LAB 3 – "FLESHING OUT THE STUBS" REPORT

Authors: Long Nguyen and Chase Arline
ECE/CSE 474, Embedded Systems
University of Washington – Dept. of Electrical and Computer Engineering

Date: 16th February 2020

TABLE OF CONTENTS

LIST OF FIGURES

1.0 ABSTRACT

The goal of this project is to develop the safety precautions of a simple BMS that would be used for monitoring a high voltage electric transportation system. This phase will further the touch screen interface, implement an analog reading system to simulate values, design a more dynamic scheduler, and trip certain emergency alarms. It will inform the user of measurements, alert the user with alarms, and take input from the user via the touchscreen. Interrupts will be used to establish the time-base of the system and for triggering an important safety mechanism (high voltage interlock fault). The system will be tested to ensure everything runs in a timely manner, measures values in specific ranges, and that all safety precautions are challenged. In conclusion, this project developed many safety precautions for a simple BMS and is able to alert the user if any of the alarms are tripped. The values used for tripping these alarms are measured through analog inputs and are converted to specific ranges relevant to their units. All code runs efficiently and within the required timeline, with a small exception that is addressed in the unresolved errors section of this report.

2.0 INTRODUCTION

This project will implement a task queue for our scheduler and build a system of alerting the user if any safety mechanisms are triggered. Values on the BMS will be simulated through analog measurements using potentiometers. An interrupt will be used to set the time base of the scheduler. Another interrupt will be used to detect when the high voltage interlock loop is broken. This will trigger a safety mechanism that will shut off the contactor. The display interface is further developed in order to keep up with the demands of the safety precautions.

3.0 DESIGN SPECIFICATION

The design of this portion of the system is defined by both the hardware and software architecture and requirements. The microcontroller will continue to interface with any I/O as it did in the previous revision, with modifications as follows. This includes the touchscreen, the simulated contactor, the HVIL, and new additions. Three potentiometers are now used for setting the voltage, current, and temperature values on the system. The HVIL loop is implemented in hardware the same as before but is read differently in software. The software now has an interrupt that reads when the HVIL has transitioned from closed to open status.

This interrupt routine allows the system to immediately shut off the contactor, as would be done in a real system if there is an issue with high voltage components. The scheduler now runs through a task queue and executes on a 100ms time-base. This time-base is set by a hardware timer interrupt inside of the microcontroller. Alarms are now triggered through specific states as defined by their state diagrams. The HVIL alarm blocks control of the contactor when active. The user will not be able to navigate away from the alarm screen while any alarm is not acknowledged. The user will acknowledge alarms via a button on the alarm screen. Certain sections of the code will be deemed critical, and precautions will be taken to ensure that interrupts do not interfere with these processes.

4.0 SOFTWARE IMPLEMENTATION

This project uses the Arduino Mega 2560 microcontroller to interface the TFT display and several GPIO pins. It currently uses a single digital input pin (pin 21), a single digital output pin (pin 24) and 3 analog input pins: pin A13, pin A14 and pin A15. These 2 digital pins will be used for the high voltage interlock, and the simulated contactor respectively. The 3 analog pins are used for temperature, HV voltage and HV current respectively. The high-level picture of this system is represented in **Figure 6** below.

The main architecture of the software revolves around the scheduler. This project transmits from the round-robin scheduler from the previous lab to a more dynamic scheduler. A task is stored in a Task Control Block which holds the function pointer and data pointer for the task. All the tasks are linked together by the TCB pointer previous and next to create a doubly linked list. The Scheduler will loop through all the tasks in the linked list, sequentially executing each of one by calling the function via function pointer with its task data pointer, as shown in **Figure 10**. All the tasks in the linked list are executed sequentially at a rate of 10Hz, which creates a time based system. In order to create that time based system, there will be a hardware timer interrupt service which sets the flag for the main loop to to cycle again every 100ms. After executing each task, return to the main loop and set the timerFlag to be false. The implementation details of the Scheduler task, timerISR task and main loop will be represented in the Appendix A.

Another major functional block of this system is measurement task block. The measurement task keeps track and updates the following data of the system: temperature, high voltage current, high voltage voltage and the high voltage interlock loop. The high voltage interlock loop value is implemented in hardware via input digital port 22 in **Figure 6**. While the temperature, HV current and HV voltage values are implemented via analog input port A13, A15 and A14 respectively. These values are simulated by using the potentiometers. The potentiometers give the input value range from 0V-5V. This will be sent into the 10-bit DAC then the measurement task will convert the value to the correct unit range. The rest of the implementation for the measurement task is the same as in the previous lab.

State of charge (SOC) task will take care of keeping track of and updating the state of charge of the high voltage battery. State of charge will only update the SOC value and has only a sub function, called updateSoc() which takes in a float pointer, to update that value of

state of charge (**Figure 8**). In this project phase, the state of charge is set to be a constant value of 0.

The contactor task sets the value of the digit output pin which represents the simulated contactor (see pin 24 in **Figure 6**). The digital output depends on user input from the battery's "ON" and "OFF" button of the touch screen, updated by inputTask (which will be talked about later). This is clearly shown in **Figure 9**. The contactor task will have two main states, which is either the battery connection is closed or open. The contactorTask's states also depend on the HVIL status which is set in the hvilISR(). When the HVIL status is closed, and if the battery connection is closed, then the LED in the simulated contactor is lit up, and if the battery connection is open, then the LED in the simulated contactor is turned off. However, when the HVIL status is open the LED always stays off and users cannot turn it on or off (see in **Figure 23**). The whole pseudo code for this task is shown under the Appendix D.

The alarm task keeps track of the states of the three different alarms of the BMS system: HVIL alarm, overcurrent alarm and high voltage out of range alarm. The states of the three alarms are implemented based on the input value from the hardware for HV Current, HV Voltage and HVIL. These are shared data from the measurement task. The HVIL alarm updates based on states of the HVIL (see in **Figure 20**). The over-current alarm updates its states based on the HV current value read from the measurement task (see in **Figure 21**). The high voltage out of range updates its states based on the HV Voltage value read from the measurement task (see in **Figure 22**). The alarm task structure splits into three functions that each updates its respective alarm state (see in **Figure 7**). In order to update the states of each alarm correctly, each alarm will be implemented as a struct encapsulating all the information about states, measurementValue and acknowledge state (see in **Figure 8**). Each element in the alarm struct is pointer to the shared variable which will be also used in different tasks such as display tasks. The Scheduler passes the array of 3 alarm structs down to the alarmTask. The values for the state of three alarms are updated via float pointers which points to the index state for each alarm. These are volatile to avoid interrupt issues with the contactor. This state is used in the display task to get input from the alarm state array to determine the correct text to display on the screen (see in **Figure 9**). The measurementValue element in the struct is a pointer point to a shared variable, which is updated from measurementTask. It is used to update the state of an alarm to "Not Active" or "Active, Not Acknowledge." The acknowledge state is also a pointer to a shared variable, which is updated in inputTask of touchScreenTask, to update the alarm's state from "Active, Not Acknowledge" to "Active, Acknowledge". The whole pseudo code alarm task can be viewed under Appendix E.

The touch screen task structure is divided into two smaller subtasks or "pseudo tasks" which are input task and display task (shown as in **Figure 7**). They do not use a TCB but are just organizers for functionality. The reason why the touch screen task is split into two subtasks is because our BMS both creates a user interface to let the users give the input and display the result or data to the touch screen. In order to be able to display data and get input correctly, when the Scheduler invokes that touch screen, it will pass a pointer to the shared variable currentScreen to determine what screen is currently displayed, pointer to shared variable clockCount value, a pointer to the share variable changeScreen to determine

whether the user want to change screen or not, a pointer to share variable ackDrawn to keep track if the acknowledge button is drawn or not, a list of Screen struct and a list of 3 alarms to display to screen for emergency case correctly (see in **Figure 8**).The implementation of the Screen is detailed in the previous revision of this project. The inputTask is broken down into the sub functions such as getTouchInput() and isButton() to determine which button is clicked by the users (show in **Figure 7**). Moreover, the inputTask also updates the shared variable, acknowledge state, of each alarm when the acknowledge button is clicked. In the inputTask, when there is an acknowledge button and the button is clicked, then the inputTask will go through and update all alarms' acknowledge state. It only updates the acknowledgement state of alarms that are active, unacknowledged.

The display task is broken down into the sub functions which are: drawScreen(), drawLabel() and drawData() (shown in **Figure 7**). The drawScreen function's implementation is detailed in the previous revision of this project. There is a change in this revision whenever we are changing screens. Instead of clearing the entire screen, we precisely delete every item on the previous screen and then draw items on the new screen. This saves a lot of time each time the screen is changed. The displayTask will also use the passed down array of alarms to determine whether there is an emergency state (one of the alarms is in state "Active, Not Acknowledge") to display the correct screen, display the acknowledge button, and also to block the users from leaving the alarm screen by disabling the "next" and "previous" button. The passed down pointer to the shared variable, ackDraw, is used to draw the acknowledge button on the screen, and not to redraw if it is already drawn. The whole pseudo code for touch screen tasks can be viewed under Appendix F.

For the external interrupt from the switch for the HVIL status, the attach interrupt is initiated in the setupTask with the mode of "RISING". When the switch is go from closed to open, the interrupt will invoke the hvilIRS() to immediately turn of the contactor LED by doing digitalWrite to port 24, then it will set the shared variable, batteryOnOff to 0, and also set the HVIL alarm's status to "ACTIVE, NOT ACKNOWLEDGED".

For the user interface design and uses of each screen, there are three screens for this BMS which are measurement screen, alarm screen and battery on/off screen. The measure measurement screen will have the title at the top of the screen, then the datas (which are soc value, temperature, HV current, HV voltage and HVIL), the next and previous buttons at the bottom of the screen (shown in the **Figure 13**). In order to draw the measurement screen, the display task calls the drawScreen() function which deletes the previous screen and draws all relevant data that is on the measurement screen (see **Figure 12**). In the measurement screen, the user can view data, or navigate to the next screen or navigate to the previous screen. This is specified in the use case diagram in **Figure 11**.

The alarm screen also has a similar design as the measurement screen with title at the top of the screen, then the data about alarms' states displayed, the next and previous buttons at the bottom of the screen (shown in the **Figure 16**). In order to draw the measurement screen, the display task calls the drawScreen() function which deletes the previous screen and draws all relevant data that is on the measurement screen (see **Figure 15**). In the alarm, the user can view data, or navigate to the next screen or navigate to the previous screen. In the case of an emergency (one of three alarms is "active, not acknowledged " state), there will be a button on the screen for the user to acknowledge the alarm. However, in this case, the user cannot

click the "next" and "previous" button until the alarm is acknowledged. This is specified in the use case diagram in **Figure 14**.

The battery on/off screen has a different layout than other screens. Battery on/off screen has two buttons: "ON" and "OFF" buttons that are at the top of TFT the screen, displaying the battery connection state, the next and previous button at the bottom of the page (see **Figure 19**). In order to draw the battery screen, the display task calls the drawScreen() function which deletes the previous screen and draws all relevant data that is on the battery screen (see **Figure 18**). In the battery on/off screen, the user can view data, or navigate to the next screen or navigate to the previous screen or turn on/off battery. This is specified in the use case diagram in **Figure 17**. The battery toggle button on this screen allows the user to click to turn on or off, which will update the contactor status through the shared variable, batteryOnOff(see in **Figure 23**).

