

21COA202 Coursework

F128451

Semester 2

Contents

1	FSMs	3
2	Data structures.....	6
3	Additional Information.....	10
3.1	Debugging.....	11
3.2	Error Checking	12
3.3	Ordering the Channels	13
3.4	Testing	14
4	Reflection	16
5	UDCHARS.....	17
6	FREERAM	19
7	HCI.....	21
8	EEPROM.....	24
9	RECENT	28
10	NAMES.....	33
11	SCROLL.....	34

1 FSMs

A Finite State Machine (FSM) is an abstract model of computation that represents a system built on a limited number of mutually exclusive states. Each state has a transition towards and away from itself, usually initiated by a given input (e.g. a button press, a variable change, etc...). FSMs can be used in many areas, including business modelling to determine the processes linking each section of a business, and in philosophy for modelling behaviour and mood, to help identify patterns in human emotion and interaction.

In this coursework, an FSM has been used to build a data-management system and display. To accomplish this, an FSM was designed using the following states: Synchronisation, Awaiting Input, Awaiting Release. The UML state diagram for this FSM is pictured in Figure 1.

Synchronisation: Upon start-up, the Arduino will immediately enter the Synchronisation state. The purpose of this state is to delay the main program starting until the Arduino is synced with the accompanying Python program through the Serial Interface. If this state was not included, there may not be a reliable connection between the Arduino and Python program, leading to the Arduino reading incorrect inputs, which would cause unexpected errors in the program. It does this by setting the Liquid Crystal Display (LCD) backlight to purple and repeatedly sending the character 'Q' from the Arduino to the Serial Interface every second. It does this until the Python program sends the character 'X' to the Serial Interface. 'X' indicates the Arduino and Python program are synced and the Arduino can now reliably receive input, therefore can set the LCD backlight to white and move into the main state (Awaiting Input) of the program.

Awaiting Input: This state is responsible for collecting input from the user. It is the state the program will spend most of its time in, being idle until it receives an input. It is split into two sub-sections: dealing with input from the buttons on the Adafruit RGB LCD shield, and with text input through the Serial Interface.

To deal with the button input, the Arduino examines the last button input and acts accordingly. For example, if the up button is pressed, the program attempts to scroll upwards through the channel list (it will do this unless the first channel in the array is already being displayed). The same concept applies for pressing the down button, although

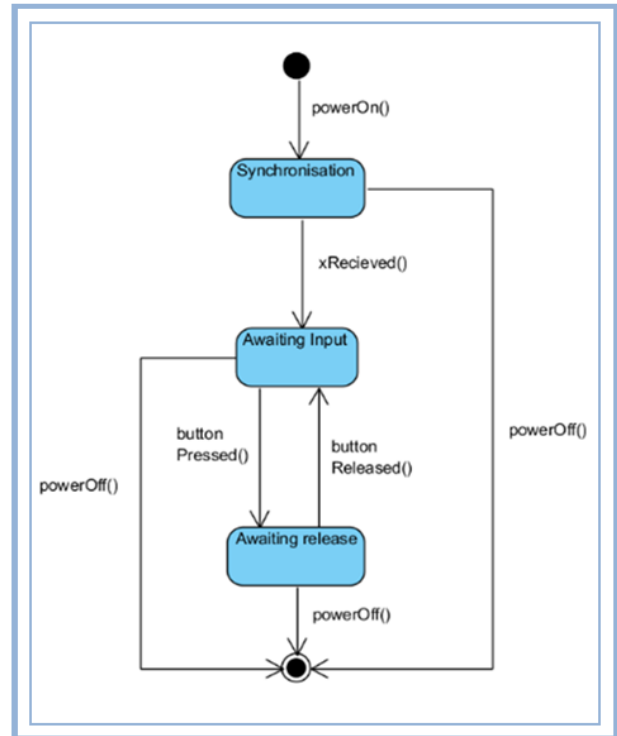


Figure 1: UML State Diagram

scrolling down through the array of channels instead. There is no effect from pressing the select button in this current state, however pressing any of these buttons move the FSM into the Awaiting Release state.

To deal with text input through the Serial Monitor the program first checks if any input has been received through the Serial Monitor. Input should be of the format “[Command][ID][Value]”, where command is ‘C’, ‘N’, ‘V’, or ‘X’, ID is a single character between A-Z, and value is a one, two, or three digit number, or word/phrase(only if ‘C’ is supplied as the command). If input has been given, it will enter into the section of this state that is equipped to deal with the input. It starts by determining which command has been given (this is a choice of ‘C’ for channel creation/altering the channel description; ‘V’ for setting the current value of a channel; ‘X’ for setting a maximum value that the channel’s current value can take; and finally, ‘N’ for setting a channel’s minimum value) then dealing with the input accordingly:

- **‘C’** – In this case, the input will fit the form “C[ID][Description]”, e.g. “CAMain”. If the character ‘C’ is the first character in the received input, the program must either create a new channel or update the description of an existing channel. It then updates the lcd with this new information. If a new channel has been added (because the channel ID being referenced does not exist in the channels array) the new channel will be inserted into the the array. On the other hand, if the channel does already exist within the array, it will simply update the channel’s description and the LCD with the new description. The channel will stay in the same position on the LCD.
- **‘V’** – The input will be given in the form “V[ID][Value]”, e.g. “VP56”. If the channel exists, its current value with be updated with the new given value and this will be displayed on the LCD. If the channel does not exist, the command will be dropped, with an error message being sent to the Serial Monitor.
- **‘N’** – The input will be received in the format: “N[ID][Value]”. If the program receives this command, it will first check if the channel with the given ID exists. If it does, it will update the channel’s minimum value from its previous value (when the channel is first created this value will be set to 0). If the channel does not exist, the command is abandoned and an error message is printed to the Serial Monitor.
- **‘X’** – The input is in the form “X[ID][Value]. If the channel exists, it will update the channel’s maximum value (by default, this value is 255). If the channel does not exist, an error is printed to the Serial Monitor and the command is discarded.



