PROJECT 2: "STUBBING OUT THE SYSTEM"

WORKING WITH SYSTEM ARCHITECTURE, A ROUND ROBIN SCHEDULER, TASK CONTROL BLOCKS, INTER-TASK COMMUNICATION, & GENERAL I/O TO IMPLEMENT THE FIRST PHASE OF A BATTERY MANAGEMENT SYSTEM

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

REV: 19 Jan 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 **first phase** in the development of a simple battery management system for high voltage electric transportation applications. The current phase focuses on "stubbing out the system": design and development of the basic system architecture, modeling the instrumentation, establishing the high-level data/control flow, managing a basic scheduler (round robin), creating an initial display driver, and alarm annunciation functions.

The **initial deliverables** in this project include the high-level system architecture, the ability to perform a subset of the necessary control, and portions of the display components. Subsequent phases (the next few projects) will continue to evolve the system architecture and flow of control, extend the driver development, and incorporate additional capabilities into both the measurement and display subsystems.

The **final subsystem** will be capable of collecting data from several different types of sensors, processing the data from those sensors, displaying it locally, and then transmitting it over a local area network to a remote terminal.

1.2 Layout of the Project / Guidelines for Success:

This series of projects is intended to mimic the design and development life cycle for bringing up an embedded system. Each project will present a set of software and hardware specifications…. lucky you!... the specs are already written! **These specifications will guide you in how to implement both your code and the surrounding hardware which simulates the system.**

A **revision table**, shown in Section 2.0 will serve as your guide in the development process. As we move into further development phases (future projects), the revision table will help you understand what has been changed / added / removed from the previous project. Hopefully this will get you in the habit of versioning both your specifications and your code. This is vital to traceability and will play an important role in your industry career.

Keep in mind that all projects from this point on will build on each other – it is important to ensure your project functions well!

1.3 Recommended Design Approach

This project involves designing, developing, and integrating a number of software components. On any such project, the approach one takes can greatly affect the ease at which the project comes together and the quality of the final product. To this end, we strongly encourage you to follow these guidelines:

1. **Develop all of your UML diagrams** first. This will give you both the static and dynamic structure of the system.
2. **Block out the functionality of each module**. This analysis should be based upon your use cases.
   This will give you a chance think through how you want each module to work and what you want it to do.
3. Do a **preliminary design of the tasks and associated data structures**. This will give you a chance to look at the big picture and to think about how you want your design to work before writing any code.
   This analysis should be based upon your UML class/task diagrams.
4. Write the **pseudo code** for the system and for each of the constituent modules.
5. Develop the **high-level flow of control** in your system. This analysis should be based upon your activity and sequence diagrams. Then code the top-level structure of your system with the bodies of each module stubbed out.
   This will enable you to verify the flow of control within your system works and that you are able to invoke each of your procedures and have them return the expected results in the expected place.
6. When you are ready to create the project in the Arduino IDE, it is strongly recommended that you follow these steps:
   a. Build your project.
   b. Correct any compile errors and warnings.
   c. Test your code.
   d. Repeat steps a-c as necessary.
   e. Write your report
   f. Demo your project.

g. Go play.

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

3.0 BACKGROUND

Did well on Project 1, put in at least 100 hours per week. No need to sleep – it's over rated anyway.

3.1 Cautions and Warnings

Never try to run your system with the **power turned off**. Under such circumstances, the results are generally less than satisfying.

Always keep only a single copy of your source code. This ensures that you will always have a maximum amount of disk space available for games, email, and some interesting pictures. **If a code eating gremlin happens to destroy your only copy, not to worry, you can always retype and debug it again….**

4.0 PROJECT OBJECTIVES

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

- Introduce and work with **formal design methodologies and specifications**:
  - Utilize **UML diagrams** to model static and dynamic aspects of the system - Use Cases, Functional Decomposition, Sequence Diagrams, State Charts, and Data and Control Flow Diagrams
  - Identify **major functional blocks**
  - **Architect the system software** as a collection of **tasks**
- Use software delay/timing functions to develop a simple **time-based operating system kernel** and **scheduler** that will schedule and dispatch tasks
- Introduce task control block structures to perform **intertask / interprocess communication**
- Introduce simple tasks and a task queue**: Stub out, model, and simulate the desired behavior** of tasks
- Use **basic input/output operations:**
  - Begin the design of a **driver** to control a touch screen **display**
  - Use external circuits to **simulate digital I/O**
- **Implement and test** the system
- Empirically determine **execution time** of each such task
- Share data between tasks using **C pointers and data structs**: develop and build background on C 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

The following two sections define the system architecture in terms of both the (1) Hardware Architecture and (2) Software Architecture. The system architecture specifies the functional components and their interactions in hardware and software. As such, the following sections are divided between the hardware and software architectures.

