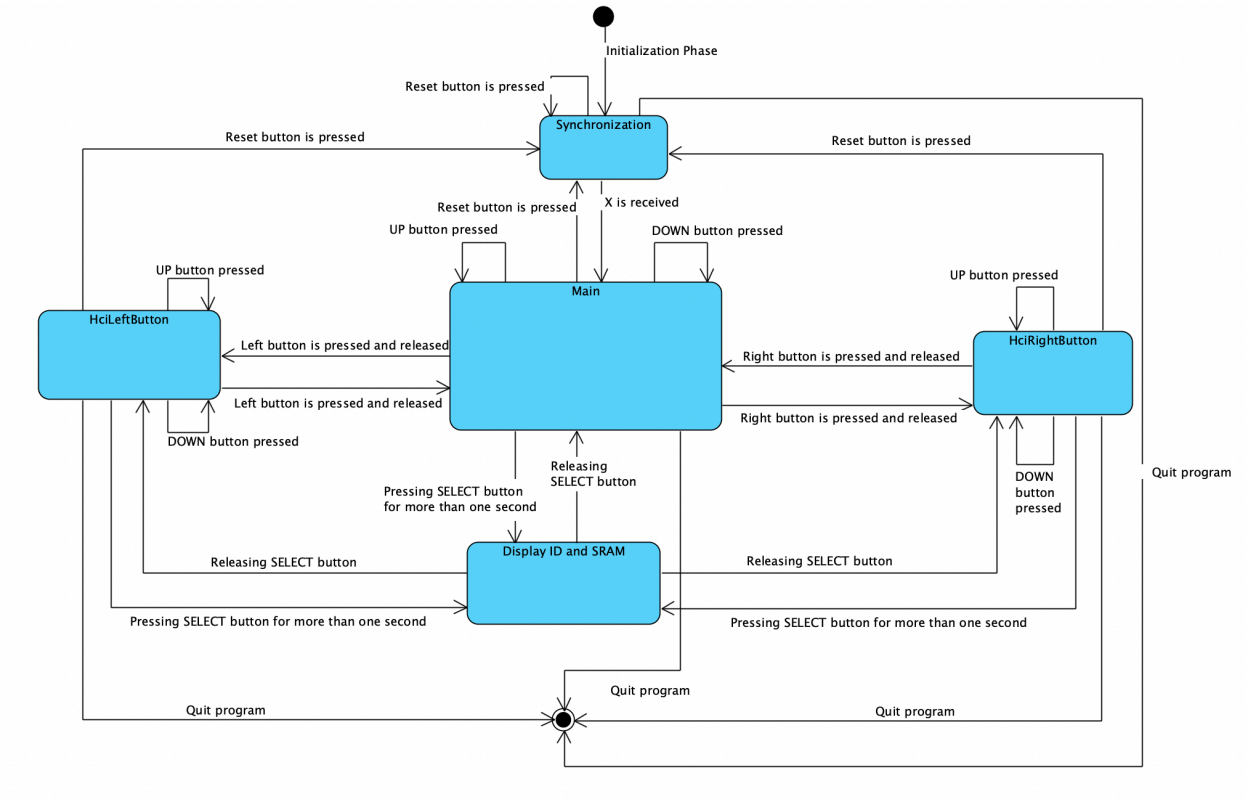
1 FSMs



The centerpiece of the implementation is a finite state machine (FSM), which guides the transition between various states based on user inputs. The FSM includes several states: Initialization, Synchronization, Main Phase, HCI Right Button, and HCI Left Button. Each of these states has specific transitions that dictate the flow of the program.

- **Initialization State**: This is the starting state of the program.
- **Synchronisation State**: The program transitions from Initialization to this state when the 'X' button is pressed.
- **Main Phase State**: This is the central state of the FSM. Users can scroll through a list using the 'Up' and 'Down' buttons.
    - Transition to **HCI Right Button State**: The program transitions to this state when the right button is pressed in the Main Phase. Here, only 'ON' devices are displayed, and the 'Up' and 'Down' buttons are used to scroll through them.
    - Transition to **HCI Left Button State**: The program transitions to this state when the left button is pressed in the Main Phase. Here, only 'OFF' devices are displayed, and the 'Up' and 'Down' buttons are used to scroll through them.

- **HCI Right Button State & HCI Left Button State**:

- Holding down the 'Select' button allows the user to view the student ID and SRAM.
- Releasing the 'Select' button returns the user to the previous state.
- The user can quit the program at any time.
- Pressing the 'Reset' button brings the FSM back to the Synchronization state.

