22COA202 Coursework

F211372

Semester 2

1 FSMs

The finite state machine (FSM) at the core of my implementation is designed to manage the different phases of operation and user interactions within the smart device control program.

Here are the key states and the transitions between them:

1. **Beginning Phase:** This is the initial state when the program starts. The program then moves to the Synchronisation phase.
2. **Synchronisation Phase:** This phase is triggered by the input 'X' from the serial monitor. After synchronisation, the state machine transitions to the Main phase.
3. **Main Phase**: This is the standard display state where devices can be added, removed, and their power and states updated. It serves as a hub for the following transitions:
   - **To Display ON Devices:** Triggered by a press and release of the right button. If the right button is pressed again, it reverts to the Main phase.
   - **To Display OFF Devices:** Triggered by a press and release of the left button. If the left button is pressed again, it reverts to the Main phase.
   - **To Display ID and Free RAM:** Triggered by a long press of the select button for more than a second. Once the select button is released, it reverts to the prior display state (Main, Display ON Devices, or Display OFF Devices).
4. **Display ON Devices:** This state shows only the devices that are currently ON. It can transition back to the Main Phase or to the Display ID and Free RAM state.
5. **Display OFF Devices:** This state shows only the devices that are currently OFF. It can transition back to the Main Phase or to the Display ID and Free RAM state.
6. **Display ID and Free RAM:** In this state, the program displays the student ID and the amount of free RAM. Upon release of the select button, it returns to the previous state.
7. **Quit Program:** This is the final state, where the program stops executing.

The up and down buttons are used to navigate through the devices in alphabetical order, regardless of the current state (except Beginning and Quit Program states).

Pressing the reset button from any phase takes the Finite State Machine back to the Synchronisation phase.

This Finite State Machine structure allows a clear and efficient control of the device display and interactions, facilitating smooth user experiences.

Drawn below is a diagram of my finite state machine and the guards (Pseudo code) are used to describe the transition from each state.

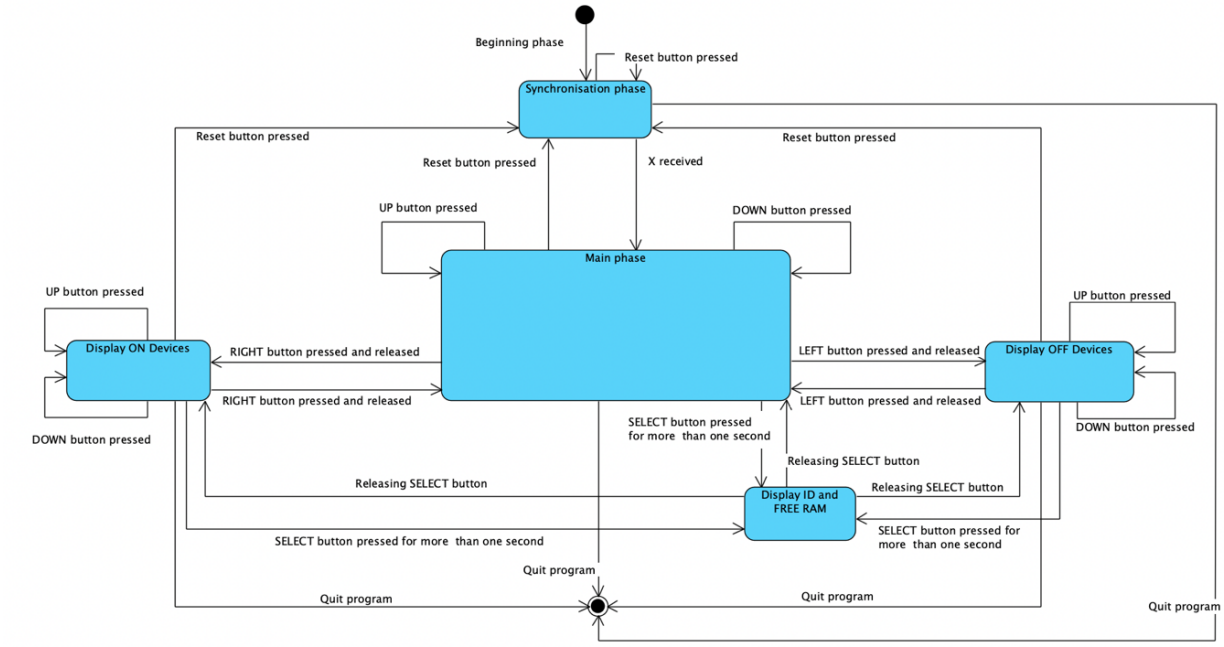

Figure 1:Synchronisation Phase

Figure 2:Display ID and FREE RAM Phase

Figure 3:Main Phase

Figure 4:Display OFF Devices Phase

Figure 5:Display ON Devices Phase

2 Data structures

```
// A struct to hold smart device data.
struct SmartDevice {
  String id;
  char type; // S,O,L,T,C
  String location;
  String state; // all devies, on or off
  int power; // for speaker and light
  int temperature;  // for thermostats
};
```

