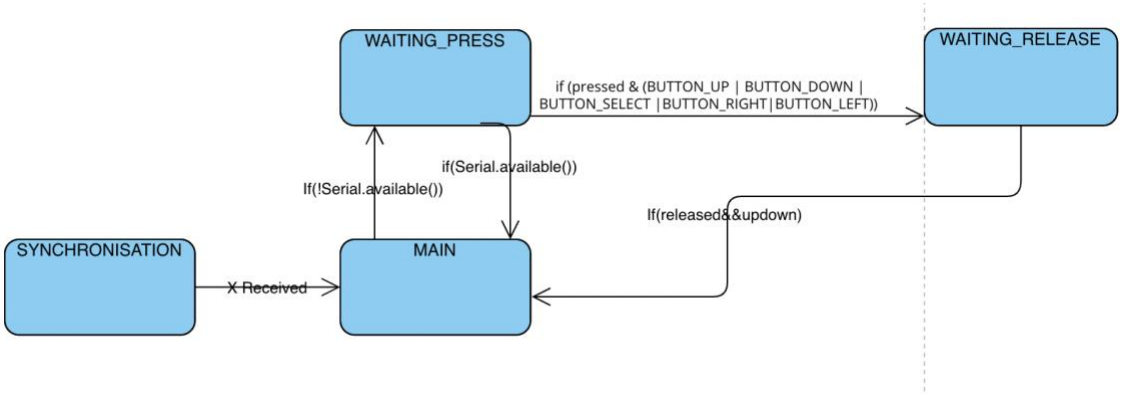
1 FSMs



I wanted to avoid a hierarchical FSM, so have no sub-FSMs, as to avoid arbitrary complexity.

SYNCHRONISATION – This state is supposed to stall the program from running until it is synced with the python script for running it. It achieves this by continuously sending the character “Q” to the serial monitor until it receives the letter “X” in the serial monitor, which will have been sent by the python script, signifying that it is able to communicate to the Arduino. This then will change the state to the MAIN state.

MAIN – Continuously waits for Serial Monitor input and processes it. It does this by switching to the awaiting button press state when nothing is in the Serial Monitor; so, it doesn’t need to process the buttons and text input simultaneously. Then when it does receive input, it will see if it is in the acceptable format, if not, it will simply await another input. If it does receive an acceptable input, it will then do the desired operations to handle the data given.

WAITING_PRESS – Continuously waits for button presses and processes each press accordingly. It does this by seeing if a button is pressed, then checking whether the button press is within up, down, left, right, select. When it registers which button it is, it will then transition to whatever action that specific button does, such as up and down scrolling, left and right activating HCI. If the button is pressed for longer than [enter time here] then it will transition to the WAITING_RELEASE state.

WAITING_RELEASE – Similar to WAITING_PRESS, in that it will differentiate between which button is being held and run code in accordance with the function of each button.

Expansion on MAIN state

The main state allows four operations:

C – Change the description of a channel.

V – Change the value of a channel.

If the first character of the input string is V, the program then converts everything after the [Operation][Channel] part of the string to string and saves this as valueString and finds the corresponding [Channel] in 'channels'. It then takes the valueString, converts it to integer, then replaces the selected element in 'channels' .value element to the integer value.

X – Change the maximum of a channel.

Works much the same as V, only changes the maximum value instead. However, it does also check whether the new input maximum is bigger than the minimum before setting it.

N – Change the minimum of a channel.

Works the same as X, however, checks whether new input minimum is smaller than maximum for the selected element.

C – Change the description of a channel

If the first character is C, the program first converts everything after [Operation][Channel] to string; finds the corresponding [Channel] in the array of Channel_dict structures defined as 'channels' and finds this element's .description element, then writes the string value to this field. If it encounters an already filled description field, it will simply overwrite it.

Code Expansion

```
639     if(!Serial.available()){
640     state=WAITING_PRESS;};
641       Serial.print("Recieved input");
642       Serial.println(freeMemory());
643       String x = Serial.readString();
644       //Serial.println(x);
645       //lcd.clear();
646       //lcd.setCursor(0,1);
647       //lcd.print((char)x[0]);
648       int pos=return_alpha(channels,x[1]);
649       if((char)x[0]=='C' or (char)x[0]=='V' or (char)x[0]=='X' or (char)x[0]=='N'){
650         set_alpha(channels,pos);
651       }
```

If there is nothing typed in the serial monitor, the program will change state to WAITING_PRESS, as it does not need to process the serial monitor input, as there is none.

If there is an input, it will print that it has received input, store the contents of the second letter of the string in the pos variable, as this is the letter of the channel to be changed. Then it will check if the first letter is any of the available operation letters, if it isn't, it will simply wait for the next input. Otherwise, it will first give the desired channel its channel letter (e.g. if user selected channel A, it would set the channel field of the struct in position 0 in the array to A).

```
652     if((char)x[0]==(char)'C'){
653       //follow by channel number and description
654       //String textData=partial_string(x,2,x.length());
655       String textData=x.substring(2,17);//15 characters long
656       textData.trim();
657       //lcd.clear();
658       //lcd.setCursor(0,1);
659       //lcd.print(textData);
660       channels[pos].text=textData;
661       pointer=pos;
662       lcd_channels(channels,pointer);
663       //lcd.setCursor(0,0);
664       //lcd.print(channels[pos].text);
665     }
```

If the ‘C’ operation is selected, the program will take the substring of the user input representing the 15 letters of the description. It will then trim this of any whitespace and set the text field of the desired channel to this description. It then makes the pointer (representing the value printed on the top line of the lcd) to the value of pos, so the screen shows the channel on which the change was made.

```
666     else if((char)x[0]==(char)'V'){
667       //follow by channel number then a value between 0 and 255
668      //int val=(int)partial_string(x,2,x.length());
669      //int val= int atoi((const char * str)
670      //int val=convert_to_int(x,2,x.length());
671      //String textData=partial_string(x,2,x.length());
672      String textData=x.substring(2);
673      int val=textData.toInt();
674      //lcd.clear();
675      //lcd.setBacklight(2);
676      //lcd.print(val);
677      if(val>=0&&val<=255){
678      if(val>channels[pos].maximum){
679        over_max=true;
680      }
681      if(val<channels[pos].maximum){
682        over_max=false;
683      }
684      if(val<channels[pos].minimum){
685        over_min=true;
686      }
687      if(val>channels[pos].minimum){
688        over_min=false;
689      }
690      Serial.println("Over Max"+(String)over_max);
691      Serial.println("Over Min"+(String)over_min);
692      if(over_min&&!over_max){
693        lcd.setBacklight(2);
694      }
695      else if(over_max&&!over_min){
696        lcd.setBacklight(1);
697      }
698      if(!over_max &&!over_min){
699        lcd.setBacklight(7);
700      }
701      else if(over_min && over_max){
702      lcd.setBacklight(3);
703    }
```

```
704       //delay(500);
705       channels[pos].value=val;
706       //Serial.println(val);
707       delay(500);
708       write_my_eeprom_single(channels,pos);
709       lcd_channels(channels,pos);
710       //eeprom_read(0);
711       /*for(int y=0;y<=64;y++){
712         Serial.println(avgArray[y]);
713       }*/}
714       //lcd.clear();
715       //lcd.print(channels[pos].value);
716
717       }
```

If the 'V' operation is selected, the program first stores the substring of the input from position 2 to the end of the string, it then converts this to int and stores it in the val variable.

It first checks if the value is above or equal to 0 or less than or equal to 255, as these are the bounds set out in the spec (in addition, this also checks if an input rather than a blank has been received.). If it is not, it rejects the input and awaits the next input. Otherwise, it goes on to check whether it is above the channel's designated maximum or below the designated minimum.

It uses the variables over_min and over_max for this. It sets over_min to true if val is less than the channel's minimum and sets it to false if val is more than the channels minimum. It sets over_max to true if val is more than the channel's maximum and sets it to false if val is less than the channel's maximum.