2 Data structures

Data Structures

1. "**Device**": A struct that represents a single device. It has the following fields:
   - "**id**": A String that represents the unique identifier of a device.
   - "**type**": A character that represents the type of a device, which can be a Speaker ('S'), Light ('L'), Thermostat ('T'), Camera ('C') or Socket ('O').
   - "**location**": A String that represents the location of a device.
   - "state": A String that represents the state of a device, which can be either "ON" or "OFF".
   - "**outputPower**": An integer that represents the output power of a device.
   - "**temp**": An integer that represents the temperature of a device, used when the device is a thermostat.

2. "**State**": An Enum that represents the current state of the application. It can be in synchronisation state, main state, or handling HCI button states.

Global Variables

1. "**lcd**": An instance of the Adafruit_RGBLCDShield class, which is used to control an LCD display.
2. "**devices**": An array that stores the list of devices managed by the application.
3. "**currentState**": A variable that keeps track of the current state of the application.

Functions

The code includes several functions to manage the devices:

- "**handleAddDevice**()": Adds a new device to the list.
- "**handleRemoveDevice()**": Removes a device from the list.
- "**handleUpdateState()**": Updates the state of a device.
- "**handleUpdateOutputPower()**": Updates the output power of a speaker or light, or the temperature of a thermostat.
- "**sortDevices()**": Sorts the devices by their ID alphabetically.
- "**displayLocation()**", "**scrolling()**": These functions handle the display of device location on the LCD.

The code also includes functions to calculate the available memory ("**freeMemory**()"), and functions to handle the different states of the application, As an aside, the code includes two custom character definitions for an up arrow and a down arrow, which could be used in the UI for scrolling or selection purposes. The code is a solid example of a device manager for a small microcontroller like the Arduino, showcasing how it's possible to interact with various types of devices, manage their state, and display information on a small LCD screen.

4 Reflection

Upon reflecting on the code, several areas of improvement stand out. These areas, which did not perform optimally, serve as a roadmap for future enhancements.

1. The first major concern lies in the realm of error handling. The existing structure lacks a resilient exception management system, thereby exposing the program to the risk of unhandled exceptions and potential crashes. A more comprehensive and robust error handling approach would mitigate these risks. By offering clear, understandable feedback to the user, such a system would more gracefully navigate unforeseen errors.
2. A specific functional issue was observed in the Human-Computer Interaction (HCI) states, specifically the left and right arrows. Unlike in the main phase, where an arrow disappears to indicate the end of a list, this feature failed to work in the mentioned states. To remedy this, I propose developing a function dedicated to controlling the arrows' activity. This function can then be called in each state as needed.
3. A related improvement would be the inclusion of the select button in the switch case. Assigning a distinct state to each button would enhance the code's operational fluidity.
4. Lastly, the code's documentation could be made more comprehensive. While some comments and documentation exist, they fall short of providing a full understanding of the code's purpose and flow. Striving for clarity and completeness in comments, particularly for methods, classes, and critical variables, will significantly improve code readability and maintainability.

In conclusion, while the code fulfills its intended function, it reveals several opportunities for enhancement. Through targeted improvements, it can be made more robust, efficient, user-friendly, and maintainable.

5 UDCHARS

To integrate the UDCHARS extension, custom characters were defined and stored as byte arrays, each signifying a 5x8 pixel pattern for LCD display.

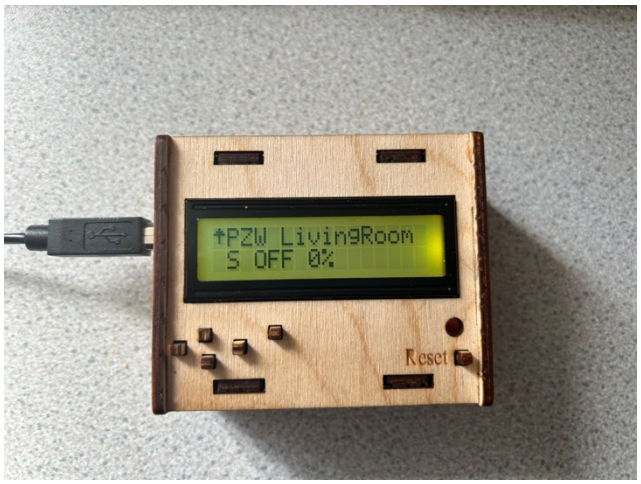
```
// ------------------
// UDCHARS EXTENSION
// ------------------

byte upArrow[8] = {
  0b00100,
  0b01110,
  0b11111,
  0b00100,
  0b00100,
  0b00100,
  0b00000,
  0b00000
};

byte downArrow[8] = {
  0b00000,
  0b00000,
  0b00100,
  0b00100,
  0b00100,
  0b11111,
  0b01110,
  0b00100
};
```

These characters were added to the LCD through the createChar() function in the setup() method, linking each character to an LCD memory index. For improved user interaction, conditional logic was used to control arrow displays based on the user's device list position. When at the last device, only the up arrow appears, indicating no more devices below. When at the first device, only the down arrow shows, signifying more devices below.

