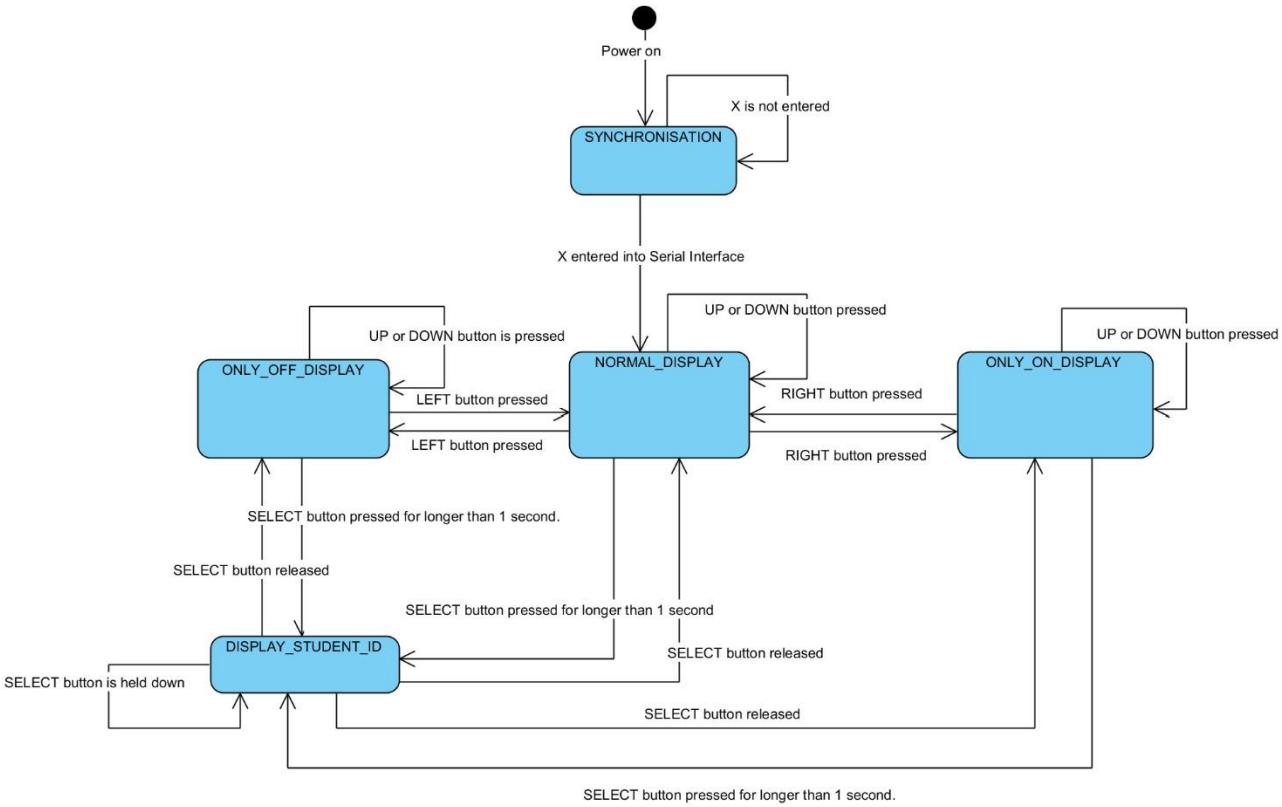
22COA202 Coursework

F229639

Semester 2

1 FSMs



The smart monitor starts in the SYNCHRONISATION state after being powered on. When in this state, and prints Q onto the serial monitor at one Q per second. It remains in this state until the user enters “x” into the serial interface; at this point, the state is changed to the NORMAL_DISPLAY state.

The NORMAL_DISPLAY state is the first state the project enters when it escapes the SYNCHRONISATION state. All smart devices, regardless of if they are on or off, can be displayed in this state. The up and down arrows allow for the display of different devices on the smart monitor but do not change the state of the smart monitor. Also, during this state, the smart device monitor still takes in inputs from the serial interface, allowing the devices on the smart monitor to be updated in real time. The only way to change the state from the NORMAL_DISPLAY state is by either pressing the left button, which will take you to the ONLY_OFF state, by pressing the right button, which will take you to the ONLY-ON state, and by holding down the select button for 1 second will take you to the DISPLAY_STUDENT_ID state.

