21COA202 Coursework

F120840

Semester 2

Contents

1 FSMs

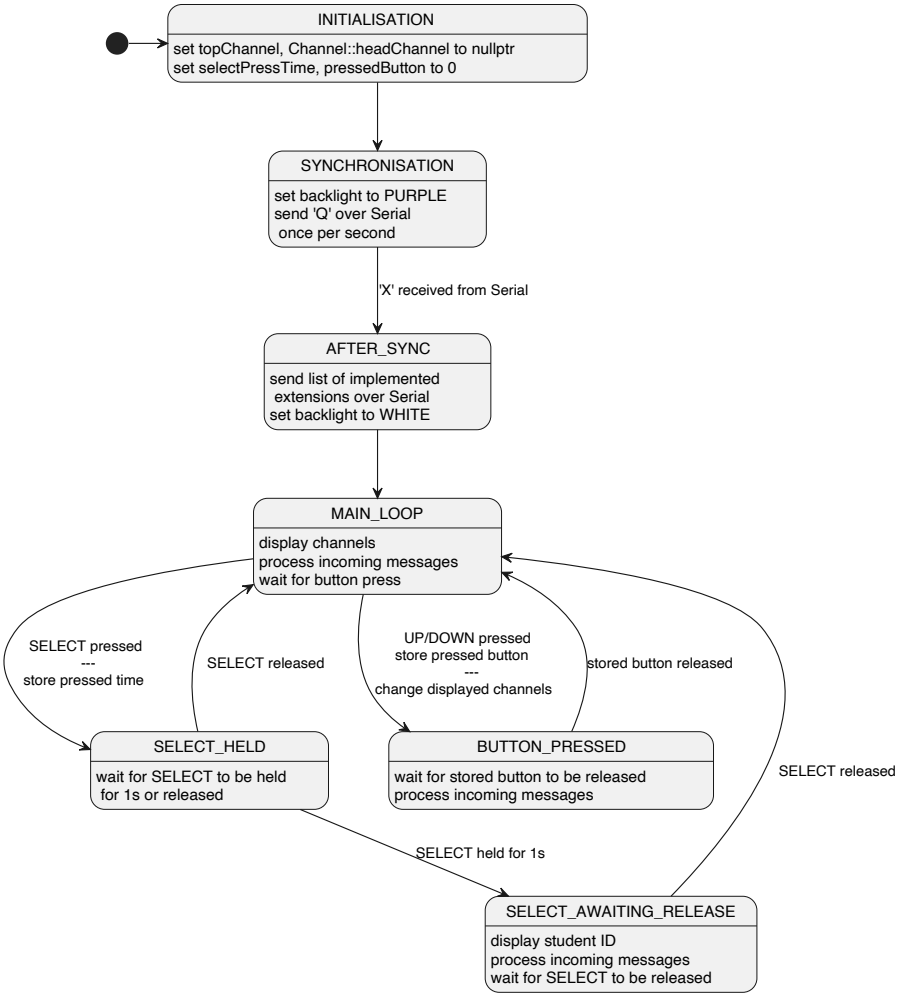

Figure 1: Main FSM

My code includes other FSMs in the extensions. With all extensions implemented, the main FSM changes to (changes highlighted by italics):

INITIALISATION

This state carries out the actions involved in initialising and setting up the system. Once complete, the system enters the `SYNCHRONISATION` state.