The "next" and "previous" button is working and implemented as the previous part. However, there is an exception in changing screens. Whenever, there is an active alarm, the screen will automatically go to the alarm screen and block the user from leaving alarm screen until they hit the acknowledge button (see in **Figure 24**). In this case, the "next" and "previous" button are disabled, and users cannot use them. In order to accomplish this, the touchInputTask checks the emergency status by going through the list of alarms passed down from the touchScreenTask. If any alarms are unacknowledged, the "next" and "previous" wouldn't update the currentScreenPtr value even though users click on it. This creates a lock that forces the user to acknowledge any unacknowledged alarms before leaving the alarm screen.

5.0 TEST PLAN

The test plan includes three parts:

1. Requirements
2. Test coverage
3. Test cases

5.1 Requirements

The task queue scheduler will run off of a one hundred millisecond time-base.
The HVIL alarm will immediately turn the contactor off.
The HVIL alarm will restrict the user's ability to turn on the contactor.

Any unacknowledged alarm will restrict the user's ability to navigate away from the alarm screen.
Any alarm being triggered will switch the screen to the alarm screen.
Alarms will follow the behavior defined in their respective state diagram.
Measurements from potentiometers will be scaled to their respective units. This is as follows:

Temperature is -10 to 45 Celsius
Current is -25 to 25 Amps
Voltage is 0 to 450 Volts

5.2 Test Coverage

The timer interrupt will trigger every one hundred milliseconds, which will set a flag for the task queue scheduler in the main loop. This ensures a time base of one hundred milliseconds.
The HVIL alarm interrupt will directly write to the output pin of the contactor to turn it off.
While the HVIL alarm is active, the user will not be able to turn on the contactor. They will be able to after the alarm transitions to "NOT ACTIVE".
If any alarm is unacknowledged, the user will not be able to use the previous or next buttons to navigate away from the alarm screen. This is done by simply checking if any alarms are unacknowledged.
If any alarm is triggered while on another screen, the screen will change to the alarm screen where the user is given the option to acknowledge it via a button that only appears if any alarm is unacknowledged.
Each alarm will transition between states as defined inside of their respective state diagram.
The measurement from each potentiometer is scaled to relevant units. They are each initially read as an analog input from the 10-bit DAC. This converts 0-5V into 1024 states and each measurement is scaled as follows:

Temperature: -10 to 45 Celsius, scaled from 0-1023
Voltage: 0 to 450 V, scaled from 0-1023
Current: -25 to 25 Amps, scaled from 0-1023

5.3 Test Cases

The time between each hardware timer interrupt is measured via the serial console. It should be 100ms. This will be seen on the serial monitor.
The HVIL input and contactor output are both tracked visually from both the display and their simulated LED input/outputs. When the contactor is on, and HVIL is turned off, the contactor should immediately turn off. This is measured visually via the LEDs.
When the HVIL alarm is active, the user pressing the buttons on the battery screen will not turn the contactor on. This is measured visually by the user.

When any alarm is active, the user will not be able to navigate off of the alarm screen. If the user presses the previous / next buttons, they will not change screen. This is seen visually by the user.
If any alarm is triggered while on another screen, the display will transition to the alarm screen. This is seen visually when an alarm is triggered.
While any alarm is unacknowledged, a button is displayed on the alarm screen which the user can press to acknowledge any unacknowledged alarms. This can be visually seen on the display.
The measured values on the screen should range accordingly as their potentiometer is turned:

Temperature is -10 to 45 Celsius
Current is -25 to 25 Amps
Voltage is 0 to 450 Volts

6.0 Presentation, Discussion, and Analysis of the Results

Our design accomplishes all test requirements according to the test cases. The program executes as stated by the project objectives with no errors. When an alarm is triggered, the user is immediately taken to the screen and they are not able to leave it until there are not any unacknowledged alarms. The time-base is measured to be 100ms +/- 1ms. When the contactor is closed and the HVIL loop is opened, the contactor immediately opens. While the HVIL is open, the contactor is not able to be closed. The user is able to close the contactor whenever the HVIL is closed. Only when any alarm is unacknowledged does a button appear on the alarm screen allowing the user to acknowledge all unacknowledged alarms. Our measurement values were correctly scaled according to 0-5V, but it was hard to represent this due to the voltage at the microcontroller never fully reaching either 0V or 5V. All alarms transitioned between their states as defined in the state diagrams.