This implementation enriches the user interface, enabling intuitive device list navigation on the LCD.



6 FREERAM

To incorporate the free memory feature, I modified the code by introducing new variables and functions. One key function is `**freeMemory()**`, an integer function designed to calculate the amount of available memory at any given point. This function is invoked within another function, `**handleSelectButton()`,** which is tasked with displaying the student ID and free memory. `**handleSelectButton()**` is then triggered within the `**updateStateFromButtons()**` function, which is initially called at the beginning of the loop before the Finite State Machine (FSM). This arrangement ensures the free memory can be viewed in all states except for the synchronization phase. The `**updateStateFromButtons()**` function is integral, comprising functions that dictate how the free memory should be displayed. By integrating the `**freeMemory()**` function within this broader function, we ensure that the information about available memory is accurately and appropriately displayed to the user.

```
// Function to handle the pressing of the select button on the LCD screen
void handleSelectButton() {
  Serial.println("Entering handleSelectButton function");
  if (!displayingStudentID) {
    displayingStudentID = true;
    delay(1000);
    lcd.clear();
    lcd.setBacklight(5);
    lcd.setCursor(0, 0);
    lcd.print("F213826");
    lcd.setCursor(0, 1);
    lcd.print("SRAM LEFT:");
    lcd.print(freeMemory());
  }
}
```

```
}
// Function to update the state based on the button presses on the LCD screen
void updateStateFromButtons() {
  static unsigned long lastButtonPressTime = 0;
  unsigned long currentTime = millis();
  static bool rightButtonPreviouslyReleased = true;
  static bool leftButtonPreviouslyReleased = true;

  if (currentTime - lastButtonPressTime < 50) {
    // Debounce delay - ignore button presses within 50 ms of the last press
    return;
  }

  uint8_t buttons = lcd.readButtons();

  if (buttons & BUTTON_SELECT) {
    handleSelectButton();
  } else if (displayingStudentID) {
    displayingStudentID = false;
    displayNeedsUpdate = true;

  }
```

7 HCI

The HCI extension was incorporated to provide an interactive and intuitive interface for the user. This was achieved by creating unique functions to handle the input from different buttons on the LCD screen. Functions such as "**handleUpButton()**", "**handleDownButton()**", "**handleRightButton()**", "**handleLeftButton()**", "**handleHciRightButton()**", and "**handleHciLeftButton()**" were implemented. Each of these functions manipulates a global "**currentDevice**" variable, which tracks the current device index selected by the user. If the Up or Down button is pressed, "**currentDevice**" is respectively decremented or incremented, considering boundary conditions to prevent going out of the device list. Similarly, "**handleRightButton()**" and "**handleLeftButton()**" functions change the state of the Finite State Machine (FSM) to "**HciRightButton**" and "**HciLeftButton**" respectively, prompting a different display on the LCD screen.

The FSM was updated to include new states "**HciRightButton**" and "**HciLeftButton**". In these states, the LCD displays the list of devices that are either on or off, respectively. This is controlled by the "**displayOnDevices()**" and "**displayOffDevices()**" functions, which filter and display the devices based on their status. The "**updateStateFromButtons()**" function, called at the start of each loop, allows for real-time updating of the system state based on user inputs. It debounces the buttons to prevent false readings due to button bounce, and calls appropriate handler functions based on the buttons pressed and the current state.Thus, the HCI extension provides a user-friendly and interactive interface, with custom characters and device lists that update based on the state of the system and user input.

8 SCROLL

The Scroll Extension feature provides a seamless user interface for device locations that exceed the LCD display limit of 11 characters. This is achieved by dynamically scrolling the display text, allowing users to view the entire device location. The "**scrolling()**" function implements this feature. It uses two static variables: `count`, which tracks the current scrolling position, and "**lastChecked**", which records the last time the display was updated. It calculates the current time with "**millis()**", and if at least 500ms has passed since the last check, the function updates the display. It extracts a substring of the location starting from the "**count**"th character and appends additional characters from the start of the location string if necessary. The scrolling effect is achieved by incrementing "**count**" at each update, creating a circular scrolling effect. The "**displayLocation()**", "**displayLocationHciLeft()**", and "**displayLocationHciRight()**" functions are modified to include the scrolling feature. They check if the length of the device location exceeds 11 characters, and if so, call the "**scrolling()**" function. The "**displayDeviceList()**" function has been updated to include the display of the device's location. It calls "**displayLocation()**", which then decides whether to use the scrolling feature or not based on the length of the location string. These modifications have been integrated into the FSM. For instance, states "**HciRightButton**" and "**HciLeftButton**" call "**displayLocationHciRight()**" and "**displayLocationHciLeft()**", respectively, thereby ensuring a consistent user interface. The Scroll Extension, therefore, enhances the user experience by ensuring all relevant device information is accessible, regardless of character limits.