22COA202 Coursework

F223882

Semester 2 / SAP

1 FSMs

The main finite state machine is placed within the void loop function. The state of the system is determined and tracked by a global variable declared at the start of the code. Hence, every time the void loop function is looped, this variable is checked and will execute the section for the correct state only, and nothing else.

This FSM has three states: Synchronization, Reading, and Main Phase.

Synchronization gives purple background and Q's every second.

Synchronization passes into Reading as soon as an 'X' is received from the serial monitor from the user.

Reading retrieves devices stored in EEPROM and puts them into a sorted array of devices.

Reading passes into Main Phase once all devices have been loaded into the array.

Main Phase prepares for user inputs, including buttons and serial monitor inputs (A, S, P, and R).

The secondary finite state machine is placed within the Reading state and the Main Phase state of the main finite state machine. It is also kept track of by a global variable declared at the start of the code. This finite state machine is used primarily for the HCI extension task, determining which devices, based on their own ON/OFF states, should be loaded into the array instead of every device.

This secondary FSM has three states as well: Normal, On, and Off.

Normal adds all devices read from EEPROM into the device array.

Normal passes into On when BUTTON_RIGHT is pressed on the Arduino Unit.

On adds only devices with state of ON into the device array from the EEPROM.

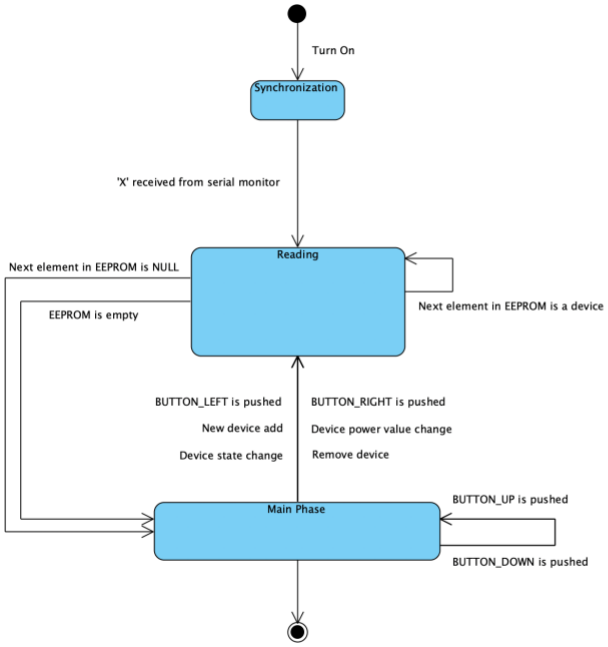
On returns to Normal state when either BUTTON_LEFT or BUTTON_RIGHT is pressed.

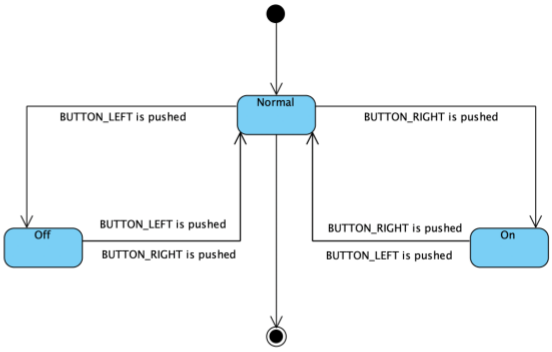
Normal passes into Off when BUTTON_LEFT is pressed on the Arduino Unit.

Off adds only devices with the state of OFF into the device array from the EEPROM.

Off returns to Normal state when either BUTTON_LEFT or BUTTON_RIGHT is pressed.

Main Finite State Machine





Secondary Finite State Machine

2 Data structures

I had initially decided to use classes to represent each device type, but after a series of attempts to make it work, I decided it was wiser to use structures instead.

I tried my best to stay away from creating string data and use character arrays as much as possible as I believe they require less space. Character arrays are trickier to use, especially being accustomed to a more powerful language such as Python. One key feature that had slipped my mind early on was the extra character needed to label the end of the array.

I had attempted to use an enum structure to contain the names of the states of the finite state machine, but then found it a bit redundant as I can represent them using whole numbers. I suppose when there is only a small number of states, using numbers would still work. However, if this system were to grow and more states are to be added, then an enum structure may be necessary.