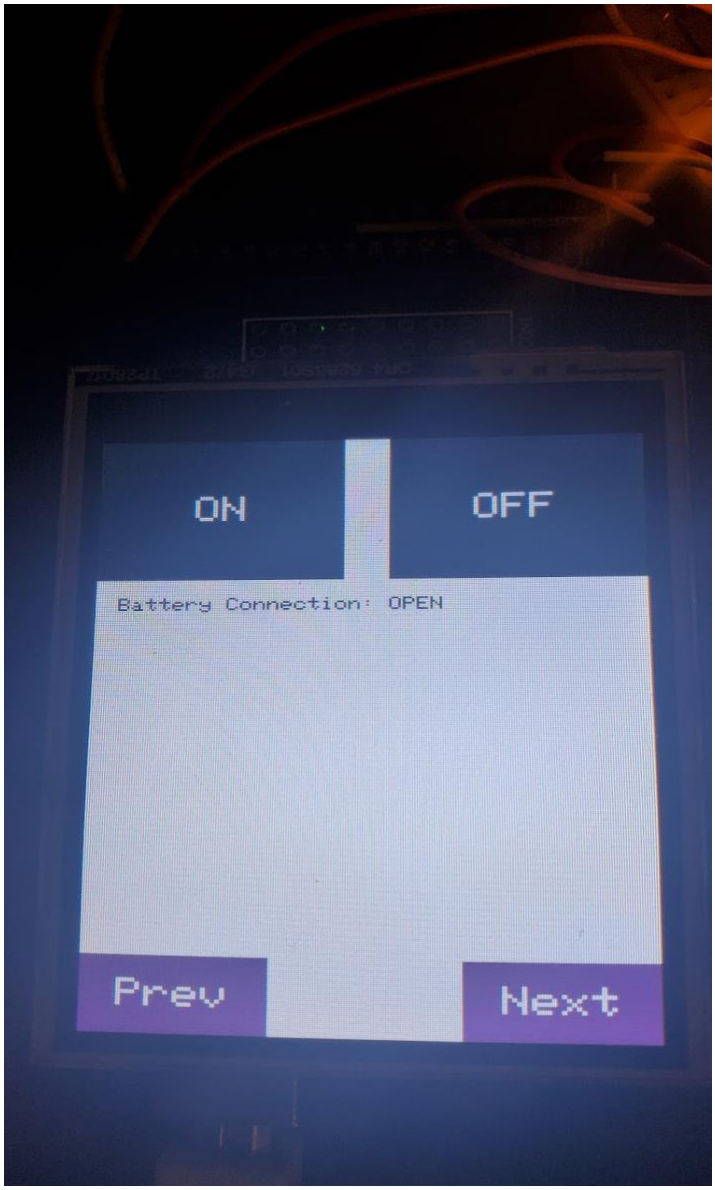


Figure 1. Battery Toggle Screen

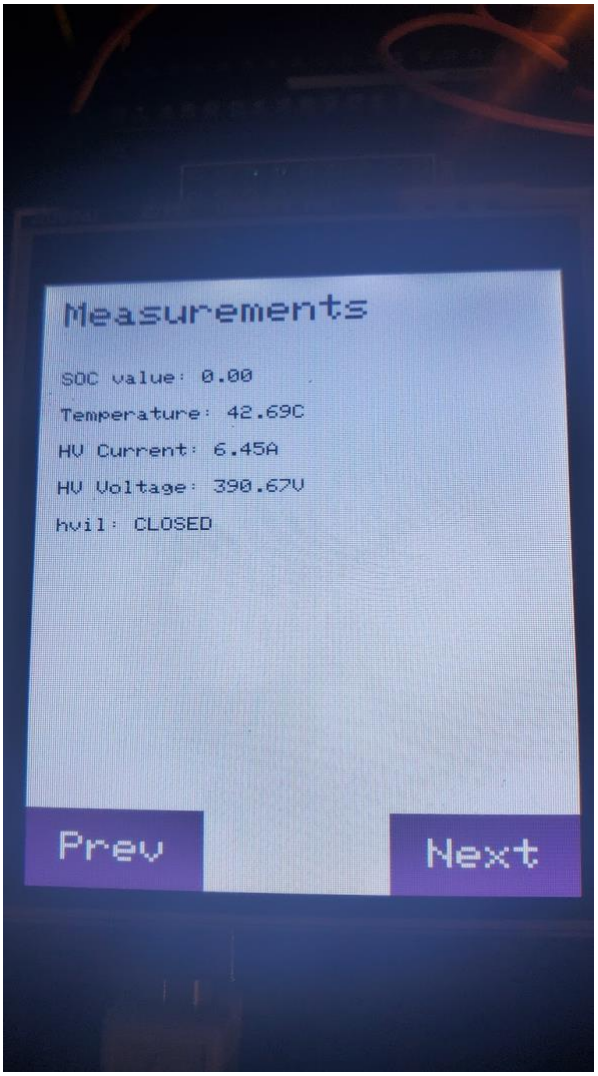


Figure 2. Measurements Screen

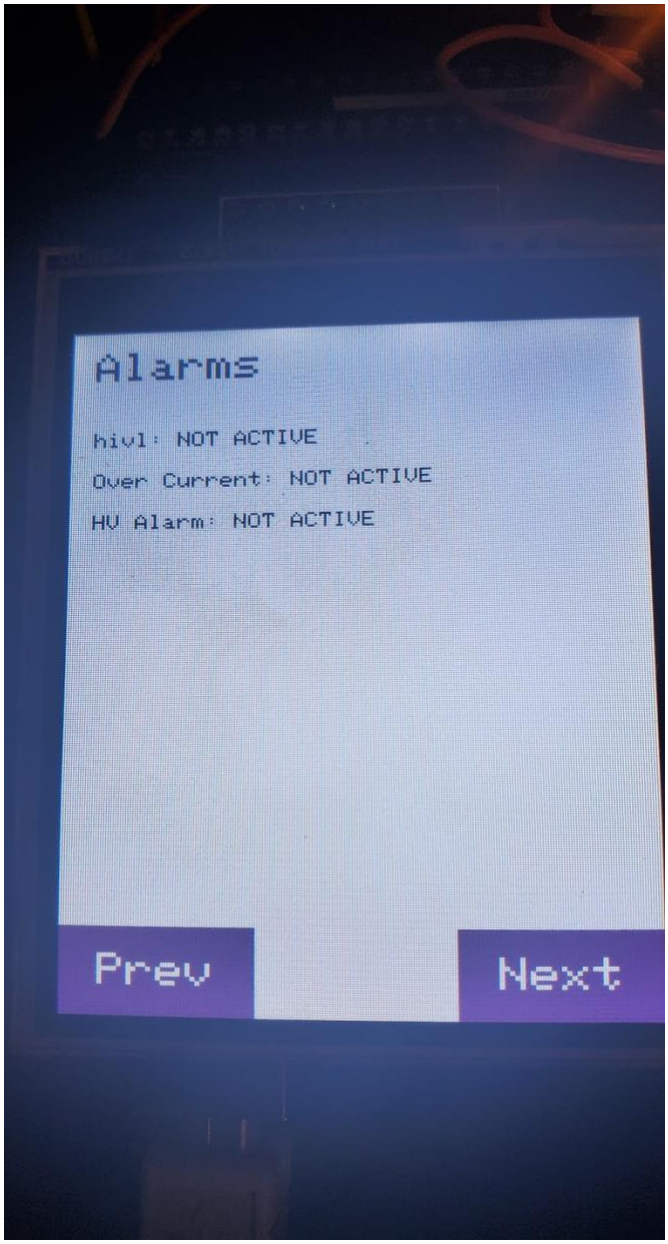


Figure 3. Alarm screen when all alarms are not active

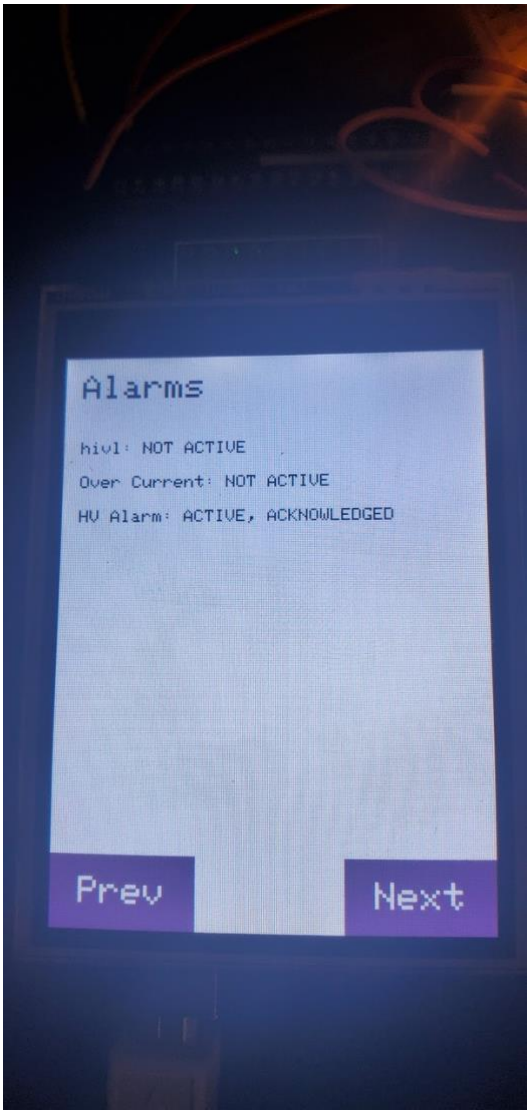


Figure 4. Alarm Screen when there are some alarms in "Active, Acknowledged" and "Not, Active"

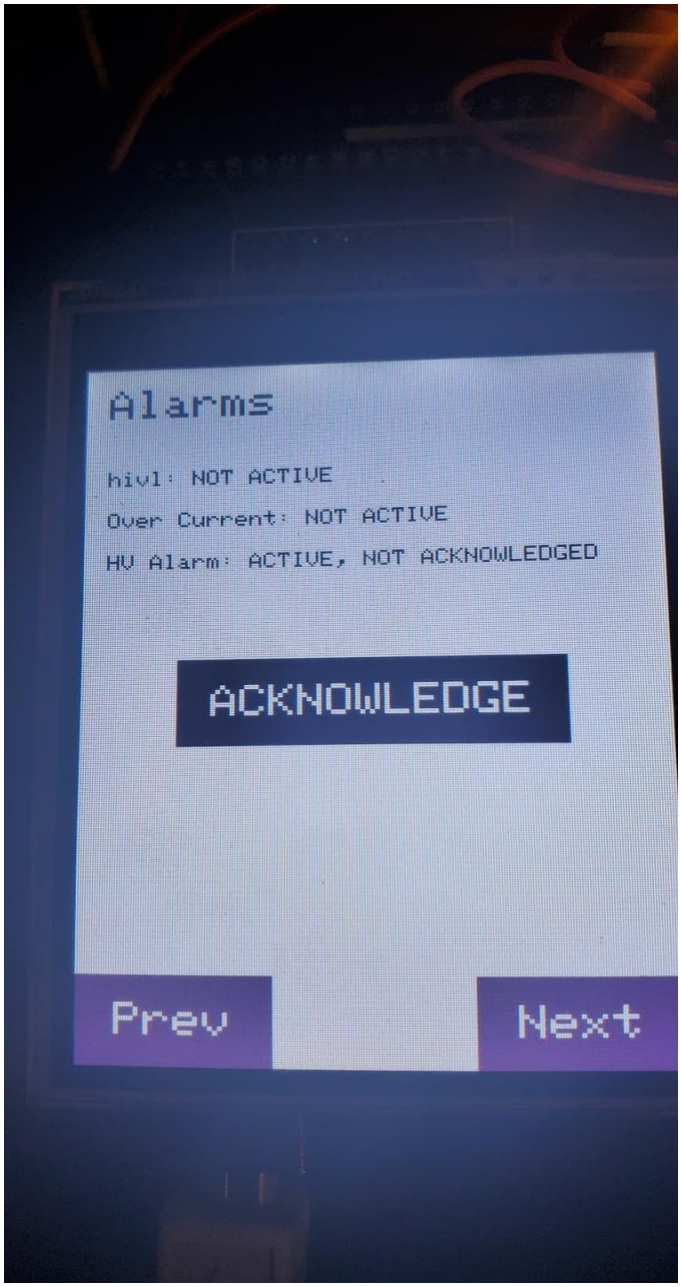


Figure 5. Alarm screen when there are some alarms are in "Active, Not Acknowledged" state.

6.1 Analysis of Any Resolved Errors

We had two errors during development that were resolved.
The first was having to be able to draw the screen faster to better fit the 100ms time-base. This was done by changing how we drew things on the display as well as how we switched screens. To draw fasters, we ditched the display libraries and made our own function for drawing over changed data. This was faster than the libraries given, and we were able to get all screens working under 100ms. We were also able to speed up the rate at which we could switch screens by deleting specific pieces of the screen instead of clearing the entire screen. To do this we simply had to keep track of slightly more data and also draw boxes over those areas whenever we switched screens.
The other error was that we had to make specific variables volatile (contactor status and hvil alarm status). This caused us an error because it meant we had to change our struct definitions.
At first, we believed this to be a major error that would refactor our code greatly. Instead, changing the struct definition to volatile allowed us to use both volatile and non-volatile variables to initialize the member in the struct. This essentially caused us no large issues and became trivial once we figured this out, but it was the best way to solve this type of problem we had.
This change to volatile variables actually allowed us to remove any "critical" sections of our code. This means we never have to turn off our interrupts for certain sections of the code because each variable that would need it is volatile, and it is being checked against another volatile variable that is used in the interrupt. Effectively the value is directly referenced (not locally cached) so there is no need to disable the interrupts in any area when we are setting the hvil alarm status, or the contactor status/writing the contactor output pin.

6.2 Analysis of Any Unresolved Errors

There are not any unresolved errors according to the lab spec, but we did have one which is that the rate at which we can change screens is limited. Simply drawing buttons / text on the screen takes too much time to be able to run in a 100ms period. So our normal screens can easily run in this amount of time, but switching from screen to screen takes anywhere from 200-300ms instead of under 100ms.

7.0 QUESTIONS

1. The resolution of each analog input is as follows:
   The voltage is 0-450V with 1024 states → 450/1024 → 0.44V per step
   The current is -25 to 25A with 1024 states → 50/1024 → 0.049A per step

The temperature is -10 to 45C with 1024 states → 55/1024 → 0.054C per step

2. Inputs:
    a. Touchscreen: Digital
    b. HVIL: Digital
    c. Temperature: Analog
    d. HV Voltage: Analog
    e. HV Current: Analog
3. Outputs:
    a. Touchscreen: Digital
    b. Contactor: Digital

8.0 CONCLUSION

In conclusion our program works very well according to the standards of the project. All interrupts execute correctly, each potentiometer gives the correct output scaled, and all inter-task communication is effective. The program itself is very dynamic which allows it to be easily modified. All of the test cases were successful. The issue we had in this project was not being able to lower the delay between changing screens even when optimizing it as much as we could. Also figuring out where to place atomic code was difficult considering our variables being changed were volatile so theoretically it did not make a difference if the interrupt was called at that specific moment.

9.0 CONTRIBUTIONS

We both worked equally on this project. Almost all of the time spent working on this project we were in a zoom call, so we were both providing the same amount of input.

10.0 APPENDICES

10.1 Pseudocode

A. The following is the pseudo code for Scheduler Task:

```
Scheduler:
    for each task in linkedlist:
        executes task

timerISR:
    timerFlag = true
```

```
loop:
    if (timerFlag):
        Scheduler()
        timerFlage = false
```

B. The following is the pseudo code for Measurement Task:

```
measurementTask(dataPtr):
   parse the dataPtr
   updateTemperature (temperature value reference, temperature pin)
   updateHVIL(hvil value reference, hvil pin)
   updateHvCurrent(hv current reference, hv current pin)
   updateHvVoltage(hv voltage reference, hv voltage pin)

updateHVIL(hvilPtr, pinNo):
    update hvil value by reading logic level on the hvil pin

updateHvCurrent(hv current reference, hv current pin):
    update Hv current by reading analog value from hvCurrent pin and scale by using
    resolution

updateHvVoltage(hv voltage reference, hv voltage pin):
    update Hv Voltage by reading analog value from hvVoltage pin and scale by using
    resolution

updateTemperature (temperature value reference, temperature pin):
    update temperature by reading analog value from hvVoltage pin and scale by using
    resolution
```

C. The following is the pseudo code for Soc Task:

```
socTask(dataPtr):
    parse the dataPtr
    updateStateOfCharge(soc reference)

updateStateOfCharge(socValPtr):
    update state of charge value every cycle
```

D. The following is the pseudo code for Contactor task:

```
comtactorTask(dataPtr):
    parse the dataPtr
    updateContactor(contactor status reference, contactor pin)

updateContactor(contactorStatusPtr, contactorPin):
    write contactorStatus to contactor pin.
```

