LAB 2 – "STUBBING OUT THE SYSTEM" REPORT

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

Date: 2nd February 2020

TABLE OF CONTENTS

LIST OF FIGURES

1.0 ABSTRACT

The goal of this project is to create the first piece of a simple BMS that would be used for monitoring a high voltage electric transportation system. This phase will create sections of the display components, input/output stubs for data, and a basic scheduler for running tasks. It will inform the user of measurements, alert the user with alarms, and take input from the user via the touchscreen.

2.0 INTRODUCTION

This project will implement a basic time-based scheduling system that will run tasks designated for updating measurements and interacting with both the GPIO and the TFT display. Some physical components will be stubbed out and will use software-generated values. The scheduler will be a round-robin system. The other goal is to establish a driver for the TFT display to create a user interface.

3.0 DESIGN SPECIFICATION

The design of this system is defined by both the hardware and software architecture and requirements. The microcontroller used is the Arduino Mega 2560 R3. The hardware requirements are to use a microcontroller for GPIO access and to interface with the TFT display. The GPIO is currently used for the HVIL input and the simulated contactor output. The TFT display is used for displaying measurements/alarms/states and taking input from the user via touch. The user will be able to close/open the battery/contactor circuit via a toggle button on the screen. Both the HVIL input and contactor output will be represented by LEDs and also a status on the display.

The software architecture consists of two main blocks: the TFT driver and the round-robin scheduler. The round-robin scheduler allows us to encapsulate our functions and data structures into task control blocks. These control blocks are run sequentially inside the scheduler, and afterwards the rest of the loop time is delayed completing a one second time period. These blocks currently encapsulate the functionality for measuring and updating the task data, as well as inter-task communication. The TFT driver allows the microcontroller to efficiently interface with the display. It provides the ability to gather input via the touchscreen and also to display information through labels, buttons, and printed data. This functionality is also encapsulated in a task control block but is separated in the software into two main functions: the input from the display, and the output to the display.

The task control blocks update the following values with either GPIO input, software generated values, or input from the touchscreen: state of charge status, all measurement values (temperature, HV voltage, HV current, HVIL input), and alarms (over-current, high voltage interlock, high voltage out of range). All of this data is also displayed onto the touchscreen using the task control blocks.

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 22) and a single digital output pin (pin 24). These will be used for the high voltage interlock, and the simulated contactor respectively, which is represented in the **Figure 4** below.

The main architecture of the software revolves around the scheduler. This project uses a round-robin style scheduler that executes all tasks sequentially. A task is stored in a Task Control Block which holds the function pointer and data pointer for the task. A queue of tasks is built, of which the round-robin scheduler executes. The Scheduler will loop through all the tasks in the queue, sequentially executing each of one by calling the function via function pointer with its task data pointer, as shown in **Figure 8**. After executing each task, the main loop delays to create one second period. The implementation details of the Scheduler task 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 high voltage interlock loop. Among those values, state of charge, temperature, high voltage current, high voltage voltage are implemented by software with different defined updating rates for each of them, while the high voltage interlock loop value is implemented in hardware via input digital port 22 in **Figure 4**. The temperature value will be updated every one second, while the HV current will be updated at every two seconds and the HV voltage is updated every three second. In order to make these values updated correctly with those rates, the measurement task needs to use the shared variable, called clockCount that is incremented by one every one second, to keep track of the number of one-second cycles so that we can update the value at the correct rate. This is represented in the pseudo code in Appendix B. The cycleCount variable is passed to measurement tasks via clockCountPtr pointer from Scheduler (see in **Figure 5**). The measurement task will be using float pointers to update temperature, HV current value, HV voltage, state of charge and HVIL values. These pointers that measurement task uses are also represented clearly in **Figure 6**. Indeed, the measurement task is broken down into four sub functions, each taking care of updating one of those four values (see in **Figure 5**). And the measurement task will call and pass the corresponding value pointer to each sub function. The whole pseudo code for the measurement task can be viewed in Appendix B. These updated data such as temperature, HV current, HV voltage and HVIL will then be presented onto the touch screen display by subtask of touch screen task (called display task), which will be talked about later, shown in **Figure 7**- Data Flow diagram.

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 at a one second rate (**Figure 6**). In order to update the SOC value at the correct rate, the SOC task also uses the same global variable, clockCount, passed via pointer from the scheduler (see in **Figure 5**). The SOC task updates the SOC value via pointer that is passed down from the Scheduler, like represented in **Figure 5-** structure diagram. The pseudo code for the complete state of charge task is shown in Appendix C. The SOC value after updating will be later displayed on the touch screen by subtask of touch screen task (called display task), which will be discussed about later, see data flow in **Figure 7**.

The contactor task sets the value of the digit output pin which represents the simulated contactor (see pin 24 in **Figure 4**). The digital output depends on user input from the batteryToggle of the touch screen, updated by inputTask (which will be talked about later). This is clearly shown in **Figure 7**. The contactor task only consists of one sub function which is used to update the value of the digital output pin. Data for the digital output pin is passed down from Scheduler via float pointer and the pin number to set is passed down by a constant value. The reason why the data for the digital output pin is passed via float pointer is because all of our data on the touchscreen is updated through float pointers in the PrintedData struct (see in **Figure 6**). Pseudocode for this can be found in Appendix F: printDataToString. The contactor task will have two main states, which is either the battery connection is closed or open. 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 (see in **Figure 21**). 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 in software at specific updating rates. HVIL alarm updates its states at one second rate (see in **Figure 18**). Overcurrent alarm updates its states at two seconds rate (see in **Figure 19**). High voltage out of range updates its states at three seconds rates (see in **Figure 20**). The alarm task structure splits into three functions that each updates its respective alarm state (see in **Figure 5**). Similar to the measurement task, in order to update these alarms with the correct rate, the alarm task uses the shared variable clockCount which is passed via a float pointer from the Scheduler. The values for the state of three alarms are updated via float pointers which points to the index state for each alarm. 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 7**). 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 5**). 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 and a list of Screen struct. A Screen encapsulates information of each screen including a list of XYbutton struct in each screen (called buttonPtr), array of PrintedData struct pointers (called dataPtr), and the number of data printed on the screen, called dataLen. XYButton struct which encapsulates the coordinates of the buttons (next, previous and battery toggle) on the screens and also the other display information. Also, the PrintedData struct encapsulates the information about display information of a single data in each screen. The Screen struct, XYButton struct, PrintedData struct can be viewed in **Figure 6**. The value from the shared variable clockCount is used to handle the edge case when the screens are drawn for the first time ever as the program is just loaded in.

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 5**). In order to be able to get the touch input correctly based on the touch point on the touch screen from the users, the touch input uses the XYButton passed from the Scheduler. Also, the inputTask updates the shared variable, called batteryOnOff, for the contactor task by dereferencing the PrintedData struct.

The display task is broken down into the sub functions which are: drawScreen(), drawLabel() and drawData() (shown in **Figure 5**). The drawScreen will use the information stored in the Screen object to draw the buttons and information in the PrintedData to draw the label. Then, the PrintedData datas stored in the Screen struct is passed to the drawData() function to draw the only updated data. Also, the PrintedData, it stores the old value of that data and also the reference to the shared variable of the new value of that data in order to avoid printing not updated data and fasten up the clearing old value on the screen of a data without having to clear the whole screen. It also does the work of converting the float values in PrintedData to their String equivalents. This is done using an enum PRINT_TYPE in the printDataToString function. The whole pseudo code for touch screen tasks can be viewed under Appendix F.

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 11**). In order to draw the measurement screen, a display task called the drawScreen() function to draw buttons and drawLabels, then it calls the drawData() function to display all the data belonging to the measurement screen to the touch screen (see **Figure 10**). 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 9**.

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 14**). In order to draw the measurement screen, a display task called the drawScreen() function to draw buttons and drawLabels, then it calls the drawData() function to display all the data belonging to the measurement screen to the touch screen (see **Figure 13**). In the alarm, 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 12**.

The battery on/off screen has a different layout than other screens. Battery on/off screen has a battery toggle button that occupies half of the screen, displaying the battery connection state, the next and previous button at the bottom of the page (see **Figure 17**). In order to draw the measurement screen, a display task called the drawScreen() function to draw buttons and drawLabels, then it calls the drawData() function to display all the data belonging to the measurement screen to the touch screen (see **Figure 16**). 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 15**. 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 21**). The next and previous button on each screen allows the user to scroll through the three screens circularly (see in **Figure 22**). If the users keep clicking on the next button, they will circularly back through all the screens. And the previous button also gives the same functionality, but in the reverse order.

5.0 TEST PLAN

The test plan includes three parts:

1. Requirements
2. Test coverage
3. Test cases

5.1 Requirements

The round-robin scheduler will run all task control blocks and delay for the rest of the time period (time period is one second). This means all tasks run once a second.
The TFT display will have three screens with each their own functionality:
“Alarms” displays:
The HV out of voltage range alarm (updates every three seconds)
The over-current alarm (updates every two seconds)
and the HVIL alarm (updates every second)
“Measurements” displays:
The temperature (updates every second)
The state of charge (updates every second)
The HV current (updates every two seconds)
The HV voltage (updates every three seconds)
The HVIL status (updates every second from digital pin 22) (“OPEN/CLOSED”)

“Battery Toggle Screen” displays:

The status of the contactor (updates once a second based on user input) ("OPEN/CLOSED")
and takes input from:
The battery toggle button (ON/OFF) (provides input from user touch once a second)

The contactor output is updated once a second to the digital output pin 24.

5.2 Test Coverage

The main loop runs every second.

The value of the alarms will cycle between: "ACTIVE", "ACTIVE, NOT ACKNOWLEDGED", and "ACTIVE, ACKNOWLEDGED" at their specified rate.

The temperature value cycles between: [-10, 5, 25] at 1s rate
The HV current value cycles between [-20, 0, 20] at 2s rate
The HV voltage value cycles between [10, 150, 450] at 3s rate
The state of charge value cycles between [0, 50, 100] at a 1s rate.
The HVIL input will update once a second depending on the state of digital pin 22. When low, it will state "CLOSED", when high, it will state "OPEN".
The status of the contactor will state "CLOSED" if the battery is closed/on. It will state "OPEN" if the battery is open/off.
The battery toggle button will take input once a second and switch between "OFF" and "ON"
The contactor output is updated once a second to the digital output pin 24. A "CLOSED" status will output a logic high. A "OPEN" status outputs a logic low.

5.3 Test Cases

The scheduler + delay time will be printed to the serial monitor (removed in final version) in milliseconds. This should be 1000 milliseconds.
The contactor output will be tracked visually from the display and the LED (pin 24).
The HVIL input will be tracked visually from the display and the LED (pin 22).
All values on the measurements/alarm's screens will be tracked visually. Each cycle is one second, so it is easy to track.
The battery toggle switch will take input once a cycle (seen visually and from the contactor output).
The same goes for the previous and next buttons. They will take input once a second, and switch to the next/previous screen accordingly.
Most test cases for this project are visual which makes it generally an easier test.