It then checks if either of over_min or over_max are true, if both of them are or if neither of them are. If over_min is true and over_max is not, the display backlight is set to 2 (green). If over_max is true and over_min is not, the display backlight is set to 1(red). If both over_max and over_min are true, the display backlight is set to 3(yellow). If neither of them are true, the display backlight is set to 7(white).

It will then set the value field of the channel to the entered val, then output this to the lcd, using the lcd_channels function.

```
718     else if((char)x[0]=='X'){
719       //maximum value for channel; channel number then value; 255 default
720       //int check_max=convert_to_int(x,2,x.length());
721       //int check_max=(partial_string(x,2,x.length()).toInt());
722       int check_max=(x.substring(2)).toInt();
723       if(check_max>=channels[pos].minimum&&check_max>=0&&check_max<=255){
724         channels[pos].maximum=check_max;
725       }
726       else{
727         Serial.print("Maximum cannot be smaller than minimum of:");
728         Serial.println(channels[pos].minimum);
729       }
730
731     }
```

If the 'X' operation is selected, the substring 2 till the end of string is taken again, and converted to an integer, check_max. If check_max is both less than or equal to 255, bigger than or equal to 0 and biggeror equal to the channel's current minimum, it is set as the new maximum, otherwise an error is output.

```
732       else if((char)x[0]=='N'){
733         //minimum value for channel; channel number then value; 0 default
734         //int check_min=(partial_string(x,2,x.length()).toInt());
735         int check_min=(x.substring(2)).toInt();
736         //int check_min=convert_to_int(x,2,x.length());
737        if(check_min<=channels[pos].maximum&&check_min>=0&&check_min<=255){
738           channels[pos].minimum=check_min;
739         }
740         else{
741           Serial.print("Minimum cannot be bigger than maximum of:");
742           Serial.println(channels[pos].maximum);
743         }
744       }
```

If the 'N' operation is selected, the substring 2 till the end of string is taken again and converted to an integer, check_min. if check_min is both less than or equal to the channel's current maximum, bigger than or equal to 0 and less than or equal to 255, it is set as the channel's new minimum. These checks are done to make sure the number is within the bounds 0-255 by the spec and so that the minimum is not greater than the maximum. Otherwise, an error is output.

Clarification of input string

Furthermore, the input must follow the pattern of [Operation][Channel][Value]. Thusly, the program checks if the first character ('x[0]') is either C, V, X or N, if it isn't it simply awaits another input, outputting that the input is not accepted to the serial monitor. Similarly, if the channel does not exist, it does the same.

Expansion on SYNCHRONISATION state

This state continuously sends the character 'Q' to the Serial Monitor, with a 1 second delay between each send. It does this until it receives an input from the Serial Monitor, at which point it will print 'BASIC' if the input is 'X' and set the backlight to 7 (which is purple), otherwise it will continue outputting 'Q' until 'X' is received. It then outputs the channels and their values and descriptions (which will be at their default settings at this point) and switches the state to main.

Code Expansion

```
757 case SYNCHRONISATION:{
758     if(!Serial.available()){
759       delay(1000);
760       Serial.print("Q");
761       }
762     String x=Serial.readString();
763     if(x=="X"){
764       Serial.println("\nBASIC");
765       lcd.setBacklight(7);
766       lcd_channels(channels,0);
767       state=MAIN;
768      }}
769      break;
```

If there is no input submitted in the serial, ‘Q’ will be printed at a rate one per second. If there is an input and it is ‘X’, ‘BASIC’ is output, the backlight is set to 7(white) and the state is changed to MAIN.

Expansion on WAITING_PRESS state

The first thing the WAITING_PRESS state checks is whether there is an input received in the Serial Monitor, if there is, it will switch to the MAIN state to process this input. This both allows for prompt processing of input and accounts for the fact that the user won’t need to use both states at once, thus cutting out excess processing.

It then stores whether any buttons have been pressed in the variable b. It then checks if the press is in any of the necessary buttons to be processed, these include BUTTON_UP, BUTTON_DOWN, BUTTON_RIGHT, BUTTON_LEFT, BUTTON_SELECT.

BUTTON_UP – This will set the pointer (which is defined elsewhere and will be explained later in the documentation) to -1 from itself unless it’s equal to 0, because we don’t want the pointer to point at non-existent values. It will then call the lcd_channels function and pass it the pointer as well as the channels array. This will then update the screen to scroll the values up, and thus display them.

BUTTON_DOWN – Does much the same as BUTTON_UP, however will +1 to pointer unless the pointer is equal to one less than the length of the channels array.

Code Expansion

```
747 case WAITING_PRESS:{
748   if(Serial.available()){
749     state=MAIN;
750   }
751 int b = lcd.readButtons();
752 // We are looking for buttons that were NOT pressed
753 // and are pressed now.
754 // Logic is "now AND NOT last_time"
755 int pressed = b & ~last_b;
756 last_b = b;
757 // if either up OR down is pressed
```

If there is an input in the serial, the state is switched back to MAIN, so it can process the user input. b and pressed are instanced here, with b representing the buttons, and pressed representing a button that is being pressed that was not pressed before.

```
758 if (pressed & (BUTTON_UP | BUTTON_DOWN | BUTTON_SELECT |BUTTON_RIGHT|BUTTON_LEFT)) {
759 // now just checking for up
760 if (pressed & (BUTTON_UP)) {
761   if(pointer!=0){
762 pointer-=1;}

772 lcd_channels(channels,pointer);}
```

Checks if a button has been pressed and that this button is either the up, down, left, right or select buttons. Then, if the button pressed is up, the pointer is decremented unless it is 0, as

decrementing if it was 0 would lead to it pointing to a non-existent element. It then outputs two elements of the channels array to the screen, showing the channel at pointer on the top line and the channel at pointer+1 on the second line.

```
815 if(pressed&BUTTON_DOWN) {

835  if(pointer+1!=25){
836  pointer+=1;
837 }
838 lcd_channels(channels,pointer);}
```

If the button pressed is down, the pointer is incremented by one, unless incrementing the pointer would make it point to the last element, as the last element should be displayed on the bottom line, not the top, to avoid a blank line, so should not be pointed to by the pointer. The lcd_channels function is then called to output two elements of the channels array to the screen, showing the channel at pointer on the top line and the channel at pointer+1 on the bottom line.

Expansion on WAITING_RELEASE state

This, like WAITING_PRESS uses updown to see which button is pressed. The difference is, this implements press_time, which stores the time (millis()) at which a button is pressed it then uses millis()-press_time to see how much time the button has been held for and if it is longer than 250 milliseconds it will then classify it as being held and allow for the actions in this state to be taken.

It then checks if the select button is the button being pressed (updown==BUTTON_SELECT). It then outputs the student ID if this is the case.

Otherwise, if a button is no longer being held, it will then register it as released, return the state to MAIN, and run lcd_channels to re-output the channels to the lcd.

```
677 case WAITING_RELEASE:{
678  if(millis() - press_time>=250){
679   press_time=millis();
```

Here, it checks that 250 milliseconds has passed whilst holding a button aka, it is currently held.

```
727   if(updown==BUTTON_SELECT){;
728  // update_display_two_line("F129714",EEPROM.freeMemory());
729  update_display("F129714");}
```

If the button held is the select button, it then clears the lcd and outputs the student ID number.

```
826 }else {
827 int b = lcd.readButtons();
828 // We are looking for buttons that are not pressed
829 // now and were pressed last time.
830 // The logic below is this. ~ means not.
831 // so "NOT now AND last_time"
832 int released = ~b & last_b;
833 last_b = b; // Save
834 // Mask off the bit we are interested in and if
835 // it's been released then change state
836 if (released & updown) {
837  lcd_channels(channels,pointer);
838  //Serial.println("waiting press");
839 state = MAIN;//WAITING_PRESS
840 }
841 }}
```

If the button is not held, it calls the lcd_channels function to output two channels at pointer and pointer+1 to the screen, so that when the button is released, the lcd reverts to as it was before it was held.

This then sets the state to MAIN where it will either process incoming input or switch to WAITING_PRESS if there is no input to be processed.