E. The following is the pseudo code for Alarm task:

```
alarm_arr[] = {state1, state2, state3}

alarmTask(data):
    parse the dataPtr
    updateCurrentAlarm(over-current alarm reference)
    updateHVORAlarm(hvor alarm reference)
    updateHVIAAlarm(hvia alarm reference)

updateHVIAAlarm(hvia alarm reference):
    if alarm.HIVL.status is open and not acknowledge:
        alarm.status ← “Active, Not Acknowledged”
    else if alarm.HIVL.status is open and acknowledge:
        alarm.status ← “Active, Acknowledged”
    else if alarm.HIVL.status is closed:
        alarm.status ← “Not Active”
        alarm.acknowledge ← “Not acknowledged”

updateHVORAlarm(hvor alarm reference):
    if alarm.HVVoltage.status is out of safe range and not acknowledge:
        alarm.status ← “Active, Not Acknowledged”
    else if alarm.HVVoltage.status is out of safe range is open and acknowledge:
        alarm.status ← “Active, Acknowledged”
    else if alarm.HVVoltage.status is in safe range:
        alarm.status ← “Not Active”
        alarm.acknowledge ← “Not acknowledged”

updateCurrentAlarm(over-current alarm reference):
    if alarm.HVCurrent.status is out of safe range and not acknowledge:
        alarm.status ← “Active, Not Acknowledged”
    else if alarm.HVCurrent.status is out of safe range is open and acknowledge:
        alarm.status ← “Active, Acknowledged”
    else if alarm.HVCurrent.status is in safe range:
        alarm.status ← “Not Active”
        alarm.acknowledge ← “Not acknowledged”
```

F. The following is the pseudo code for TouchScreen task:

```
printDataToString(floatValue, PRINT_TYPE)
    check PRINT_TYPE against each case (number, boolean, alarm, label) in enum:
        convert floatValue to its String equivalent based on
        the PRINT_TYPE value
    return the string equivalent

touchScreenTask(dataPtr):
```

```
        parse dataPtr
        if the first time (use clock count):
           drawDisplay()
           draw “next” button
           draw “previous” button
        needToChangeScreen (bool) ← execute input task
        execute display task

inputTask(currenScreenPtr, screens, alarmsList):
        getTouchInput()
        needToDrawScreen  ← false
        emergency ← get from extract value state of each alarm
        if (touch on next or previous button) and not emergency:
              needToDrawScreen← true
        if on battery screen and touch on “ON” button and HVIL alarms not active:
              update value for batteryOnOff to be on
        else if on battery screen and touch on “OFF” button:
              update value for batteryOnOff to be off

        if on alarm screen and acknowledge button is click:
              go through list of alarms and update all alarms in “Acitve, Not Acknowledge”
              to “Active, Acknowledge”.
        return needToDrawScreen

displayTask(currentScreenPtr, screensList, switchScreen, alarmsList, lastScreenPtr):
         drawScreen(screensList, switchScreen, alarmsList, lastScreenPtr,
         currentScreenPtr)
         for each data in screen.dataList:
               drawData(data, switchScreen)

drawScreen(screensList, switchScreen, alarmsList, lastScreenPtr, currentScreenPtr):
         if switchScreen:
               deleteAll all labels, data and button from the screensList[lastScreenPtr]
               if on batteryScreen:
                     draw “battery toggle” button
               for each label on the screen:
                      draw label
           else if on alarm screen and emergencyCheck(alarmsList):
               draw “Acknowledge” button
           else if on alarm screen and not emergencyCheck(alarmsList):
               delete “Acknowledge” button

emergencyCheck(alarmsList):
         go through each alarm:
               if there is an active not acknowledge alarm:
                     return true
         return false

drawData(data, switchScreen):
        if data is updated new or switch to new screen:
```

print data to the screen.

10.2 Code File Names

StarterFile.ino: file that the program starts in. This file includes the startUpTask and two ISR() function.
StarterFile.h : header file for StarterFile.ino
Alarm.c : code for the alarm task
Alarm.h : header file for alarm.c
Contactor.c : code for the contactor task
Contactor.h : header file for Contactor.c
Measurement.c : code file for the measurement task
Measurement.h : header file for Measurement.c
Soc.c : code file for the state of charge task
Soc.h header file for Soc.c
Scheduler.c: code for the Scheduler task
Scheduler.h: the header file for Scheduler.c
TaskControlBlock.h : header file defining TaskControlBlock struct
TouchScreenTask.ino : code file for the touch screen task
TouchScreenTask.h : header file for TouchScreenTask.ino

10.3 Figures

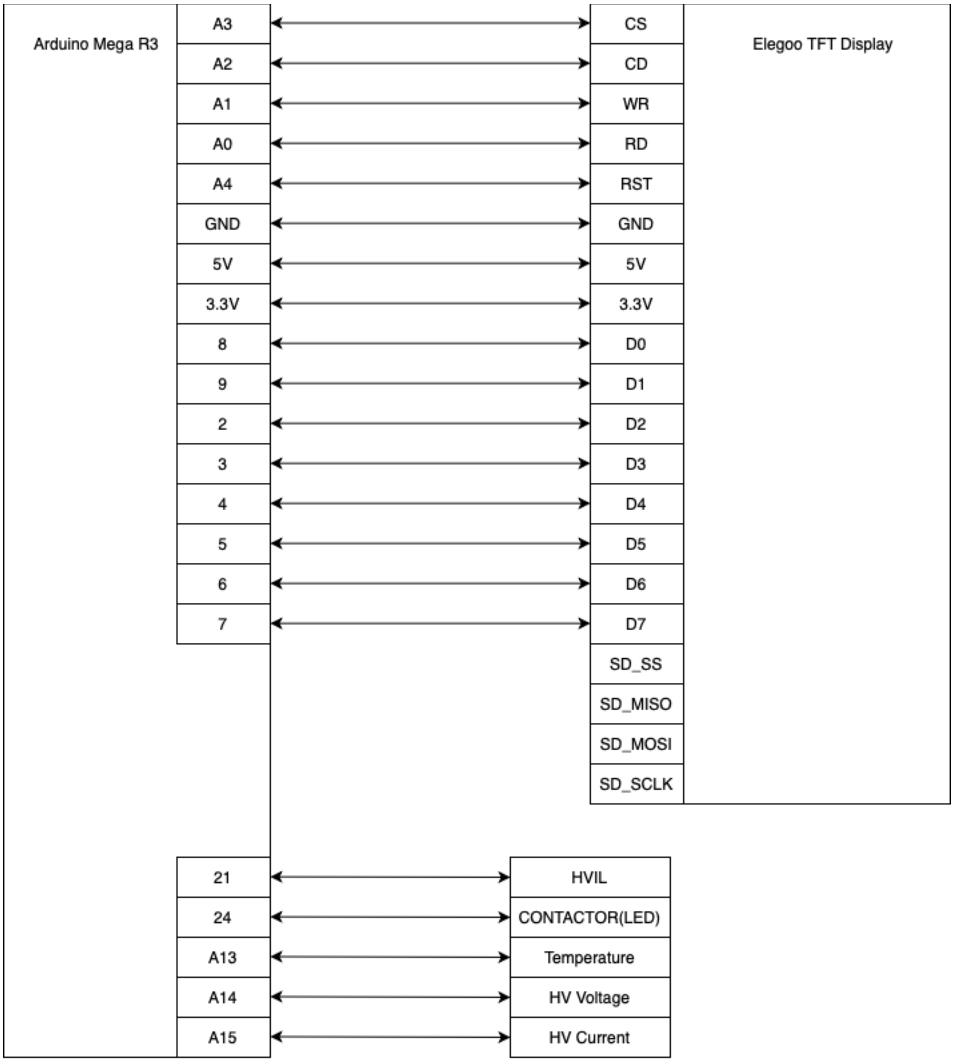


Figure 6. System Block Diagram - showing the Atmega input and output ports (and port numbers) labeled per I/O component

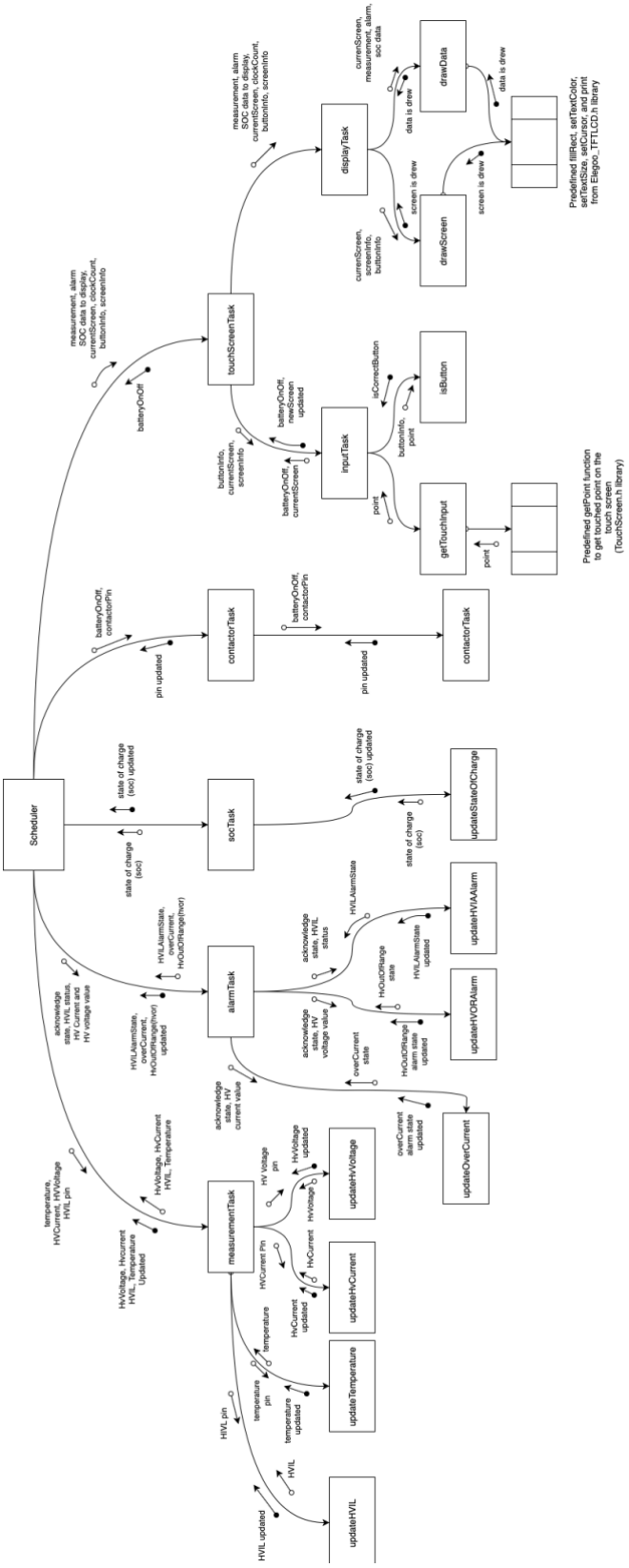


Figure 7. Structure Diagram - showing functional decomposition of tasks within the System Controller