The primary data structure being used in the coursework is the SmartDevice struct. This structure has been designed to encapsulate all the necessary information about a smart device.

id: A String representing the unique ID of the device. (3 letters)

type: A char representing the device type (S, O, L, T, C).

location: A String representing the location of the device (1-15 characters long ranging from A-B, 0-9).

state: A String representing the state of the device (on or off).

power: An int representing the power level of the speaker and light devices (0-100%)

temperature: An int representing the temperature level for thermostat devices (9-32 °C)

devices array: This is an array of SmartDevice structs, which holds the data for all the smart devices. It is an instance of the SmartDevice struct.

An instance of the SmartDevice struct is created for each smart device in the system. All instances are stored in the devices array, which serves as the primary data store for device information. deviceCount variable keeps track of the number of devices currently stored in this array.

Note:

In the process of optimizing my Arduino code, I had to manage my resources carefully, particularly the available SRAM. The amount of free RAM before adding any devices was measured at 722 bytes. However, each device added to the system consumed a certain amount of this available memory.

I conducted tests to determine the maximum number of devices I could handle without causing the code to break due to insufficient memory. As each device was

added, I carefully tracked the remaining free SRAM. After extensive testing, I determined that the system could effectively manage up to 20 devices. At this point, the free SRAM was reduced to 443 bytes.

Attempting to add more than 20 devices led to issues with the program execution due to the limited available memory. Therefore, to ensure the stable operation of my code, I decided to set a limit of 20 devices. It's an effective balance between functionality and resource management given the constraints of the Arduino microcontroller.

Listed below are the functions that help update the global struct smartDevice.

addDevice (String data): This function takes a formatted string as input, extracts the device information, creates a new SmartDevice instance, and adds it to the devices array.

updateDeviceState (String data): This function accepts a string representing the updated information of a device. It parses this string, finds the corresponding device in the devices array using the device ID, and updates the device in the appropriate SmartDevice instance.

updateDevicePower (String data): This function accepts a string representing the updated information of a device. It parses this string, finds the corresponding device in the devices array using the device ID, and updates the device's power in the appropriate SmartDevice instance.

removeDevice (String data): This function accepts a string representing the ID of a device to be removed. It locates the device in the devices array and removes it, shifting all subsequent devices in the array to fill the gap. The deviceCount global variable is also decremented to reflect the removal of the device.

sortDevices (): This function sorts the devices array based on the device ID. It is used after a device is added or removed to ensure that the devices are always displayed in order.

updateDisplay (): This function updates the LCD display with information about the device at the currentDisplayIndex in the devices array. It handles the display of device ID, type, location, state, power or temperature based on the device type.

processSerialData (String data): This function handles the updating of the devices array during the main phase of the program. It calls addDevice (String data), updateDeviceState (String data), updateDevicePower (String data), and removeDevice (String data) functions based on the command received.

3 Debugging

I have a bunch of debug statements in my code that print to the serial monitor. They were used to help me build my code step by step and notice errors that were not detected by the Arduino IDE. All of them have been commented out as instructed. However, every debug statement starts with the “DEBUG:” phrase, to help differentiate between debug statements and error statements.

I have debug statements in the following areas–

1. Where I process the serial data, to denote which switch case has been triggered and what the raw serial input is, these have been commented out.
2. After adding a device, the sortDevice () function is triggered, and then as DEBUG statements, the sorted devices are printed to the serial monitor.
3. When an error is detected, I have added debug statements to figure out what the error is (e.g., invalid device type, invalid device location, etc.)

All my DEBUG statements are managed by C macros to help reduce the SRAM usage when uncommented.

4 Reflection

Reflecting on the development and functionality of my FSM code, it's clear that several key features have been successfully implemented. The device handling, location scrolling, and user interactions via the LCD buttons work as intended, providing a robust and dynamic interface for monitoring and controlling an array of smart devices.

However, there are a few areas where my code could be improved. One such area is the HCI extension. It's functionality, as it stands, isn't as seamless as I would like it to be. Occasionally, it encounters issues when there is only one device switched off and the left button is used in an attempt to display it. In these circumstances, the device doesn't appear immediately. Rather, the up button needs to be pressed after the left button to display the sole "OFF" device. However, when multiple devices are switched off, the left button operates smoothly, and the devices are displayed without any issues.

In an ideal world, I would rectify this inconsistency in the interface so that the sole "OFF" device is displayed immediately upon the left button being pressed, mirroring the behaviour when multiple devices are in the "OFF" state. On the other hand, the right button, used for displaying "ON" devices, operates flawlessly and does not require any changes.