6.0 Presentation, Discussion, and Analysis of the Results

Our design accomplishes all test requirements according to the test cases. All screens display exactly what they should in the correct location, and every button (touch input) functions according to how it needs to. The contactor output follows the touch screen input (battery toggle) and completes all test cases. The screen itself is fluid and reactive to input / changes in data. The HVIL circuit/software implementation works as expected in the test cases.

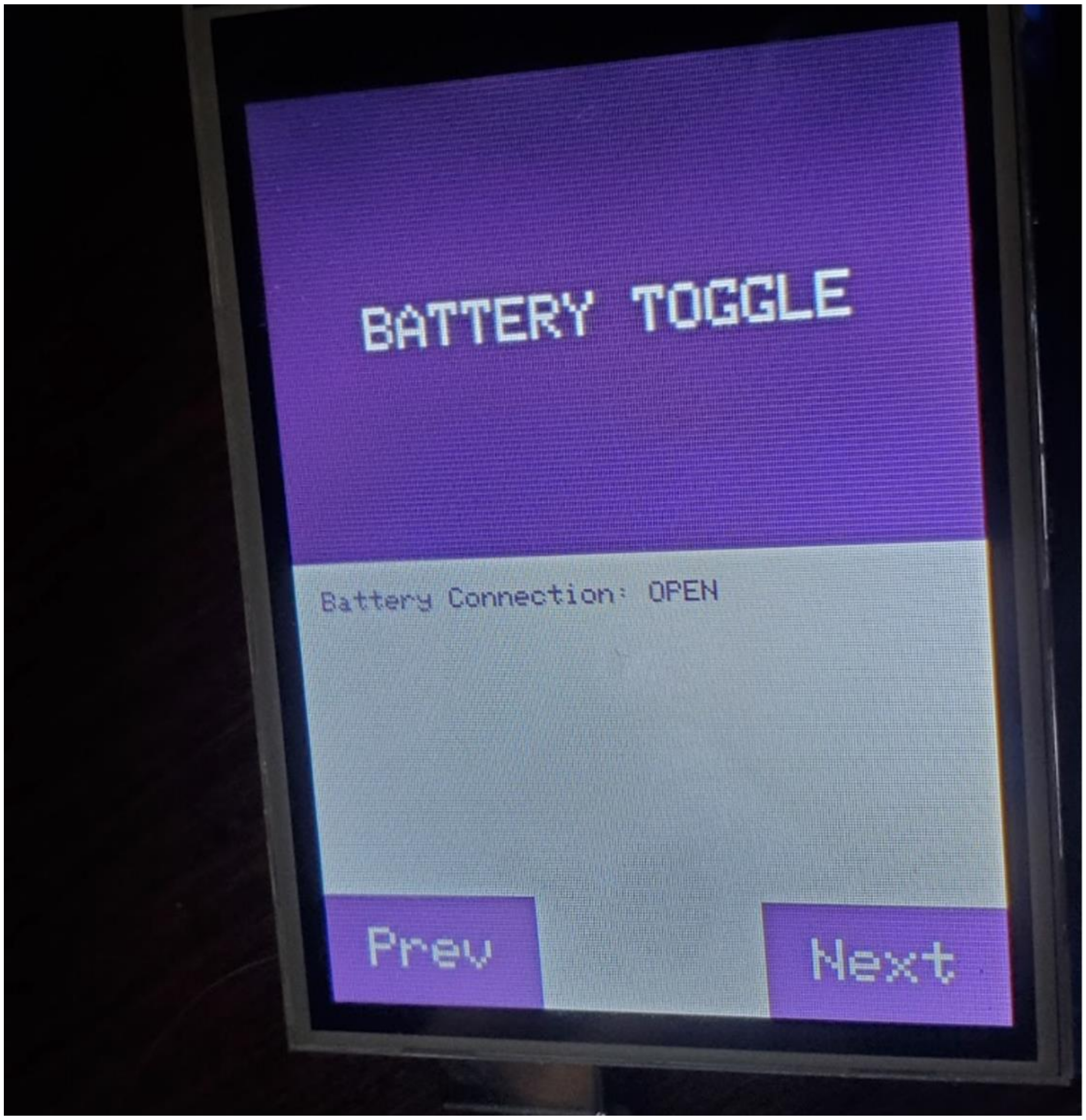
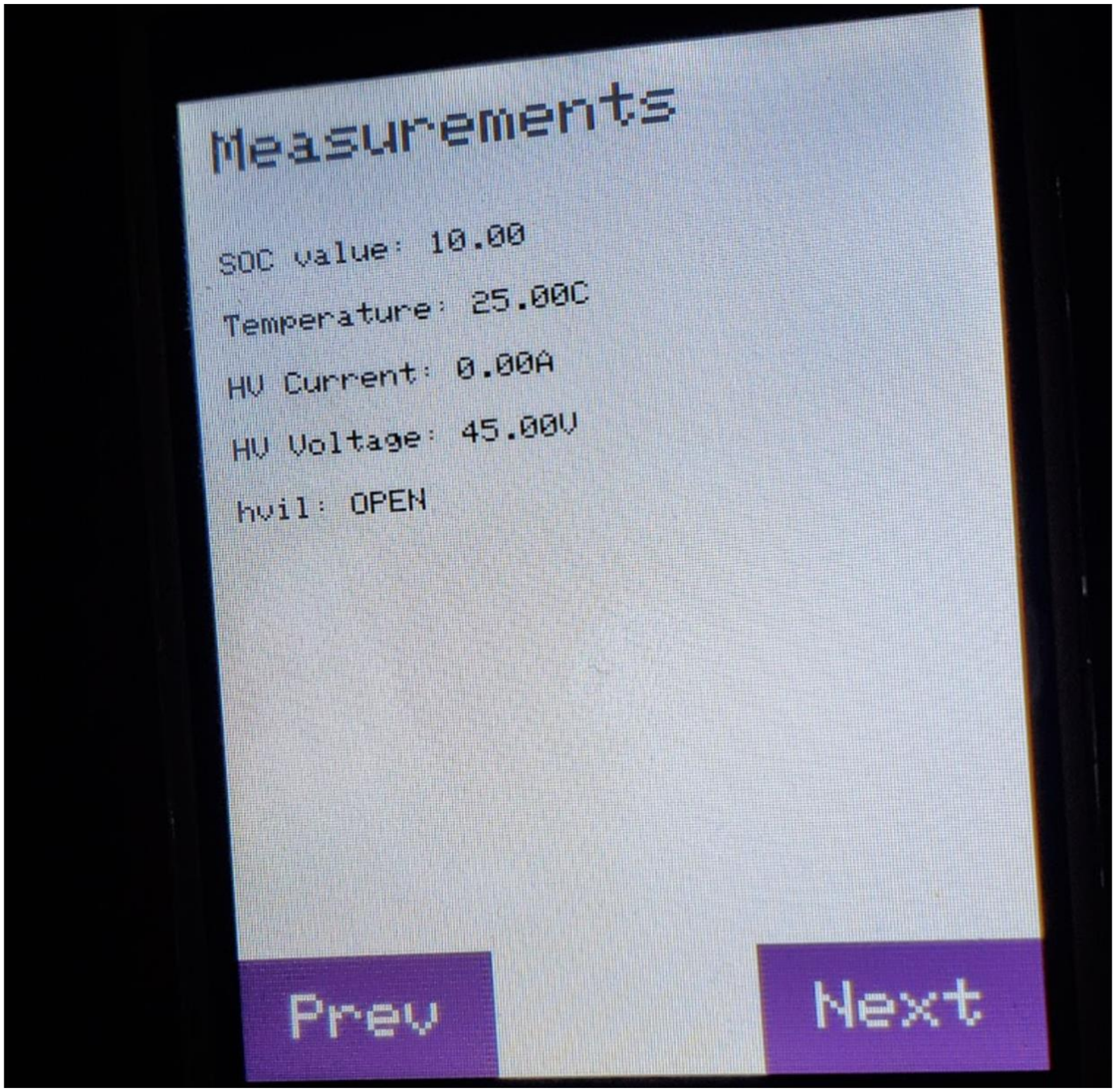


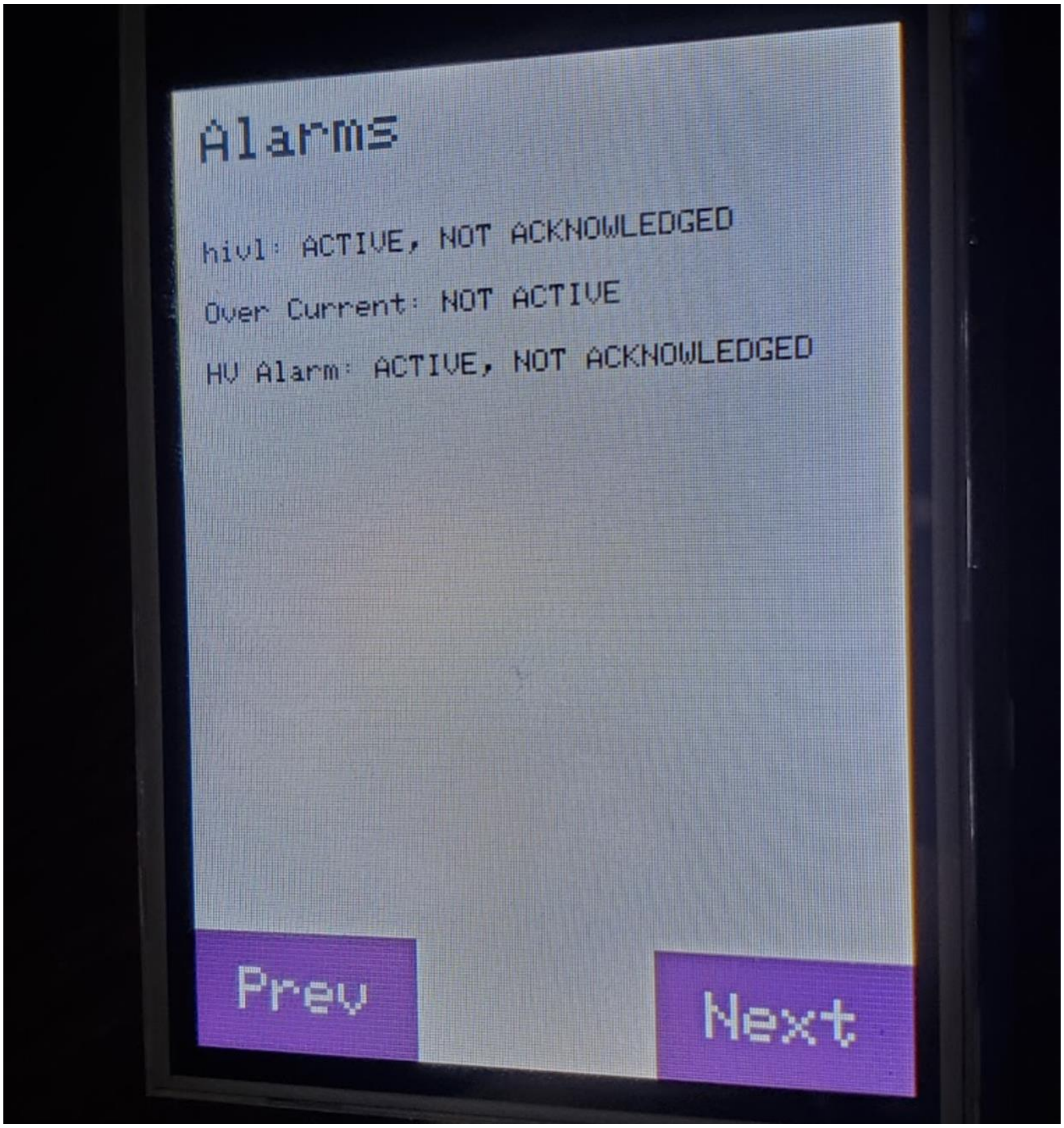
Figure 1. Battery Toggle Screen



Measurements

SOC value: 10.00
Temperature: 25.00C
HV Current: 0.00A
HV Voltage: 45.00V
hvil: OPEN

Figure 2. Measurements Screen


Alarms

hivl: ACTIVE, NOT ACKNOWLEDGED

Over Current: NOT ACTIVE

HV Alarm: ACTIVE, NOT ACKNOWLEDGED

Figure 3. Measurements Screen

6.1 Analysis of Any Resolved Errors

There were some errors during the development of this project, and all were resolved.