2 Data structures

Operations that affect global data structures/ store are detailed in the above section, as these are handled within the states rather than within functions.

Data Type	Reason
Struct	Better access to individual elements of each structure, e.g. can call structElement.value to get a value of an individual structure
Enum	States are immutable list of states, so enum lends naturally
Int	Allows storage of whole numbers
String	Allows storage of word strings
Constant	Used for values that aren't changed after they're set
Array	Allows for easily grouping items together to be easily searched

Variable	Data Type	Description
Channel_dict	Struct	Used for storing the information of each channel, has the fields channel (for storing the letter of the channel), text(for storing the description of the channel), value, maximum and minimum.
channels	Channel_dict[]	Array that holds each channel (of data type Channel_dict).
pointer	int	Used to keep track of what channels are being shown on screen.
b	int	
State_e state	enum	Used to store all available states in the FSM.
Press_time	long	Stores how long a button has been pressed.
Last_b	int	
Refine_right	Bool	Signifies whether the right button has been clicked or unclicked, as to whether it should implement HCI.
Refine_left	Bool	Signifies whether the left button has been clicked or unclicked, as to whether it should implement HCI.
Val	Int	Stores a numeric value used in the right_align and left_align functions
Over_min	Bool	
Over_max	Bool	
X	String	
textData	String	
Check_max	Int	

Check_min	Int	
Press_time	Int	
Len	Int	Stores the length of the channels array
Released	Int	
Pressed	int	
Name0x0[]	Byte	Stores the byte data for the up-arrow character
Name0x1[]	Byte	Stores the byte data for the down arrow character
I	Int	Used in loops
charArray	Char	Stores an array of characters, used to store the characters contained within the description string of a channel
highByte	Byte	Stores the higher byte of an integer value
lowByte	Byte	Stores the lower byte of an integer value
Value	Int	Stores the value field of a channel
Text	String	Stores the description field of a channel
Min	Int	Stores the minimum field of a channel
Max	Int	Stores the maximum field of a channel
valString	String	Stores the value of a channel, converted to a string, to write to EEPROM
J	Int	Used in loops
newChannels	channelDict[]	Stores a filtered set of the channels array, depending on whether right or left is clicked, as per HCI
downPoint	Bool	Stores whether the down pointer should be displayed

upPoint	Bool	Stores whether the up pointer should be displayed
displayText0	String	Stores the text to be displayed for the top line of the lcd, before it is displayed
displayText1	String	Stores the text to be displayed for the bottom line of the lcd, before it is displayed
Alpha	Char[]	Stores the letters of the alphabet in a character array, used to assign letters to the channels

Variable	Data Type	Description

name0x0	byte	Stores the byte data of up arrow
name0x1	byte	Stores the byte data of down arrow
X	String	Used to store output of serial.readline()
state_e	Enum	Used to store the available states in the FSM
Channel_dict	struct	Used for storing the information of each channel, has the fields channel (for storing the letter of the channel), text(for storing the description of the channel), value, maximum and minimum.
channels	Channel_dict[]	Array that holds each channel (of data type Channel_dict).
updown	Int	Stores which button on the Arduino was pressed.
press_time	Long	Stores the time a button on the Arduino has been pressed for (used in WAITING_RELEASE).
last_b	Int	Used to see whether a held button has been released, in conjunction with b and used in released
pointer	Int	Used to keep track of which channel should be displayed on the top line of the Arduino.
over_min	Bool	Used to signify whether an entered value is over the minimum of that channel
over_max	Bool	Used to signify whether an entered value is under the maximum of that channel
Alpha	Char[]	Stores the letters of the alphabet in a character array, used to assign letters to the channels

displayText0	String	Stores the text to be displayed for the top line of the lcd, before it is displayed
displayText1	String	Stores the text to be displayed for the bottom line of the lcd, before it is displayed
Over_min	Bool	Boolean which is false if the user inputted value is above the minimum of the channel or true if it is below.
Over_max	Bool	Boolean which is false if the user inputted value is below the maximum of the channel or true if it is above.
X	String	Used to store the input read from the serial monitor within the MAIN state.
textData	String	Used to store the substring of x that represents the value to be added to the desired field, depending on whether 'C','V','X' or 'N' is pressed
Check_max	Int	Stores the user inputted maximum value.
Check_min	Int	Stores the user inputted minimum value.

```
605 void set_alpha(Channel_dict channels[],int pos){
606   char alpha[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
607   channels[pos].channel=alpha[pos];
608 }
```

Simply alters channels array to give the desired channel a channel letter.

3 Debugging

I have used Serial.println() statements to check whether certain functions have been entered.

I also use them to check the value of certain variables, such as when seeing which address is being written to in eeprom.

These are found scattered in my code as commented out serial.println() statements.

4 Reflection

When reflecting on my code, I bring my mind back to the planning stage; of which there was very little, aside from a plan for the FSM states. On the one hand, it meant I had a working FSM pretty quickly to then model the rest of my code quickly and easily around. On the other hand, my code is sloppy, it gets the job done but not always in the most readable or efficient ways, and as such makes expansion upon it or bug fixing inherently arduous. Such was the plight I suffered when attempting to add a scroll feature to the screen output. This expansion was better suited to a screen output that was itself a state, but as I decided to not take that approach earlier on, it meant that I could now not retrofit this as a solution, but instead had to have a much clumsier implementation of scroll than I would have liked. My main criticism for my code is its inconsistency in both its efficiency and readability, both affecting the ability to expand the codebase.

I'm quite happy with the overall product but do have a few aspects I could have done better. I managed to get together the main functionality quite easily, deciding early on that I would have the states that I'm using now. The system of using an array of structs proved quite useful and helped make the user input easy to add to these channels, as I could simply select the desired element of each desired structure. Using built-in functions such as toInt and substring also ensured my code was robust and worked consistently when it came to using those. However, I did run into some issues later on. When implementing RECENT, I ran into an issue where my program ceased to work at all. When this happened, I didn't get any errors, so assumed it was an issue elsewhere. It was caused due to too much data being written to the finite space in EEPROM, so I had to compromise and write less values in RECENT. I would have liked to fix this by finding a more efficient way to store these values into EEPROM. I also did not have enough time to implement the EEPROM extension. I had the working EEPROM read and write functions from doing RECENT, but due to the way I set up my channels array it would require some restructuring to my already completed main code and I simply did not have enough time to do and test this. I would fix this by declaring my channels array and assigning them all channel letters on setup. I would then read the contents of EEPROM and write them to this array. I furthermore did not have much time to test my code in its entirety, therefore as I've programmed each part it was working at the time I programmed it, however it may not work once changes have been made to the code.

5 UDCHARS

Changes:

- Just changed in the lcd output to add bool expressions and call the pointers when necessary.