Next my code cannot handle simultaneous select button press and serial input. Even if the user passes data to the serial monitor when the select button is pressed, it updates my display only after the button has been released. I tried fixing my code to handle it simultaneously, but it would break.

Lastly, in terms of scalability, the code currently operates under the assumption that the number of devices is relatively small. If the number of devices grows significantly, it might be beneficial to implement a search or filter functionality that allows the user to quickly and conveniently navigate through the devices.

All things considered, while the code could benefit from these improvements, it already provides a solid foundation for a smart device monitoring and control system. The areas identified for enhancement offer valuable directions for the future development and refinement of the system. Considering the amount of time and effort I put into building my dummy smart home monitor, I am excited for you to test it out. Over to you :)

5 UDCHARS

This extension defines two customer characters for the LCD display, an up arrow and a down arrow. The characters are represented as 8-byte arrows, where each byte corresponds to a row in the 5 x 8-pixel character matric. The bits in each byte indicate the state of the pixels in the row, hence 1 for on and 0 for off.

I have called these arrows, when I update the display after adding/ removing a device and when viewing the only ONdevices/ only OFF devices. The up and down arrows help the user navigate through the sorted Devices array.

When it's only one device in my devices array, no arrows are displayed. But when there is more than one device and when the display reaches the first device, only the down arrow is displayed denoting that there are no more devices above, similarly when the display reaches the last device, only the up arrow is displayed denoting the last device. Every other device's display has both the up and down arrows.
Displaying only the up and down arrows is done by using a counter to check if the first/ last device has been reached.

Example:

```
// Display the up arrow only after the first device display.
if (currentDisplayIndex > 0) {
  // Set the cursor to the position where the up arrow should be displayed
  lcd.setCursor(0, 0);
  lcd.write(byte(0)); // Up arrow (custom character 0)
}
```

```
// Display the down arrow for all devices except the last device
if (currentDisplayIndex < deviceCount - 1) {
  // Set the cursor to the position where the down arrow should be displayed
  lcd.setCursor(0, 1);
  lcd.write(byte(1)); // Down arrow (custom character 1)
}
```

Figure 6: Only one device added.

Figure 7: First device, hence down arrow

Figure 8: Middle devices, hence both arrows

Figure 9: Last device: hence up arrow

6 FREERAM

To calculate the free ram in bytes and display when the select button is pressed for more than a second, I created two functions namely, int freeListSize () and int freeMemory () to return integers.

The function freeListSize () calculates the size of the free list, which represents the total amount of free memory available in the heap that has been previously allocated and freed.

Returns: The total size of the free list in bytes.

The function free Memory () calculates the available free memory in bytes by determining the difference between the stack pointer and the heap end (__brkval) or heap start (__heap_start) if __brkval is 0.

Returns: The amount of free memory available in bytes.

The free ram can be displayed when the select button is pressed in every state, after the synchronisation phase.

```
/*
 * External heap start and break value pointers.
 * These pointers represent the start of the heap and the end of the currently used
 * heap memory respectively. They are used to calculate the available free memory.
 */
extern unsigned int __heap_start;
extern void *__brkval;
/*
 * The free list structure as maintained by the
 * avr-libc memory allocation routines.
 */
struct __freelist
{
  size_t sz;
  struct __freelist *nx;
};
```

Figure 10: Display SRAM and ID, when select button is pressed.

7 HCI

I created handlers for the left button press and right button press namely, void leftButtonHandler () and rightButtonHandler ().

When the left button is pressed all the devices that are off are displayed. When the right button is pressed all the devices that are on are displayed.

To display the devices, I created two separate functions namely,

void displayOnDevices (): Displays information about devices that are currently ON. The devices array is searched up by the state and if the device state is ON it is displayed with all the details. Below is a code snippet explaining it.

```
  // Counters for the number of devices that are currently "ON".
  int onDevicesCount = 0;
  // Iterate through the device array to find the number of on devices
  for (int i = 0; i < deviceCount; i++) {
    if (devices[i].state == "ON") {
      onDevicesCount++;
    }
  }
```

If no devices are ON, then “Nothing’s on” is displayed on the lcd.

else, the ON devices are displayed, below are variables to help display them.

```
if (onDevicesCount == 0) {
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("NOTHING'S ON");
    lcd.setBacklight(WHITE);
    } else {
      // Indices of devices in the devices array, used to keep track of the currently
displayed device and to find the previous and next devices in the list.
      int onDeviceIndex = 0;
      int prevOnDeviceIndex = -1;
      int nextOnDeviceIndex = -1;
```

This function is called when the right button is pressed. and the backlight is set to green.