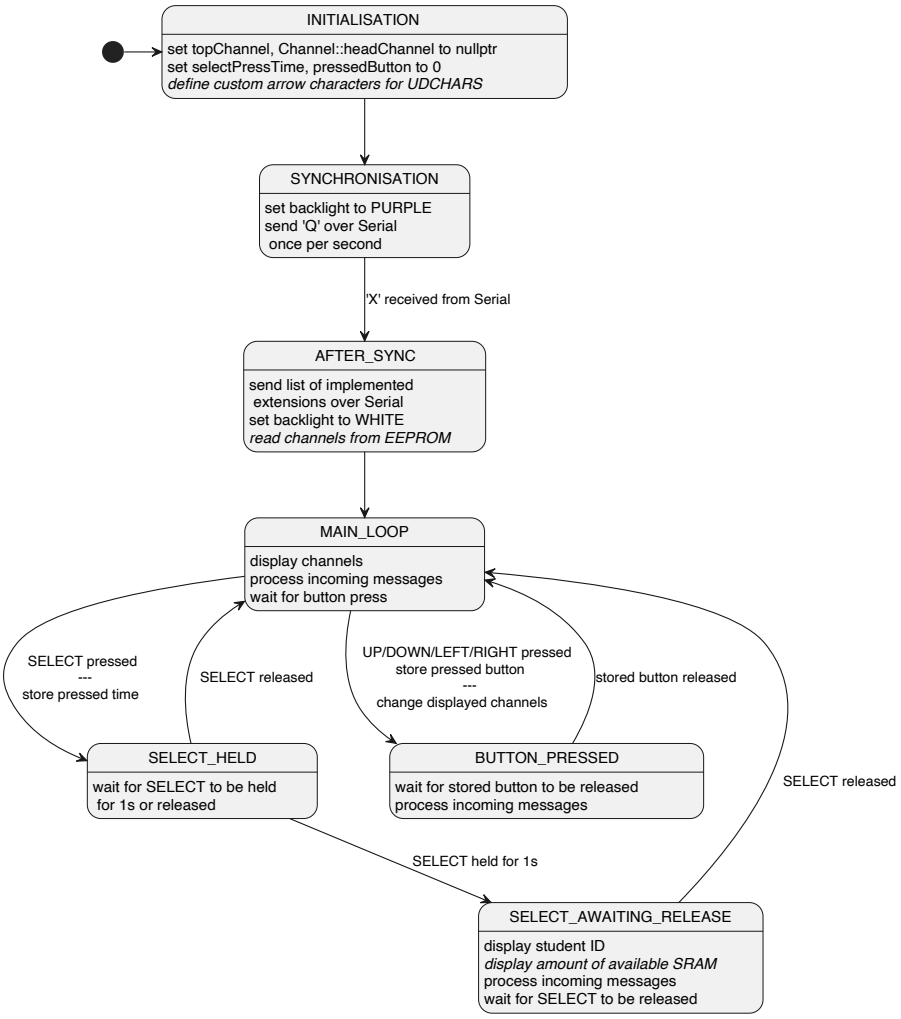

Figure 2: Main FSM With Extensions

SYNCHRONISATION

This state carries out the 'synchronisation phase' defined in the specification. The system will stay in this state until synchronisation with the host is complete, thence the system will enter the `AFTER_SYNC` state.

AFTER_SYNC

This state carries out the actions after the synchronisation phase, as defined by the specification. Once complete, the system enters the `MAIN_LOOP` state.

MAIN_LOOP

This state carries out the main phase. In this state, the system waits for messages from the host and button presses, and displays information about channels.

When `SELECT` is pressed, the system enters the `SELECT_HELD` state.

When `UP` is pressed, the system enters the `BUTTON_PRESSED` state. Once it is released, the program will display the channel before the channel currently displayed on the top line.

When `DOWN` is pressed, the system enters the `BUTTON_PRESSED` state. Once it is released, the program will display the channel after the channel currently displayed on the bottom line.

SELECT_HELD

This state waits for `SELECT` to be held for 1 second. Once that happens, the LCD backlight turns purple, my student ID is displayed, and the system enters the `SELECT_AWAITING_RELEASE` state. If `SELECT` is released before 1 second, the system enters back into the `MAIN_LOOP` state.

SELECT_AWAITING_RELEASE

In this state, the system waits for `SELECT` to be released. Once it has been released, the LCD is cleared and the system enters back into the `MAIN_LOOP` state.

BUTTON_PRESSED

In this state, the system waits for the stored button (`UP`/`DOWN`) to be released, implementing a proper button press. This is 'blocking', if multiple buttons are pressed, only the actions for the first one are carried out.

Once the button is released, the system enters back into the `MAIN_LOOP` state.