The first is that we overlooked the difference between the datatypes being printed on the display. Originally the struct only used the float data type for the data that is printed on the screen. This could not print boolean or alarm states very easily. This almost made us totally change the inside of the struct which would have been a nightmare. Instead, an enum was added inside that held the type of data being printed. A function was added that converted the float to the type of data being printed (all of it was converted to a String anyways).
This is done in printDataToString, shown in Appendix F. This worked well later as we had to add another type (labels) which don't update any data. We could then just use its datatype to String conversion as a String with length zero. This data never changes, so we never update it on the screens.

Another error was also with the touchscreen. The input had to be gathered before the data was displayed, but we had to draw the screen before any input could be gathered. Because we wanted the functionality to be completely encapsulated in the tasks, we added in the clockCount shared data to the touch screen task data. When clockCount has never been increased, we know it is the first time on the display so we can draw the screen before the input. All other times we can get the input before drawing the screen.

The touchscreen also had another error in our code. We would miss the input of the person touching the screen because the touch screen can be inconsistent with the pressure being applied. This works fine if sampling fast enough, but we are only sampling once a second. So, we create a loop that uses a timeout (currently set to 20ms) that reads the input until either the timeout occurs, or we read "good" input. This is just input that has a pressure above a minimum set pressure.

6.2 Analysis of Any Unresolved Errors

There are no unresolved errors.

7.0 QUESTIONS

Question 1&2: Almost every task took less than a millisecond (0 ms using millis()), except the display task. This one took anywhere from 100ms to 200ms depending on how many values were being updated/ if a new screen was being drawn. The execution time of the scheduler (task queue) is dynamically recorded during the program execution. This allows the one second interval to be precise and also dynamically managed. The time taken by the scheduler was recorded in milliseconds and subtracted from 1000 milliseconds. The loop then delays the subtracted number, to achieve a one second time interval for the overall loop. This is shown in **Figure 8**.
Question 3. The touch screen is an input and an output. The HVIL is a digital input. The simulated contactor is a digital output. This is shown in **Figure 7**.

8.0 CONCLUSION

In conclusion, both the scheduler and the TFT driver work very well. The TFT driver is set up to easily modify the display if needed in the future. The scheduler runs all tasks once every second. All data required to be updated/printed is done so. The digital input for the HVIL effectively updates the displayed value corresponding with the LED. The battery toggle effectively updates the contactor status on the display and the LED through its GPIO pin. Inter-task communication is accomplished through tasks such as measurement/touchscreen, alarm/touchscreen, contactor/touchscreen,
state of charge / touchscreen.

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 taskQueue:
        executes task
```

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

```
float temperatureValues[] = { -10, 5, 25};
float currentValues[] = { -20, 0, 20};
float voltageValues[] = {10, 150, 45};

measurementTask(dataPtr):
   parse the dataPtr
   updateTemperature (temperature value reference, clock count)
   updateHVIL(hvil value reference, hvil pin)
```

```
  updateHvCurrent(hv current reference, clock count)
  updateHvVoltage(hv voltage reference, clock count)

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

updateHvCurrent(hvCurrentPtr, clockCntPtr):
    update HV current every two clock cycles

updateHvVoltage(hvVoltagePtr, clockCntPtr):
    update HV voltage every three clock cycles

updateTemperature(temperaturePtr, clockCntPtr):
     update temperature value every clock cycle
```

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

```
socValues[] = {0, 50, 10}

socTask(dataPtr):
    parse the dataPtr
     updateStateOfCharge(soc reference, clock count )

updateStateOfCharge(socValPtr, clockCntPtr):
    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, clock count reference)
     updateHVORAlarm(hvor alarm reference, clock count reference)
```

```
    updateHVIAAlarm(hvia alarm reference, clock count reference)

updateHVIAAlarm(hviaValPtr, clockCntPtr):
    update the hvia alarm every clock cycle

updateHVORAlarm(hvorValPtr, clockCntPtr):
    update hvor alarm every three clock cycles

