charge:
  - Constant value of [0]

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

As discussed in Section 7.7.1 "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 7.7.1.

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:
  - The *HVIL Alarm* 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*** the *HVIL Alarm* is "NOT ACTIVE".

  NOTE: The *HVIL Alarm* is defined in Section 7.9.

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.

7.7.1 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 4. This circuit allows for software simulation with visual confirmation that the output pin is being actuated.

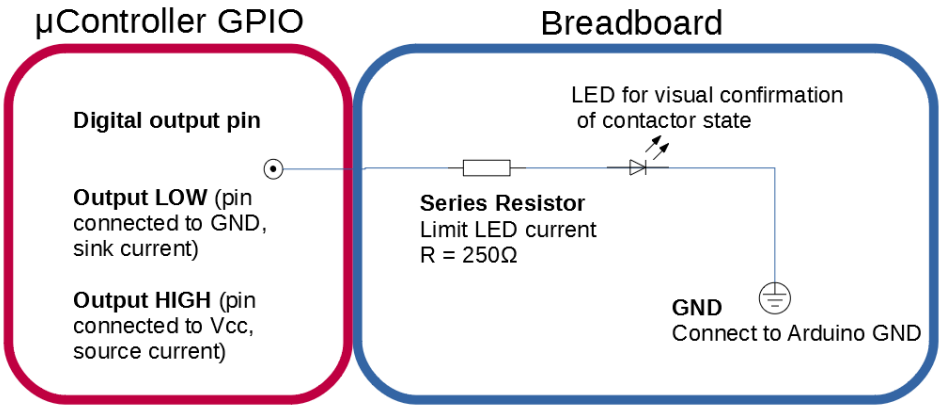

Figure 4 – DIGITAL OUTPUT: Test Circuit for Simulating Contactors

7.8 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 7.9.
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.

7.9 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

7.9.1.1 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.

7.9.1.2 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 7.9.1.1 Alarm State Transitions.
- *** The interrupt routine defined in Section 7.8 shall handle transitioning the *HVIL Alarm* to the "ACTIVE, NOT ACKNOWLEDGED" state. ***

7.9.1.3 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 7.9.1.1 Alarm State Transitions.
- The *Overcurrent Alarm* shall be set to "NOT ACTIVE" if the current measurement lies **inside** of the range (-5A, 20A).

7.9.1.4 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 7.9.1.1 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

Write your **Project Report** according to the rubric. The project report will be graded according to the rubric.

Software Implementation section:

- Update the **charts, diagrams** and **pseudocode** listed in **Project 2**
  - ➔ Each figure must be **REFERENCED** in the text. Use the figures to explain your software design…

    "…The **system block diagram** of **Figure 1** shows the ports and pin numbers that the inputs X, Y, Z go into…"

    "…The **data flow diagram** of **Figure 2** shows how data flows from input X to tasks A, B, C…"

Questions section:

- What is the **resolution** of each of the **[0, 5V] analog inputs** being simulated via potentiometer (**HV voltage**, **HV current,** and **Temperature**)?
  - Resolution refers to the physical value that each bit represents.
  - Hint: The analog input returns a 10bit value.
- List of all **inputs** and **outputs** categorized as either **digital** or **analog signals**

NOTE: In a formal report, your numbers, raw data, pseudocode, etc. should go into an **Appendix**. The body of the report is for the discussion, don't clutter it up with a bunch of other stuff. You can always refer to the information in the Appendices, as you need to.

If any of the above requirements is not clear, or you have any concerns or questions about what you're required to do, please do not hesitate to ask.

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 Working with Interrupts

When working with interrupts, it is important to keep in mind the following:

1. Interrupts are **asynchronous events** that can originate in the software or in the hardware; inside or outside of the processor.
2. The interrupt level (the **priority**) of each interrupt is either predetermined by the design of the processor or by processor-specific interface software.
3. One must write an **interrupt service routine (ISR)**. This routine provides the body of the interrupt – the code that is to be executed when the interrupt occurs. The ISR should be short and concise. It should only contain enough code to get a job done or to apprise the system that something needs to be done. The following are definite no's when writing an ISR:
   - The ISR should not block waiting on some other event.
   - The ISR should not work with semaphores or monitors.
   - The ISR should not contain a number of calls to functions or perform recursive operations (judgment call here). It's easy to blow the stack.
   - The ISR should not disable interrupts for an extended time.
   - The ISR should not contain dozens of lines of code.
4. One must store the address of the ISR in the **interrupt vector table**. The table is essentially an array of function pointers, indexed according to the interrupt number. Providing the ISR and getting its address (pointer to that function) into the interrupt vector table varies with different environments. For the Arduino environment, use the attachInterrupt() function.
5. One must **enable the interrupt**.
   This can be done globally – all interrupts are enabled, or locally – interrupt 'x' is enabled. Unless the interrupt is enabled, even if it occurs physically, it will not be recognized by the system.
6. One must **acknowledge** (or recognize) **the interrupt** when it occurs unless otherwise taken care of by the system (generally not the case). This has the effect of resetting the interrupt and allowing further interrupts of the same kind to recur. If the interrupt is not acknowledged, it may occur once then never again. The acknowledgement is usually done in the ISR. The Arduino environment automatically handles interrupt acknowledgment (see the Interrupts section of the Arduino Resources document for more information).

7. One must **exit the interrupt properly**.
   Unlike a simple function call, the return from an interrupt is a bit more involved. Generally, there is a specific statement (not a simple return) that is used for exit and cleanup.
8. All **variables accessed by the interrupt** that are not constant should be declared **VOLATILE** since the interrupt service routine can change the value of the variable outside the control of a code section in which it appears. Think of the interrupt as a concurrently executing thread.
   Note that declaring a variable VOLATILE is a directive to the compiler to load the variable from RAM instead of a temporary storage register in the CPU.