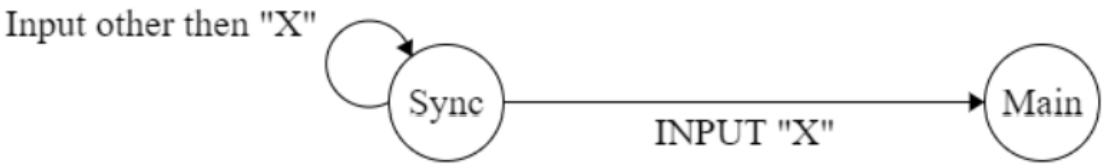
23COA202 Coursework

F330446

Semester 1

1 FSMs



The two states within my code are:

1. Synchronisation State
2. Main State

When the Arduino is turned on, initially the Synchronisation state runs, which print “Q” every second until “X” as of which the lcd backlight turns white and “UDCHARS, FREERAM” switches to Main State. If “X” not inputted, then error message shown and repeats Synchronisation state.

Once Main State the system monitors the serial interface for a user input and performs accordingly. For example, if the user inputs A with vehicle information in a certain format, then the vehicle information will be stored and displayed onto the lcd display screen. The program will, change and remove or add data according to the user input. The functions are:

1. A-adding vehicle information to program
2. S-changing vehicle payment status
3. T-changing vehicle type
4. L-changing vehicle location
5. R-removing vehicle information from program

The program also displays the vehicle information, and user can use the up and down buttons to scroll information. And the lcd screens backlight if Yellow is the payment status is NPD and green if the payment status PD.

The program also displays arrows only when it is possible to scroll to the above or below to display those elements. Also, when there is only one element, no arrows show as cannot scroll up or down.

When only on element in the parking data so no arrows appear:

When the last element in the array is displayed the upper arrow disappears:

When the last element in the array is displayed the lower arrow disappears:

When the select button is pressed by the user for more than 1 second the backlight to the lcd turns violet and my student ID and the free ram is displayed, when the select button is released returns to normal.

2 Data structures

The purpose of data structures within my code is to organize and store data effectively, allowing the data to be easily accessed and operated on. They have also enhanced the code readability, and adaptability. Below are the types of data structures I have implemented within my code.

- Arrays:

  On the left I have one-dimensional array of bytes to define a custom character for an 8x5-pixel up arrow (also done for the down arrow). The **'uint8_t upArrow []'** array represents the pattern for each row of the arrow. Using an array in this instance allows readability and easy modification due to the array for mat being easy to visually understand.

```
uint8_t upArrow[] = {
  0b00100,
  0b01110,
  0b10101,
  0b00100,
  0b00100,
  0b00100,
  0b00100,
  0b00000
};
```

- Struct:

  The **'struct parkingData'** id defined to store information regarding the inputted vehicles, including the registration number, location, type, payment status, entry and exit times. I have also used an array of this structure (**'parking[5]'**) to declare that up to five vehicles can be stored within this struct. The struct and array work together to manage information about the parked vehicles in my parking system. This structs allows logical grouping of related elements. Also. During modifications within the stored data, having a struct allowed changes to be localized, reducing error likelihood.

```
struct parkingData {
  String vehicleReg;
  String location;
  char vehicleType;
  String payStatus;
  String enterTime;
  String exitTime;
};
parkingData parking[5];
```

- String class:

  I have used the string class in my coursework to define variables (e.g. **String vehicleReg'**, **'String location'**, **'String enterTime'**) within the struct. I have also used a String class to store the user inputs as **'String input.'** Although Arduino had limited dynamic memory storage space, String class was the best to allow easy concatenation and manipulation throughout the code. Also, the String class manages memory dynamically, which avoids buffer overflow problems with the **'parking [5]'** array.

- Global Variables:

  In my code, I have declared several global variables, such as **'unsigned long lastDisplayTime'**, **'int n'** and **'int scroll.'** The global variables are convenient and simplify the communication between the states.

```
// the global variables use:
unsigned long lastDisplayTime;
int i = 0;
int count = 0;
int n = 0;       // array index
int scroll = 0;  // the scrolling index
```

3 Debugging

The process of debugging I have implemented throughout my code is now either commented out or printed on the Serial monitor beginning with “DEBUG:” and a statement explain what stages of the code have been passed. I used the process of debugging to identify errors, including logical errors, syntax errors and runtime errors by observing the Serial monitors outputs during the code running.

The main benefits of the debugging process within my code are to improve my code quality as debugging not only addressed the existing issues but also will prevent similar issues in the future operation of the code. During coding the stage of debugging also showed me opportunities to optimize the codes efficiency, by either concatenating strings or combining if statements to maintain readability.

There where certain draw acks of using this debugging process, mainly it is highly time-consuming, especially on the layered protocol codes. As well as hiving debug statements also altered the program’s behaviour, such as in Protocol A the debug statement began looping, due to a misplaced break. Hence even though debugging helps identify and fix errors there was still a risk of introducing new bugs. Lastly, debugging was an extremely mental exhaustive process for me, as it required prolonged sessions of concentration.