updateCurrentAlarm(overCurrentPtr, clockCntPtr):
    update overcurrent alarm every two clock cycles
```

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()
    needToChangeScreen (bool) ← execute input task
    execute display task

inputTask(currenScreenPtr, screens):
    getTouchInput()
    needToDrawScreen  ← false
    if touch on next or previous button:
        needToDrawScreen← true
    if on battery screen and touch on battery toggle button:
        update value for battery on/off
    return needToDrawScreen

displayTask(currentScreenPtr, screensList, switchScreen):
     use currentScreenPtr and screen list to determine which screen to draw
     drawScreen(screenToDraw, isNewScreen)
     for each data in screen.dataList:
         drawData(data, switchScreen)

drawScreen(screen, switchScreen):
     if switchScreen:
         clear the old screen
         draw “next” button
         draw “previous” button
         if on batteryScreen:
                draw “battery toggle” button
            for each label on the screen:
                draw label

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 schedulerTask and startUpTask.
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
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

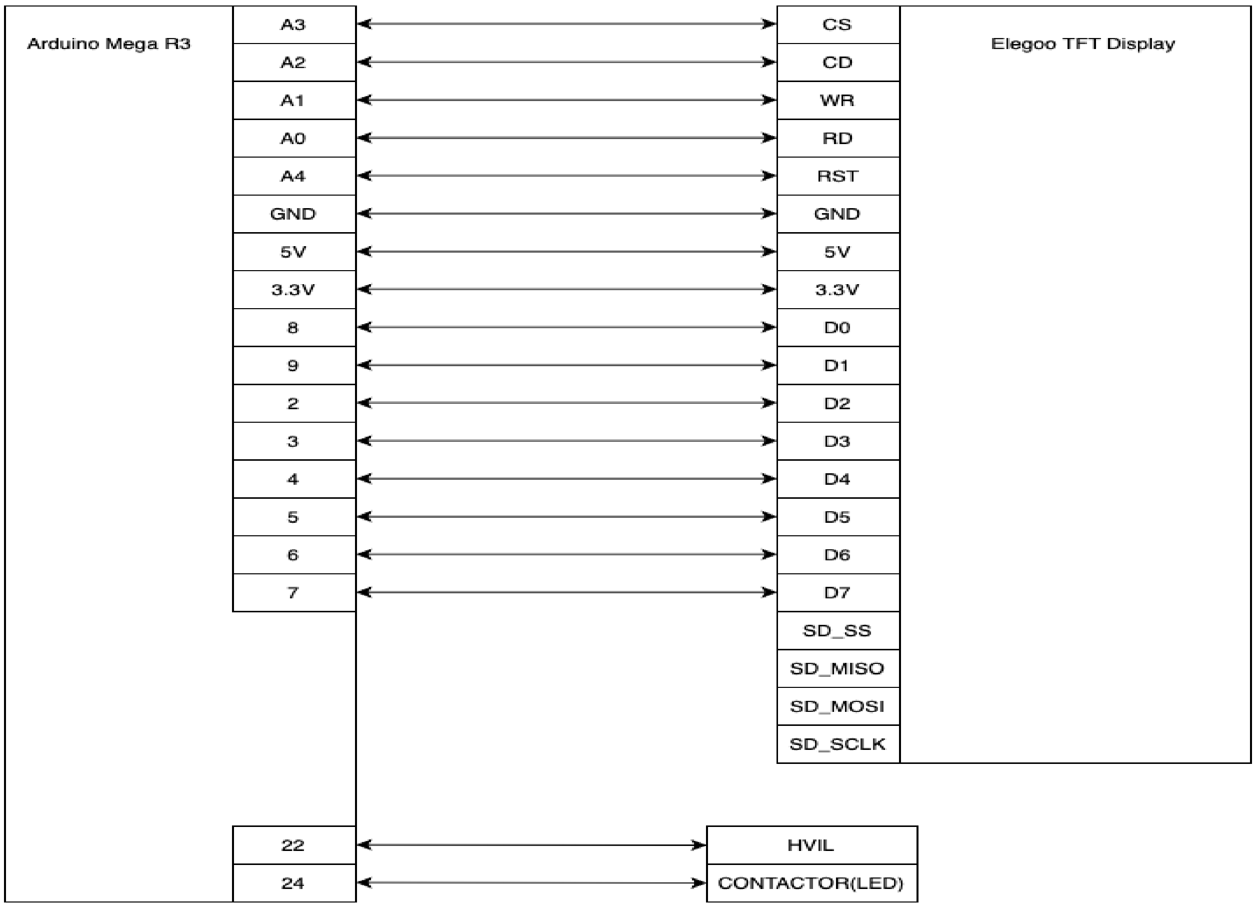


Figure 4. System Block Diagram - showing the Atmega input and output ports (and port numbers) labeled per I/O component

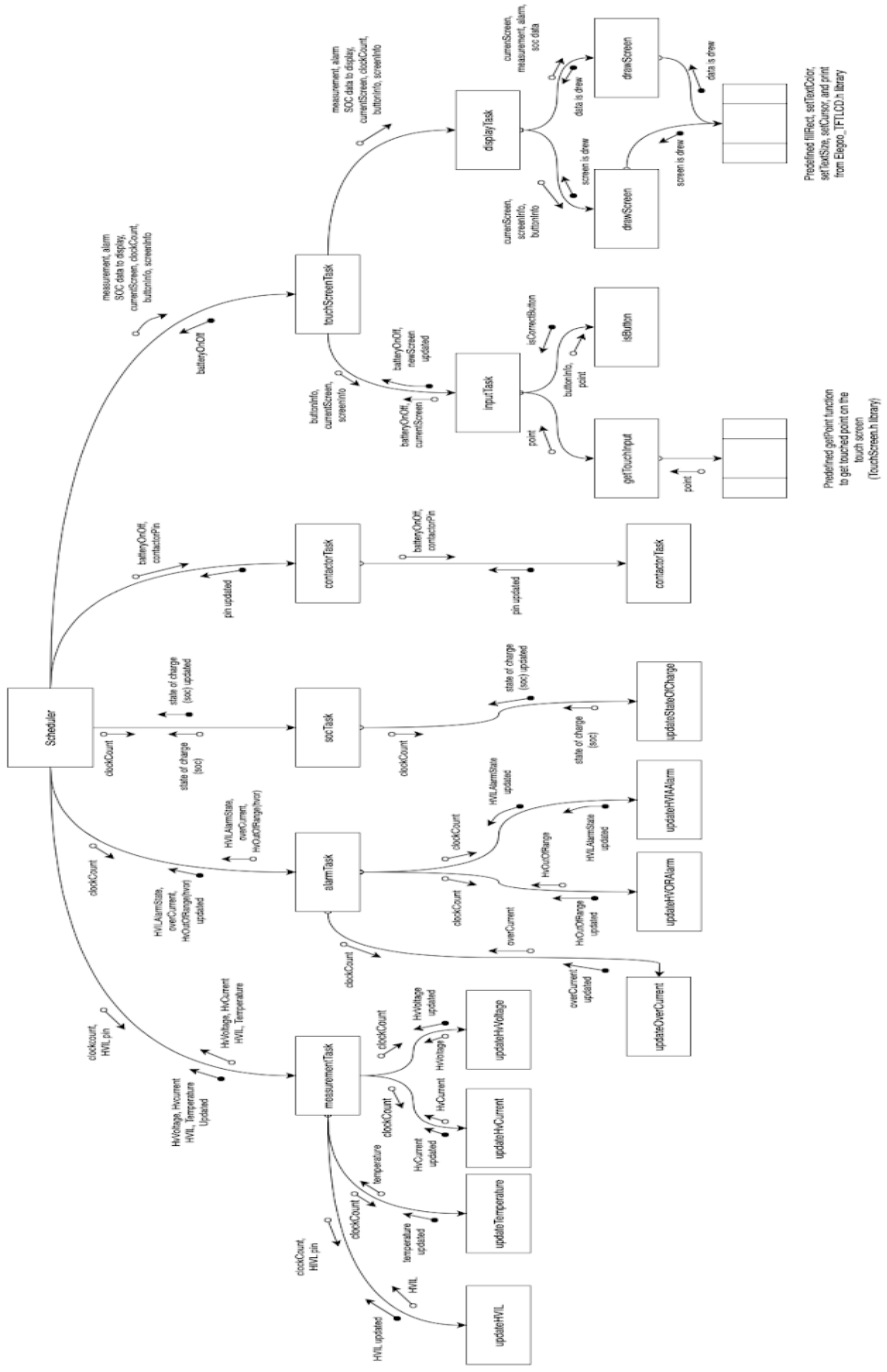

Scheduler
measurementTask
alarmTask
socTask
contactorTask
touchScreenTask
displayTask
inputTask
drawScreen
drawScreen
isButton
getTouchInput
contactorTask
updateStateOfCharge
updateHVIAlarm
updateHVORAlarm
updateOverCurrent
updateHvVoltage
updateHvCurrent
updateTemperature
updateHVIL
clockCount, HVIL pin
HvVoltage, HvCurrent HVIL, Temperature Updated
HvVoltage, HvCurrent HVIL, Temperature
clockCount
HVILAlarmState, overCurrent, HvOutOfRange(hvor) updated