void displayOffDevices (): Displays information about devices that are currently OFF. The devices array is searched up by the state and if the device state is OFF it is displayed with all the details.

```
  // Counters for the number of devices that are currently "OFF".
  int offDevicesCount = 0;
  // Iterate through the device array to find the number of off devices
  for (int i = 0; i < deviceCount; i++) {
    if (devices[i].state == "OFF") {
      offDevicesCount++;
    }
  }
```

If no devices are OFF, then “Nothing’s off” is displayed on the lcd.

else, the OFF devices are displayed, below are variables to help display them.

```
  if (offDevicesCount == 0) {
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("NOTHING'S OFF");
    lcd.setBacklight(WHITE);
  } else {
    // Indices of devices in the devices array, used to keep track of the currently
displayed device and to find the previous and next devices in the list.
    int offDeviceIndex = 0;
    int prevOffDeviceIndex = -1;
    int nextOffDeviceIndex = -1;
```

This function is called when the left button is pressed, and the backlight is set to yellow.

Since the above two displays are a subset of the main display, the up and down arrows are displayed accordingly.

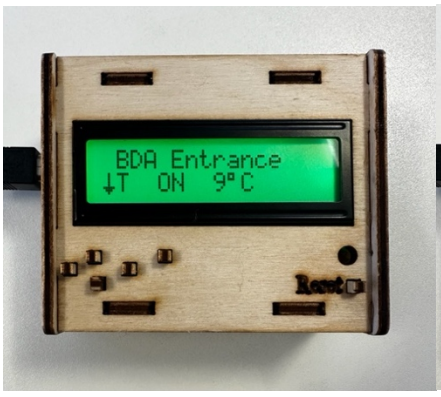

Figure 11:Main Phase

Figure 12:Display OFF Devices Phase

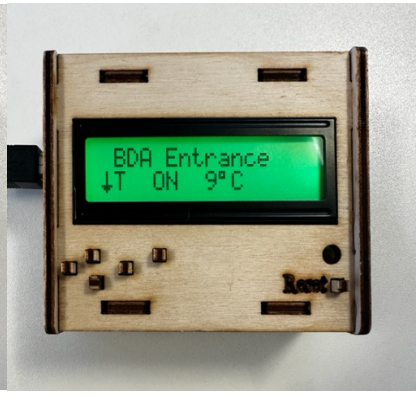

Figure 13:Display ON Devices Phase

9 SCROLL

The scrollLocation (), scrollOnDevicesLocation (), scrollOnDevicesLocation () function is responsible for providing a scrolling display of a device's location on the LCD screen, in cases where the location string is longer than the maximum allowed length that can be displayed on the LCD.

The function begins by checking if there are any devices to display. If deviceCount > 0, this means we have at least one device to show. If there are no devices, the function won't execute further.

Next, it calculates the maxLocationLength, which is the maximum number of characters that can be displayed on the LCD for the device's location. It's determined by the minimum of either 11 or the difference between 16 and the length of the current device's ID plus 2. The '16' represents the maximum number of characters an LCD line can display, and '2' is for the space between the device ID and the location.

Then, it checks if the length of the location string of the current device (devices[currentDisplayIndex]. location) is greater than maxLocationLength. If it's not, the function will not proceed further.

If the location string is longer than maxLocationLength, it checks if the scrollPosition (which keeps track of the starting index of the substring currently displayed) has reached the end of the location string. If it has, scrollPosition is reset to 0, which means the scroll display restarts from the beginning of the location string.

It then extracts a substring from the location string, starting at scrollPosition and ending at scrollPosition + maxLocationLength. This substring, displayText, is what will be shown on the LCD.

The cursor on the LCD is then set to the appropriate position, and displayText is printed on the LCD.

In your main loop () function, this scrollLocation (), scrollOnDevicesLocation (), scrollOnDevicesLocation () function is called every time the difference between currentMillis and previousMillis is greater than or equal to scrollInterval. This interval ensures that the location text scrolls at a regular pace of 2 characters per second.

scrollLocation () is also indirectly involved in the updateDisplay () function. In updateDisplay(), when a device's location is too long to fit on the LCD, a substring of the location is displayed instead. This substring starts at scrollPosition and has a length of maxLocationLength. As scrollPosition is updated in scrollLocation (), this ensures that the location text displayed in updateDisplay () is the current section to be displayed as it scrolls.

(Since the displayOnDevices and displayOffDevices are a subset of the main display,

The scrollLocation is created individually for them, namely: scrollOnDevicesLocation and scrollOffDevicesLocation.)

Figure 14: Initial display of location string. The input is "A-PZW-S-OUTSIDEGARDENSHED";
Since the location string is longer that 11 characters, it first trims the location to 15 characters, and displays on a scroll.

Figure 15: Scrolls the characters of the location string at 2 characters per second until the end, and then restores location to the beginning

Figure 16: If location string is no longer than 11 characters, no scroll effect.