Examples of debugging within my code:

```
// Protocol A:
// FIX THE REPEATING 0 ISSUE IF COMMENT OUT SERIAL PRINT LN THEN ENTERTIME DISAPPEARS SOME DIGITS
bool exists = false;
if (input.charAt(0) == 'A') {
  if (input.charAt(1) == '-') {
    //Serial.print("...");
    if (input.substring(2, 9).length() == 7) {
      if (input[9] == '-' && input[11] == '-') {
        if (input[10] == 'C' || input[10] == 'M' || input[10] == 'V' || input[10] == 'L' || input[10] == 'B') {
          //Serial.println("...");
          if (input[11] == '-') {
            if (input.substring(12, input.length()).length() >= 1 && (input.substring(12, input.length()).length() <= 11)) {
              for (int i = 0; i < 5; i++) {
                //Serial.println("DEBUG: Enters the for loop");
                if (parking[i].vehicleReg.equals(reg) && parking[i].vehicleType == (input[10])) {
                  Serial.println("ERROR: This vehicle already exists");
                  exists = true;
                  break;
                }
```

This is a part of my Protocol A code, in which I have used **‘Serial.print (“…”)’** in many places as a debugging method to check if the user input has successfully been checked against the if statements, which check for a unique input format (e.g., **‘A-GR04SCD-C-LondonRd.’**) in order to continue. Under the for-loop declaration, I have also included another debug statement to check is the correctly formatted inputs can enter the for loop successfully or not.

4 Reflection

One of the issues I did run into during coding on the Arduino IDE was that the data began was being incorrectly displayed if too many vehicles were added to my struct whilst the array **‘parking[10]’** was programmed to hold 10 different vehicle data. However, once I identified this issue and observed that the errors had no certain patterns and was many due to the overload on the memory of the Arduino itself, I had to change my fixed array size from 10 to 5 to fix this issue.

Reflecting on my code, all the basic operations and protocols the Arduino code should undergo work and have been implemented according to the coursework specification. Out of the 5 extension tasks I have completed the UDCHARS and FREERAM. I would have also liked to try coding for the EEPROM and HCI, However, due to time constraints I planned to prioritise the debugging and display code necessary for the basic code instead of these extension projects. Although, I have discussed how I would tackle the extension tasks below in their specified sections.

Overall, I am really pleased with the functionality of my code, but if I worked on this project further, I would perhaps like to refactor the protocol handling sections to make them more modular and easier to extend. This would be done by implementing protocol cases to enhance the readability of my code. Another improvement to my code would be to use Enums instead of using numeric constants for the **‘Sync’** and **‘Main’** states, this could make the code more readable.

Extension Features

5 UDCHARS

This extension was to display custom characters that symbolise up and down arrows that at positions (0, 0) and (0, 1) of the lcd screen. In order to define the custom characters, I used 2 one-dimensional arrays of bytes named **'uint8_t upArrow []'** and **'uint8_t downArrow [].'**

```
uint8_t upArrow[] = {
  0b00100,
  0b01110,
  0b10101,
  0b00100,
  0b00100,
  0b00100,
  0b00100,
  0b00000
};
```

```
uint8_t downArrow[] = {
  0b00000,
  0b00100,
  0b00100,
  0b00100,
  0b00100,
  0b10101,
  0b01110,
  0b00100
};
```

Then within my **lcdDisplay()** void which codes for all the display functionality of the Arduino lcd screen. This means the arrows will only display if n (the index of my parking array) is greater then zero, so arrows are only present whilst elements are. Then **'lcd.createChar'** calls on the custom characters stored as upArrow and downArrow.

Then within the for-loop that iterates over 5 elements (as 5 elements spaces available in the **parking** array). If there is more than one element in the array and the scroll is not at the beginning, an up arrow is displayed at the top-left corner (LCD position 0,0). If there is at least one element in the array and the scroll is not at the end, a down arrow is displayed at the bottom-left corner (LCD position 0,1).

```
void lcdDisplay() {
  if (n > 0) {
    lcd.createChar(0, upArrow);
    lcd.createChar(1, downArrow);
    for (i = 0; i < 5; i++) {
      if (n > 1 && scroll != 0) {
        lcd.setCursor(0, 0);
        lcd.write((uint8_t)0);
      }// if 1+ elements in array and scroll not 0 the
      if (n > 0 && scroll != n - 1) {
        lcd.setCursor(0, 1);
        lcd.write((uint8_t)1);
      }/*prevents the down arrow being displayed if or
      or if the last element is being displayed on the
      lcd.setCursor(1, 0);
      lcd.print(parking[scroll].vehicleReg);
```