The DISPLAY_STUDENT_ID state is entered when the select button is pressed and held down for over one second. In this state, the smart monitor displays my student ID number. In this state, the smart monitor can still take in inputs from the serial interface and update the devices on the smart monitor in real time. It does not display these devices and changes on screen in real time. To escape this state, the user has to release the select button, allowing the smart monitor to return to its previous state.

The ONLY_OFF state is entered when the user presses and releases the left button on the smart monitor while the smart monitor is in the NORMAL_DISPLAY state. In this state, only devices that are in the off state are displayed on the smart monitor. It also still takes inputs from the serial

interface and allows changes made to devices on the smart monitor to be displayed in real time. Of course, if the change to a device is from off to on, the device will be removed from the display. To escape this state, the left button on the smart monitor must be pressed and released, which will return the smart monitor to the previous NORMAL_DISPLAY state. Another way to escape this state is by pressing and holding the select button for over one second, which will take you to the DISPLAY_STUDENT_ID state.

The ONLY_ON state is entered when the user presses and releases the right button on the smart device monitor while the smart monitor is in the NORMAL_DISPLAY state. In this state, only the devices that are on are on display. It also takes inputs from the serial interface, allows changes to be made to the smart monitor's devices, and updates and shows these changes in real time. To escape this state, repressing and releasing the right button takes you back to the NORMAL_DISPLAY state, and holding down the select button for longer than one second will take the smart monitor into the DISPLAY_STUDENT_ID state.

2 Data structures

`lcd` was in the instance of `Adafruit_RGBLCDShield` and is used throughout my project when making changes or displaying to the LCD.

The data structures that I was using in my project:

Device: This is a class data structure I used in my project to store all of the information on each device/ create device objects that will be stored on the smart monitor. It has the private attributes :

- `ID`: A character array of 4 characters that stores the device's ID.
- `Location`: A character array of 16 characters that stores the device's location.
- `Type`: a character representing what device's type, 'S' for speaker, 'T' for thermistor, 'L' for light, 'C' for camera and 'O' for socket.
- `State`: a character array of 4 characters that stores the state of a device either ON or OFF.
- `Power`: an integer representing the power output of a device.
- `Temperature`: an integer representing the temperature of a device. Between 9 and 32 degrees Celsius.
- `EPflag`: a Boolean value that is true when the device was originally read from the EEPROM. And false when the device was recently added.

Each attribute in the Device class has a getter method which is:

- `getID`: this method returns the device's ID as a string.
- `getLocation`: this method returns the device's location as a string.
- `getType`: this method returns the device's type.
- `getState`: this method returns the device's state.
- `getPower`: this method returns the device's power output.
- `getTemperature`: this method returns the device's temperature.
- `getEPflap`: this device gets the EPflag value of a device.

Each attribute also has a setter method which is:

- `setID`: this method is used to set the device's ID;

- `setLocation`: this method is used to set the device's location.
- `setType`: this method is used to set the device's type.
- `setState`: this method is used to set the device's state
- `setPower`: this method sets the device's power output.
- `setTemperature`: this method is used to set the device's temperature
- `setEPflap`: this device sets the EPflag value of a device.

Instances of the Device class are:

- `sDevices`, an array of Device objects I used this device array to store the devices that are on the smart monitor, (sDevices = smart devices).
- `sDevicesCopy` is the same as `sDeviceCopy`, (sDevicesCopy = smart devices copy).

State: is an enumerator that holds values representing the state the smart monitor can be in. i.e. SYNCHRONISATION, NORMAL_DISPLAY, ONLY_ON, ONLY_OFF, DISPLAY_STUDENT_ID.