HVILAlarmState, overCurrent, HvOutOfRange(hvor)
state of charge (soc)
state of charge (soc) updated
pin updated
batteryOnOff, contactorPin
batteryOnOff
measurement, alarm SOC data to display, currentScreen, clockCount, buttonInfo, screenInfo
buttonInfo, currentScreen, screenInfo
batteryOnOff, currentScreen
batteryOnOff, newScreen updated
point
buttonInfo, point
isCorrectButton
currentScreen, screenInfo, buttonInfo
screen is drew
currentScreen, measurement, alarm, soc data
data is drew
Predefined fillRect, setTextColor, setTextSize, setCursor, and print from Elegoo_TFTLCD.h library
Predefined getPoint function to get touched point on the touch screen (TouchScreen.h library)
HVILAlarmState updated
HvOutOfRange
HvOutOfRange updated
overCurrent
overCurrent updated
HvVoltage
HvVoltage updated
HvCurrent
HvCurrent updated
temperature
temperature updated
HVIL
HVIL updated
clockCount, HVIL pin

Figure 5. Structure Diagram - showing functional decomposition of tasks within the System Controller

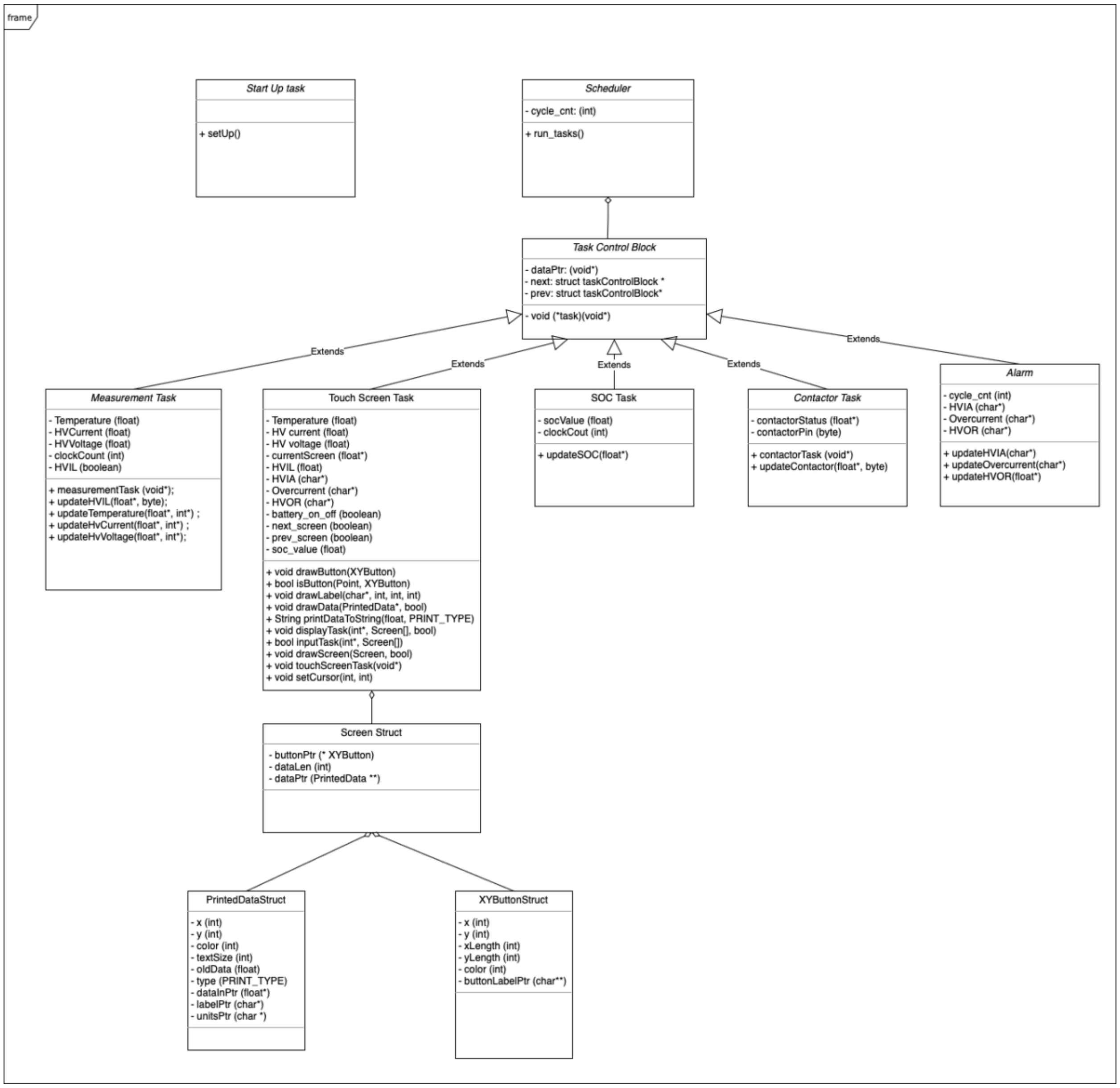


Figure 6. Class diagram - showing the structure of the tasks within the System Controller as reflected in the Structure Diagram.

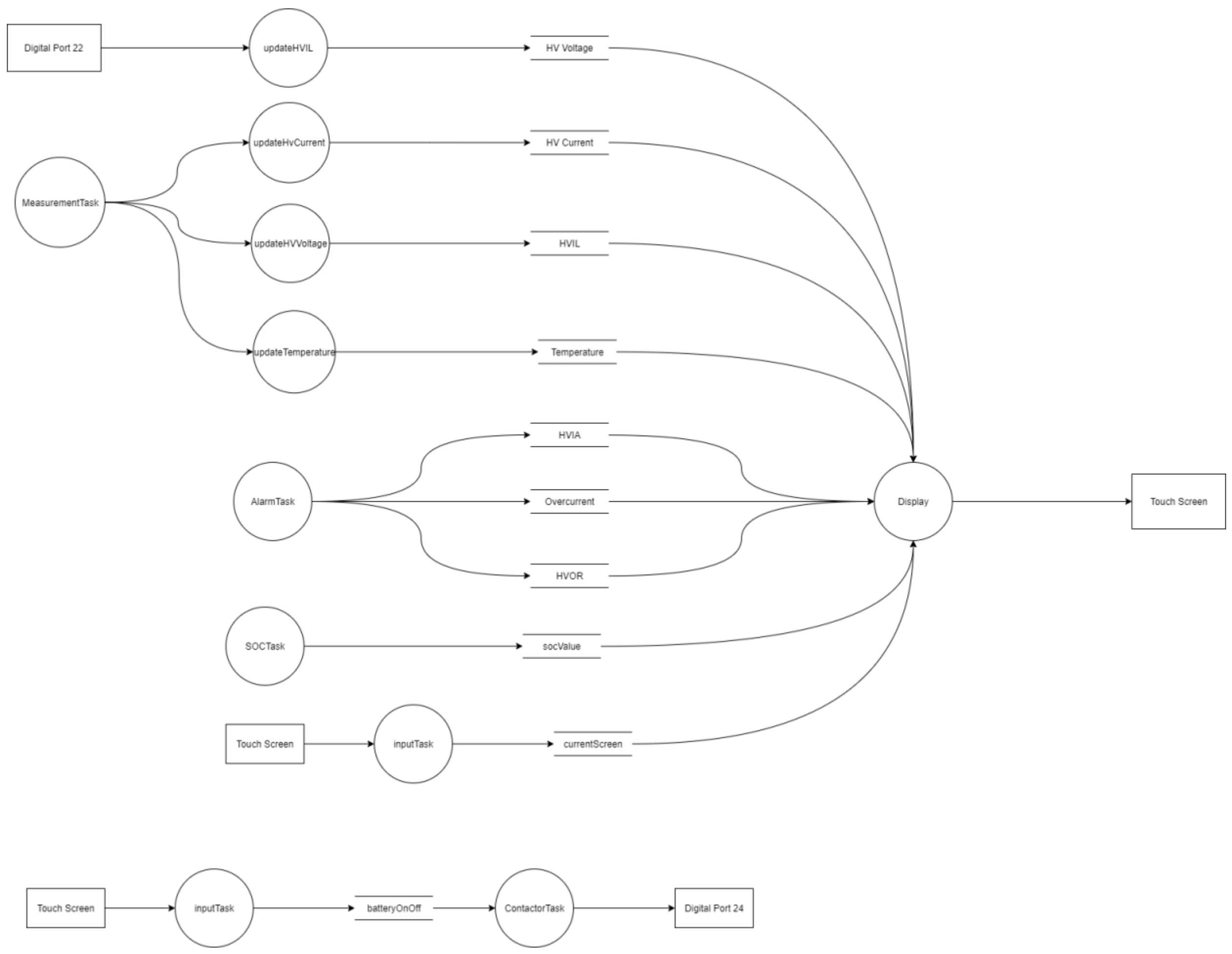


Figure 7. Data flow diagrams - shows data flow for inputs/outputs

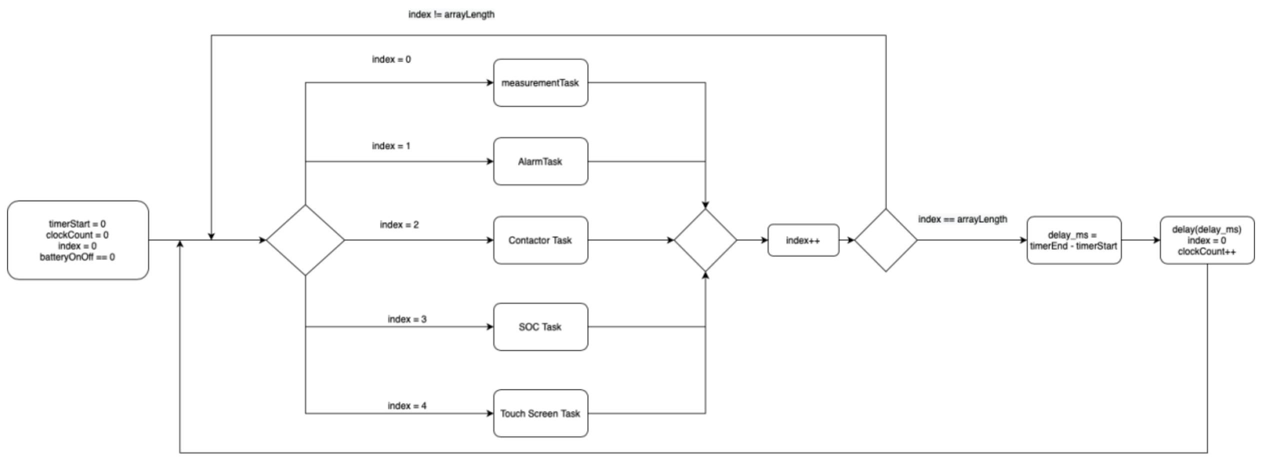


Figure 8. Activity Diagram - shows the System Controller's dynamic behavior from the initial entry in the loop() function

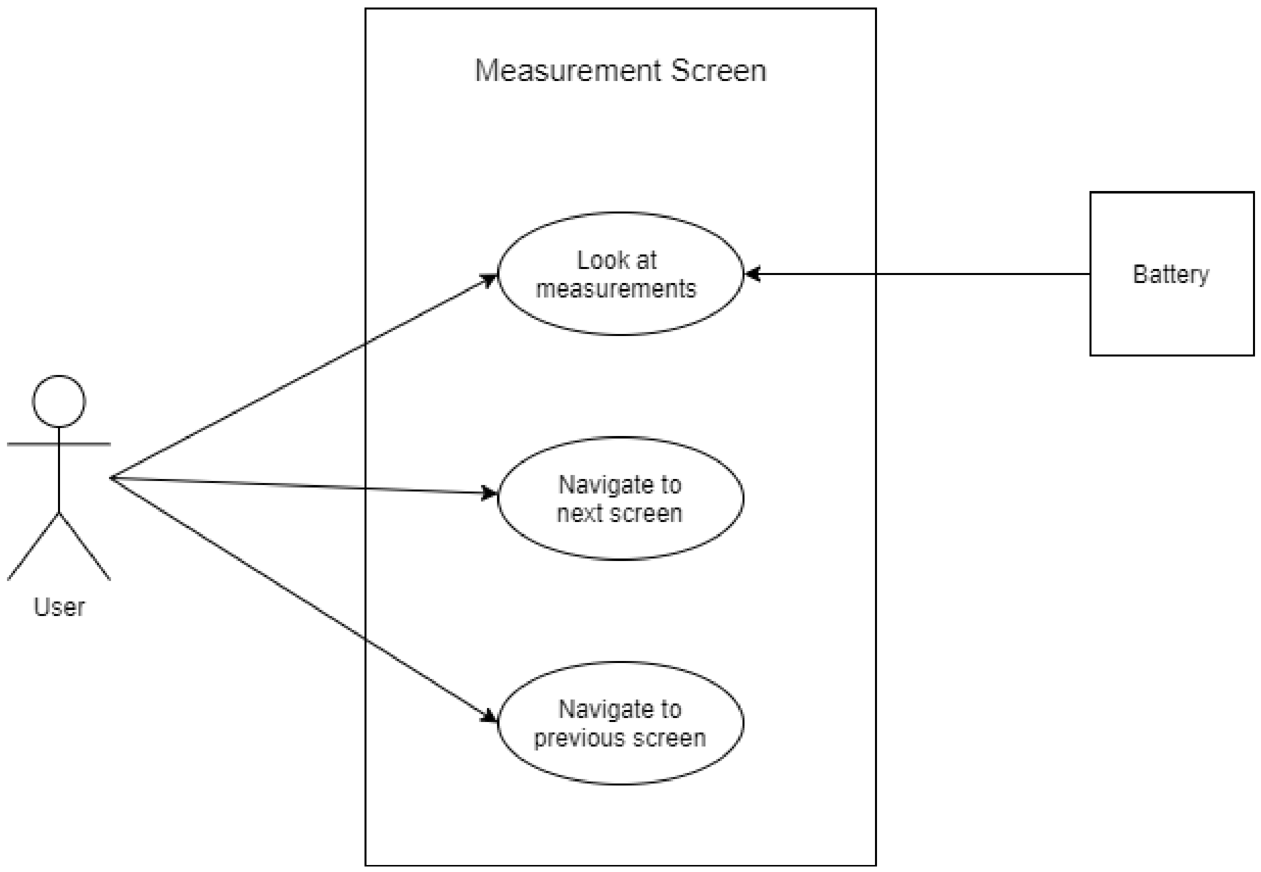


Figure 9. Use Case Diagram for Measurement Screen

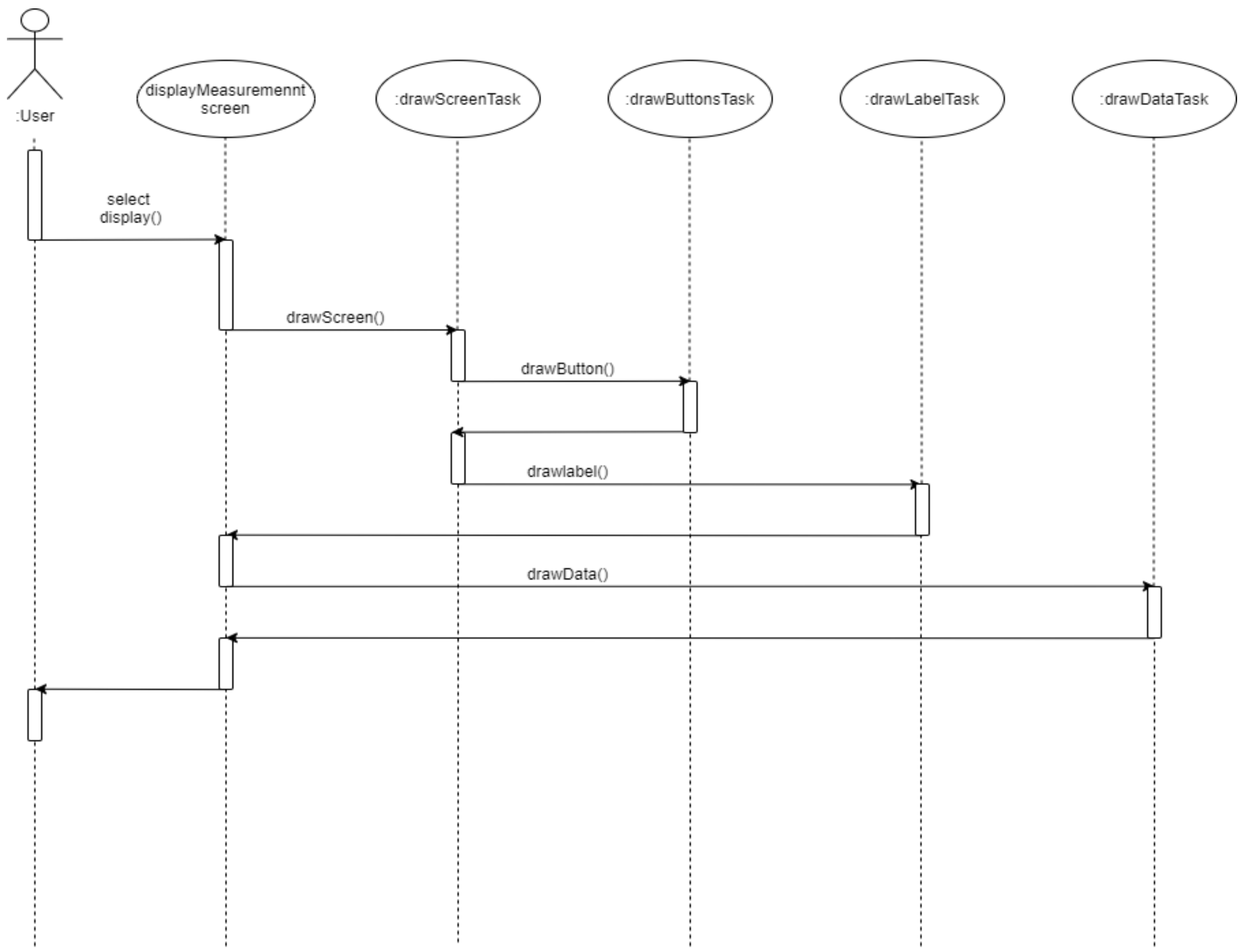


Figure 10. Sequence Diagram for Measurement Screen

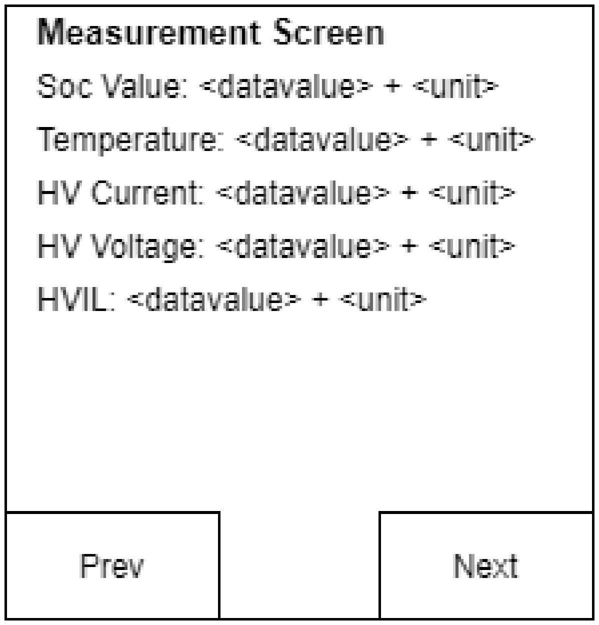

Measurement Screen
Soc Value: <datavalue> + <unit>
Temperature: <datavalue> + <unit>
HV Current: <datavalue> + <unit>
HV Voltage: <datavalue> + <unit>
HVIL: <datavalue> + <unit>
Prev
Next

Figure 11. Front Panel Design for Measurement Screen

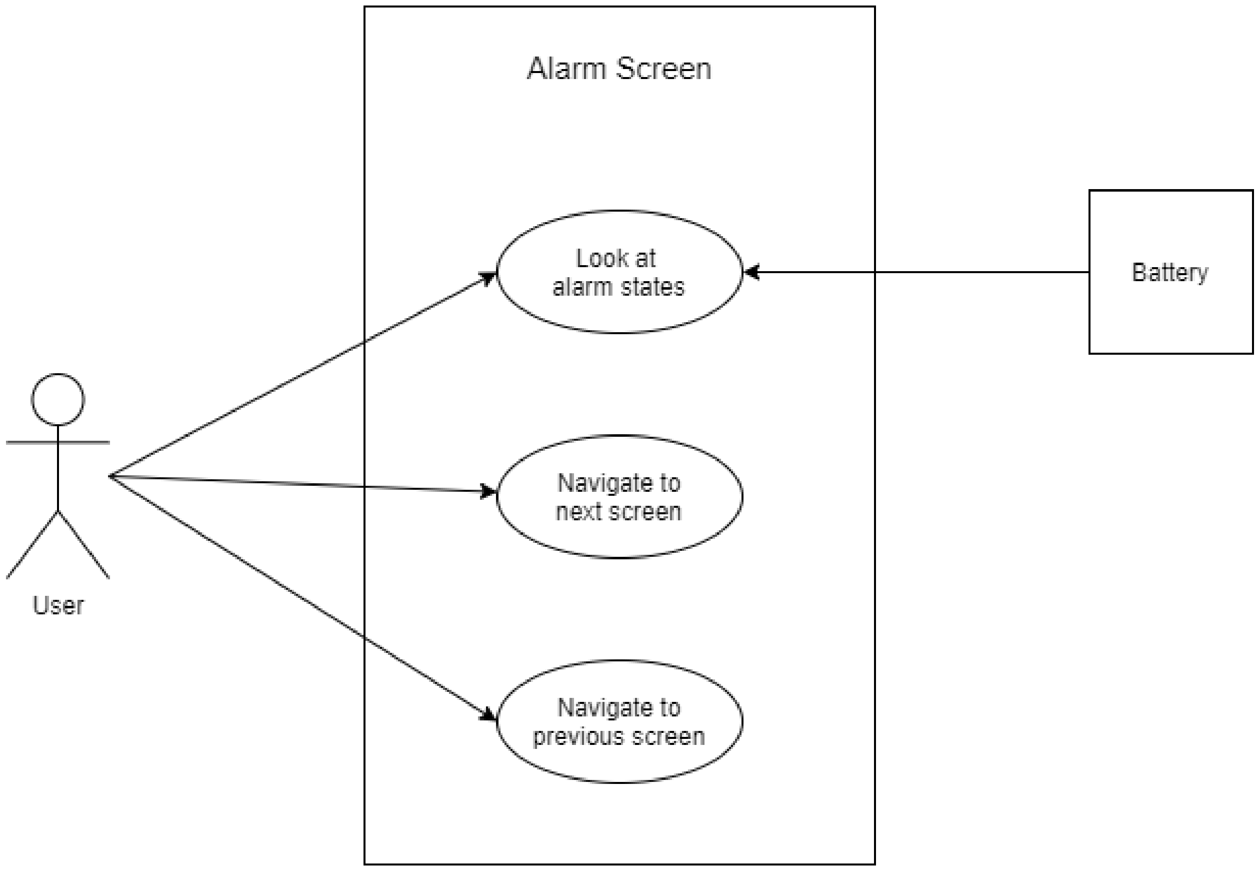