pls_end_this

```
#include <Wire.h>
#include <Adafruit_RGBLCDShield.h>
#include <utility/Adafruit_MCP23017.h>
#include <EEPROM.h>

Adafruit_RGBLCDShield lcd = Adafruit_RGBLCDShield();
#define UP_CHAR 0
#define DOWN_CHAR 1
byte name0x0[] = { B00100, B01110, B11111, B00100, B00100, B00100, B00100, B00100 };
byte name0x1[] = { B00100, B00100, B00100, B00100, B00100, B11111, B01110, B00100 };

//then can do channels_maxmin(alpha_index('A'),56);, etc.
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
lcd.begin(16, 2);
lcd.setBacklight(5);
lcd.clear();
lcd.createChar(UP_CHAR,name0x0);
 lcd.createChar(DOWN_CHAR,name0x1);
 //eeprom_read(0);
  //for resetting eeprom
  /*for (int i = 0 ; i < EEPROM.length() ; i++) {
```

```
void lcd_channels(Channel_dict channels[],int pointer){
  bool downPoint=false;
  char alpha[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
  bool upPoint=false;
  //change this part to not change channel element and instead just print the letter
  /*if(channels[pointer].channel==NULL){
    channels[pointer].channel=alpha[pointer];
  }
  if(channels[pointer+1].channel==NULL){
    channels[pointer+1].channel=alpha[pointer+1];
  }*/
  if(pointer+1!=25){
    downPoint=true;
  };
  if(pointer!=0){
    upPoint=true;
  }
  lcd.clear();
  lcd.setCursor(0,0);
  if(upPoint==true){
    //lcd.createChar(0,name0x0);
    //lcd.setCursor(0,0);
    //Serial.println("in up char");
    lcd.print(char(UP_CHAR));
  }
  else{
    lcd.print(" ");
  }
  //lcd.print(channels[pointer].channel);
  lcd.print(alpha[pointer]);
  //lcd.print(" ");
  //lcd.print(channels[pointer].value);
```

```
  right_align(channels,channels[pointer].value);
  lcd.print(" ");
  //lcd.print("AVG ");
  lcd.print(return_avg(pointer));
  String displayText0=(channels[pointer].text);
  displayText0.trim();
  scroll(displayText0,0);
  //lcd.print(displayText0);
  lcd.setCursor(0,1);
  if(downPoint==true){
    //lcd.createChar(1,name0x1);
    //lcd.setCursor(0,1)
    lcd.print(char(DOWN_CHAR));
  }
  else{
    lcd.print(" ");
  }
  //lcd.print(channels[pointer+1].channel);
  lcd.print(alpha[pointer+1]);
  //lcd.print(" ");
  //lcd.print(channels[pointer+1].value);
  right_align(channels,channels[pointer+1].value);
  lcd.print(" ");
  //lcd.print("AVG ");
  lcd.print(return_avg(pointer+1));
  String displayText1=(channels[pointer+1].text);
  scroll(displayText1,1);
  displayText1.trim();
  //lcd.print(displayText1);
  //lcd.print("UR MOM");

}
```

With this extension, I used an online tool to design the up and down arrows and provide the relevant bit data to store them as the variables name0x0 and name0x1 respectively.

I then instantiate these as characters on the lcd in the setup function, defining them as the constants UP_CHAR and DOWN_CHAR, so they can be used as characters on the lcd display.

In my function, lcd_channels, which outputs the channels and their data to the lcd screen, I then define two new variables, downPoint and upPoint, these are set to false at the top of the function and are used to tell whether the up arrow and/or down arrow need to be displayed on the screen.

To achieve this, I implement two if statements. The first; checking if the channel on the first line is not A, which is done by checking if the pointer passed to the function is not 0. This then sets upPoint to true if the statement holds, thus meaning the up pointer appears in front of any channel that is not the first channel.

The opposite is done with downPoint, checking if the pointer is not 25, which is the position of channel Z, and only outputting the down pointer if that statement holds.

Variable	Data Type	Description
upPoint	Bool	Used to see whether the up pointer is displayed; true if it should be, false otherwise. True when the channel on the top line is not the 0^{th} channel.
downPoint	Bool	Used to see whether the down pointer is displayed; true if it should be, false otherwise. True when the channel on the bottom line is not the last (26^{th}) channel.
UP_CHAR	Constant	Stores the byte data of the up pointer character.

DOWN_CHAR	Constant	Stores the byte data of the down pointer character.
Name0x0	Byte	Stores the byte data of the up pointer character.
Name0x1	Byte	Stores the byte data of the down pointer character.

6 FREERAM

```
45 void update_display_two_line(String str,String str1){
46  lcd.clear();
47  lcd.print(str);
48  lcd.setCursor(0,1);
49  lcd.print(str1);
50 }
```

```
51 #ifdef __arm__
52 // should use uinstd.h to define sbrk but Due causes a conflict
53 extern "C" char* sbrk(int incr);
54 #else // __ARM__
55 extern char *__brkval;
56 #endif // __arm__
57 int freeMemory() {
58 char top;
59 #ifdef __arm__
60 return &top - reinterpret_cast<char*>(sbrk(0));
61 #elif defined(CORE_TEENSY) || (ARDUINO > 103 && ARDUINO != 151)
62 return &top - __brkval;
63 #else // __arm__
64 return __brkval ? &top - __brkval : &top - __malloc_heap_start;
65 #endif // __arm__
66 }
```

```
821  if(updown==BUTTON_SELECT){;
822 // update_display_two_line("F129714",EEPROM.freeMemory());
823 //update_display("F129714");
824 update_display_two_line("F129714",String(freeMemory()));}
825  //update_display("down");
826 }else {
827 int b = lcd.readButtons();
828 // We are looking for buttons that are not pressed
829 // now and were pressed last time.
830 // The logic below is this. ~ means not.
831 // so "NOT now AND last_time"
832 int released = ~b & last_b;
833 last_b = b; // Save
834 // Mask off the bit we are interested in and if
835 // it's been released then change state
836 if (released & updown) {
837  lcd_channels(channels,pointer);
838  //Serial.println("waiting press");
839 state = MAIN;//WAITING_PRESS
840 }
841 }}
```

I define the function freeMemory() as provided from the week 3 lab worksheet. I then call this function, and convert the output to String, then display said output to the screen.

This output is put in the WAITING_RELEASE state with the student id output and required a new function to be made to output two lines at once instead of just the student id.

The function, called update_display_two_line, simply takes two strings as parameters, clears the lcd and outputs the first string on the first line and the second string on the second line.

This function is only active whilst the select button is held as per the WAITING_RELEASE state. When the button is let go the screen will return to normal as the lcd_channels() function is called to output the normal channel data to the screen again and the state is changed back to main.

7 HCI

Changes:

- Made 4 functions for it and had to change WAITING_PRESS

Explanation

```
int count_right_rule(Channel_dict channels[]){
  int j=0;
  for(int i=0;i<=26;i++){
    if(channels[i].value>channels[i].maximum){
      j+=1;
    }
  }
  return j;
}
Channel_dict * make_array_right_rule(Channel_dict channels[],int count,int pointer){
  Channel_dict newChannels[count];
  //malloc(newChannels);
  int j=0;
  for(int i=0;i<=26;i++){
    if(channels[i].value>channels[i].maximum){
      newChannels[j]=channels[i];
      j+=1;
    }
  }
  //return newChannels;
  if(count!=0){
  lcd_channels_len(newChannels,count,pointer);}
  else{
    return;
  }
}
```

```
int count_left_rule(Channel_dict channels[]){
  int j=0;
  for(int i=0;i<=26;i++){
    if(channels[i].value<channels[i].minimum){
      j+=1;
    }
  }
  return j;
}
Channel_dict * make_array_left_rule(Channel_dict channels[],int count,int pointer){
  Channel_dict newChannels[count];
  //malloc(newChannels);
  int j=0;
  for(int i=0;i<=26;i++){
    if(channels[i].value<channels[i].minimum){
      newChannels[j]=channels[i];
      j+=1;
    }
  }
  //return newChannels;
  if(count!=0){
  lcd_channels_len(newChannels,count,pointer);}
  else{
    return;
  }
}
//void scroll(String displayText1,int row){//change to account for two simultaneously on screen
```