Figure 2: Arduino displaying channel ID, value, (RECENT placeholder) and description

- If none of the specified commands are received or the command does not fit the accepted format, the input will be rejected and error will be displayed on the Serial Monitor.



Figure 3: Minimum backlight



Figure 4: Maximum backlight



Figure 5: Minimum and maximum backlight

Additionally, when the commands 'V', 'X', or 'N' are received the program will check if the backlight of LCD needs updating.

Typically at run time, the LCD backlight will be set to white. This signifies that all channels are valid, i.e. their current value is within their minimum and maximum values. If any channel's current value is below its minimum boundary, the LCD backlight should be set to green. Likewise, if any channel's value is above its maximum value, the LCD backlight should be red. If both cases are true, the colour should be yellow.

Awaiting Release: This state is activated when the up, down, or select button is held. For the up and down buttons, this provides a simpler way of scrolling through the array of channels. Initially, it will scroll through the array one channel every 0.35 seconds. After the same button has been held down for 1.5 seconds, it will scroll through the array faster: one channel per 0.15 seconds. If the select button is held for a second, it will clear the lcd screen set the LCD backlight to purple, then display then display the ID number. When this button is released, it will clear the screen again and return to its previous display, including restoring the backlight colour (this may be white, green, red, or yellow). When any of these buttons are released, the state switches back to Awaiting Input.

2 Data structures

Structures:

	Variable within Structure	Data Type of Variable	Description of Variable
t_channel: This structure is used to define the data that each channel should store.	ch_name	Character	The ID of the channel.
	ch_descr	Char[]	A very short illustration of what the channel is.
	ch_val	Byte	The current integer assigned to the channel.
	ch_min	Byte	The minimum integer ch_val should take.
	ch_max	Byte	The maximum integer ch_val should take.
	below_min	Boolean	Whether the channel's current value is below its minimum value.
	above_max	Boolean	Whether the channel's current value is above its maximum value.

Enumerators:

	Values
state_e: This enumerator is used to list all states needed in the FSM	SYNCHRONISATION, AWAITING_INPUT, AWAITING_RELEASE

Variables of the above types:

	Instance of:	Description of Variable
channels	t_channel	An array of t_channel structures that stores the data for every channel used in the program; its maximum size is 26 elements (one channel for every letter of the alphabet).
channel	t_channel	Occasionally used within functions when only one channel from channels array needs to be referenced.
state	state_e	Stores the Enum value that indicates what state the FSM is currently in. Once certain conditions are met, the value of state will change.

Constants:

	Description of Constant
ID	Student ID number (F128451).
Extensions	List of all extension tasks completed (for now, this is set to "BASIC").

Functions affecting Structures:

The following functions update and use the data stored in the channels (t_channel) array or data associated within channels (e.g. next_empty_channel, and current_channel):

	Parameters	Description of Function	Effect on channels
Update_display()	ch_idx (int), the index of the channel being displayed on the top row of the LCD screen.	Takes the data from channels and displays the channel ID, and description, and calls the function to display the channel's value. Additionally, determines if '^' or 'v' should be displayed on the side of the LCD screen depending on if the first or last channel is being displayed (should be removed if at start of end of array).	No effect – all data stays the same.
	Channels (t_channel), the array of channels.		
	next_empty_channel (byte), the index of the last channel plus one. This indicates the length of channels.		
Display_value()	ch_val (byte), the value of the channel that is being displayed on the LCD.	Displays the supplied value on the LCD screen and right-aligns it.	No effect.
	row (byte), the row of the LCD screen the channel will be displayed on		
Check_below_min()	channel (t_channel), the single data structure that is being checked for validity.	Checks if the supplied channel's current value is above its minimum boundary.	Sets the channel's below_min Boolean to True or False accordingly.

Check_above_max()	channel (t_channel), the channel being checked for its range.	Checks if the channel's current value is above its maximum value.	Sets the channel's above_max Boolean accordingly.
Set_backlight()	channels (t_channel), the array of all channels in the program.	Checks all channels for being out of range of their boundaries and sets the backlight accordingly: green if a channel is below its minimum; red if a channel is above its minimum; yellow if a channel is below its minimum and another channel is above its maximum.	No effect.
	channel_amount (byte), the total amount of channels that have been defined.		
Scroll_button_pressed()	btn_pressed (int), indicates which button has been pressed (either up or down on the LCD shield).	Checks which button was pressed: if up was pressed, and the current index of the channel is not the start of channels, it scrolls up; if down was pressed and the channel index is not at the last index of channels array, scroll down.	Increments or decrements current_channel according to which button was pressed.
	*current_channel (int), pointer to current_channel, which signifies which channel index is being displayed on the top row of the LCD.		
	Channels (t_channel), array of all channels in the program.		

3 Additional Information

This section covers the following:

- Debugging
- Error checking
- Ordering the channels
- Testing

3.1 Debugging

Debugging is managed by a C macro, called 'DEBUG'. This includes a simple print statement that takes the supplied piece of data and prints it to the Serial Monitor. For ease of use, every statement printing data is preceded by a short description of what the data is. For example, in the function that sets the LCD backlight "Min: 0" or "Min: 1" is printed to the Serial Monitor to indicate whether a given channel's value is, respectively, above or below its minimum boundary.

Using a C macro, makes debugging incredibly simple as it is largely the same as printing to the Serial Monitor with the added benefit of being able to disable and enable all of the debugging statements at the same time. This means polishing the program is very fast, whilst also keeping all the debugging available for further expansion of the program.

```
#define DEBUG

#ifdef DEBUG

#define debug_print(description,value) Serial.print(description);
Serial.println(value)

#else

#define debug_print(description,value)

#endif
```

Figure 6: Defining the Debugging Macro

Important!

When testing the program, comment out line 12: '#define DEBUG'. If this macro is not commented out, the SCROLL extension causes the UDCHAR arrow characters to be printed in place of an empty description (i.e. when the channel has not been defined and there is an empty row on the LCD). This only happens when the Macro is left in the program.