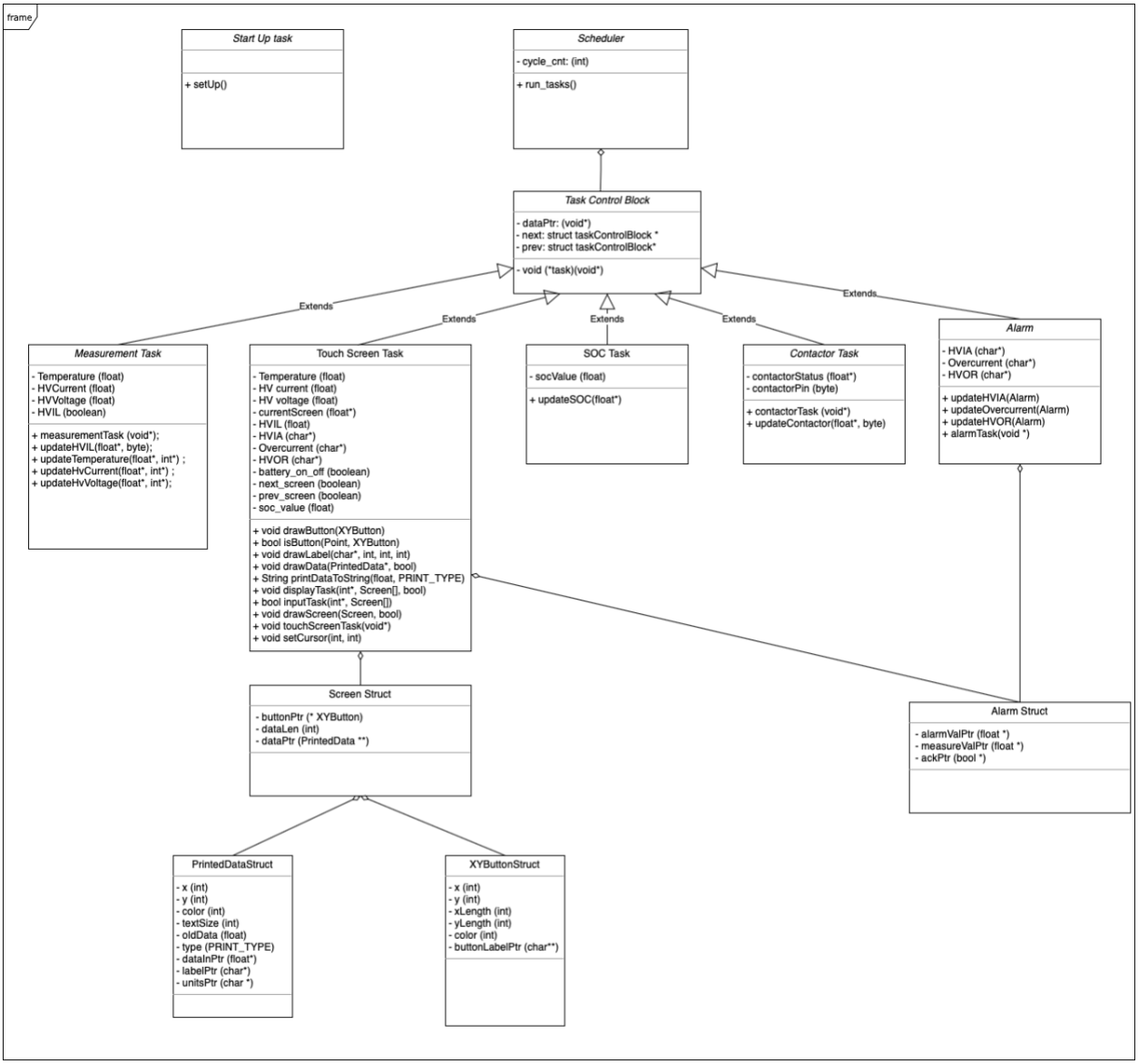


Figure 8. Class diagram - showing the structure of the tasks within the System Controller as reflected in the Structure Diagram.

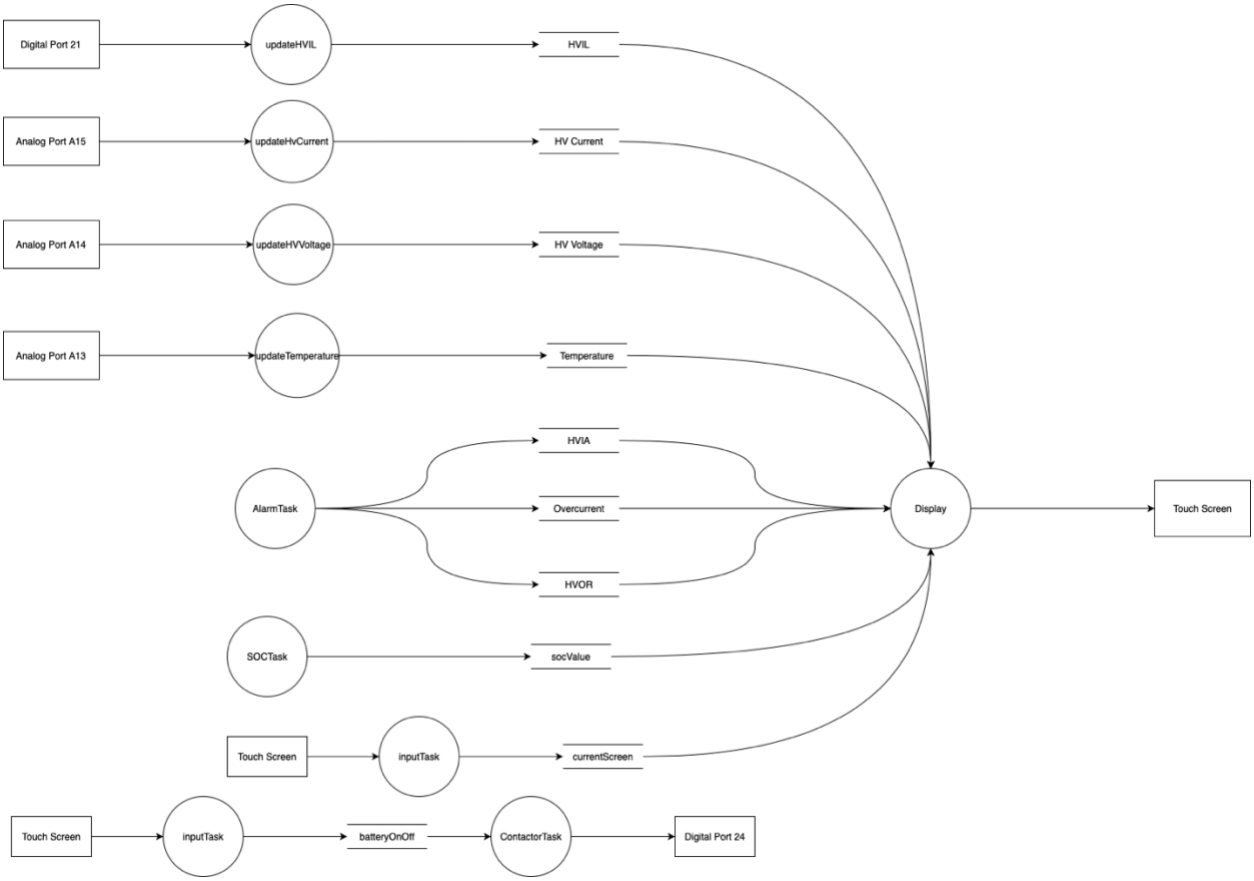


Figure 9. Data flow diagrams - shows data flow for inputs/outputs

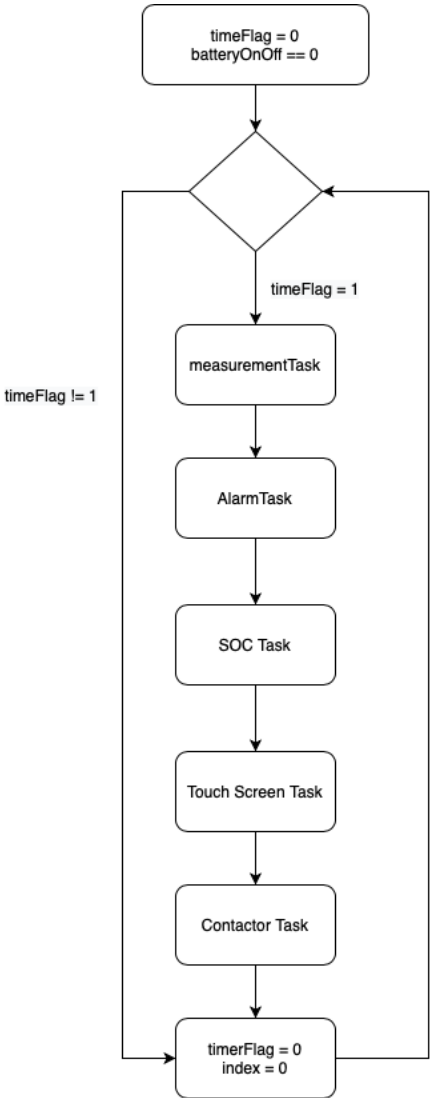


Figure 10. Activity Diagram - shows the System Controller's dynamic behavior from the initial entry in the loop() function