Instances of the State enumerator are:

- `state`, which is a variable that stores the current state of the smart monitor
- `pState` is a variable I use to store the previous state of the smart monitor. (pState = previous state)

Other global variables are:

- `start` is an integer representing the index of the first element in the sDevices array.
- `end` is an integer used to represent the index of the last device in the sDevices array.
- `Display` is an integer representing the device's index in the array displayed on the smart monitor.

Functions that make changes to global variables:

- `setTheEnd()`, this function takes in a list of device arrays and then sets the end variable to the index of the last device in the device array passed into the function.

- `sortDevices()`, this function takes in a list of device arrays and then sorts them alphabetically by their ID.
- `addDevice()` is the function used to add a new device to the `sDevices` array. It also replaces a device's information with an already-used ID.
- `changeDeviceState()` is the function of changing a device's state inside the `sDevice` array.
- `changeDevicePower()` changes a device's power attributes (power or temperature) in the `sDevice` array.
- `RemovingDevice()` is the function to remove a device from the `sDevice`.

3 Debugging

I approached debugging my project in many different ways. When trying to follow my code's flow, I would put a print statement to the serial monitor displaying the project's state. Testing and checking if particular functions ran at the right time were done the same way as I would have a print statement at the beginning of the functions stating what function was being run at the time. To show that functions were being called properly.

My main approach to debugging was using print statements on the serial monitor, which proved to solve many of the problems I had. Although I found that these statements took up a lot of memory, so I did have to remove them after using them after using them.

Another way I debugged my code was by printing out the IDs of the devices currently inside my `sDevices` array. This allowed me to see what was currently in the array because much of my code depended on what Device IDs were in the array. And it allowed me to see if the functions like remove and add worked properly.

4 Reflection

In my project, I completed almost every functionality in the specification to complete perfection, within reasonable limits. However, I did encounter many issues/ problems, most of which were relatively easy to solve and overcome. I mainly had problems with memory, and the Arduino would run out of RAM, crash, and make errors that were very hard to debug. If I could rewrite my project, I would write my code in a way that requires less String, as I found that this was the reason for my high use of memory, and in turn, it would have made my code more flexible and robust.

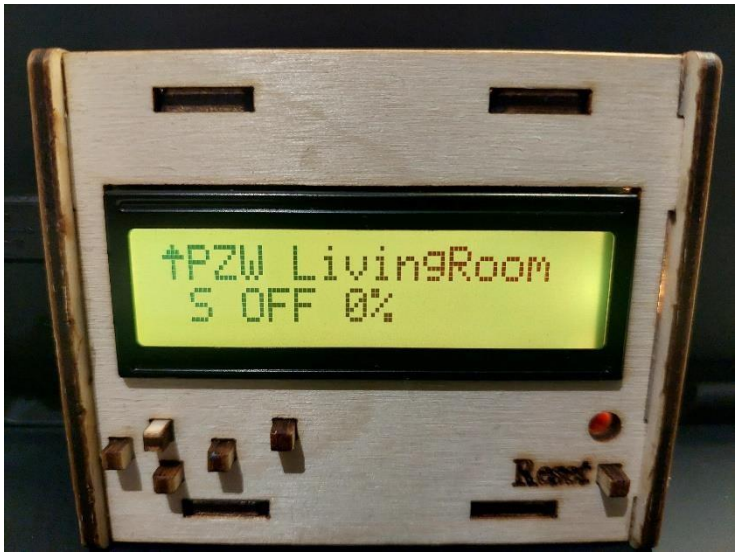
A feature that could be improved in my project is the timing of things. I had issues getting features such as; holding the select button for one second, and the user is taken to DISPLAY_STUDENT_ID, and scrolling a device's location on display by one character every two seconds, which proved to be quite a difficult task to get exactly correct because I did not fully understand how to use timers in my project. But in the end, I managed to get these features working to an acceptable level. If I could try this again, I would spend more time learning other ways to implement a timer system into my project.

Overall I am very happy with the results of my projects. However, I could improve on some things, like making my code more memory efficient and taking advantage of timers and functions that can better handle interrupts..

5 UDCHARS

I did not need to change the FSM for this extension. The process of creating custom characters on the Arduino was simple. The code I used to create the characters was the variable `upArrow` was used to create and store the up arrow character, and the variable `downArrow` was used to create and store the down arrow character. Then the method `lcd.createChar()` was used to make the characters I had made usable on the LCD.