3.2 Error Checking

Every time the user enters a command into the Arduino through the Serial Interface, it is necessary to validate the input. This is because, as per the specification, the Arduino is only expecting a relatively limited range of commands through the Serial Interface, so should not have to process any erroneous commands. The program needs to deal with what happens if the user accidentally makes a typographical error, or if an untoward user deliberately attempts to break the program. Therefore, the program filters commands using the following rules:

- Only accept 'V', 'N' and 'X' commands if they refer to a channel that exists (a channel exists if it has been created using the 'C' command).
- Channel ID must be between A-Z (specifically capital letters).
- Values given for 'V', 'N', and 'X' commands must be 0-255 (inclusive).
- Only accept the input if the first character is 'C', 'V', 'N', or 'X'.
- Inputs must fit the format '[Command][ID][Value]' or '[Command][ID][Description]'.

Finally, the length of the channel description should be limited to fifteen characters. However, entering a channel description longer than this should not cause the program to throw an error. In this case, the description is clipped after 15 characters to conserve memory. This is as per the specification.







3.3 Ordering the Channels



Part of the specification asks that the channels are displayed in alphabetical order. This implies that the channels are stored in alphabetical order too. I intended to store the channels contiguously in an array (i.e. no indices exist in the middle where there are no active channels) which created the need to use an insertion algorithm when a new channel was created.

The insertion algorithm first checks if there are any channels in the array. If not, the new channel can simply be placed in the first index of the array, and the algorithm finishes. Another case it checks for is that the ID of the channel being inserted (the new channel) is less than the current first channel in the array. If so, it pushes everything currently in the array back by one index and inserts the new channel in the first index. Lastly, there is the case where the new channel needs to be inserted 'somewhere' in the array. Therefore, it needs to iterate backwards through the array and find the first index where the new channel's ID is greater than the current channel's ID. When it finds this position, it pushes everything from that index back by one position and finally inserts the new channel at that index.

3.4 Testing

The program has been tested using a combination of manual inputs and inputs through the Serial Interface, using an accompanying Python program. Below is a table detailing various tests that have been completed following the completion of the BASIC specification:

Test	Input	Success?
Creating a channel	Once program is synchronized, enter "CLTest".	Yes 
Setting a channel's value	Following on from the previous test, enter "VL255".	Yes 
Channel's value is below its minimum	Enter the following commands: "CYMin", "NY100", "VY50".	Yes 
Channel's value is above its maximum	Continuing from test 2, enter "XL167".	Yes 
One channel is below its minimum, another is above its maximum	Combine the previous two tests.	Yes 
Scrolling	Create the following channels: "CAMain", "CBSecondary", "CCNew", "CDOld".	Yes 

Hold select for a second displays ID number	After the synchronisation phase, hold the select button.	Yes	
Synchronisation phase	Power the Arduino	Yes	

4 Reflection

Overall, I believe my code fulfils the specification for the BASIC tasks. The code is neat and commented, and variable and function names are sensible and descriptive.

On the other hand, there are several elements I wish to improve in my program. Firstly, I would like to make the channels array dynamic, so its size changes as new channels are added. This is because there is likely not much need to allocate 312 bytes to store 26 channels all the time. Depending on the intended use of this data manager and display, I would expect 10-20 channels to be used most of the time, which means a significant amount of bytes in the limited SRAM are allocated unnecessarily. A dynamic array would have to be created with a small initial size (perhaps four elements), then, when that array fills up, a new array would have to be created with one or two extra elements. The contents of the first array would have to be copied into the second array, then the memory allocated for the first array would need to be freed. The reason I did not include this is because freeing memory leaves gaps in the heap, so over time, it becomes fragmented and it is not always possible to insert data into the gaps. This makes stack and heap memory more likely to collide, causing the program to crash.

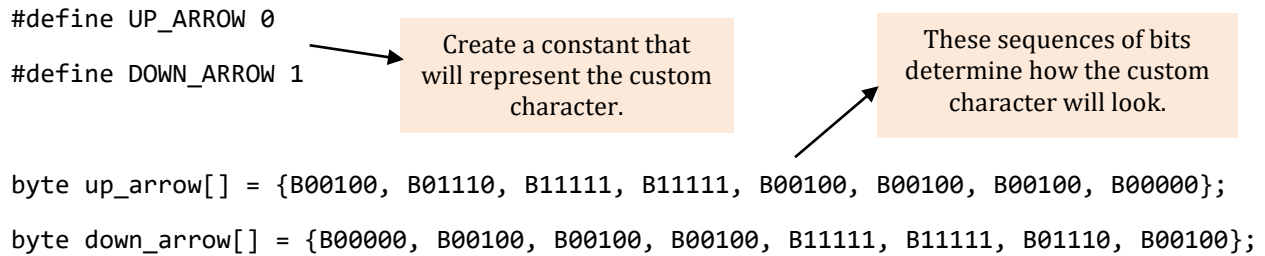
Additionally, I would like to tidy up how data is displayed on the LCD: when the user presses the up or down button, the arrow characters should remain on the LCD, unless the first or last channel are displayed. Currently, the display function uses `lcd.clear()` which removes everything from the LCD. This means that the arrow flickers when the button is pressed, when, it should be constantly shown on the screen. To fix this, I believe I would have to remove `lcd.clear()`, instead writing over the arrow characters with a space when the first or last channel is being displayed. Otherwise, the arrow characters should be displayed.

Lastly, I would like to increase the amount of values the Arduino holds for each channel (for the RECENT extension). Currently, it calculates the average of the last 32 values given to the channel, but ideally, it should be able to calculate the average of the last 64 inputs. To do this, I would need to take greater care with memory management, perhaps even investing into making the channels array dynamic. If the program tracked how much memory was being used up, it could allow more values to be stored if less channels were created. For example, if 10 channels were in use, it could allow 64 recent values for each channel, whereas if 20 channels were created, it may only allow for 48 values. This would not allow for 26 channels to store 64 values each. However, I believe for a general use case, this management of memory would be sufficient to properly fulfil the specification.