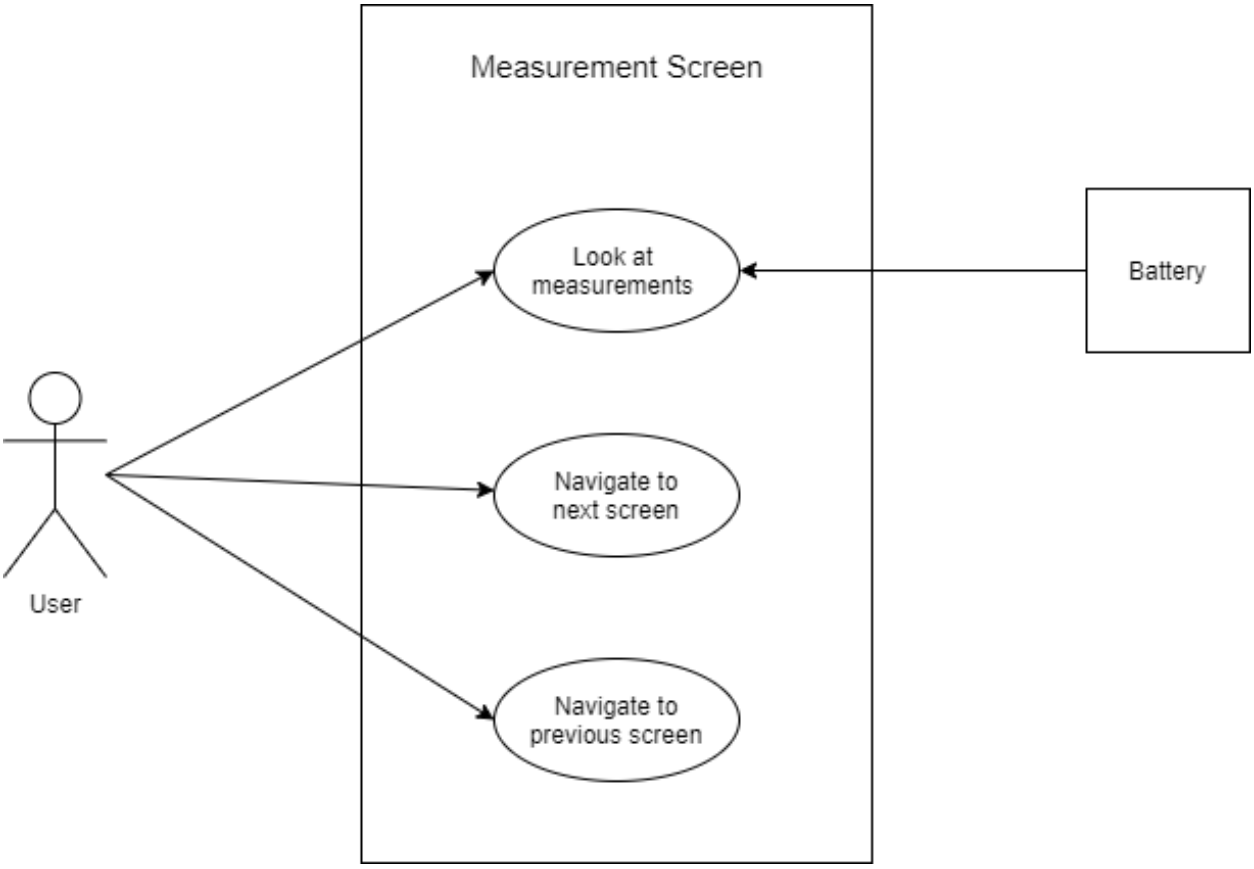


Figure 11. Use Case Diagram for Measurement Screen

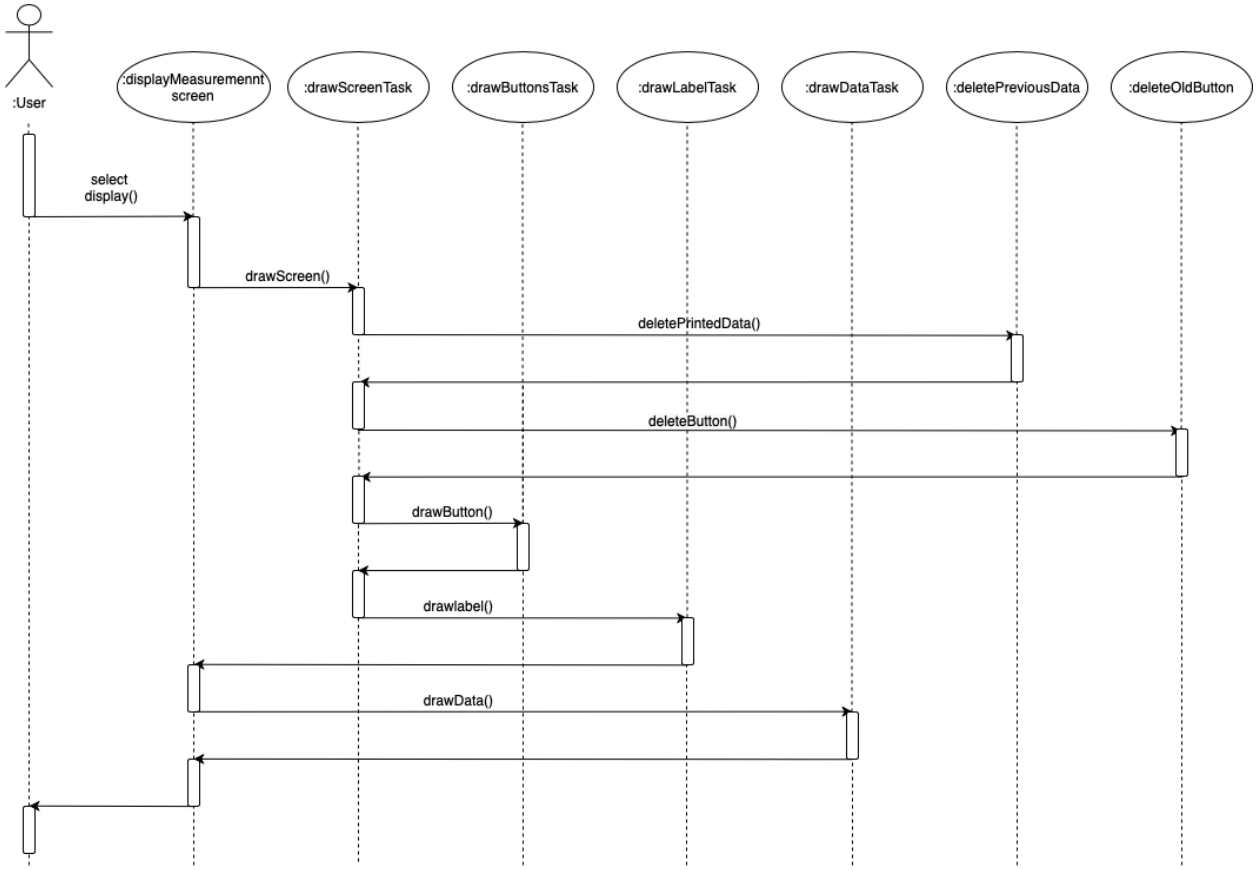


Figure 12. Sequence Diagram for Measurement Screen

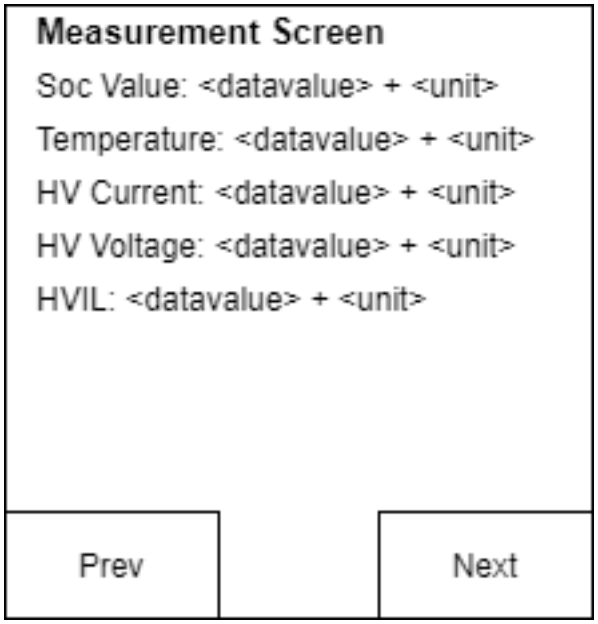


Figure 13. Front Panel Design for Measurement Screen

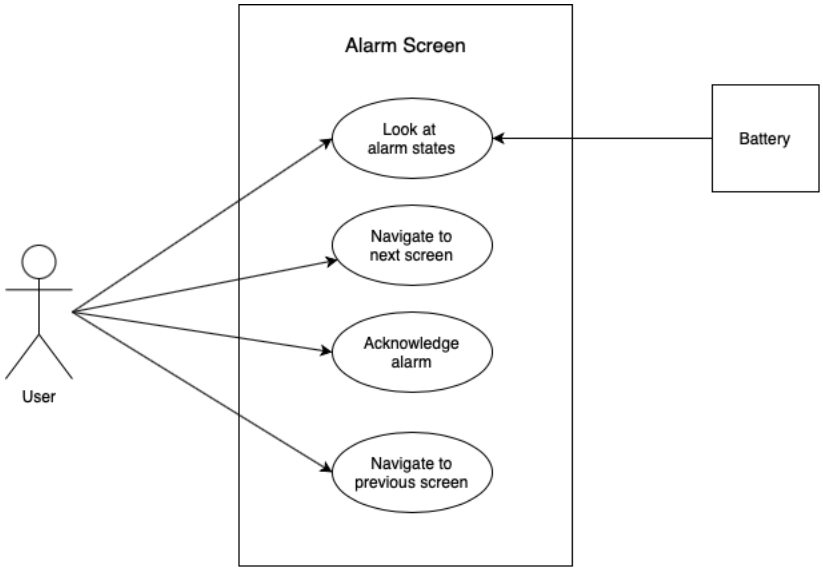


Figure 14. Use Case Diagram for Alarm Screen

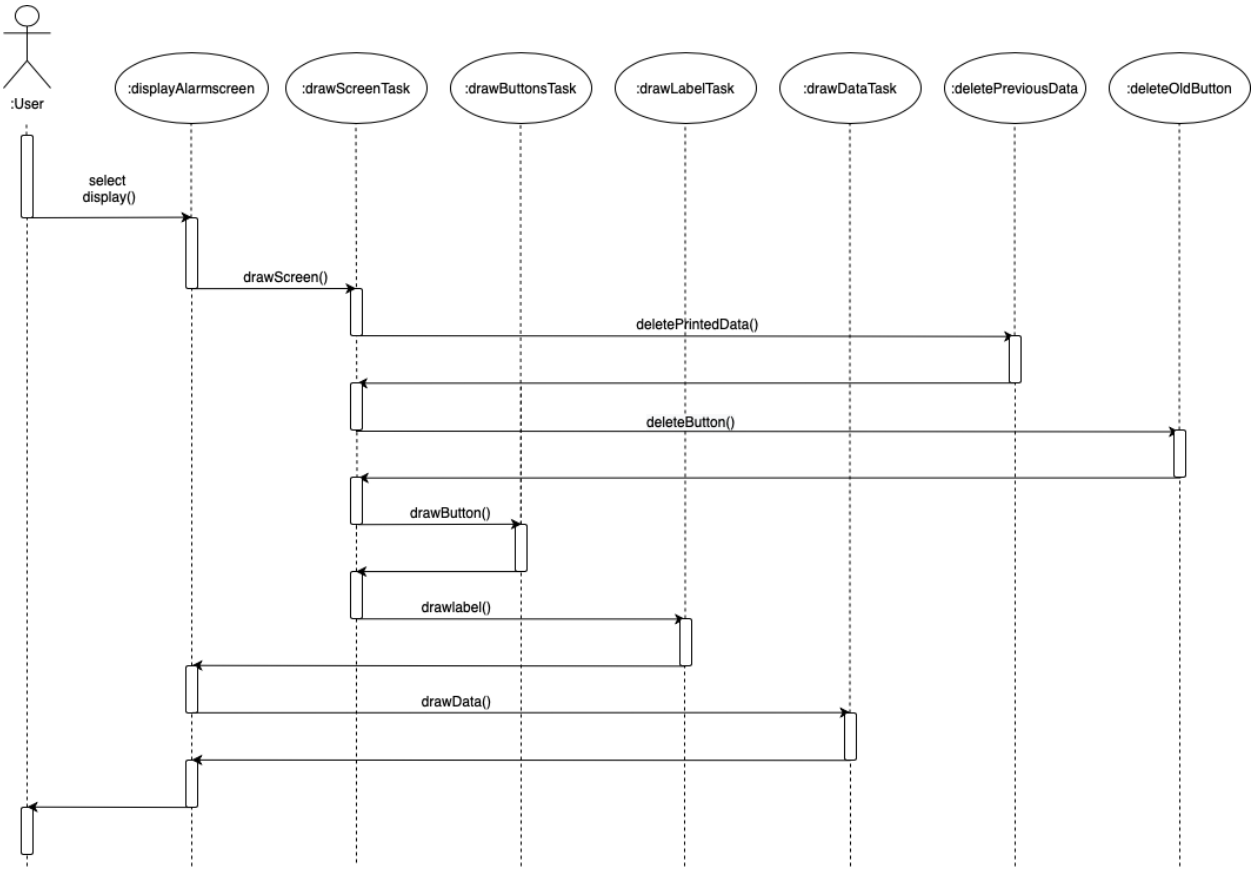
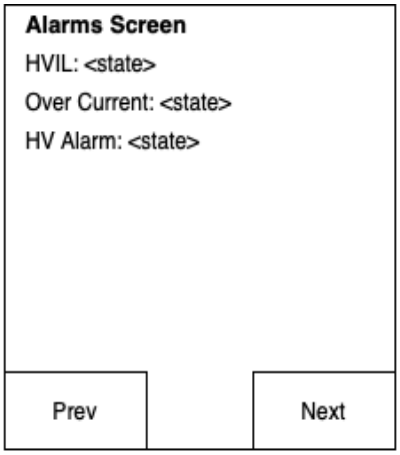
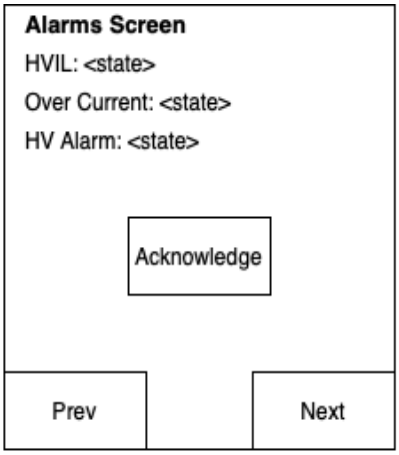