Figure 12. Use Case Diagram for Alarm Screen

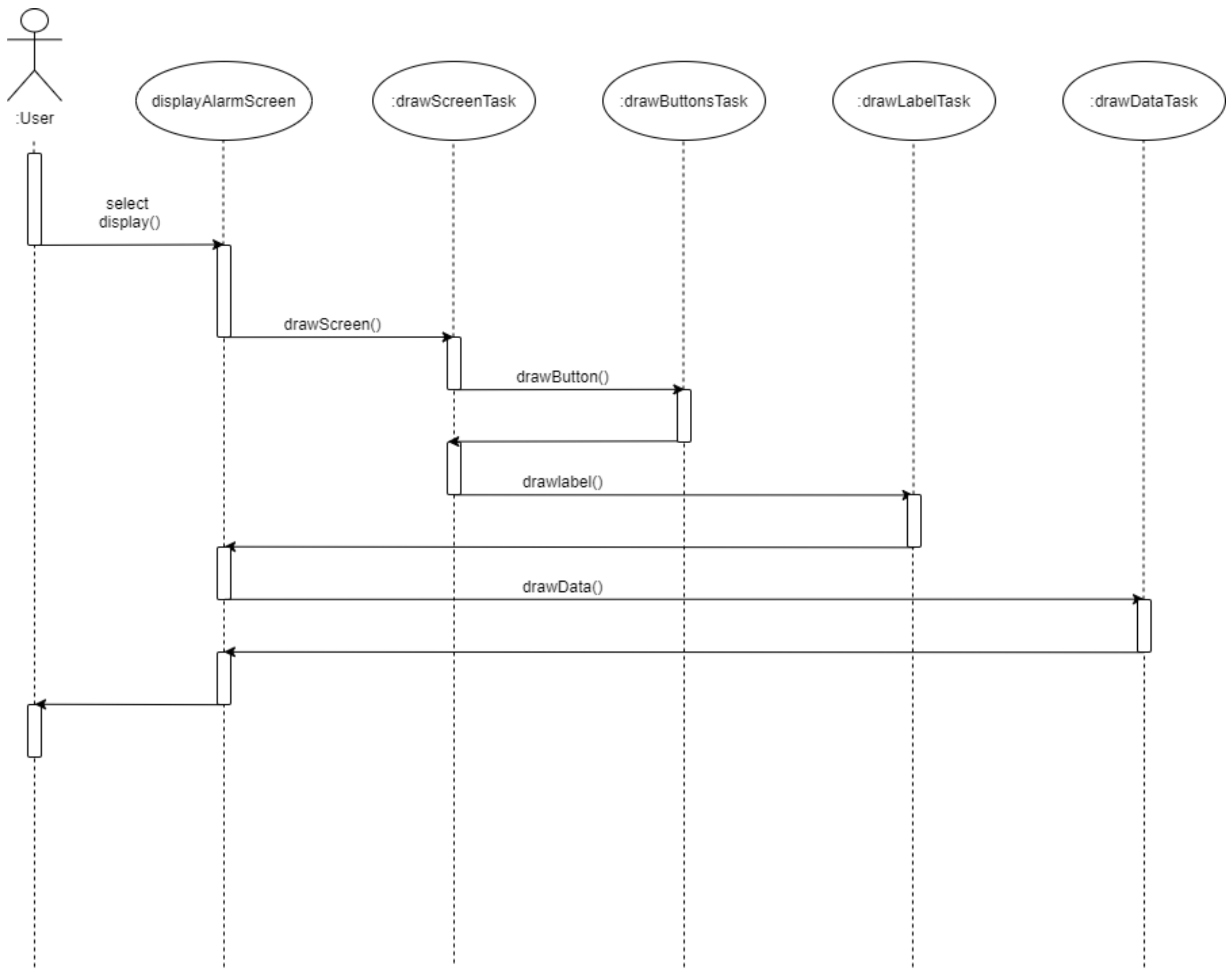


Figure 13. Sequence Diagram for Alarm Screen

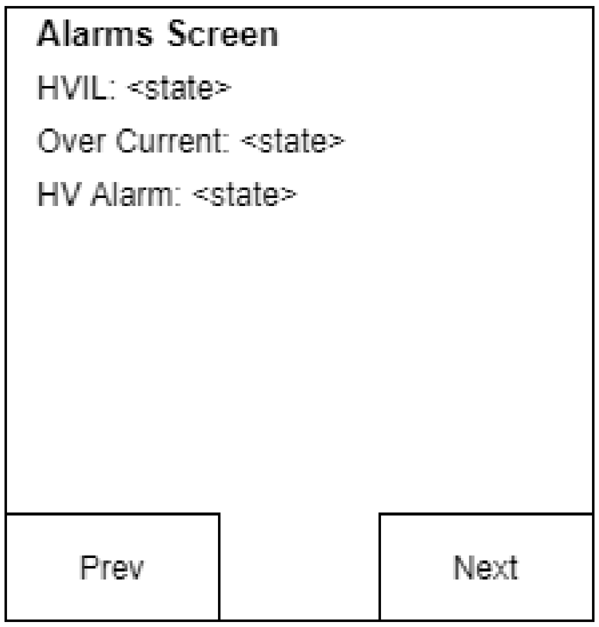


Figure 14. Front Panel Design for Alarm Screen

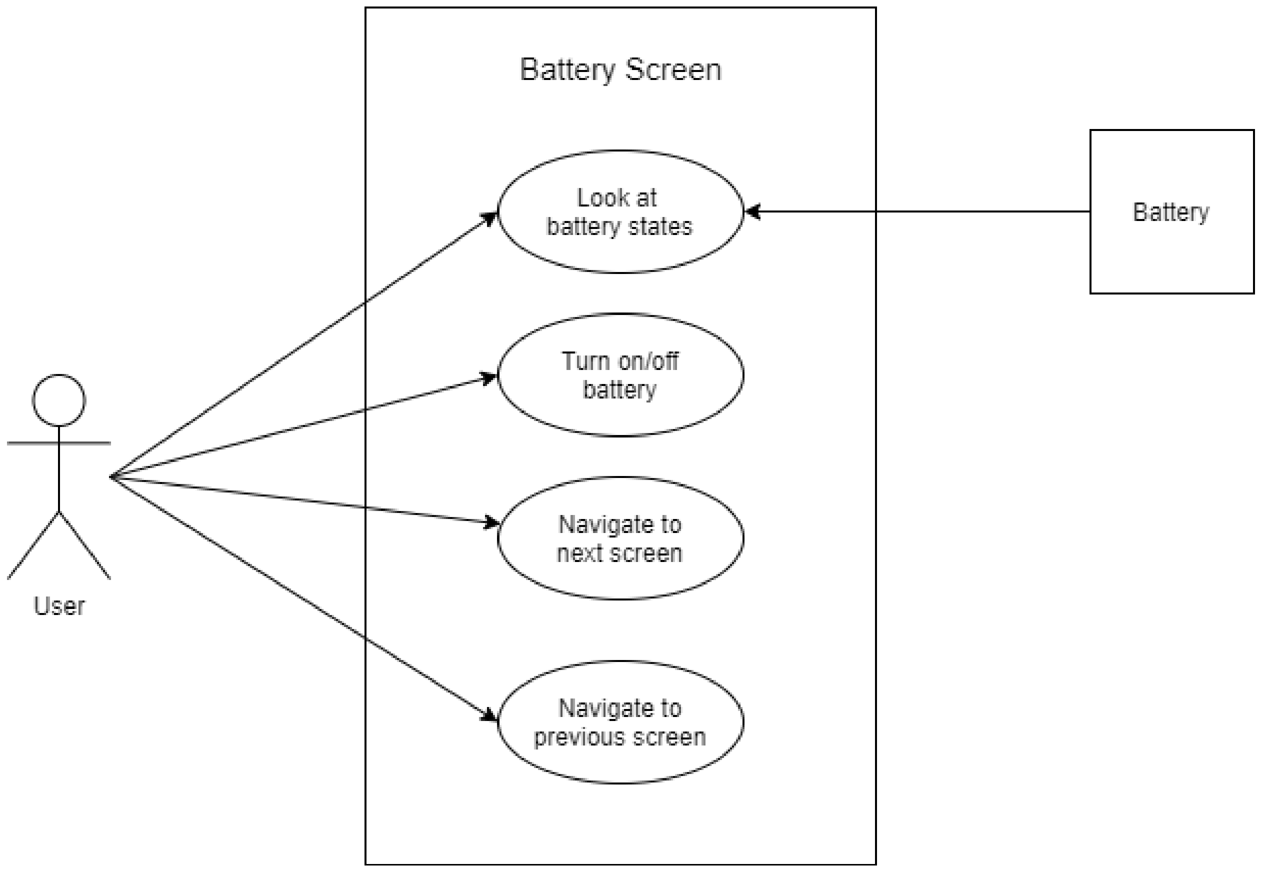


Figure 15. Use Case Diagram for Battery Screen

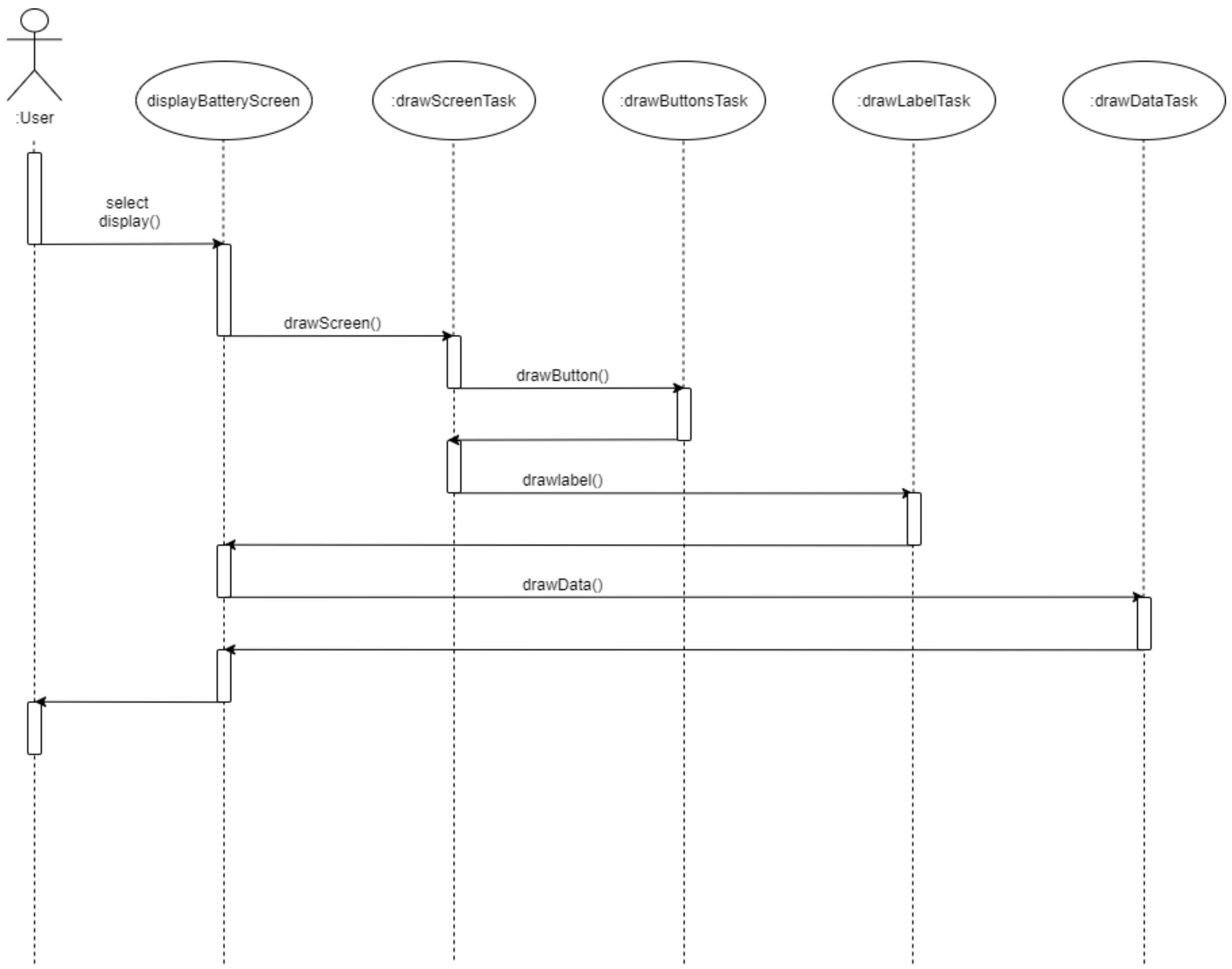


Figure 16. Sequence Diagram for Battery Screen

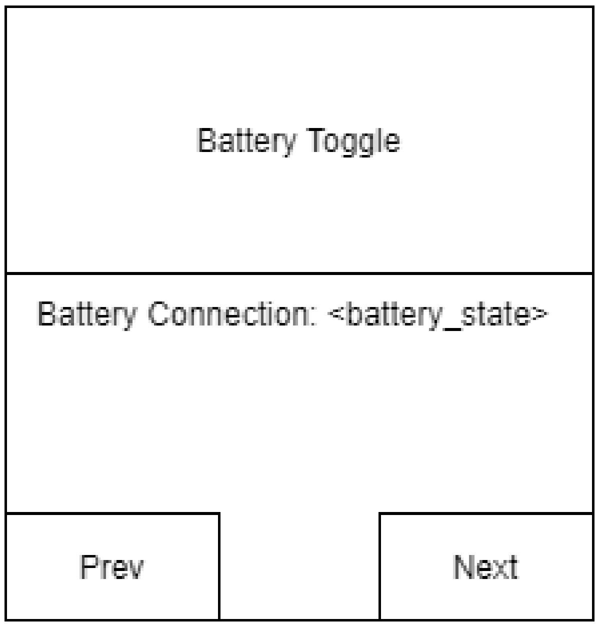


Figure 17. Front panel Design for Battery Screen

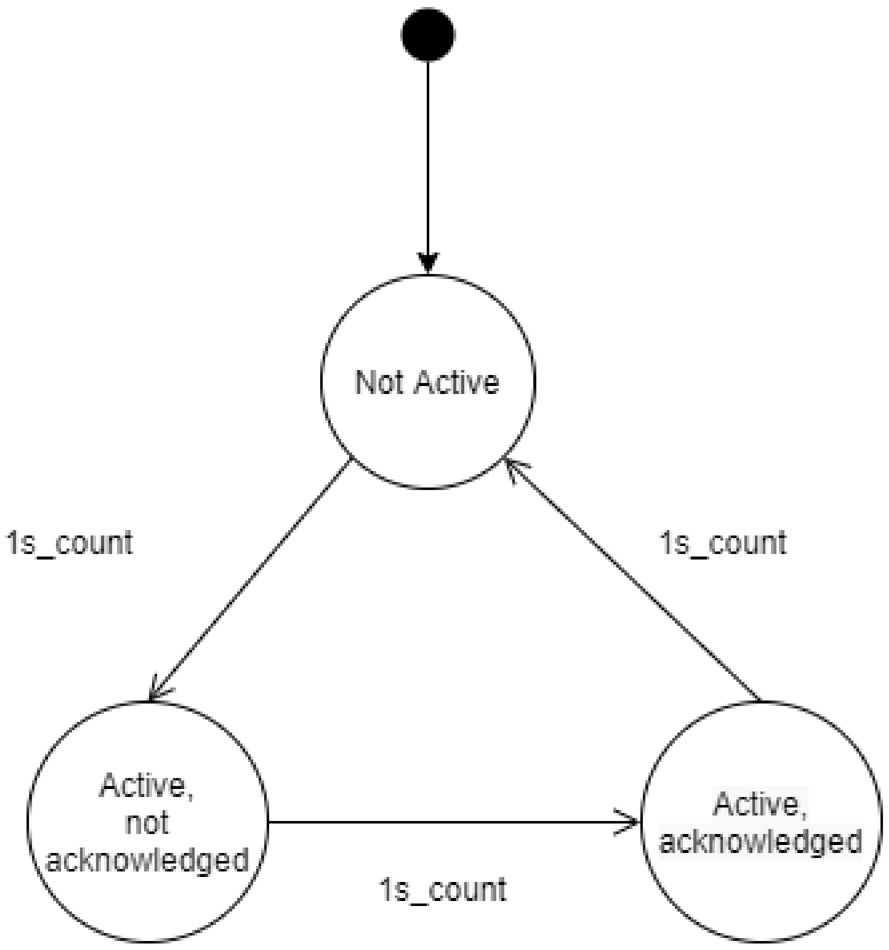


Figure 18. State Diagram for HVIL Alarm

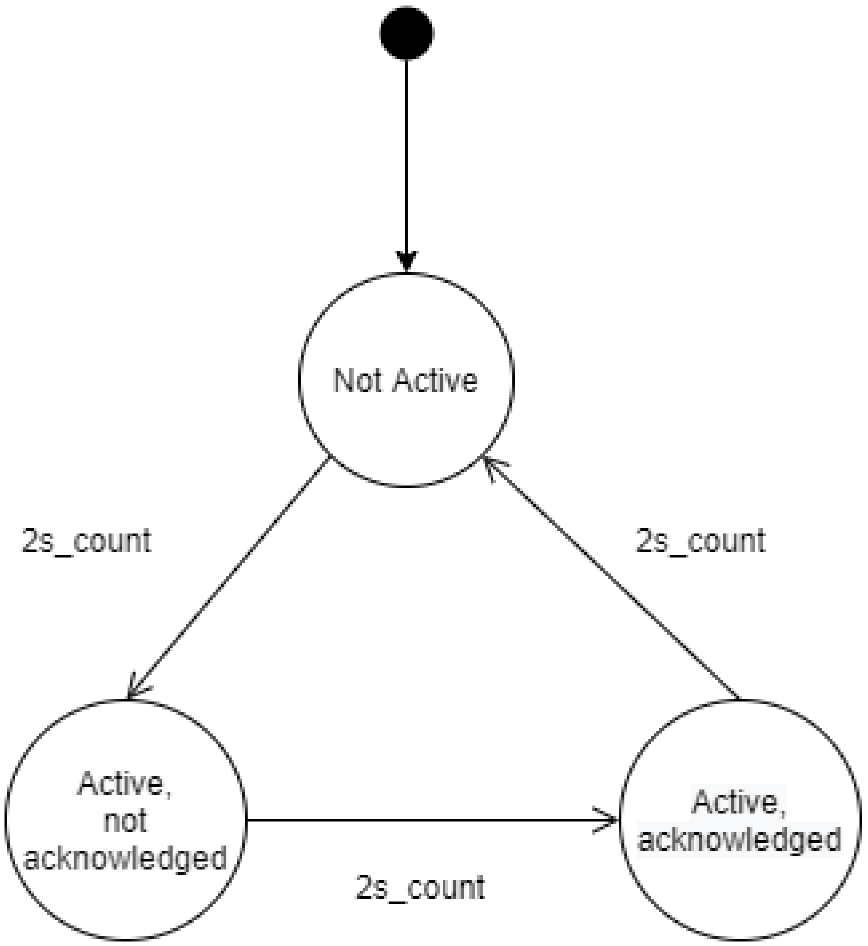


Figure 19. State Diagram for Overcurrent Alarm

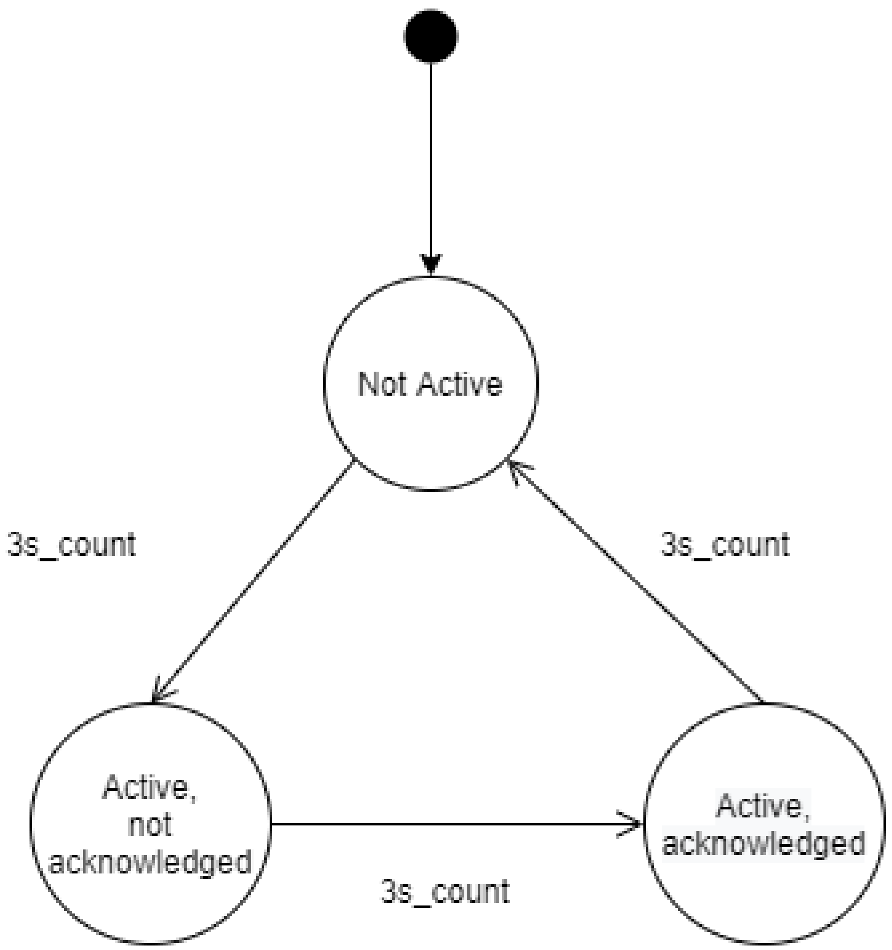


Figure 20. State Diagram for High Voltage out of Range Alarm

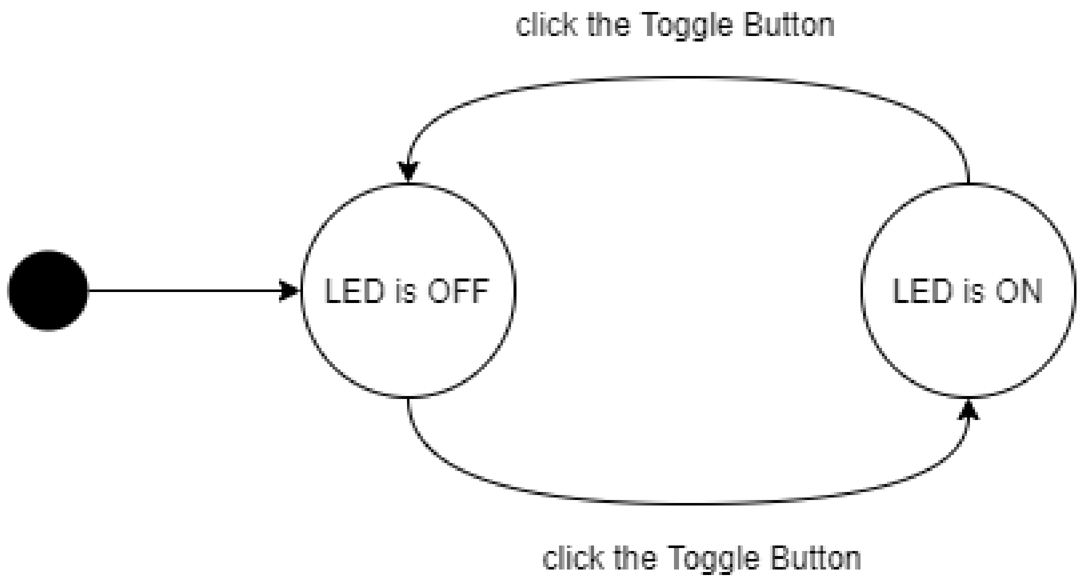


Figure 21. State Diagram for Contactor

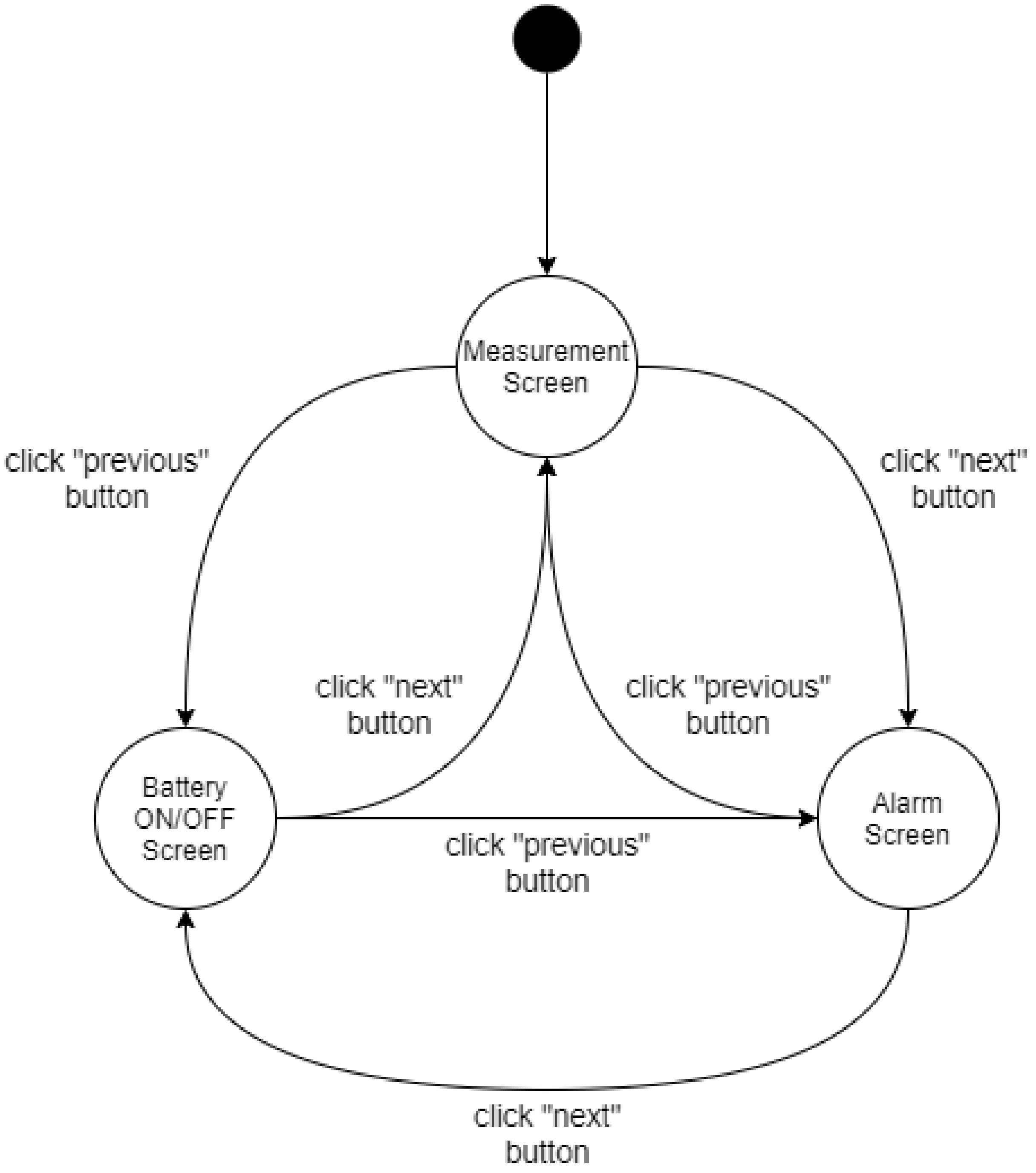


Figure 22. State Diagram for Touch Screen Display