5 UDCHARS

The specification of UDCHARS tasks the programmer with creating custom arrow characters to display on the LCD in place of the 'v' and '^' characters. The functionality of the new and old arrow characters should be identical, with both arrows being shown on the screen, unless the first channel is displayed (in which case, remove the up arrow), or the last channel is displayed (remove the down arrow). If both the first and last channels are present on the LCD, neither arrow characters should be shown.

Conceptually, no change needs to be made to the FSM for this extension task as the intended use of the arrow characters does not change.



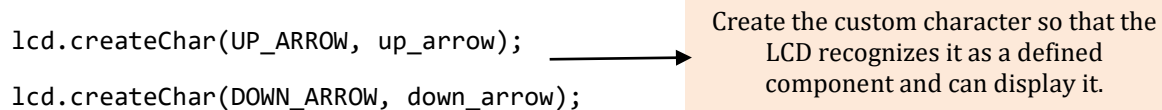
```
#define UP_ARROW 0
#define DOWN_ARROW 1

byte up_arrow[] = {B00100, B01110, B11111, B11111, B00100, B00100, B00100, B00000};
byte down_arrow[] = {B00000, B00100, B00100, B00100, B11111, B11111, B01110, B00100};
```

Create a constant that will represent the custom character.

These sequences of bits determine how the custom character will look.

Figure 7: Defining the custom characters (global scope)



```
lcd.createChar(UP_ARROW, up_arrow);
lcd.createChar(DOWN_ARROW, down_arrow);
```

Create the custom character so that the LCD recognizes it as a defined component and can display it.

Figure 8: Creating the custom characters (in setup())

```

if (ch_idx > lower_bound && lower_bound != -1)
{
    lcd.write(UP_ARROW);
}
lcd.setCursor(0, 1);
if (ch_idx < upper_bound && upper_bound != -1) {
    lcd.write(DOWN_ARROW);
}

```

Checks if the last first or last channel is being displayed. If not, then print the up or down arrow, respectively, to the LCD.

Figure 9: Using the custom characters on the LCD (in update_display())

Creating custom characters:

The LCD can display characters as a series of 1's and 0's in a 5x8 grid, where each element in the byte array is one of the eight rows.. '1' represents the cell in the grid is illuminated, whereas '0' indicates the cell is off. For example, to create a smiley face:

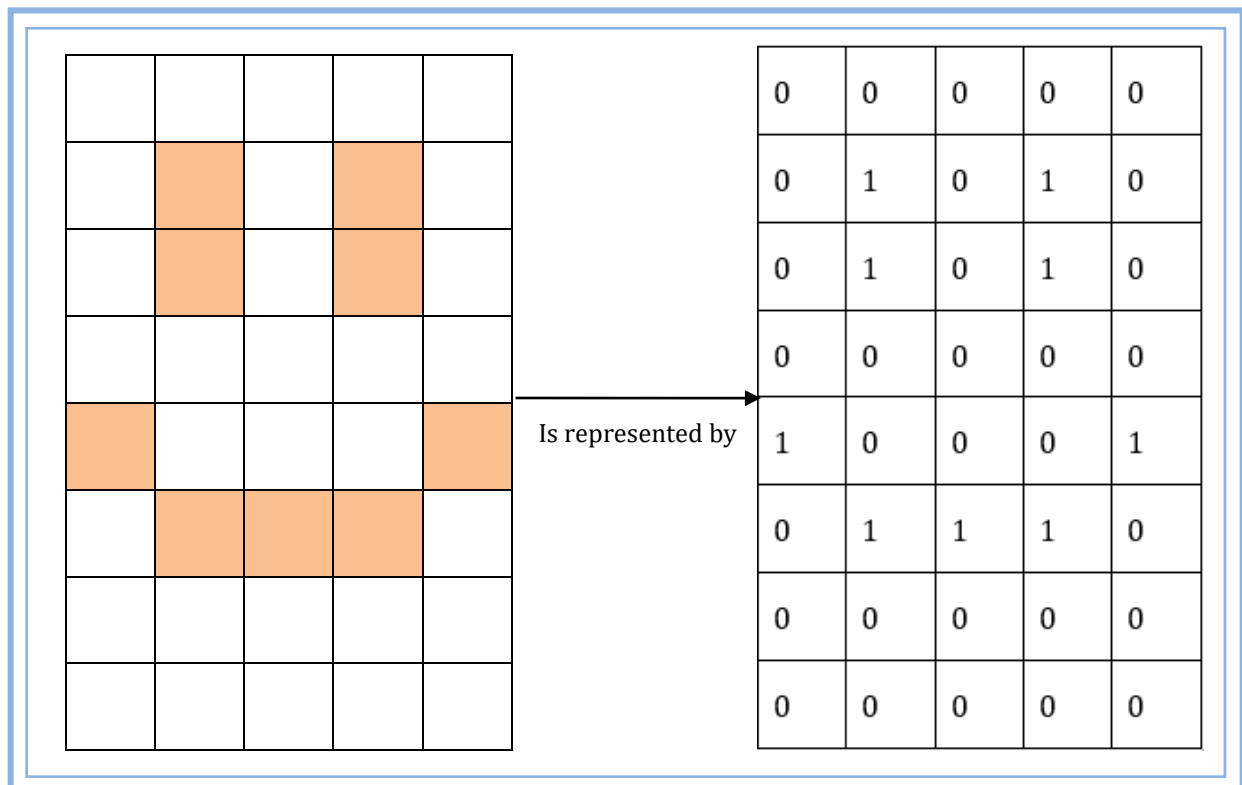


Figure 10: The concept behind custom characters

6 FREERAM

This extension task asks the programmer to extend the purpose of holding the select button to display the amount of free SRAM (in bytes), in addition to showing the programmer's ID number. Given this is an expansion of an existing feature, there is no changes to be made to the FSM.

```
int get_free_ram() {
    /*
        Calculates the amount of free SRAM left in the Arduino. This is calculated
        by subtracting the combined total of memory taken up by stack and heap
        memory from the total amount of memory available in the Arduino
    */
    extern int __heap_start; // Refers to the variable storing the start of the heap
    memory in an external file
    extern int *__brkval; // Refers to the pointer for memory in an external file
    int free_mem; // The amount of free SRAM which will be calculated


    if (__brkval == 0) { //If stack memory is empty, then the amount of free SRAM is the
    amount of memory not taken up by the heap memory
        free_mem = (int)&free_mem - (int)&__heap_start;
    }
    else {
        free_mem = (int)&free_mem - (int)__brkval; //If stack memory is not empty, then the
    amount of free memory is what is left after the pointer
    }
    return free_mem;
}
```

Figure 11: `get_free_ram()` function

This design is based on an algorithm by Mpflaga ([Link to the algorithm](#)). SRAM consists of both stack and heap memory, therefore, the idea behind the above calculation uses the pointers of heap and stack memory (as these indicate the endpoints of the two memory

sections). We can use our knowledge of how big the memory is in total and these pointers to calculate how much memory is remaining, by subtracting the pointers from the total memory.