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 **digital inputs highlighted in grey** will be implemented in hardware and the **analog input sensors highlighted in blue** will be stubbed out in code.

Digital:

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

Analog:

- HV terminal voltage
- HV terminal current
- Temperature
- Accelerometer

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 prototype will be implemented using an ATMega development board, a touch screen display, and external circuitry to mimic a HV battery system.

While **we will not be implementing everything shown at this point**, the diagram in Figure 1 is intended to give you an idea of how the final system architecture will be structured from a hardware perspective. This diagram 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**.**

Note this is not a complete block diagram, you will need to format the diagram properly by adding hardware I/O information and signal information. ONLY INCLUDE ITEMS IN GREY FOR YOUR BLOCK DIAGRAM.

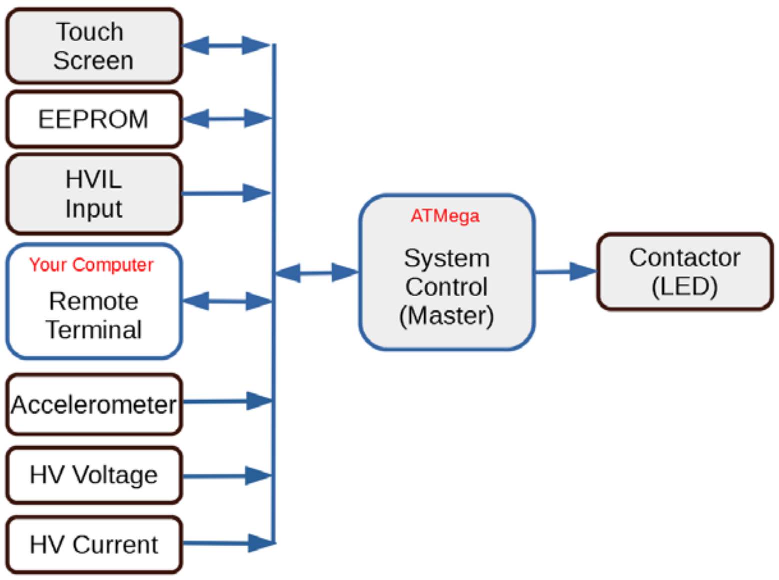

Figure 1 – High-Level *Partially Complete* Block Diagram of the Battery Management System (BMS). *NOTE: Items in light grey are implemented in this project*

7.0 Software Architecture

The software architecture describes the structure and functionality of the software. Software architecture is about making fundamental structural choices that are costly to change once implemented. These are documented in UML (Unified Modeling Language) diagrams, aka software architecture diagrams.

Creating solid code structure is an integral part of embedded system design. Doing so aids in debugging and maintaining/updating code. Your code shall be structured using the TCBs described in the Appendix, **Section 9.1 Implementing the TCB**.

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 ((see **Section 9.1** on how to declare & initialize these))
  - Task data structures
  - Task control blocks (TCB)

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 begin with the **Round Robin Scheduler,** and as the quarter progresses, the complexity of our scheduler will progressively increase until we've created a real-time embedded operating system.

7.2.1 Round Robin Scheduler

The Round Robin (RR) scheduler is one of the simplest of all scheduler algorithms. It runs each task from the task queue in succession, assigning each task an equal time slice (time quanta) to run. For now, we will assume that each of our tasks take the same time to execute.

- 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 not be pre-emptable (each task will run to completion without interruption).
- If a task has nothing to do, it shall exit immediately.

In this lab, the Arduino software timer function, **delay()**, shall pace execution of all tasks within the main loop – more on this below.

7.2.2 Round Robin Task Queue Implementation

How to build the task queue?
The task queue is an array of elements consisting of pointers to variables of type TCB. The *Schedule Task* executes each TCB in the array and then returns to the main loop where it is called again once the software delay is expired.

The TCB elements in the queue correspond to tasks identified in Section 7.0. Extra elements in the array can provide space for future capabilities.