2 Data structures

2.1 Types

2.1.1 Enum `State`

The states in the main FSM (see figure 1) are defined in the typedef'd enum `State`. This current state is statically stored in `loop()#state`.

An enum was used as it allowed me to define named constants and use them with some type safety, through using an `enum class` would have provided more type safety.

2.1.2 Struct `Channel`

Channels are implemented as an ordered singly-linked-list, ordered by channel ID. This is implemented using the typedef'd struct `Channel`:

Table 1: `Channel` struct

Type	Name	Description
char	id	This channel's ID (A-Z)
const char*	desc	This channel's description/name/title (max. 15 chars)
byte	descLen	The length of this channel's description
byte	max	This channel's maximum
byte	min	This channel's minimum
channel_s*	next	Pointer to the next created channel (by ID)
byte	scrollIndex	SCROLL: the start index of the currently displayed description (see §11)
unsigned long	lastScrollTime	SCROLL: the time the description was last scrolled
ScrollState	scrollState	SCROLL: the current state of scrolling this channel's description

NOTE: This struct has more attributes, which are determined by the macro `RECENT_MODE`, see §9.

The instances of `Channel` are:

- `Channel::headChannel`

- `loop()#topChannel`
  - the channel currently displayed on the top line

The head of the channel linked list (LL) is stored statically in `Channel::headChannel`. When a new channel is created using `Channel::create(char, char*, byte)`, the LL is updated using `Channel::insertChannel(Channel*)`, which will insert the new channel into the appropriate position, according to the channel's ID.

Table 2: Static `Channel` functions

Function Signature	Description
`insertChannel(Channel* ch)`	Inserts the given channel into the LL of channels in its appropriate position
`create(char id, const char* desc, byte descLen)`	Creates a new channel with the provided ID and description if not already created, else updates description of channel
`channelForId(char)`	Returns the channel with the provided ID, or `nullptr` if not yet created
`firstChannel(HciState)`	HCI: Returns the first channel in the LL that matches the current HCI state (see §7)
`channelBefore(Channel*, HciState)`	Returns the channel before the provided channel in the LL, that matches the current HCI state
`channelAfter(Channel*, HciState)`	Returns the channel after the provided channel in the LL, that matches the current HCI state
`canGoUp(Channel*, HciState)`	Returns whether there exists a channel before the provided channel in the LL, that matches the current HCI state
`canGoDown(Channel*, HciState)`	Returns whether there exists a channel after the provided channel in the LL, that matches the current HCI state

When `UP` is pressed, `Channel#canGoUp` is called to check that there exists a channel before the one currently displayed on top, and if there is then `Channel#channelBefore` is called to get it and `loop()#topChannel` is updated. When `DOWN` is pressed, the same process occurs but using `Channel#canGoDown` and `Channel#channelAfter`.

I chose to use a LL over an array of pointers as although it makes the logic more complicated, I felt it was a cleaner solution and it made getting the channel before/after a channel easier if there uncreated channels.

2.1.3 Struct `SerialInput`

I encountered a problem when processing incoming messages while `SELECT` is being held: sometimes the serial receive buffer buffer would not have everything entered to the Serial Monitor, e.g. I entered `VA5` but `Serial.available()` would return 2 not 4, though later calls would eventually return 4.

This forced me to store what has been previously read from the Serial interface, which is stored in the typedef'd struct `SerialInput`:

Table 3: `SerialInput` struct

Type	Name	Description
static constexpr byte	INPUT_LEN	The maximum number of bytes to read from the Serial interface
char[ ]	input	What has been read so far
byte	inputLen	The number of characters read so far
Channel**	topChannelPtr	Pointer to the pointer that stores the channel displayed on the top line
HciState	hciState	HCI: The current state of the HCI FSM (see §7)

Using a struct also made it easier to pass around information - instead of a function having multiple parameters representing each aspect of `SerialInput`, it would just take in a `SerialInput`.

The function `handleSerialInput(Channel**, hciState)` statically stores the only instance of `SerialInput` and is responsible for handling any messages received from the host. When a message has been completely received, if it conforms to the protocol then `handleCreateMessage(SerialInput&)` (first char C) or `handleValueMessage(SerialInput&)` (first char V, X or N) will be called and carry out the actions specified by the documentation.