Testing:

Test	Input	Success?
Display free SRAM and ID number.	During the awaiting_input state, hold the select button for at least one second.	Yes 

7 HCI

HCI challenges the programmer to create a method of displaying only channels where their current value is above their maximum boundary, or below their minimum boundary, when the right or left button is pressed. No changes to the FSM are necessary for this task. Below is the main code for displaying the minimum channels. The concept is identical for displaying maximum channels.

This code is complex so it will be explained in greater detail. First it is important to understand how the standard display works. By default, `ch_idx` is set to the given index that is passed to the function from the scroll function. This is a number between 0 and 25. `Next_ch_idx` is `ch_idx` plus one. When data is displayed to the LCD it gathers the channel information pointed to by `ch_idx` and `next_ch_idx` then displays them (channel pointed to by `ch_idx` is displayed on the first row, and channel pointed to by `next_ch_idx` is displayed on the second row). The most important part to understand is how `ch_idx` and `next_ch_idx` are being used in displaying the data on first and second row on the LCD.

When the minimum channels are shown, it will override `ch_idx` and `next_ch_idx` and set them to '-1'. This value represents no channel exists in the array where its value is below its minimum value. This is used to prevent errors occurring when less than 3 channels meet the criteria. From there, the channels array is iterated through. If a minimum channel is found, its index is assigned to `ch_idx` (if not, it is set to -1). Then, similar logic is applied to finding `next_ch_idx`. These are displayed on the LCD in an identical manner to the BASIC program.

This function is called when the scroll buttons are pressed to determine where the `next_ch_idx` lies in the array. This is important because it is used to prevent scrolling further through the minimum and maximum displays when the last channel that fits the criteria is displayed on the second row of the LCD.

```

// Initialise next_ch_idx as -1, which represents that a second channel fitting the
criteria does not exist
int next_ch_idx = -1;
if (min_shown) {
    bool ch_idx_exists = false;
    if (!scrolling_up){
        for (int i = *ch_idx; i < next_empty_channel ; i++) {
            if (check_below_min(channels[i])) {
                // Find the first channel in the array (from the current channel being
                displayed) that is below its minimum

                *ch_idx = i;
                ch_idx_exists = true;
                break;
            }
        }
    }
    else{
        for (int i = *ch_idx; i >= 0; i--) {
            if (check_below_min(channels[i])) {
                // Find the first channel in the array (from the current channel being
                displayed) that is below its minimum

                *ch_idx = i;
                ch_idx_exists = true;
                break;
            }
        }
    }
    debug_print(F("ch_idx: "), *ch_idx);
    if (ch_idx_exists == false) {
        debug_print(F("no channels are below minimum"), ' ');
        *ch_idx = -1;
    }
}

```

Figure 12.a: Algorithm for identifying which channels should be displayed in the minimum list

```



else {
    // If ch_idx is given a value this means at least one channel exists that
    matches the criteria, so it is possible another channel exists too

    // Now, we search for it
    for (int i = *ch_idx + 1; i < next_empty_channel ; i++) {
        if (check_below_min(channels[i])) {
            next_ch_idx = i;
            break;
        }
    }
    debug_print(F("next_ch_idx: "), next_ch_idx);
}
}

```

Figure 12.b: Algorithm for identifying which channels should be displayed in the minimum list (continued)

Testing:

Test	Input	Success?
Display minimum channels	Enter the following commands: "CA1", "CB2", "CC3", "CD4", "NA50", "ND167", "NC7", "VA49", "VC1", "VD200" then press the left button.	Yes 
Display maximum channels	Enter the following commands: "CA1", "CB2", "CC3", "CD4", "XB45", "VB150", "XD254", "VD255", then press the right button.	Yes 
Returning to BASIC display	Following on from test 2, press the right button again.	Yes

8 EEPROM

This task challenges the programmer to interact with the EEPROM. This task is split into two sections: reading from EEPROM on start-up and writing to EEPROM as the program runs. In order to tackle the first part, a new state must be created for the FSM – Initialisation. This state will provide the Arduino with time to access the EEPROM and read its data into the channels array. For writing to the EEPROM, this can be done during the program, every time a new channel is created, a description is changed, or a minimum or maximum value is set.

Data in EEPROM:

All the channels in the program are written to EEPROM as an array. This means that the first byte in EEPROM is an integer (1-26) which indicates how many channels are stored in EEPROM. This value is updated every time a new channel is created. It is also the first value to be read from EEPROM on start-up and is then used in a for loop to read all channels in EEPROM and nothing more. This is the mechanism for determining what is previously by this program, and not random data from previous programs. Each channel is stored using only its ID, description, minimum value, and maximum value (in this specific order). The ID occupies the first byte of each channel, the description takes up 15 bytes (descriptions that are shorter than 15 characters are extended to ensure consistency and make it easier to read from EEPROM later), and minimum and maximum values both also take only one byte.