6 FREERAM

For this extension I had to download Memory Free library to Arduino IDE as it is not a standard library present beforehand. I included the library at the tope of my code with the other libraries I have used. Memory Free is a library that provides a function to display the amount of free memory available on an Arduino device.

```
1   //The libraries I have used within the code:
2   #include <Wire.h>
3   #include <Adafruit_RGBLCDShield.h>  // for the lcd screen
4   #include <TimeLib.h>                //for the time display
5   #include <utility/Adafruit_MCP23017.h>
6   #include <MemoryFree.h>  // for the freeSRAM display
7   Adafruit_RGBLCDShield lcd = Adafruit_RGBLCDShield();
```

The **'uint8_t buttons= lcd.readButtons()'** reads the state of buttons on the LCD shield and stored the result in the variable **'buttons.'**

The **'if(buttons & BUTTON_SELECT)'** checks if the select button has been pressed. If pressed the **'selectStart'** variable is set 0, and the **'selectStart'** will record the current time using **'millis().'** If the time since pressing the Select button is more than 1000 milliseconds and **'select_pressed'** is false, then the lcd Backlight is set to violet and my student ID and the **'getFreeMemory'** function is used to display the amount of free memory alongside **'bytes'** is displayed on the lower line.

It then sets **'select_pressed'** to true and resets the variable **'selectStart'** to 0.

```
{
  uint8_t buttons = lcd.readButtons();
  static unsigned long selectStart = 0;
  static bool select_pressed = false;

  if (buttons & BUTTON_SELECT) {
    if (selectStart == 0) {
      selectStart = millis();
    }
    if (millis() - selectStart > 1000) {
      if (!select_pressed) {
        lcd.setBacklight(VIOLET);
        lcd.setCursor(1, 0);
        lcd.clear();
        lcd.print("F330446");
        lcd.setCursor(0, 1);
        lcd.print(getFreeMemory());
        lcd.print("bytes");
        select_pressed = true;
        selectStart == 0;
      }
    }
  } else {
    if (select_pressed) {
      lcd.clear();
      lcd.setBacklight(WHITE);
      lcdDisplay();
      select_pressed = false;
      selectStart = 0;
    }
  }
```

7 HCI

I haven’t implemented this extension, but I would have liked to if I had more time as it is not much different to the previous structures used to display the parking array on the lcd screen.

1. Firstly, I would create a new function outside of my states called **‘lcdDisplaySubset’**, which would have a bool variable to indicate if **‘vehicleStays’** then I would call on the custom characters for up and down Arrow. This will be the copy of **lcdDisplay**, only the Boolean, the necessity of a string input of the paid and if no **‘vehicleStays’** then prints ‘NO VEHICLES’ will be different.
2. Then within the main state I would add **‘if (buttons & BUTTON_RIGHT)’** then **lcd. clear(),** set Backlight to green and call on the **lcdDisplaySubset(“PD”).**
3. **‘Else if (buttons & BUTTON_LEFT)’** then **lcd. clear()** and set Backlight to yellow and call on **lcdDisplaySubset(“NPD”)**
4. Else just **lcd. clear** and set Backlight to white and call on the normal **lcdDisplay().**
5. Break the for-loop.

8 EEPROM

I have not implemented this extension into my code. But if I when to I would:

1. Begin by downloading and **‘#include <EEPROM.h>’** library into by code.

2. Then I would create a different struct **‘EEPROMparking’** to represent the data to be stored in EEPROM this would the same as my **‘parkingData’** struct but instead of the string class for each variable I would choose the char class as that would be best suited for storing a fixed-character length of data. I would also specify the length of each variable in this struct.

3. Then I would probably define global variable to store the parking data in EEPROM as appropriate. 2 global variables would be needed: **Start** which would be 0 and **Spaces** which would be 5.

4. Then I would add a void **‘readEEPROMparking()’** which would include an EEPROM function that retrieve and read the EEPROM parking data.

5. Then I would implement another void **‘writeEEPROMparking()’** which commit to the changes of parkingData to EEPROMparking.

6. Then within the setup function I would initialize the **‘EEPROMparking’** array. To do this I would use a for-loop to equal all the **‘parkingData’** variables to the corresponding **‘EEPROMparking’** variables.

7. Then within the loop function, whenever the **‘parkingdata’** is changed will need to write that data into **‘EEPROMparking.’**

9 SCROLL

I have not implemented this function but to implement it I would:

1. To modify the **'parkingData'** struct to include '**scrollCurrent**' position as an int class.
2. To updating the **lcdDisplay ()** to include an if statement to claim that if the character length if larger then 7 is true then the '**scrollIndex**' which is an int would be the current time expressed in **'(millis()/1000) % (scrollLocation.length() + 2);'**. The '**scrollIndex**' of '**scrollCurrent**' is equal to the '**parking[scroll].**'
3. The '**scrollLocation**' will be equal to the '**scrollLocation.substring(scrollIndex, scrollIndex +7)**' which would extract the substring to display. Then lcd.print the '**scrollLocation**.'
4. Also need to update the L protocol to include '**parking[i].scrollCurrent = 0**' within the for-loop of the protocol.

Coding the scroll this way will allow scrolling of the current parking location to the left at 2 characters per second and return to the start again when the full location has been displayed.