The next step after I made the characters was implementing them into the smart monitor. The way I did this was I just replaced the characters in my `displayDevice`() function that I used to represent arrows with the newly created arrow characters.

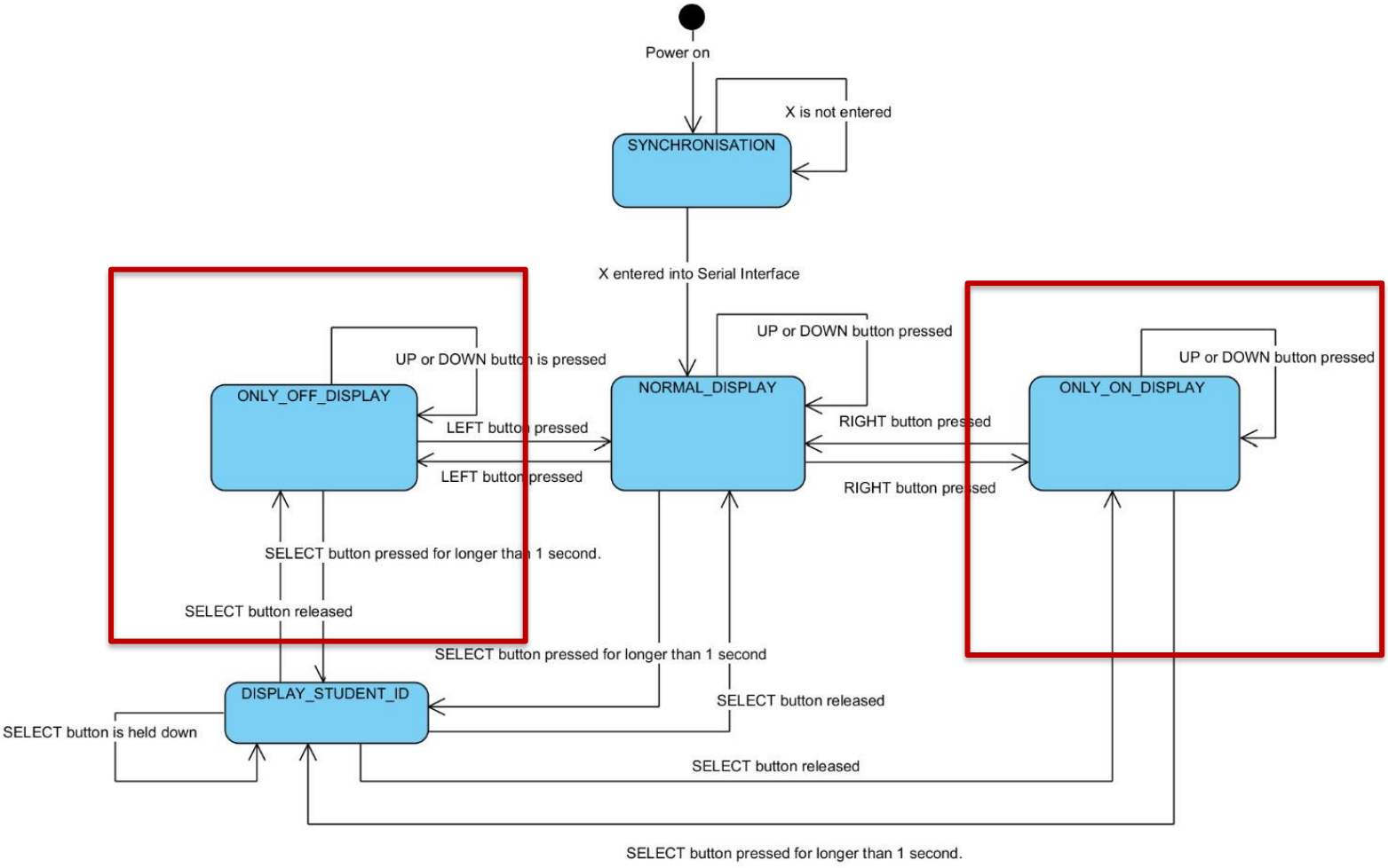


6 FREERAM

This extension did not require me to make changes to the FSM, I Added a snippet of code at the top of my project that allowed me to use a function `freememory()` that allowed me to collect and show the amount of available SRAM on the smart monitor. I then added this function to the `displayStudentID_State()` function, which allowed me to display my student ID and the amount of free RAM on the smart monitor.