7.2.3 Creating the Round Robin Scheduler

Write a program that will run forever. Typically this consists of an infinite while loop; however in the Arduino language, we use the **loop()** function which runs indefinitely. Your code should look something like this:

```
unsigned long time_in_ms;
void loop()
{
  Scheduler();
  delay(time_in_ms);
}
```

The Scheduler() task should index through the task queue and call each of the functions in turn. Be sure to implement your queue so that when it gets to the last element, it wraps back around to the head of the queue.

7.2.4 Associating Real Time with the Scheduler

You will **add a timing delay to your loop** so that you can associate real time with your main loop.

For example, if the Scheduler task takes 80ms to run and the delay is set to 20ms, each task in the scheduler executes once per 100ms. By incrementing a **counter** each time the loop executes, we can associate that counter with the cycle time of the loop – 100ms in this case.

This is how we will create **timers / counters** to implement the cycling rates for each of the measurement items – see Section 7.4 Measurement Tasks: Sensor Measurements.

To accomplish this, we use the delay function: "**delay(unsigned long time_in_ms)**". Simply call this function with the delay in milliseconds as its argument. Remember Project 1…

The scheduler shall execute all tasks at a **1sec period (1Hz).** In other words, each task shall execute once per second. You will need to figure out what delay is required for this to occur.

7.3 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.

7.3.1 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.

7.3.1.1 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.

7.3.2 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*.

7.3.2.1 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 only the HVIL Status (highlighted in grey) will be implemented in hardware at this phase. All other data will be software generated according to specifications in 7.4 and 7.5.

7.3.2.2 Alarm Screen

The **Alarm Screen** shall display the value of each of the alarms as listed in Section 7.7.

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

7.3.2.3 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.

7.4 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.

At this stage in the project, most of the measurements are simulated by software other than HVIL. As the project progresses, all measurements will be physically sampled.

7.4.1 Temperature, HV Current & Voltage Measurements

The measurement task shall update the following measurement values as described. As mentioned in Section 7.3.2, the touch screen display shall access this data via shared measurement variables.

Each measurement value should be held for the amount time indicated.
For a 1s rate, each value is held for 1s.

- Temperature:
  - Cycle thru values [-10, 5, 25] at a 1s rate

- HV Current:
  - Cycle thru values [-20, 0, 20] at a 2s rate
- HV Voltage:
  - Cycle thru values [10, 150, 450] at a 3s rate

7.4.2 High Voltage Interlock Loop (HVIL)

At this stage, there is only **one physical measurement being taken:**

- HV Interlock Loop

This measurement shall be stored in a shared variable for inter-task data exchange with the display, similar to the other measurement values.

7.4.2.1 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 2 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!

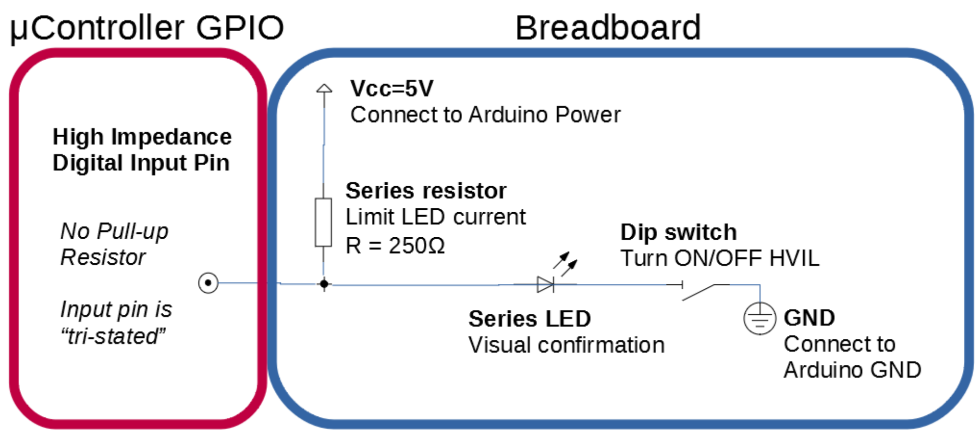

Figure 2 – DIGITAL INPUT: Test Circuit for High Voltage Interlock Loop (HVIL)

7.5 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 values shall be used for the SOC. As mentioned in Section 7.3.2, the touch screen display task will share access to the SOC variable.

- State of Charge:
  - Cycle thru values [0, 50, 100] at a 1s rate

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

As discussed in Section 7.6.1 "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 7.6.1.