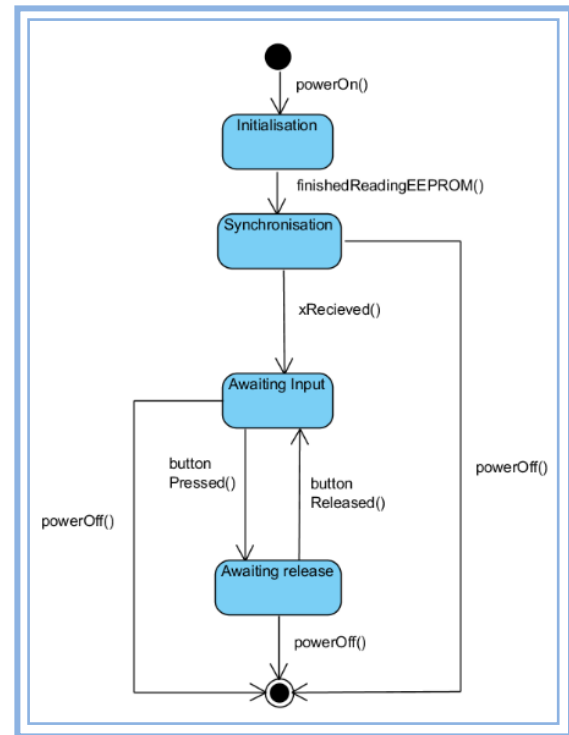


Figure 13: Altered State Diagram

When data is read from EEPROM, it is read sequentially, placing each channel inside the channels array for the main program. On the other hand, when data is written to EEPROM, the program either jumps to the first available space (using the amount of channels in EEPROM in the calculation of bytes) or jumps to the location of the channel with the specified ID.


```

void write_channel_to_EEPROM(int address, t_channel channel) {
    EEPROM.update(address, channel.ch_name);
    address += 1;
    String ch_descr = channel.ch_descr;
    byte len = ch_descr.length();
    if (len < 15) {
        for (int i = len; i < 15; i++) {
            ch_descr = ch_descr + " ";
        }
    }
    EEPROM.update(address, len);
    address += 1;
    for (int i = 0; i < 15; i++)
    {
        address++;
        EEPROM.update(address, ch_descr[i]);
    }
    address += 1;
    EEPROM.update(address, channel.ch_min);
    address += 1;
    EEPROM.update(address, channel.ch_max);
    Serial.println("Written");
    Serial.println(address);
}

```

Write channel ID to EEPROM.

Extend the length of the channel description to 15 character then write to EEPROM

Write channel minimum value to EEPROM.

Write channel maximum value to EEPROM.

Figure 14: Writing to the EEPROM algorithm


```

//First value indicates how many channels are stored in EEPROM
int address = 0;
byte channel_amount = Exported(address);
Serial.println(channel_amount);
address++;
for (int i = 0; i < channel_amount; i++) {
    address++;
    channels[i].ch_name = EEPROM.read(address);    // Read channel ID
    address++;
    int descr_len = EEPROM.read(address);
    char data[descr_len + 1];    // Read channel description as char array
    address++;
    for (int j = 0; j < 15; j++) {
        address++;
        data[j] = EEPROM.read(address);
    }
    data[descr_len] = '\0';
    channels[i].ch_descr = (String)data;    // then assemble into string
    address++;
    channels[i].ch_min = EEPROM.read(address);    // Read minimum value
    address++;
    channels[i].ch_max = EEPROM.read(address);    // Read maximum value
    Serial.println(channels[i].ch_name);
    Serial.println(channels[i].ch_descr);    // Debugging
    Serial.println(channels[i].ch_min);
    Serial.println(channels[i].ch_max);
    next_empty_channel = channel_amount;
}

```

Figure 15: Reading to EEPROM algorithm

Testing:

Test	Input	Success?
Data is correctly written to and read from EEPROM	Enter the following commands: "CANew", "CGImage", "CPdata", "NG100", "XP200". Then, reset the Arduino, and enter the following commands: "VG49", "VP231". The display should go green then yellow.	Yes 

9 RECENT

This extension asks the programmer to store the 64 most recent values given to a channel, calculate their average and output the average on the LCD. I believe this task is designed to be nigh-on impossible as storing 64 values for 26 channels would cost 1,664 bytes, more than 75% of the Arduino Uno's limited SRAM. This means that some reasonable compromises must be made to fulfil the specification in the best way possible. These compromises could have included not storing 64 values (maybe only 32), not including other EXTENSION tasks, reducing the maximum length of the description, etc.... I decided very quickly that the ideal compromise would be reducing how many values would be stored per channel. It is clear to me that the purpose of this task is not to specifically display the channel's average: it is to store large amounts of data. This immediately scraps the idea of storing a sum and count variable per channel and simply dividing them. Below are some of the concepts I created for completing this task:

- Firstly, the spec could have been followed quite literally, by simply storing as many values per channel as possible and averaging them to display on the LCD. However, in this solution, only 12-13 values would be able to be stored per channel, which is a bit lackluster and would not produce particularly accurate averages when more than 13 values are inputted. The first values to be inputted would be discarded too soon to make way for the new values coming in.
- Building on this idea, it could have been possible to use a combination of SRAM and EEPROM to store a large amount of values. The EEPROM EXTENSION uses approximately 200 bytes of EEPROM, leaving 800 bytes to use in this task. Combined with SRAM, I would have almost 1,200 bytes to work with, which would have allowed the Arduino to store 46 values per channel. It was also possible to store an extra value if the most recent value was only stored in the channel's `ch_val` variable and then the 2nd most recent value (and so on) was stored in the array of previous values. However, I believe this would be writing to and reading from the EEPROM unnecessarily, which is bad for the EEPROM's limited read-write cycle, as the specification does not ask us to store the channel's value persistently and that is what this concept would be doing. Therefore, in the interests of preserving EEPROM, I decided to not use this method either.
- Finally, I created what I believe is a good compromise: each channel stores a combination of the most recent values and the most recent averages. For example, to average the most recent 64 values, it would use an array of length 4 which contains the four most recent values inputted for that channel, and an array of length 16 that stores the last 16 averages. The way the last averages would be calculated would be using the four most recent values at the time a new average needed to be created. Then it would be possible to average these recent averages and also factor in the most recent values that are not yet stored in the averages array. When the 65th value is inputted, the program should drop the first average (effectively dropping the first four values that were inputted) and be replaced by the

new value. This way, at any given time after the first 64 values have been inputted, the Arduino would always store 61-64 of the most recent values. This solution is optimal as it stores 20 values for each channel (therefore fulfilling the specification) whilst also not sacrificing too much accuracy in the average.