In WAITING_PRESS

```
if (pressed & (BUTTON_UP | BUTTON_DOWN | BUTTON_SELECT |BUTTON_RIGHT|BUTTON_LEFT)) {
  scrollPos=0;
// now just checking for up
if (pressed & (BUTTON_UP)) {
 if(pointer!=0){
pointer-=1;}
if(refine_right==true){
 int len=count_right_rule(channels);
 make_array_right_rule(channels,len,pointer);
}
else if(refine_left==true){
 int len=count_left_rule(channels);
 make_array_left_rule(channels,len,pointer);
}
else{
lcd_channels(channels,pointer);}
}
if(pressed&BUTTON_SELECT){
 //write_my_eeprom(channels);
 //read_my_eeprom(2);
 update_display("Student ID");
}
if(pressed&BUTTON_LEFT){
 //update_display("left");
 refine_left=not refine_left;
 if(refine_left==true){
   pointer=0;
   int len=count_left_rule(channels);
   if(len==0){
     update_display("No Values");
     delay(50);
     refine_left=false;
   }
    make_array_left_rule(channels,len,pointer);
```

```
911  }
912  else{
913    lcd_channels(channels,pointer);
914  }
915  }
916 if(pressed&BUTTON_RIGHT){
917  //update_display("right");
918  refine_right=not refine_right;
919  if(refine_right==true){
920    pointer=0;
921    int len=count_right_rule(channels);
922    if(len==0){
923      update_display("No Values");
924      delay(50);
925      refine_right=false;
926    }
927     make_array_right_rule(channels,len,pointer);
928    //lcd_channels_len(newChannels,len,pointer);
929  }
930  else{
931    lcd_channels(channels,pointer);
932  }
933  }
934 if(pressed&BUTTON_DOWN) {
935 if(refine_right){
936  int len=count_right_rule(channels);
937  if(pointer+1<(len-1)){
938  pointer+=1;
939 }
940  make_array_right_rule(channels,len,pointer);
941  //Serial.println("HELLLLOOOOO");
942  //lcd_channels_len(newChannels,len,pointer);
943 }
944 else if(refine_left){
```

```
945  int len=count_left_rule(channels);
946  if(pointer+1<(len-1)){
947  pointer+=1;
948 }
949  make_array_left_rule(channels,len,pointer);
950  //Serial.println("HELLLLOOOOO");
951  //lcd_channels_len(newChannels,len,pointer);
952 }
953 else{
954  if(pointer+1!=25){
955  pointer+=1;
956 }
957 lcd_channels(channels,pointer);}
958 }
959 updown = pressed;
960 press_time = millis();
961 state = WAITING_RELEASE;
962 }}
963 break;
964 }}
965
```

To implement HCI I had to create 4 new functions and make some adjustments to the WAITING_PRESS state.

The functions come in pairs, count_right_rule and make_array_right_rule and their complementary pair, count_left_rule and make_array_left_rule.

Count_right_rule takes the channels array as a parameter and uses a for loop to count the number of channels where the value of the channel is higher than the maximum. It then returns this total number. Thereby, returning the number of channels that should be displayed in HCI when right is pressed.

This total is then used after it is called, where it is used as a parameter for make_array_right_rule. It is used to create an array of the size of this total. Make_array_right_rule then goes through all the channels in the channels array and adds any channel that obeys the right rule (of the value being above maximum) to the array newChannels. It then passes this array to lcd_channels to be output to the lcd.

Count_left_rule and make_array_left_rule perform much the same but instead of checking against the right rule, of value being above maximum, it checks against the left rule, of the value being below the minimum.

I define 3 new variables here, refine_left, which is used to tell the display when to display the HCI left, refine_right, which is used to tell the display to display the HCI right. I also define len, which is simply the channels array passed into either count_left_rule or count_right_rule, so this length can then be passed to either make_array_left_rule or make_array_right_rule respectively.

Now, in the WAITING_PRESS state, when the right button is pressed, refine_right is toggled (false it was true or true if it was false). If refine_right is true, this then passes channels array to count_right_rule, to get the length of the array to be displayed in HCI, and also resets the pointer to 0. The pointer is set to 0 here so that when HCI is displayed, it will display from its first element instead of a potentially non-existent element. It then checks if the variable len is 0 or not, and outputs that there are no values if the length is 0. Otherwise, it will pass channels array, len and the pointer to the make_array_right_rule function. On the other hand, if refine_right is false, the normal lcd_channels function is called, displaying the channels to the lcd as normal.

Pressing the left button has much the same functionality, but instead changing the variable refine_left instead of refine_right and calling the function count_left_rule to return the length and calling make_array_left_rule instead of make_array_right_rule.

Up/ Down Arrow

Changes also had to be made to the logic for the up and down button presses. Both these buttons have the same changes made to them which I will explain here. Instead of just incrementing/decrementing the pointer and calling lcd_channels as usual, refine_left and refine_right are now utilised here.

Down arrow

In the button down press, it will first check if either refine_right or refine_left are true. If refine_right is true, it will then store the call of count_right_rule in the len variable, it will then use this output to check if incrementing the pointer will still result in a value that is in the new array, by checking if pointer+1 is less than the length of the array. If not, the pointer is not incremented, otherwise, it is. It will then call the make_array_right_rule function.

When refine_left is true, it will first store the call of count_left_rule to the len variable. It will then check if incrementing the pointer will result in a value that is not in the new array, by checking if pointer+1 is less than the length of the array. If it is, then the pointer is incremented, otherwise it is not. It will then call the make_array_left_rule.

If both refine_left and refine_right are false, then the lcd_channels function is called to display all channels as usual.

Up Arrow

This uses much the same logic as the down arrow. The only difference is that the pointer is decremented. This thus changes the way the pointer is checked if refine_left or refine_right are

true. Instead of checking if incrementing the pointer is larger than the length of the array, it instead checks if the pointer is equal to 0, as if it was it would result in a value not in the array.

Variable	Data Type	Description
newChannels	Channel_dict[]	Stores a filtered version of the channels array. Filtered by the right filter or left filter (below min or above max) depending on button pressed.
Refine_left	Bool	Used to see if HCI left should be displayed. True if it should be, false otherwise. Toggled on and off by pressing left button.
Refine_right	Bool	Used to see if HCI right should be displayed. True if it should be, false otherwise. Toggled on and off by pressing right button.
Len	Int	Stores the length of returned filtered HCI list. Used to check pointer operations (so when pressing up and down can't select a value that doesn't exist) and when displaying up and down pointers (not displayed for 1st or last values)

8 EEPROM

Mechanism employed to not include in max and min operations is that max and min operations are only done when user inputs a ‘V’ operation.

Look at the commented code under each “EEPROM PROOF OF CONCEPT”.

```
/*EEPROM PROOF OF CONCEPT
  struct Channel_dict channel;
Channel_dict channels[26]={};*/
 //make a function to look up alphabet's index
 //then can do channels_maxmin(alpha_index('A'),56);, etc.
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
lcd.begin(16, 2);
 lcd.setBacklight(5);
 lcd.clear();
 lcd.createChar(UP_CHAR,name0x0);
  lcd.createChar(DOWN_CHAR,name0x1);
  /*EEPROM PROOF OF CONCEPT
    char alpha[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};

for(int i=0;i<=26;i++){
  channels[i].channel=alpha[i];
  //channels[i].value=eeprom_read_value(i);
  if(eeprom_read_text(i).substring(1,2)!=NULL){
    Serial.println(eeprom_read_text(i));
  channels[i].text=eeprom_read_text(i);}
  else{
    channels[i].text="DEFAULT";
  }
  channels[i].maximum=eeprom_read_max(i);
  channels[i].minimum=eeprom_read_min(i);*/
}
  //eeprom_read(0);
   //for resetting eeprom
   /*for (int i = 0 ; i < EEPROM.length() ; i++) {
    EEPROM.write(i, 0);
  }*/
```

Here, I first declare the channels array as global so both loop() and setup() can use it, as now setup will need to access it in order to read values from eeprom at startup.