This was a very humbling experience for me as it did cause a lot of issues with my code and some error messages malfunction, I did manage to fix these bugs in the end, and it has proven to be an amazing essential part of my project.

7 HCI



For this extension, I had to make changes to my FSM. To implement this extension, I added two new states, ONLY_OFF and ONLY_ON. For this extension, I used the array `sDeviceCopy` array to create a sub-set of arrays depending on the device's state. For example;

When going into the ONLY_OFF state, the project would run the `only_off_state()` function. This function would copy the devices in the sDevices array into the `sDevicesCopy`, then remove all the on devices from the `sDevicesCopy`. This then becomes an array from which devices displayed on the smart monitor are displayed from. It works the same for the ONLY_ON state, although it runs the `only_on_state()` function and only removes the off devices as its main differences.

When implementing this extension, I did run into a few problems, one of which was memory due to the fact I was doubling the number of devices being stored (one in the

`sDevices` and the `sDevicesCopy`). This also meant I had to reduce the number of devices I could efficiently have on the smart monitor.

8 EEPROM

For this extension, I did not have to make changes to the FSM, I made two functions `writeToEEPROM()` and `readEEPROOM()`. In the `readEEPROM()` function, I used the .get() method from EEPROM to read device objects on the EEPROM. `eeDevice` is a device object. `eeAddress` is the address in EEPROM we are reading from. To read objects from the EEPROM, I used the `.get()` method on the address in EEPROM we were on. Then add on the size of the device to get the address of the next potential object in EEPROM.

In the `writeToEEPROM()` function, I used EEPROM's `.put()` method to write device objects on the EEPROM. I copied the contents of the sDevice array onto EEPROM by iterating through the array. As a device is added, add the size of a device (30 bytes) onto the `eeAdress` to allow us to add the next device onto the next available space in EEPROM.

To check that the device read from EEPROM was written by me, I added the EPFlag attribute to the device object. This was set to true when writing to the EEPROM, which told me that the device was from the EEPROM when I read from it cause only the devices I sent to EEPROM had a true EPFlag. I found a better way to check if I wrote a device to EEPROM. It is by using a unique identifier that I can save as a device attribute when they are being uploaded to the EEPROM, and this identifier will tell me that I saved the device.

The EEPROM being 1024 bytes will limit the number of devices I can store on the EEPROM. As the size of a device is 30 bytes, you can only store 1024/30 = 34 devices at once on the EEPROM, and due to another limitation of the EEPROM, you are only able to write/read from EEPROM approximately 100,000 times this limits the use of the device further into the future.

9 SCROLL

For this extension, I did not have to change my FSM. To implement this extension in my project, I had to make a condition in my `displayDevice()` function that if the location of the device on display were longer than 11 characters, then the `scrollLocation()` function would be called and then scroll the location of the device on the screen.

The `scrollLocation()` function creates a copy of the device location into a string variable (`displayedLocations`) and then displays the location on the smart monitor. Then when two seconds pass, remove the first character of the `displayedLocation` and display this on the smart monitor. I repeated this until only the last character of the location was displayed. I then reset the `displayedLocation` back to the original Location of the device.

Also, to ensure that the button presses and other features, i.e. reading from the serial interface, can still take place, I added a condition in the function to check if any significant interrupts occurred. If any did, e.g. the up button is pressed, we will break out of the loop and properly execute what was needed.

There were many issues that I ran into when implementing the SCROLL. Like originally, I did try to display a substring of the location and would increment the start point of the string every iteration of the loop. However, removing the first character and resetting it when I reached the end of the string produced a more efficient result.