Figure 15. Sequence Diagram for Alarm Screen



14.a: Alarm Screen when no alarm active



14.b: Alarm Screen when there is an active alarm

Figure 16. Front Panel Design for Alarm Screen

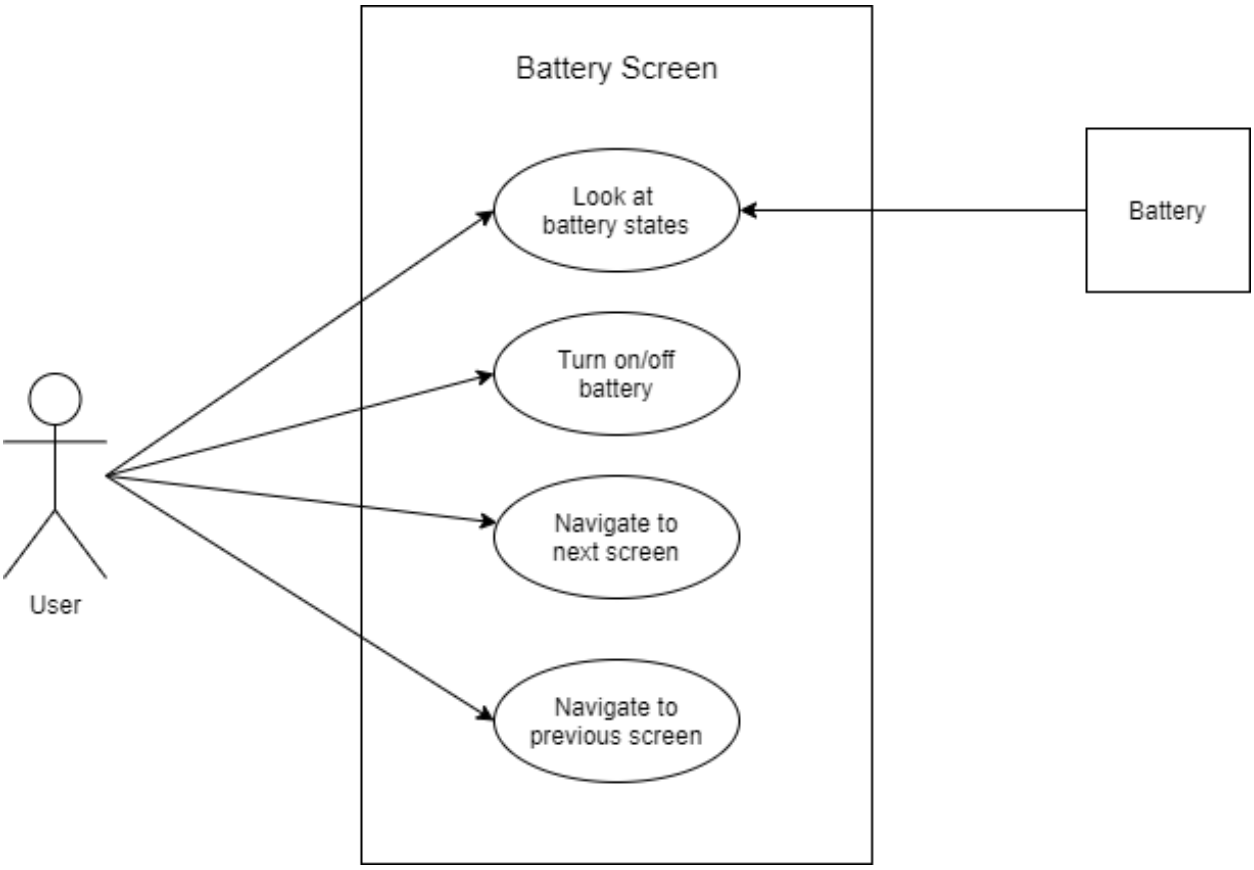


Figure 17. Use Case Diagram for Battery Screen

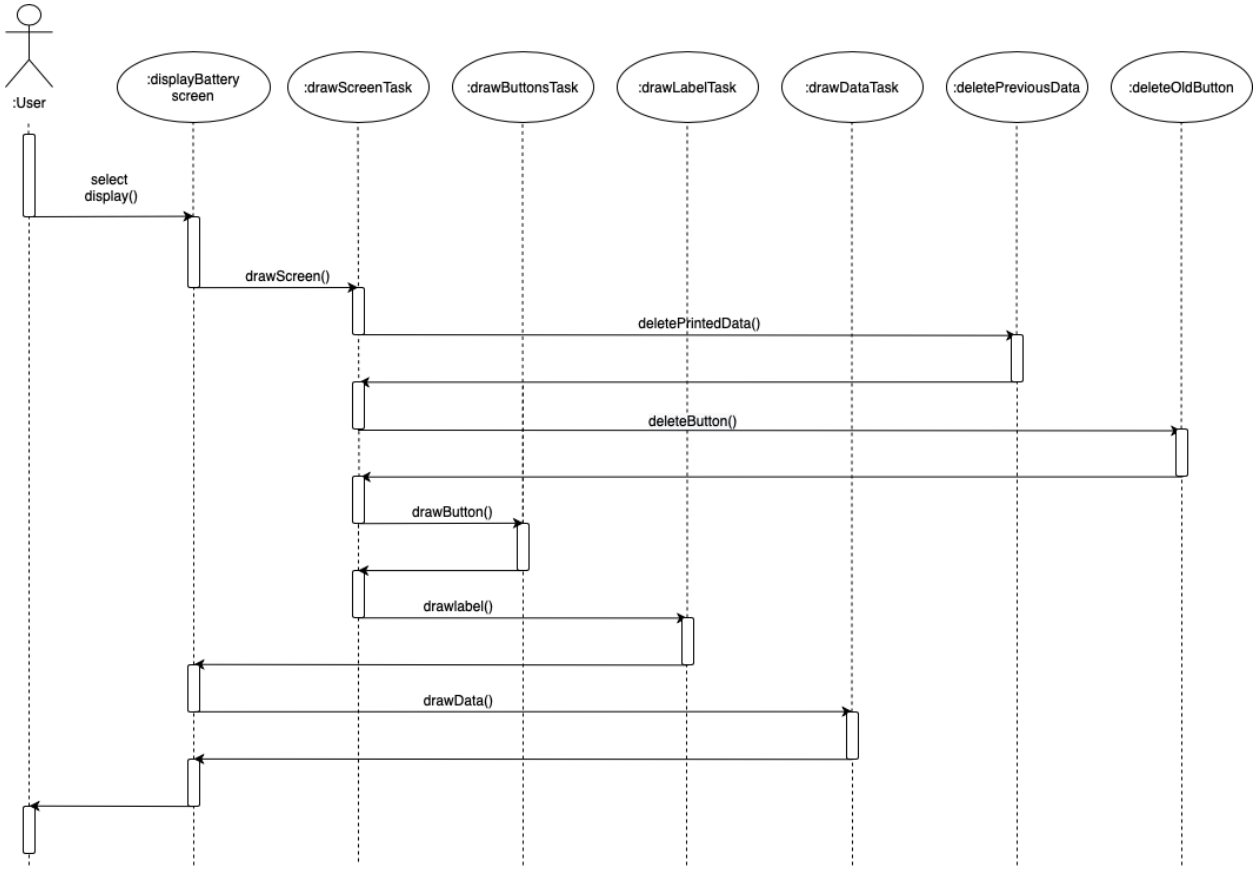


Figure 18. Sequence Diagram for Battery Screen

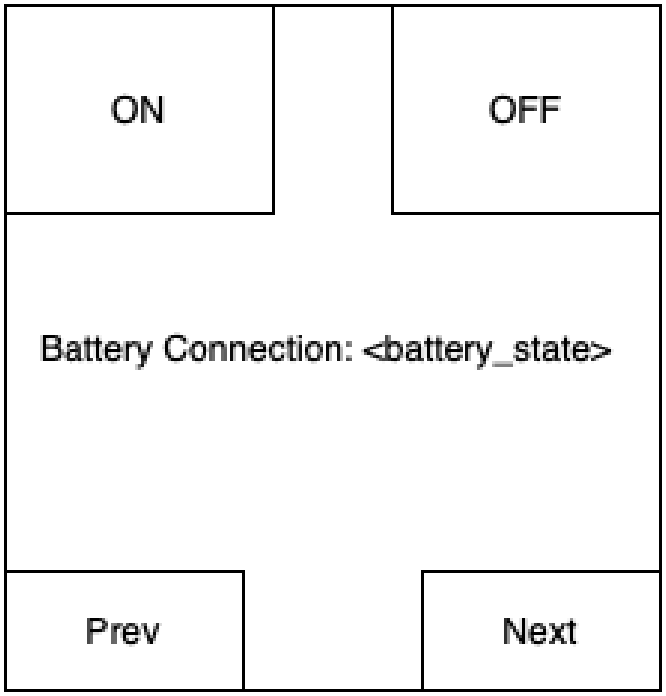


Figure 19. Front panel Design for Battery Screen