Here I assign each channel it’s channel letter in alphabetical order first. I then read it’s value from eeprom and set it’s value field as the value that’s been read. I then do the same with it’s text description, so long as the text read is not blank (meaning the text field should be DEFAULT). It’s maximum and minimum are then also read from EEPROM and set in their respective fields. It does this for every channel on setup as a for loop is used.

```
/*EEPROM PROOF OF CONCEPT
void eeprom_read(int addr){
  addr*=62;
  char charArray[15];
   Serial.println(char(EEPROM.read(addr)));
   byte highByte=EEPROM.read(addr+=1);
  byte lowByte=EEPROM.read(addr+=1);
  int value=convert_byte_to_int(highByte,lowByte);
  Serial.println(value);
   for(int i=0;i<15;i++){
    charArray[i]=EEPROM.read(addr+=1);
   }
   //Serial.println(char(charArray[0]));
   Serial.println(String(charArray));
   Serial.println(addr);
   highByte=EEPROM.read(addr+=1);
   lowByte=EEPROM.read(addr+=1);
   value=convert_byte_to_int(highByte,lowByte);
   Serial.println(value);
   highByte=EEPROM.read(addr+=1);
   lowByte=EEPROM.read(addr+=1);
   value=convert_byte_to_int(highByte,lowByte);
   Serial.println(value);
   Serial.print("addr count: ");
   Serial.println(addr);
   for(int i=0;i<=64;i++){
    if(EEPROM.read(addr+1)!=0){
    highByte=EEPROM.read(addr+=1);
    lowByte=EEPROM.read(addr+=1);
    value=convert_byte_to_int(highByte,lowByte);
    Serial.println(value);}
    else{
      addr+=2;
    }
   }

}
```

This part is just used for testing to check where the respective address positions of each element is in EEPROM.

```
int eeprom_read_value(int addr){
  addr*=62;
  char charArray[15];
   byte highByte=EEPROM.read(addr+=1);
  byte lowByte=EEPROM.read(addr+=1);
  int value=convert_byte_to_int(highByte,lowByte);
  return value;

}

String eeprom_read_text(int addr){
  addr*=62;
  char charArray[15];
  addr+=2;
  for(int i=0;i<15;i++){
    charArray[i]=EEPROM.read(addr+=1);
   }
   //Serial.println(char(charArray[0]));
   return String(charArray);

}

int eeprom_read_max(int addr){
  addr*=62;
  addr+=17;
   int highByte=EEPROM.read(addr+=1);
   int lowByte=EEPROM.read(addr+=1);
   int value=convert_byte_to_int(highByte,lowByte);
   return value;

}
```

```
162 int eeprom_read_min(int addr){
163   addr*=62;
164   addr+=19;
165    int highByte=EEPROM.read(addr+=1);
166    int lowByte=EEPROM.read(addr+=1);
167    int value=convert_byte_to_int(highByte,lowByte);
168    return value;
169 
170 }
```

Here are the functions used to read each of the desired fields from EEPROM. I multiply the given address by 62 as this is the amount of bytes for each channel. I then += to the address to skip over other unnecessary data in the EEPROM. For example,addr+=2 is used in eeprom_read_text to skip over the channel and value fields stored in EEPROM. Thus only returning the text field.

9 RECENT

Compromises/ Changes:

- Had to compromise and give each channel only 40 bytes to store the data in, so that's only 20 numbers rather than 64
- Had to change adding new values/ descriptions, as had to write these to EEPROM now.
- Had to change lcd_channels output

Explanation

First, I will describe the layout of the EEPROM used in this task.

The EEPROM is split into a section of one byte, for the channel letter, followed by a two-byte section, for the value stored in the channel. This is then followed by a 15-byte section, for the 15-character string of the description. Next, we then have two two-byte sections for the maximum and minimum respectively. This is next followed by a 40-byte section for the past 20 values stored in that channel, to be used in the calculation of the average.

pls_end_this §

```
151
152 void write_my_eeprom_single(Channel_dict channels[],int i){
153   int j=i;
154   j=j*62;
155   j-=1;
156   char alpha[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
157   //for(int i=0;i<=26;i++){
158     //Serial.println((sizeof(channels)/sizeof(Channel_dict)));
159     char channel=channels[i].channel;
160     /*if(channel==NULL){
161       channel=alpha[i];
162       //continue;
163     }*/
164     int value=channels[i].value;
165     String text=channels[i].text;
166     int max=channels[i].maximum;
167     int min=channels[i].minimum;
168     EEPROM.update(j+=1,channel);
169     String valString=String(value);
170     EEPROM.update(j+=1,return_high_byte(value));
171     EEPROM.update(j+=1,return_low_byte(value));
172     /*if(length_of(text)<15){
173       EEPROM.update(j+=1,length_of(text));
174     }
175     else{
176     EEPROM.update(j+=1,char(1));
177     EEPROM.update(j+=1,char(5));}
178     */
179     for(int k=0;k<15;k++){
180       EEPROM.update(j+=1,char(text[k]));
181     }
182     EEPROM.update(j+=1,return_high_byte(max));
183     EEPROM.update(j+=1,return_low_byte(max));
184     EEPROM.update(j+=1,return_high_byte(min));
```

```
185   EEPROM.update(j+=1,return_low_byte(min));
186   //Serial.print("addr count write: ");
187   //Serial.println(j);
188   //byte pastNums[40];
189   Avg_return pastNums;
190   pastNums=read_eeprom_avg_bytes(i);
191   /*Serial.println("PAST NUMS");
192   for(int k=0;k<40;k++){
193     Serial.println("OUTPUT OF READ AVG");
194     Serial.println(pastNums.avgArray[k]);
195     Serial.println("-------------------");
196   }
197   Serial.println("----------------");*/
198   int pos=return_first_empty(pastNums);
199   pos*=2;
200   /*Serial.print("POSITION: ");
201   Serial.println(pos);
202   Serial.println(value);*/
203   //if(pastNums.avgArray[pos]<1&&pastNums.avgArray[pos+1]<1){
204   pastNums.avgArray[pos]=return_high_byte(value);
205   pastNums.avgArray[pos+1]=return_low_byte(value);//}
206   /*for(int k=0;k<40;k++){
207     Serial.println("CHANGES MADE");
208     Serial.println(pastNums.avgArray[k]);
209     Serial.println("-------------------");
210   }*/
211   /*for(int k=0;k<40;k++){
212     //if(k%2!=0){
213       //Serial.println(k);
214     //has 0, 32, then want it after
215     if((pastNums.avgArray[k]<1)&&(pastNums.avgArray[k+1]<1)){
216       /*Serial.println("Writing value");
217       Serial.println(value);
218       Serial.println(return_high_byte(value));
219       Serial.println(return_low_byte(value));
220       byte highByte=pastNums.avgArray[k];
221       byte lowByte=pastNums.avgArray[k+1];
222       int num=convert_byte_to_int(highByte,lowByte);
223       Serial.print("num: ");
224       Serial.println(num);
225       Serial.println(highByte);
226       Serial.println(lowByte);*/
227        /*pastNums.avgArray[k]=return_high_byte(value);
228        pastNums.avgArray[k+1]=return_low_byte(value);
229       break;
230     }
231     }*/
232   //}
233
234   for(int k=0;k<40;k++){
235     //Serial.print("k: ");
236     //Serial.println(pastNums.avgArray[k]);
237     EEPROM.update(j+=1,pastNums.avgArray[k]);
238     //Serial.print("location: ");
239     //Serial.println(j);
240   }
241   //Serial.print("j: ");
242   //Serial.println(j);
243   //write to empty spaces at end of eeprom the averages.
244 //   Serial.println("writes :"+String(writes));
245   /*
246   String maxString=String(max);
247   for(int k=0;k<2;k++){
248     EEPROM.update(j+=1,(char)maxString[k]);
249   }
250   String minString=String(min);
251   for(int k=0;k<2;k++){
252     EEPROM.update(j+=1,(char)minString[k]);
```

This function takes the channels array and the index of the channel to be written to eeprom, as parameters.

I will explain below first how I have split up the EEPROM before discussing how the function works.

I have split the EEPROM as follows: 1 Byte for the channel letter, 2 bytes for the channel value, 15 bytes for the channel text, 2 bytes for the maximum, 2 bytes for the minimum, 40 bytes for the most recent numbers (at 2 bytes per number). I originally had 64 bytes allocated to this, but having this caused unintended errors elsewhere in my program, so I had to compromise and decrease the amount of bytes I could allocate.

The function first sets the integer j equal to i, this will be used as the pointer to EEPROM memory slots. It then multiplies by 62, as 62 is the size of each channel in EEPROM, this way, it is pointing to the start of the memory allocated to this channel. It then subtracts one from j as the function uses +=1 operations, so to point at address 0 from a +=1, the value of j would need to be one less than that.

Next we call the EEPROM.update function with the parameters j+=1 as the location and channel as the value to be written. Using j+=1 increments this value ready for the next write and the value of channel is already a char, which can be written straight to EEPROM without conversion. In addition, using update means that EEPROM is not cleared when we are amending it.

Next for the value, we split this value up into its high byte and low byte, how these functions work will be explained below. These bytes are then written to the next two memory addresses.