2.2 Constants (Macros)

I stored my student ID and my list of implemented extensions in the macros `STUDENT_ID` and `IMPLEMENTED_EXTENSIONS`.

2.2.1 LCD Backlight

To help with changing the colour of the LCD backlight, I used macros to define every color:

```
48   #define BL_OFF   0
49   #define RED      1
50   #define GREEN    2
51   #define YELLOW   3
52   #define BLUE     4
53   #define PURPLE   5
54   #define TEAL     6
55   #define WHITE    7
```

Figure 3: Backlight Colour Macros

2.2.2 LCD

I used macros defining the column of where each characteristic of a channel will be displayed:

```
63   // column
64   #define ARROW_POSITION   0
65   #define ID_POSITION      1
66   #define DATA_POSITION    2
67   #define AVRG_POSITION    5
68   #define DESC_POSITION    10
```

Figure 4: LCD Macros

2.2.3 Timeouts

To held with the length of timeouts, I used macros:

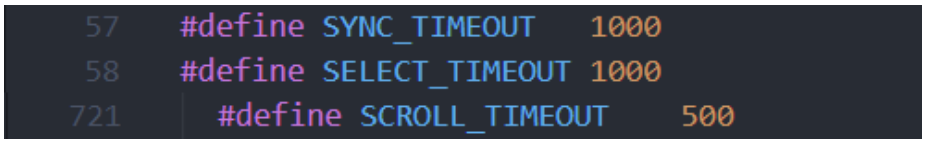

Figure 5: Timeout Macros

3 Debugging

Debug functions generally start with '_', and are commented with '`// debug`'.

The program will only send debug messages if the `DEBUG` macro is defined as a non-zero number.

Right after syncronisation is complete, the program sends multiple debug messages about the channels that have been created using values read from the EEPROM.

Whenever an erroneous message (not conforming to the protocol) is sent, a debug message about it is sent just before the error message.

4 Reflection

I am mostly happy with my code, everything works as desired.

Though I am kind of unhappy that I mixed C & C++ constructs and did not really try to stick to one.

I am also unhappy that my implementation of RECENT is not the best - when `RECENT_MODE` is `LL`, not as many values are stored as I would have liked. But I only store few values to ensure the Arduino does not ever run out of SRAM, at least because of my code. It is possible to store more values but that does not ensure the Arduino will not ever run out of SRAM.

I am also happy that I was able to mostly eliminate my use of intermediate, short-lived strings (e.g. substrings) by directly accessing the initial string buffer, saving some memory on the stack. For example, when displaying the channel description for the NAMES extension (`NAMES_SCROLL::displayChannelName(int, Channel*)`), instead of storing the substring of the channel's description (for scrolling purposes) then displaying it, I directly accessed the buffer.

I also feel like my program didn not properly reflect a proper finite-state machine. Some things could have been states, for example:

- when an incoming message does not conform to the protocol, the program could transfer to the state `ERROR`
- `ERROR` would essentially do what `processError(SerialInput)` does, then the system would enter back into the `MAIN_LOOP` state

But I felt that this did not make sense as a state, because the program flow for this would be more akin to a flowchart than a state machine - the system would not *stay* in that state. Furthemore, some states were not needed and could have been a transition, for example `AFTER_SYNC`.

Extension Features

5 UDCHARS

The namespace `UDCHARS` contains the code relating to the UDCHARS extension.

The following macros were defined, for use when creating and displaying the custom characters:

- `UP_ARROW_CHAR` 0
- `DOWN_ARROW_CHAR` 1

The change to the FSM was that in the `INITIALISATION` state, the custom characters are defined:

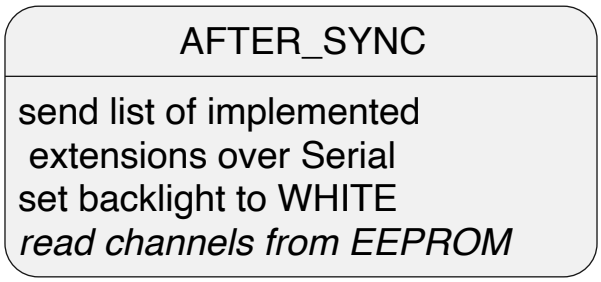

Figure 6: UDCHARS FSM Change

In my code, this change was realised by calling `UDCHARS::createChars()` in the `INITIALISATION` state (line 781):

```
775     switch (state) {
776     case INITIALISATION:
777       hciState = NORMAL;
778       topChannel = Channel::headChannel = nullptr;
779       selectPressTime = 0;
780       pressedButton = 0;
781       UDCHARS::createChars(); // UDCHARS
782       state = SYNCHRONISATION;
783       break;
```

Figure 7: UDCHARS FSM Code Change

5.1 Defining the arrows

The custom characters are defined in `UDCHARS::createChars()`, they were designed as 2 chevrons pointing in the appropriate direction (mirrored vertically), using chareditor.com:

Figure 8: Up Chevron Design

```
551     byte upChevron[] PROGMEM = { B00100, B01010, B10001, B00100, B01010, B10001, B00000, B00000 };
552     lcd.createChar(UP_ARROW_CHAR, upChevron);
```

Figure 9: Up Chevron Code

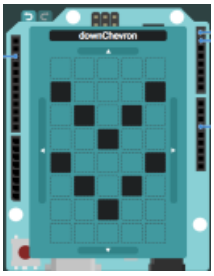

Figure 10: Down Chevron Design

```
553     byte downChevron[] PROGMEM = { B00000, B10001, B01010, B00100, B10001, B01010, B00100, B00000 };
554     lcd.createChar(DOWN_ARROW_CHAR, downChevron);
```

Figure 11: Down Chevron Code

5.2 Displaying the arrows

The arrows can be displayed to the LCD using `UDCHARS::displayUpArrow(bool)` and `UDCHARS::displayDownArrow(bool)`.

These functions have a single parameter `bool display`, which determines whether the arrows is printed to the LCD or a space is printed instead. This is indicative of whether a channel exists above the current `topChannel` or a channel exists below the current `btmChannel`.

```
557   void displayUpArrow(bool display) {
558     byte ch = display ? UP_ARROW_CHAR : ' ';
559     lcd.setCursor(ARROW_POSITION, TOP_LINE);
560     lcd.write(ch);
561   }
562
563   void displayDownArrow(bool display) {
564     byte ch = display ? DOWN_ARROW_CHAR : ' ';
565     lcd.setCursor(ARROW_POSITION, BOTTOM_LINE);
566     lcd.write(ch);
567   }
```

Figure 12: Code to Display Arrows

Which are called in `updateDisplay(Channel*, HciState)`:

```
1122  void updateDisplay(Channel *topChannel, HciState hciState) {
1123    updateBacklight();
1124
1125    // UDCHARS,HCI
1126    UDCHARS::displayUpArrow(Channel::canGoUp(topChannel, hciState));
1127    UDCHARS::displayDownArrow(Channel::canGoDown(topChannel, hciState));
```

Figure 13: Displaying of Arrows

5.3 Example

For example, if channels A, B and C have been created, the Arduino should look like:

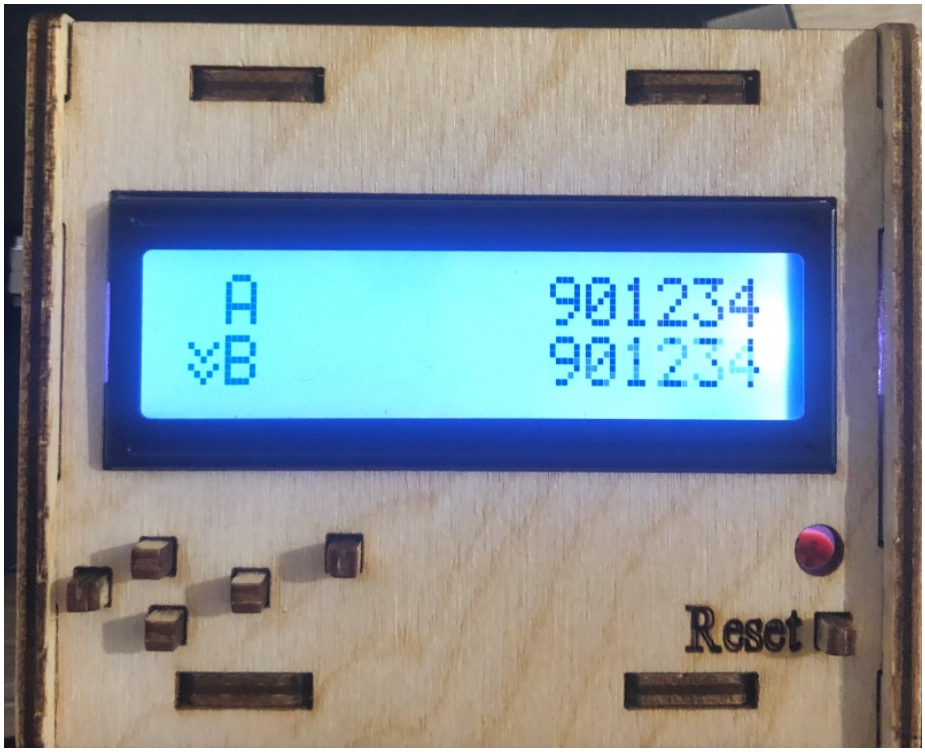

Figure 14: UDCHARS Example Arduino Display

As there are no channels before (“above”) A, the up arrow is not displayed, a space is displayed instead. As there is a channel after (“below”) B, the down arrow is displayed.

6 FREERAM

The namespace `FREERAM` contains the code relating to the FREERAM extension.

The function `FREERAM::<unnamed>::freeMemory()` returns the number of bytes currently available in the Arduino's SRAM.

This can displayed to the screen, left justified, using `FREERAM::displayFreeMemory(byte)`:

```
590   void displayFreeMemory(byte row) {
591     lcd.setCursor(1, row);
592     lcd.print(F("Free bytes:"));
593     lcd.print(freeMemory());
594   }
```

Figure 15: `FREERAM::displayFreeMemory(byte)`

The change to the FSM was that in the `SELECT_AWAITING_RELEASE` state, the Arduino will also display the amount of available SRAM:

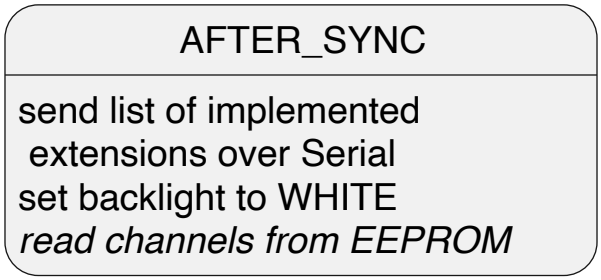

Figure 16: FREERAM FSM Change

In my code, this was realised by calling `FREERAM::displayFreeMemory` in `selectDisplay()`.

Once SELECT has been held for at least 1 second. The Arduino should look like:

Figure 17: FREERAM Arduino Display

7 HCI

HCI is implemented using a finite state machine with the states:

- NORMAL
  - display all channels
- LEFT_MIN
  - display channels where the current value is beyond the minimum
- RIGHT_MAX
  - display channels where the current value is beyond the maximum