Unfortunately, in practice, the Arduino was only realistically capable of storing four of the most values and eight of the most recent averages before starting to display warnings about memory. However, this does mean the Arduino is able to, in effect, find the average of the last 32 values, which I believe is still a respectable amount, albeit not as much as I was looking for. Furthermore, due to a very slim amount of memory left over from this implementation before warning messages start being sent, it was not possible to include a variable for each channel that stored the overall average of the last 32 values. Therefore, it was necessary to calculate the average each time the `update_display()` function was called which is a waste of time when the values do not change between calls. However, it is not the end of the world, as `update_display()` is not called too often that it would be an issue.

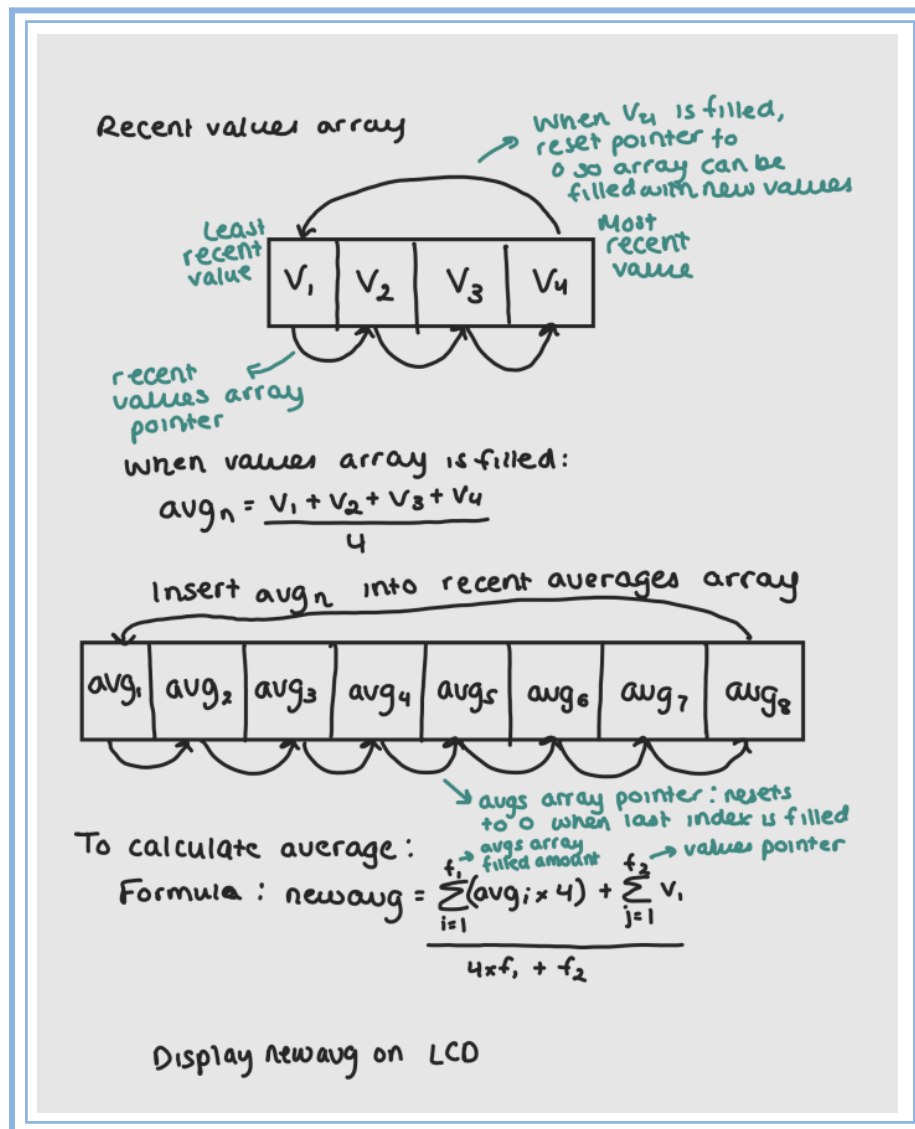


Figure 16: Diagram illustrating how RECENT works

```

// Add new value to ch_recent_vals

channels[existing_channel].ch_recent_vals[channels[existing_channel].vals_pointer] =
inp.toInt();

channels[existing_channel].vals_pointer++;

// Check if val_pointer = 4, need to set back to 0, and average ch_recent_vals and
put into ch_recent_avgs

if (channels[existing_channel].vals_pointer >
sizeof(channels[existing_channel].ch_recent_vals) / sizeof(byte) - 1) {

    channels[existing_channel].vals_pointer = 0;

    int sum = 0;

    for (byte i = 0; i < sizeof(channels[existing_channel].ch_recent_vals) /
sizeof(byte); i++) {

        sum += channels[existing_channel].ch_recent_vals[i];

    }

    // Calculate average of last 4 values and add to avgs array

    byte avg = sum / sizeof(channels[existing_channel].ch_recent_vals) / sizeof(byte);

    channels[existing_channel].ch_recent_avgs[channels[existing_channel].avgs_pointer]
= avg;

    channels[existing_channel].avgs_pointer++;

    if (channels[existing_channel].avgs_pointer >
sizeof(channels[existing_channel].ch_recent_avgs) / sizeof(byte) - 1) {

        channels[existing_channel].avgs_pointer = 0;

        // Need to set avgs_reset when the avgs array is filled and pointer moves back to 0
        // Because now when we calculate overall average, we need to know all of avgs is
        // filled but not to use a given index
        // When that index is being currently filled by new values

        channels[existing_channel].avgs_reset = true;

    }

}