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 transition to CLOSED when user inputs a request to TURN ON BATTERY (i.e. CLOSE contactors).

  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.

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

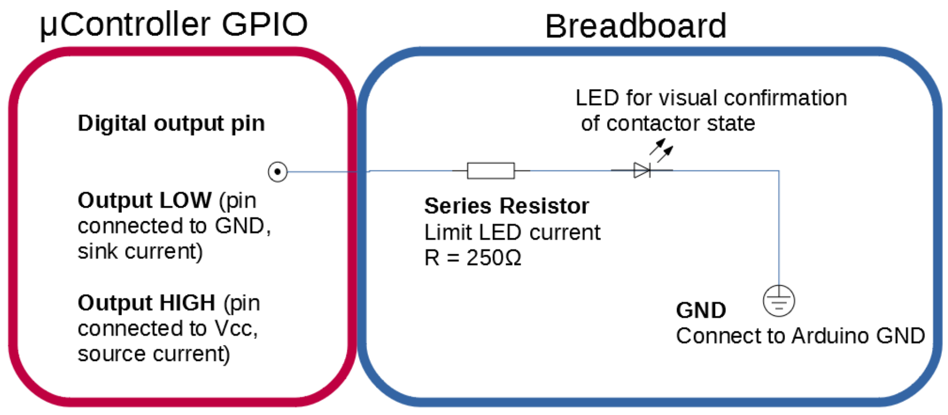
µController GPIO
Breadboard
Digital output pin
Output LOW (pin connected to GND, sink current)
Output HIGH (pin connected to Vcc, source current)
LED for visual confirmation of contactor state
Series Resistor
Limit LED current
R = 250Ω
GND
Connect to Arduino GND

Figure 3 – DIGITAL OUTPUT: Test Circuit for Simulating Contactors

7.7 Alarm Task

There are three alarms defined for the battery management system, each has three states:

1. <state1>: "NOT ACTIVE"
2. <state2>: ACTIVE, NOT ACKNOWLEDGED
3. <state3>: ACTIVE, ACKNOWLEDGED

The following values shall be used for the shared global variables associated with each alarm. As mentioned in Section 7.3.2, the touch screen display will access these shared global alarm variables to display.

- HVIL (High Voltage Interlock Alarm):
  - Cycle thru values [<state1>, <state2>, <state3>] at a 1s rate
- Overcurrent
  - Cycle thru values [<state1>, <state2>, <state3>] at a 2s rate
- High Voltage Out of Range
  - Cycle thru values [<state1>, <state2>, <state3>] at a 3s rate

8.0 DELIVERABLES

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

Include figures of the following in the **Software Implementation** section.

➔ 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…"

- **System block diagram** showing the ATmega input and output ports (and port numbers) labeled per I/O component
- **Structure chart** showing functional decomposition of tasks within the System Controller.
- **Class/task diagram** showing the structure of tasks within the System Controller as reflected in the structure chart.
  - These diagrams should include the task name, global variables used in the task and how they are structured with respect to each other.
- **Data flow diagrams** for all inputs and outputs
  - Hint: The touch screen can be considered as one input/output unit
- **Activity diagram** for the showing the System Controller's dynamic behavior from initial entry into the loop() function.
  - This will include the Scheduler task and all the tasks that it calls.
  - Hint: The scheduler task can be represented by branch and merge – the branch condition is the for loop index, and each branch is a task. After the scheduler, the activity diagram should enter the delay function and loop back.
- **Touch screen**
  Hint: There should be a use case diagram, sequence diagram and front panel display for EACH SCREEN
  - **Use case diagram** for the touch screen display.
  - **Sequence diagram** for the touch screen display.
  - **Front panel design** for the touch screen display.
- **State diagrams:**
  - Each alarm (Hint: Each alarm value is a state).
  - Contactors
  - Touch screen display (Hint: Each display screen is a state).

- Pseudocode for all tasks per the functions listed in Section 7.0 Software Architecture…. pseudocode makes excellent comments in your code!
    - **Place pseudocode in the appendix and reference it in the Software Implementation section..** "Pseudocode for each task is located in the Appendix". Points for pseudocode will be assigned in the Software Implementation section of your report.