Two integers were declared globally to track the main state of the programme from Synchronisation to Reading, to Main Phase, and the substate inside these states with ON and OFF.

Two integers were declared globally to tracks the position of the cursor to print the next Q while looping through Synchronisation.

An integer was declared globally to tracks the index of the device in the sorted array on display.

An integer was declared globally to tracks the index of the last device in the sorted array.

An integer was declared globally to tracks the index of the last occupied byte in EEPROM.

Two character variables were declared globally as arrow variables that are changed according to the situation, to be displayed.

A Float was declared globally as a testing variable to tell if the next byte in EEPROM is available or not.

4 Reflection

I must be honest and say that I deeply dislike programming in this project. I have gotten used to more powerful languages such as Python and Java, with less constraints and more abstraction, and doing C++ for Arduino has been painful. Figuring out deeper level concepts such as how structures would be stored in memory; that a variable value isn't actually referenced using the variable name; that character arrays require an extra bit to signify its end; they were all really challenging to figure out.

Which made it all the more satisfying when it worked out in the end.

I didn't make use of any debugging tools or programmes, but I'm confident that the code I wrote has succeeded in delivering the specification of this coursework.

I had trouble detecting a long press on the SELECT_BUTTON to show my student ID. This feature is still in my code, but it flickers every so often when pressed down. It can still read commands from the user in the serial monitor during this phase, but I couldn't get rid of the flicker. I believe I had to use pinMode() to manipulate the button but I couldn't figure out the pin number on the board that SELECT_BUTTON is attached at.

I believe I got lucky figuring out my main finite state machine as I had implemented it when I was still coming to grasps with it. If I had placed it in anywhere that wasn't the void loop function I would have probably wasted a lot more of my time.

All in all, everything was excruciatingly painful up until the point of success.

5 UDCHARS

There were no changes to the FSM during this extension.

I declared two patterned bytes as global variables and created them as custom characters in void setup function, called to them when I needed to use them in the lcdDisplay function.

This extension was simple and straightforward, once I understood that custom characters can be created to be contained in a byte.

6 FREERAM

There were no changes to the FSM during this extension.

This extension made use of a reusable function to return the SRAM the unit had left. If I'm being honest, I don't fully understand that function, but enough to know that it works a charm and return an integer.

This extension fits into the SELECT_BUTTON feature which I have expressed my dissatisfaction before, but I hadn't enough time to keep improving on that.

7 HCI

This extension introduced a secondary FSM into my main FSM.

At first, I believed I should just add additional states into my main FSM, but that created a lot of repeating code as these states would be nearing identical except for two lines of code.

Come to think of it, this secondary FSM is not really embedded inside of the main one, it's more intertwined. The secondary FSM would have three different states showing all states, On states, and Off states inside the Reading state of the first one, and then again in Main Phase. But the states in Reading and Main Phase are the same. Whether or not it would count in total as three different FSMs, I'm not sure.

8 EEPROM

This extension was the reason the Reading state came into existence for my main FSM.

Initially, my idea was to iterate through EEPROM using the eepromAddress counter, but this means that as users add devices out of order of their ID, my display list would also be out of order. My solution was to create an array of device at the start and populate it in the Reading state with devices from EEPROM.

This caused another problem in which the device array would require its own update alongside EEPROM every time a user makes a change to any device. My solution to that was to forgo updating the array every time a change occurs; to only update EEPROM and change the state of the main FSM back to Reading to populate the array again.

I understand that this is not the most efficient or optimal solution, but it was the one I compromised on under consideration of time and practicality. I understand that EEPROM has a limited number of puts and gets; but in the scope of this coursework, it's a problem I decided to overlook.

I had not attempted the subtask of designing a function to determine whether the values in EEPROM was made by me or left behind from someone else. I hadn't enough time, so nothing much to say about that.

9 SCROLL

I hadn't attempted this extension.

If I were to have had enough time, I believe I would put a second secondary FSM inside the module which displays the device on the LCD display.

Every time it is looped through, there would be a counter to keep track of how close the last letter that can be seen on the display is to the actual end of the character array holding the location of the device.

This would be checked again at the start of every loop and the substring of the character array would be moved down according to that counter. Once it reaches zero, meaning the end of the character array is show, the next iteration would return it to the start of the character array.