Then we use a for loop to parse the text value of the channel, this loop starts from k=0 to 14, for the 15 memory addresses allocated for the text. It then will take the character at the location of k within the text and then convert this to the character data type before writing it to EEPROM using j+=1 to write it to the next available address, doing this until k reaches 14 and the loop ends.

For writing the maximum and minimum we use much the same process as writing the value to EEPROM.

For writing the RECENT values, I will first need to explain the data structure Avg_return and the function read_eeprom_avg_bytes as well as return_first_empty.

```
struct Avg_return{
 byte avgArray[40];
};
```

The structure of Avg_return is defined so read_eeprom_avg_bytes can return an array, avg_return simply contains a byte array of length 40.

```
Avg_return read_eeprom_avg_bytes(int addr){//TODO the issue is here, writing is fine
  addr=addr*62;
  addr+=22;//22
  Avg_return a;
  //byte avgArray[40];
  for(int i=0;i<40;i++){
    //if(i%2!=0){
      //Serial.print("Location: ");
      //Serial.println(addr+i);
      //byte highByte=EEPROM.read(addr+i);
      //byte lowByte=EEPROM.read(addr+i+1);
      //a.avgArray[i]=highByte;
      a.avgArray[i]=EEPROM.read(addr+i);
      //Serial.print("High byte: ");
      //Serial.println(highByte);
      //Serial.print("Low byte: ");
      //Serial.println(lowByte);
      /*Serial.println("IN RETURN AVERAGE BYTES");
      Serial.println(a.avgArray[i]);
      Serial.println("-----------------------");*/
   // }
  }
  return a;
}
```

This function takes the index of the channel as a parameter. It then multiplies this address by 62 (as this is the size of each channel in EEPROM) to get the address of the channel in EEPROM and adds 22 (which is the size of the channel in EEPROM without the RECENT values). It also defines a Avg_return structure as ‘a’.

It then uses a for loop to read the next 40 values in EEPROM. It appends these values to the avgArray element of ‘a’, and returns ‘a’ when the loop ends. Thus returning an array of the RECENT values in EEPROM of a desired channel.

```
int return_first_empty(Avg_return a){
  int numArray[64];
  for(int i=0;i<40;i++){
    if(i%2==0){
    byte highByte=a.avgArray[i];
    byte lowByte=a.avgArray[i+1];
    int num=convert_byte_to_int(highByte,lowByte);
    /*Serial.println(i);
    Serial.println("CONVERTED NUM");
    Serial.println(highByte);
    Serial.println(lowByte);
    Serial.println(num);
    Serial.println("-------------");*/
    numArray[i/2]=num;
    }
  }
  for(int i=0;i<64;i++){
    if(numArray[i]<1){//==0
      return i;
    }
  }
  return 0;
}
```

This takes an instance of Avg_return as a parameter ‘a’ and then defines an integer array of length 64 under the variable numArray. It loops through the 40 values in ‘a’ with a loop increment of 2 (only when I is a multiple of 2 etc. 0,2,4,6, will the code in the loop run). It then takes the value in ‘a’s avgArray element at the position i and assigns it to the variable highByte and assigns the value at i+1 to lowByte. It then passes the highByte and lowByte to the convert_byte_to_int and stores the result of this in the variable num. As highByte and lowByte together took two places in the array and num only takes one, it can be surmised that num can be stored in the numArray at half the position of these values, which is done in numArray[i/2]=num.

It then loops through the numArray and returns the position of the first occurrence of the number 0, which is the first ‘empty’ position in the array.

Now back to the write_my_eeprom_single function.

It defines pastNums of type Avg_return and assigns this to the value returned by the function read_eeprom_avg_bytes with the parameter of the index of the current channel. This then means pastNums contains an array of all the RECENT elements of the current channel.

Next, the value returned from return_first_empty with the parameter of pastNums, is then assigned to the integer pos. This is then multiplied by 2, as the byte array contained in pastNums is double the length of the integer array used in return_first_empty.

It then uses pos to set the values at the positions pos and pos+1 in the avgArray element of pastNums to the high byte and low byte of the channel’s value respectively.

It finally loops through pastNums’ avgArray, again using j+=1 to update each EEPROM address with it’s RECENT value.

EXPLAIN RETURN HIGH BYTE AND RETURN LOW BYTE.

```
100
101 byte return_high_byte(int num){
102   //Serial.print("num: ");
103   //Serial.println(num);
104    //byte highByte = ((num >> 8) & 0xFF);
105    byte highByte=highByte(num);
106    //Serial.print("calc: ");
107    //Serial.println(highByte);
108    return highByte;
109 }
110
111
112 byte return_low_byte(int num){
113 //byte lowByte = ((num >> 0) & 0xFF);
114 byte lowByte=lowByte(num);
115 return lowByte;}
116
```

The functions return_high_byte and return_low_byte simply return the values of the number given to them using the in-built highByte and lowByte functions respectively.

Variable	Data Type	Description
Channel	Char	Stores the character of the channel name (which is a letter A-Z)
Text	String	Stores the channel description of the given channel.
Max	Int	Stores the maximum of the given channel.
Min	Int	Stores the minimum of the given channel.
highByte	Byte	Stores the higher byte of a given value.
lowByte	Byte	Stores the lower byte of a given value.
Value	Int	Stores the value of the given channel.
J	Int	Used to increment which eeprom address is written to.

Pos	Int	Used to store the position at which the new value can be entered to the array of recent values.
pastNums	Avg_Return	Stores the eeprom data of the recent values
A	Avg_Return	Used to store the eeprom data of the recent values
numArray	Int[64]	Used to store the integer representations of the recent values.

10 NAMES

Already had this implemented in base code

I store the channel names as part of the struct Channel_dict data structure, it is defined as the text field and is set to “DEFAULT” if there is no value currently written to it.

```
void lcd_channels(Channel_dict channels[],int pointer){
  bool downPoint=false;
  char alpha[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
  bool upPoint=false;
  //change this part to not change channel element and instead just print the letter
  /*if(channels[pointer].channel==NULL){
    channels[pointer].channel=alpha[pointer];
  }
  if(channels[pointer+1].channel==NULL){
    channels[pointer+1].channel=alpha[pointer+1];
  }*/
  if(pointer+1!=25){
    downPoint=true;
  };
  if(pointer!=0){
    upPoint=true;
  }
  lcd.clear();
  lcd.setCursor(0,0);
  if(upPoint==true){
    //lcd.createChar(0,name0x0);
    //lcd.setCursor(0,0);
    //Serial.println("in up char");
    lcd.print(char(UP_CHAR));
  }
  else{
    lcd.print(" ");
  }
  //lcd.print(channels[pointer].channel);
  lcd.print(alpha[pointer]);
  //lcd.print(" ");
  //lcd.print(channels[pointer].value);
  right_align(channels,channels[pointer].value);
  lcd.print(" ");
  String displayText0=(channels[pointer].text);
  displayText0.trim();
  lcd.print(displayText0);
  lcd.setCursor(0,1);
```

```
  if(downPoint==true){
    //lcd.createChar(1,name0x1);
    //lcd.setCursor(0,1)
    lcd.print(char(DOWN_CHAR));
  }
  //lcd.print(channels[pointer+1].channel);
  lcd.print(alpha[pointer+1]);
  //lcd.print(" ");
  //lcd.print(channels[pointer+1].value);
  right_align(channels,channels[pointer+1].value);
  lcd.print(" ");
  String displayText1=(channels[pointer+1].text);
  displayText1.trim();
  lcd.print(displayText1);

}

void right_align(Channel_dict channels[],int val){
  Serial.println(val);
  if(val<10){
    lcd.print("  ");
    lcd.print(val);
  }
  else if(val<100 && val>=10){
    lcd.print(" ");
    lcd.print(val);
  }
  else if(val<1000 && val>=100){
    lcd.print(val);
  }
  else{
    lcd.print("ERR");
  }
}
```

To output the NAMES values, I make amendments in the lcd_channels function, which the program uses to output the channels’ data to the lcd screen. First, I simply print a space, then define a String (displayText0 for the top line and displayText1 for the bottom line), which takes the .text field of the channel at the position of the pointer (for the top line) or pointer+1 (for the bottom line). I then trim this string, to remove any excess whitespace and then print this to the lcd.