Include this in the **Questions section**:

- Execution time of each task
    - Hint: Execution time can be found by toggling an output port and reading that signal on an oscilloscope OR by using the millis() function.
- What delay is required for a 1Hz cycle time?
- List of all inputs and outputs

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 Implementing the TCB

9.1.1 Tasks and Task Control Blocks (TCBs)

The BMS is comprised of a number of tasks. The function(s) and data for each task are located in a **TCB (Task Control Block)** structure – implemented as a C struct. The TCB struct contains all data required to both call the task function and pass data to the task function. Each task has its own TCB.

Each TCB has **four members** as shown in the C typedef declaration from Figure 4:

i. Pointer to a **function** taking a void* argument and returning a void.
ii. Pointer to void used to reference the **data** for the task.
    See the next Section 9.1.2 "Data Structures for Task Control Blocks (TCBs)" for a definition of this pointer.
iii. Pointers to the next and previous TCB in a **linked list** data structure, "next" and "prev". **These pointers are not used for the Round Robin Scheduler and shall be initialized to NULL.**

Such a structure allows various tasks to be handled using function pointers.

The following gives a C typedef declaration for such a TCB:

```
struct MyStruct
{
    void (*myTask)(void*);     // Pointer to function "myTask()"
    void* taskDataPtr;         // Pointer to data structure "taskData"
    struct MyStruct* next;     // Linked list pointer "next"
    struct MyStruct* prev;     // Linked list pointer "prev"
};
typedef struct MyStruct TCB;
```

Figure 4 – C Struct Typedef for the Task Control Block (TCB) with Linked List

NOTE: Within the C language, using "void" as a datatype allows one to type cast to different datatypes at compile-time or runtime. In our case, this type casting would apply to both the task's function and data structure, "myTask" and "taskDataPtr" respectively from the example above. In this way, the TCB typedef is generic to all functions in our system!

Implementation Notes:

1. All function declarations (one function per task) shall accept a pointer to void with a return of void.
2. The pointer in the task argument (function argument for "myTask()" function in Figure 4) must be **re-cast as a pointer to that task's data structure type before it may be dereferenced**… see the next section on constructing each task's data structure.
3. Each task shall have its own TCB!

9.1.2 Data Structures for Task Control Blocks (TCBs)

The TCB data structure consists of pointers to global variable data required/modified by the task (the function "myTask" in this example). This data structure is represented by TCB member "taskDataPtr" as seen in Figure 4.

A representative example of **declaring and initializing a task's data structure** is given in Figure 5, where the global variable "fooData" would be an integer required by the "fooTask" TCB's function.

```
/**  Data Structure for the “fooTask” TCB  **/

   int fooVariable;                        // Create “fooVariable” global variable

   typedef struct fooDataStruct            // Create data structure typedef
   {                                       //   “fooTaskData” of type “fooDataStruct”
       int* fooVariablePtr;
   } fooTaskData;

   fooTaskData fooData;                    // Create data structure “fooData” to be
                                           //   used in TCB
   fooData.fooVariablePtr = &fooVariable;  // Initialize fooData’s variable(s)
```

Figure 5 – **Task Data Structure** for the "fooTask" TCB: C Data Struct Declaration

The data structure's variables must have **global scope**. This facilitates **inter-task data exchange/communication** through **shared variables**, meaning that tasks needing to communicate/share data with each other will share a variable (i.e. they both use the same variable). Typical programming practices recommend limiting the number of global variables to reduce chances of data corruption (since all functions can access those variables); however, by encapsulating these variables in a data structure, it strictly limits the variables to which functions have access.

Based upon the requirements specification, the shared variables are defined to hold measurement data, status, alarm, and flag information. To properly encapsulate and ensure access by the correct tasks, all shared global variables must be passed to tasks using the task data structures!

9.1.3 Task Queue & Scheduling: Initializing Elements of the Task Queue Array (TCB Pointer Array)

Each element of the task queue array shall be a pointer to type TCB, representing one of the tasks identified in Section 7.0 Software Architecture.