```

Figure 17: Creating the array of recent values and averages (when a new value is entered)

```

byte get_avg(t_channel channel) {
    int avg_sum = 0;

    // Average all recent avgs and vals and print

    // Need to get estimate of previous averages -> avg = sum/count implies sum =
    avg*count

    for (byte i = 0; i < sizeof(channel.ch_recent_avgs) / sizeof(byte); i++) {
        if (i != channel.avgs_pointer || channel.vals_pointer != 0) {
            avg_sum += channel.ch_recent_avgs[i] * sizeof(channel.ch_recent_vals) /
            sizeof(byte);
        }
    }

    for (byte i = 0; i < channel.vals_pointer; i++) {
        // Add the values in the recent_vals array to the avg_sum

        avg_sum += channel.ch_recent_vals[i];
    }

    // Count depends on whether ch_recent_avgs has been filled before then reset

    // If it has not, then count will be how many indices of ch_recent_avgs are filled
    by a value * 4 + the vals_pointer value

    byte count = ((sizeof(channel.ch_recent_vals)/sizeof(byte)) *
    (channel.avgs_pointer) + channel.vals_pointer);

    if (channel.avgs_reset) {
        // Otherwise, we want to say 28 values (7*4) + vals_pointer

        count =
        ((sizeof(channel.ch_recent_vals)/sizeof(byte))*(sizeof(channel.ch_recent_avgs)/sizeof
        (byte))) + channel.vals_pointer;
    }

    // Calculate average




    byte avg = avg_sum / count;

    return avg;
}

```

Figure 18: Calculating the average function

Testing:

Test	Input	Success?
Calculates average of 16 values	Create channel 'A', and give it values 0-15. The average displayed on the LCD should be 7.	Yes 
Calculates average of 33 values	Create channel 'A' and give it values 0-30,100,217. The average should be 23.	Yes 
Calculates average of 35 values for all channels	Create channel A-Z, and give each 32 values. Every channel should display its average. More importantly, the Arduino should not run out of memory!	Yes 

10 NAMES

This extension asks the programmer to capture and display each channel's description. There were two possible data structures to use in this task: firstly, a string, limited to at most fifteen characters, or a character array set to exactly 15 characters long. I chose to implement the latter solution as the data type 'String' is well known for causing memory issues in small embedded systems, with limited memory, such as this system. This did slightly complicate the task as, instead of being able to use a simple assignment operator, I had to rely on the 'strcpy()' function, which I was not accustomed to.

Many of the prerequisite features for this extension are already covered in the BASIC specification, such as truncating channel descriptions after fifteen characters, which makes implementing this extension fairly straight-forward.

```
lcd.setCursor(10, 0);  
lcd.print(channels[ch_idx].ch_descr);
```

Figure 19: Printing channel description to the LCD

11 SCROLL

This extension challenges the programmer to scroll a channel's description to make all of it visible to the user. To accomplish this task, the TimerOne library was used. This library facilitates the use of the Arduino Uno's 16-bit timer in a way that is a lot simpler than interacting directly with the Arduino's output pins. For the sake of this extension, the interrupt period was set to $\frac{1}{2}$ second, meaning it would call the Interrupt Service Routine (ISR) every 500 milliseconds. Because of the high rate of the timer, it was not possible to contain the whole scrolling function within the ISR, as the time it takes to complete the ISR must be less than the interrupt period. Instead, the ISR was used to set a Boolean value, named 'scroll' to true. Then in the main loop, specifically the AWAITING_INPUT state, it would check the state of 'scroll' and if it was true it would scroll top and bottom channel descriptions along by one character, then set 'scroll' back to false, ready for the next interrupt.

```
Timer1.initialize(500000); // initialize timer1, and set a 1/2 second period
Timer1.attachInterrupt(call_scroll); // attaches call_scroll() as ISR
```

Figure 20: Initialising Timer1

```
void call_scroll(){
    scroll = true;
}
```

Figure 21: The Interrupt Service Routine

```

if (scroll){
    byte next_channel_index = current_channel + 1;
    if (min_shown || max_shown){
        next_channel_index = get_channel_display_indices(channels, &current_channel,
next_empty_channel, min_shown, max_shown);
    }
    byte descr_length = 0;
    // Calculate size of descr and descr2 that is actually occupied by description -
    looking for null-terminating character
    for (byte i = 0; i < sizeof(channels[current_channel].ch_descr)/sizeof(char); i++){
        if (channels[current_channel].ch_descr[i] == '\0'){
            break;
        }
        descr_length++;
    }
    ...
    if (descr_length > 6){
        lcd.setCursor(10,0);
        if (topPos < descr_length - 5){
            for(byte i = topPos; i < topPos + 6 ; i++){
                lcd.print(channels[current_channel].ch_descr[i]);
            }
            topPos++;
            if (topPos > descr_length - 4){
                topPos = 0;
            }
        }
        else{
            lcd.setCursor(10,0);
            lcd.print(channels[current_channel].ch_descr);
            lcd.print(F("    "));
        }
    }
    ...
    scroll = false;
}

```

Get the indices of the two channels being displayed on the LCD.

Go through the same process for the description on the bottom row of the LCD.

If the description is more than 6 characters long, scroll the description.

This line stops the scrolling when the last character of the description is in the last position on the LCD.

Prints the subset of the description to the LCD.




When the end of the description is reached, reset the position to 0 to repeat the scroll. It pauses for a second before resetting.

If the description is shorter than 7 characters, do not scroll.

Go through the same process for the description on the bottom row of the LCD.

Figure 22: Scrolling the description

Testing:

Test	Input	Success?
Scrolling on top and bottom	Create two channels: "CALongDescription", "CBHelloooo". Both descriptions should scroll and reset.	Yes 
Scrolling on just the top	Create two channels: "CALongDescription", "CBMain". Only the top description should scroll, the bottom description should be stationary.	Yes 
Scrolling when there is only one channel displayed	Create one channel: "CALongDescription".	Yes 
Scrolling in min/max display (HCI)	Create two channels: "CALongDescription", "CBDescription". Give both channels minimum value 10, and set their values to 1. Both descriptions should scroll on the minimum display.	Yes 