Variable	Data Type	Description

displayText0	String	Stores the channel description of the channel to be displayed on the top line of the lcd.
displayText1	String	Stores the channel description of the channel to be displayed on the bottom line of the lcd.

11 SCROLL

```
491 void lcd_channels(Channel_dict channels[],int pointer){
492   bool downPoint=false;
493   char alpha[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
494   bool upPoint=false;
495   //change this part to not change channel element and instead just print the letter
496   /*if(channels[pointer].channel==NULL){
497     channels[pointer].channel=alpha[pointer];
498   }
499   if(channels[pointer+1].channel==NULL){
500     channels[pointer+1].channel=alpha[pointer+1];
501   }*/
502   if(pointer+1!=25){
503     downPoint=true;
504   };
505   if(pointer!=0){
506     upPoint=true;
507   }
508   lcd.clear();
509   lcd.setCursor(0,0);
510   if(upPoint==true){
511     //lcd.createChar(0,name0x0);
512     //lcd.setCursor(0,0);
513     //Serial.println("in up char");
514     lcd.print(char(UP_CHAR));
515   }
516   else{
517     lcd.print(" ");
518   }
519   //lcd.print(channels[pointer].channel);
520   lcd.print(alpha[pointer]);
521   //lcd.print(" ");
522   //lcd.print(channels[pointer].value);
523   right_align(channels,channels[pointer].value);
524   lcd.print(" ");
```

```
525   //lcd.print("AVG ");
526   lcd.print(return_avg(pointer));
527   String displayText0=(channels[pointer].text);
528   displayText0.trim();
529   scroll(displayText0,0);
530   //lcd.print(displayText0);
531   lcd.setCursor(0,1);
532   if(downPoint==true){
533     //lcd.createChar(1,name0x1);
534     //lcd.setCursor(0,1)
535     lcd.print(char(DOWN_CHAR));
536   }
537   else{
538     lcd.print(" ");
539   }
540   //lcd.print(channels[pointer+1].channel);
541   lcd.print(alpha[pointer+1]);
542   //lcd.print(" ");
543   //lcd.print(channels[pointer+1].value);
544   right_align(channels,channels[pointer+1].value);
545   lcd.print(" ");
546   //lcd.print("AVG ");
547   lcd.print(return_avg(pointer+1));
548   String displayText1=(channels[pointer+1].text);
549   scroll(displayText1,1);
550   displayText1.trim();
551   //lcd.print(displayText1);
552   //lcd.print("UR MOM");
553
554 }
555
```

Had to make a new function, change lcd_channels and alter WAITING_PRESS

I added calls to the scroll function in lcd_channels, passing the text from NAMES, that needs to be printed on that line, and the row it is on to the scroll function.

```
324
325 void scroll(String displayText1,int row){//change to account for two simultaneously on screen
326   /*if(scrollPos>displayText1.length()){
327     scrollPos=0;
328   }*/
329   if (displayText1.length()>6){
330     lcd.setCursor(10,row);
331     //return partial string between x and the end of the string and change every half second
332     lcd.print("          ");
333     lcd.setCursor(10,row);
334     if(row==0){
335     //lcd.print(partial_string(displayText1,scrollPos,displayText1.length()-1));
336     lcd.print(displayText1.substring(scrollPos));
337     scrollPos+=1;}
338     else{
339       //lcd.print(partial_string(displayText1,scrollPos-1,displayText1.length()-1))
340       lcd.print(displayText1.substring(scrollPos));
341       scrollPos1+=1;
342     }
343   }
344 }
345
```

First I will explain why I use displayText1.length()>6. This is explained simply by the fact that only 6 characters can be displayed on the screen, so if the text is less than this, it is unnecessary to scroll it, so it is simply output as usual if this is the case.

Otherwise, the cursor will be set to the position that the text should start at which is (10,row) with row being the row (either 0 or 1) that the text is to output to. A long sequence of blanks is then output, to clear the 6 characters that were previously displayed on the screen.

A substring of the text is then displayed, this substring starting at scrollPos and ending at the end of the string, if the row is 0; scrollPos is then incremented. If the row is 1 the substring starts at scrollPos1 and ends at the end of the string; scrollPos1 is then incremented.

pls_end_this §

```
#include <Wire.h>
#include <Adafruit_RGBLCDShield.h>
#include <utility/Adafruit_MCP23017.h>
#include <EEPROM.h>

Adafruit_RGBLCDShield lcd = Adafruit_RGBLCDShield();
#define UP_CHAR 0
#define DOWN_CHAR 1
byte name0x0[] = { B00100, B01110, B11111, B00100, B00100, B00100, B00100, B00100 };
byte name0x1[] = { B00100, B00100, B00100, B00100, B00100, B11111, B01110, B00100 };
int scrollPos=0;
int scrollPos1=0;
```

Here I define scrollPos and scrollPos1 as global.

```
void loop() {
  // put your main code here, to run repeatedly:
  //for(int i=0;i<=26;i++){
    //channels[i].channel=alpha[i];};
static enum state_e state = SYNCHRONISATION;
static int updown; // Store which button was pressed
static long press_time; // Store the time the button was pressed
static int last_b = 0; // Store for the last button press
static int pointer = 0;
static bool refine_right=false;
static bool refine_left=false;
//static int val=0;
static bool over_min=false;
static bool over_max=false;
static long scroll_time;//stores time between scrolls
static long scroll_time1;//stores time between scrolls
```

Here I define scroll_time and scroll_time1 as static long within the loop.

```
case WAITING_PRESS:{
  if(((millis()-scroll_time>=500)&&scrollPos!=0)||((millis()-scroll_time>=2000)&&scrollPos==0)){//or statement for scrollPos==0 gives a 2 second delay before scrolling again
    if(scrollPos==0){//so it doesn't flash/ re-output the text after the 2 sec delay
      scrollPos=1;
    }
  if(scrollPos<channels[pointer].text.length()){
    scroll_time=millis();
  scroll(channels[pointer].text,0);}
  else{
    endScroll(channels[pointer].text,0);
    scrollPos=0;
  }}
  if(((millis()-scroll_time1>=500)&&scrollPos1!=0)||((millis()-scroll_time1>=2000)&&scrollPos1==0)){//or statement for scrollPos==0 gives a 2 second delay before scrolling again
    if(scrollPos1==0){//so it doesn't flash/ re-output the text after the 2 sec delay
      scrollPos1=1;
    }
  if(scrollPos1<channels[pointer+1].text.length()){
    scroll_time1=millis();
  scroll(channels[pointer+1].text,1);}
  else{
    endScroll(channels[pointer+1].text,1);
    scrollPos1=0;
  }}
```

Within WAITING_PRESS, I first check if the difference between the time stored in scroll_time, and the current time since the program started, is bigger than or equal to half a second. This is because to achieve the 2 characters per second scroll, my program scrolls one character every half second. This is then joined with an or statement to another argument. This argument uses the same logic as the first to see if two seconds has passed and in addition checks if the scrollPos is 0, this allows the program to show the start of the text for 2 seconds before starting to scroll again.

The next if statement is explained in a comment, but I'll restate it here. It simply sets the scrollPos to 1 if it was 0, as after the delay before starting to scroll again, it should start scrolling from the second character, to avoid re-printing the same first 6 characters of the text.

Variable	Data Type	Description
scrollPos	Int	Stores the position of the first character to displayed on the lcd for the text on the top line. Will increment every half second.
scrollPos1	Int	Stores the position of the first character to displayed on the lcd for the text on the bottom line. Will increment every half second.
Scroll_time	Long	Stores the time since scrollPos was last incremented, used to increment scrollPos every half second.
Scroll_time1	Long	Stores the time since scrollPos1 was last incremented, used to increment scrollPos1 every half second.
displayText1	String	Used to store text passed to the scroll function.