Each of the TCB pointers in the task queue array shall be initialized as shown in Figure 6, summarized as such:

- The TCB's **function pointer** shall be initialized to point to the proper task. For example, TCB element zero should have its function pointer initialized to point to the *Measure Task*.
- The **data pointer** shall be initialized to point to the proper task data structure used by that task. For example, if "MeasureData" is the data structure for the *Measure Task*, then the data pointer of the TCB should point to "MeasureData".
- The **linked list pointers**, "next" and "prev" shall be set to NULL until we transition to scheduler using a linked list.

```
/**  Function Declaration for "fooTask" TCB  **/
   void fooTaskFnc(void*);              // Function declaration for "fooTaskFnc"


/**  Initializing "fooTask" TCB  **/
   TCB fooTask;                         // Initialize TCB using typedef defined in
                                        //    Figure 4
   fooTask.myTask = &fooTaskFnc;        // Initialize TCB "myTask" to point to
                                        //    function fooTaskFnc()
   fooTask.taskDataPtr = &fooData;      //Initialize fooTask's data struct
```

Figure 6 – Full Initialization of a TCB

9.1.4 How to Access Data from your Task's Data Structure Pointer, "taskDataPtr", Variable

You may have trouble implementing your task function since the variable passed to your task function is declared as (void*).

- **Question**: How does one access data from the task's data structure pointer when the function declaration uses the (void*) datatype?
- **Answer:** In the code for your task, re-cast the (void*) pointer as a pointer to that task's data structure type before it is dereferenced.

In the examples posed in Figure 4 and Figure 5, you would need to recast the (void*) function argument for "fooTaskFnc" to type "TaskData" before being able to access the pointer to variable "data". The previous examples are used to show the re-casting process in Figure 7.

```
/** Accessing the TCB’s Data Structure from within “fooTaskFunc”  **/
   void fooTaskFnc(void* arg)             // Function declaration for “fooTaskFnc”
   {
       /** Re-cast “arg” to fooTaskFnc’s data structure type “fooTypeData” **/
       fooTaskData* localDataPtr = (fooTaskData*) arg;


       /** Access and increment “fooVariable” **/
       *(localDataPtr->fooVariablePtr) = *(localDataPtr->fooVariablePtr) + 1;

   }
```

Figure 7 – Coding the TCB Task: Re-Casting to Access the TCB's Data Structure Pointer (void*) from within the Task Function Code

9.2 Tips for Building Circuits

1. Measure the resistance of your ground connections to ensure that there is a **good ground connection**, i.e. no "floating grounds"…. Grounding problems have brought down even the best at NASA.
2. Do not place your board on a metal (conductive surface).
3. Mind the current limits of your I/O pins, you don't want to release smoke demons in the lab!
    a. Max DC current thru a single Arduino I/O pin is 40 mA
    b. Max DC current thru Vcc and GND pin from Arduino boards is 200 mA

4. Reduce the chance of **electrostatic shock** to your circuits and microcontroller boards by handling them carefully. It is recommended to hold your boards at the edges. BUT if you really do want to electrostatically shock your board, try shuffling your feet over carpet or rubbing a balloon over your hair, **then zap your board**… This is an excellent way to ruin your microcontroller!
5. Just like your code, think through your physical wiring before starting. This will save you time and headache down the road.
6. **LEDs** must have **series resistors** to limit current draw thru microcontroller pins.
7. **Software configurable pull-up resistors** are available in the Arduino microcontroller (see Figure 8 below): 50k for Mega, 20k for Uno

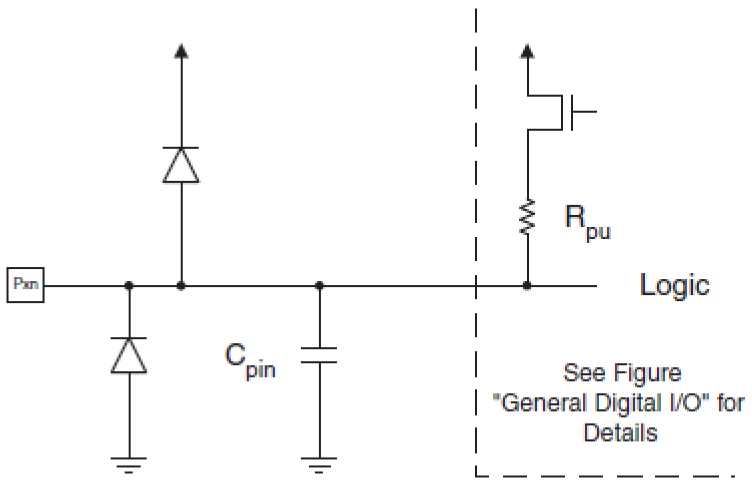

Figure 8 – ATMega GPIO showing Internal Pull-Up Resistor, Rpu = 50k (Source: Page 84 of ATmega reference manual)