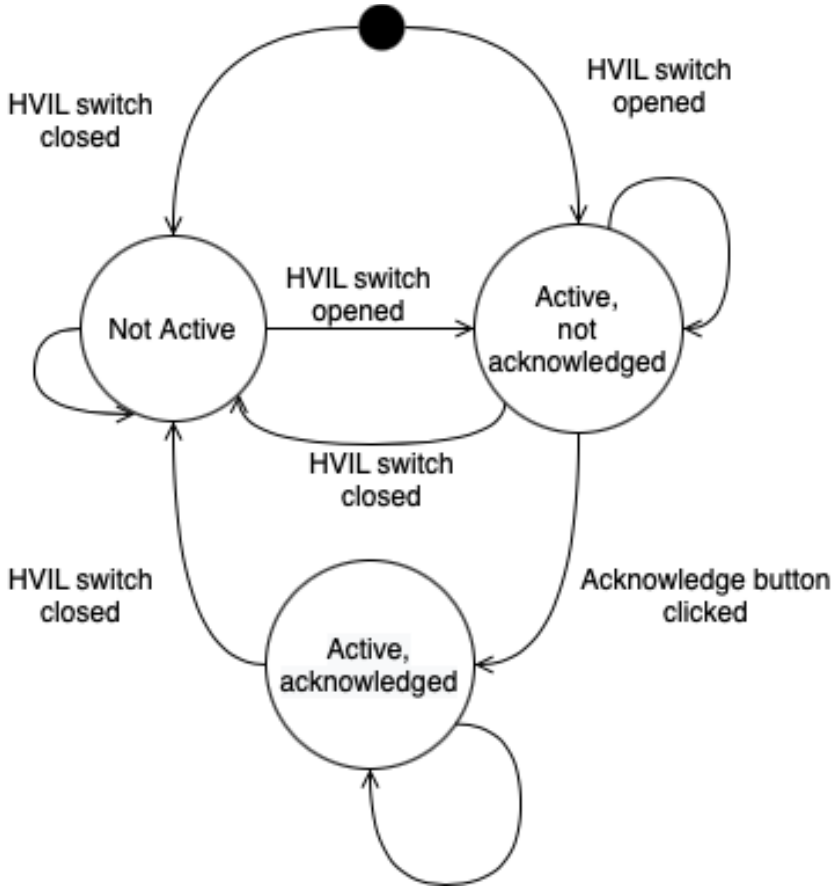


Figure 20. State Diagram for HVIL Alarm

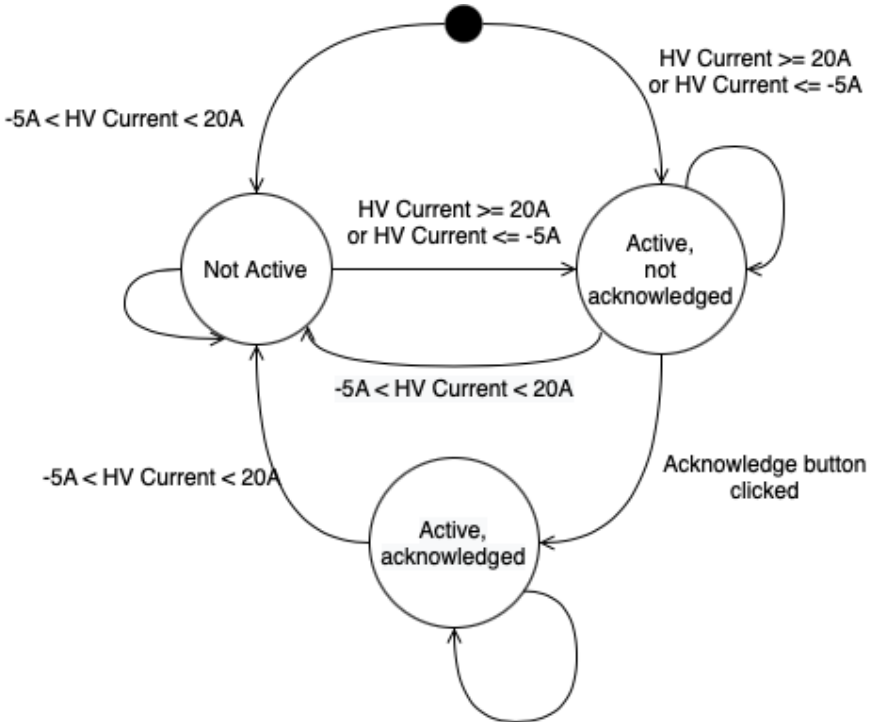


Figure 21. State Diagram for Overcurrent Alarm

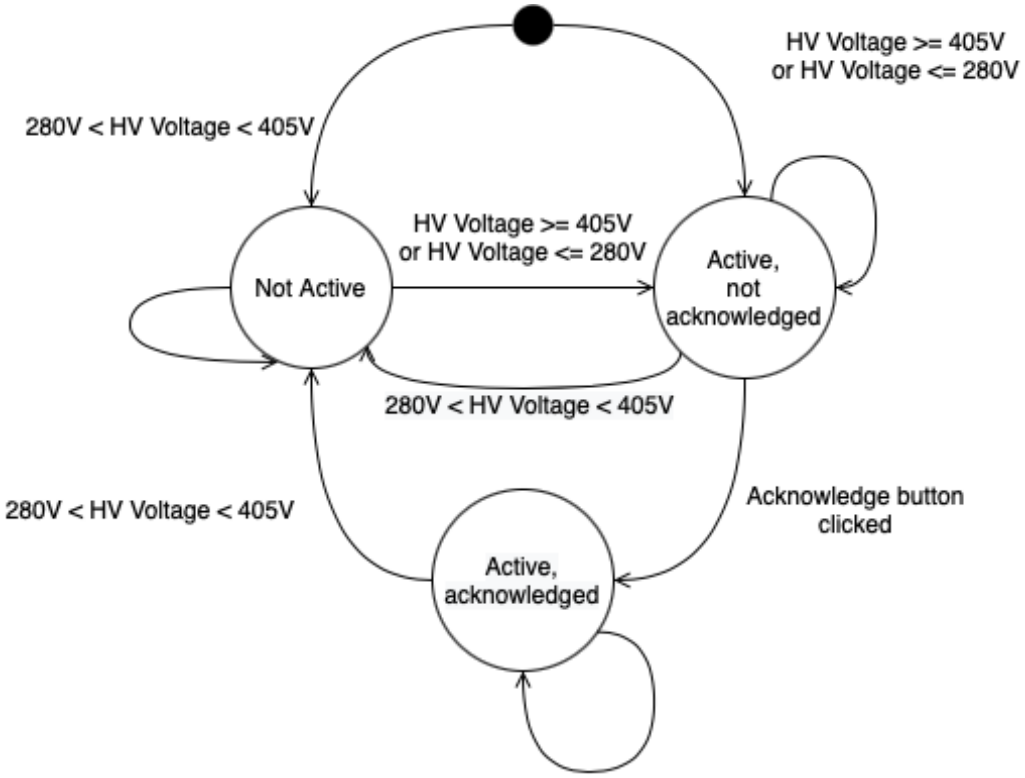


Figure 22. State Diagram for High Voltage out of Range Alarm

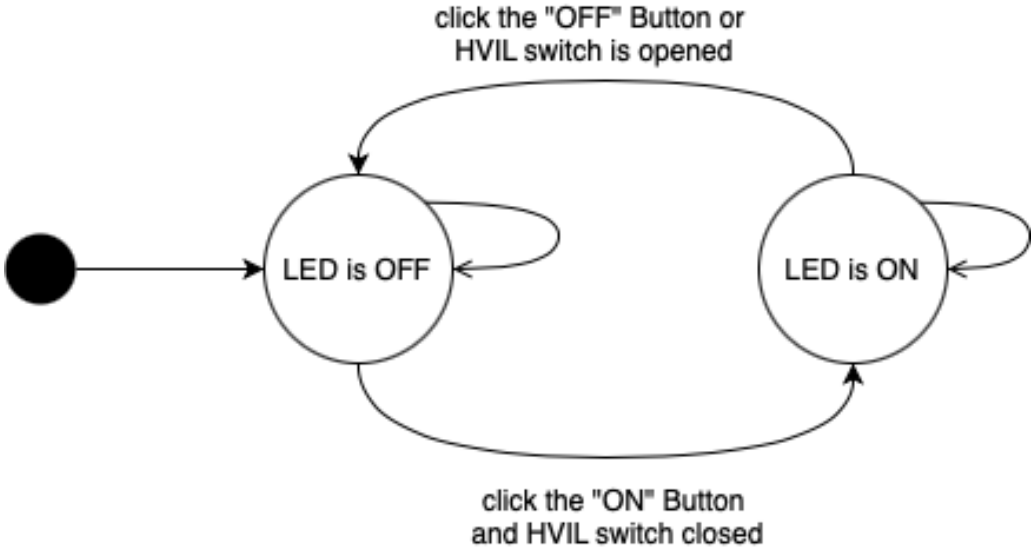


Figure 23. State Diagram for Contactor

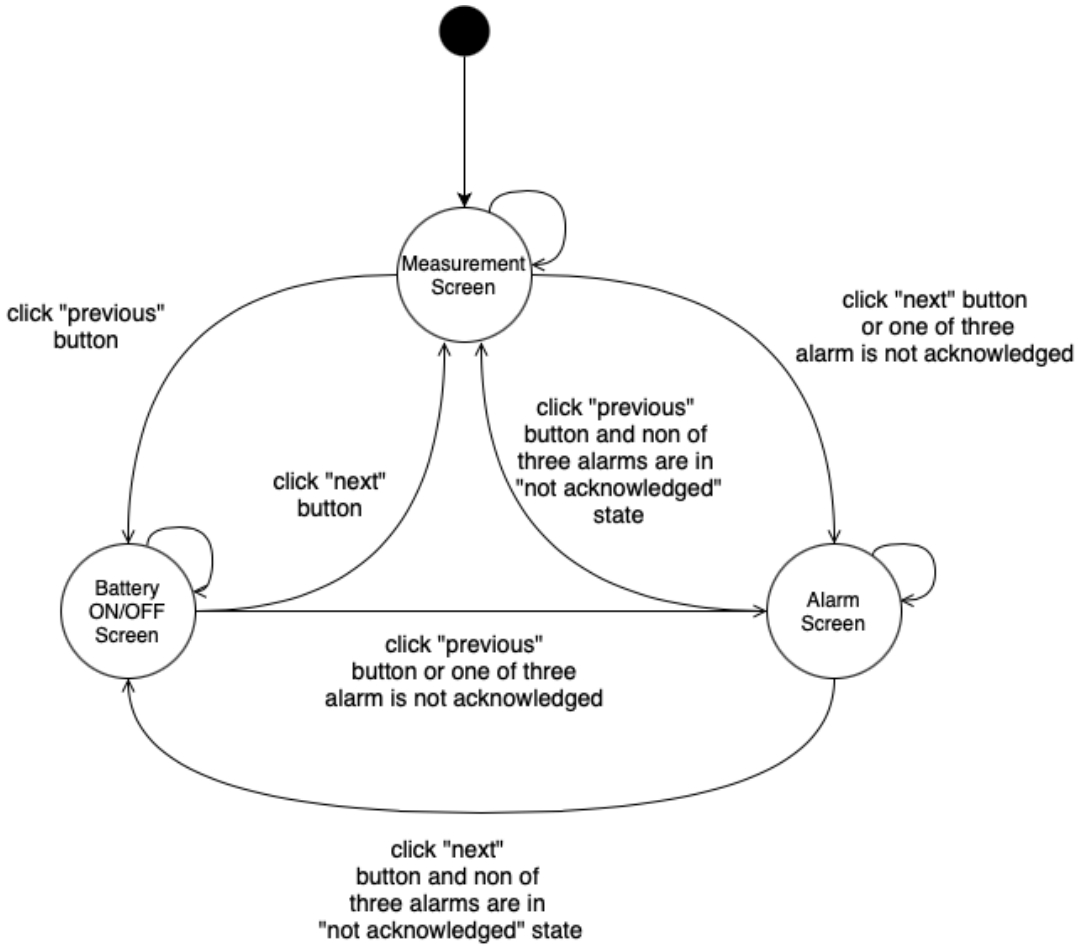


Figure 24. State Diagram for Touch Screen Display