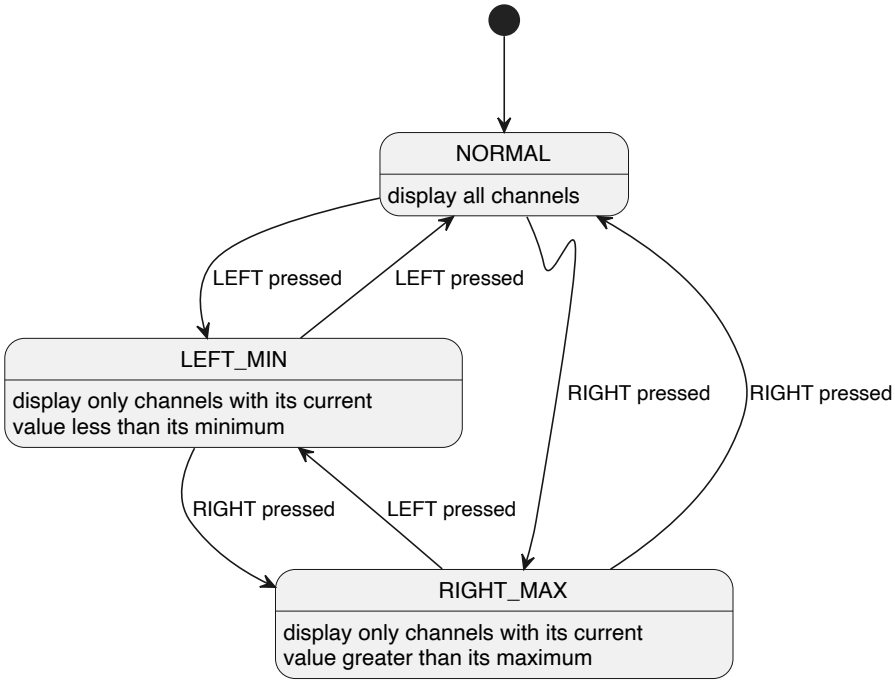

Figure 18: HCI FSM

The HCI FSM is realised by the enum `HciState` and the function `Channel#meetsHciRequirement(HciState)`.

The function `Channel#meetsHciRequirement(HciState)` returns whether or not a channel is meant to be displayed according to the current state of the HCI FSM.

For example, channel A with {data: 30, min: 50, max: 255}, the display should look like:

After `LEFT` has been pressed, the display should look like:

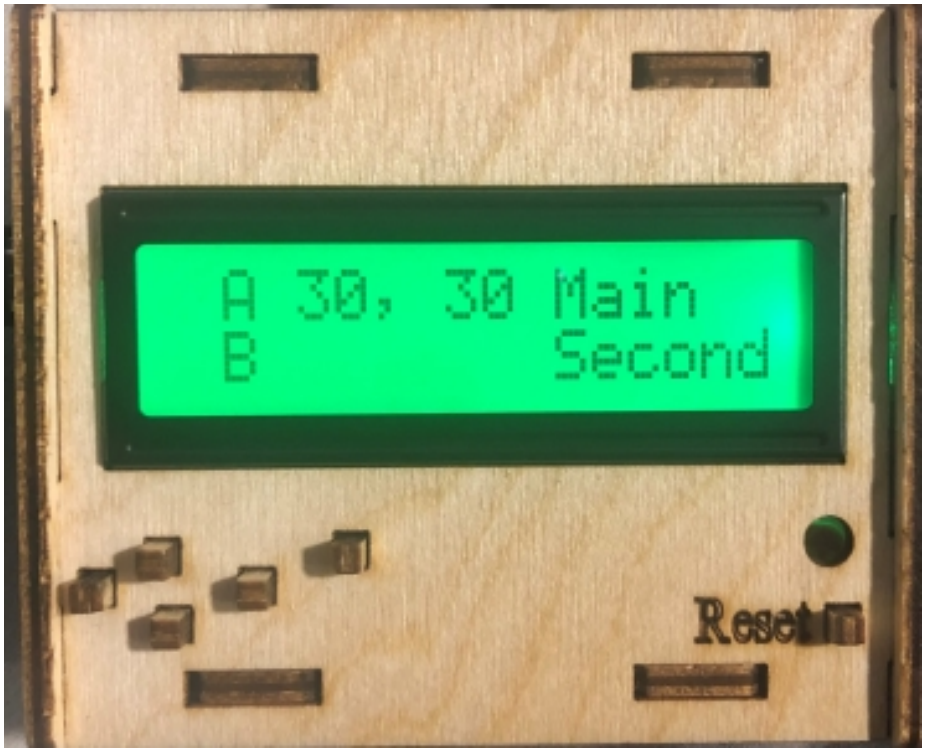

Figure 19: HCI Arduino Display

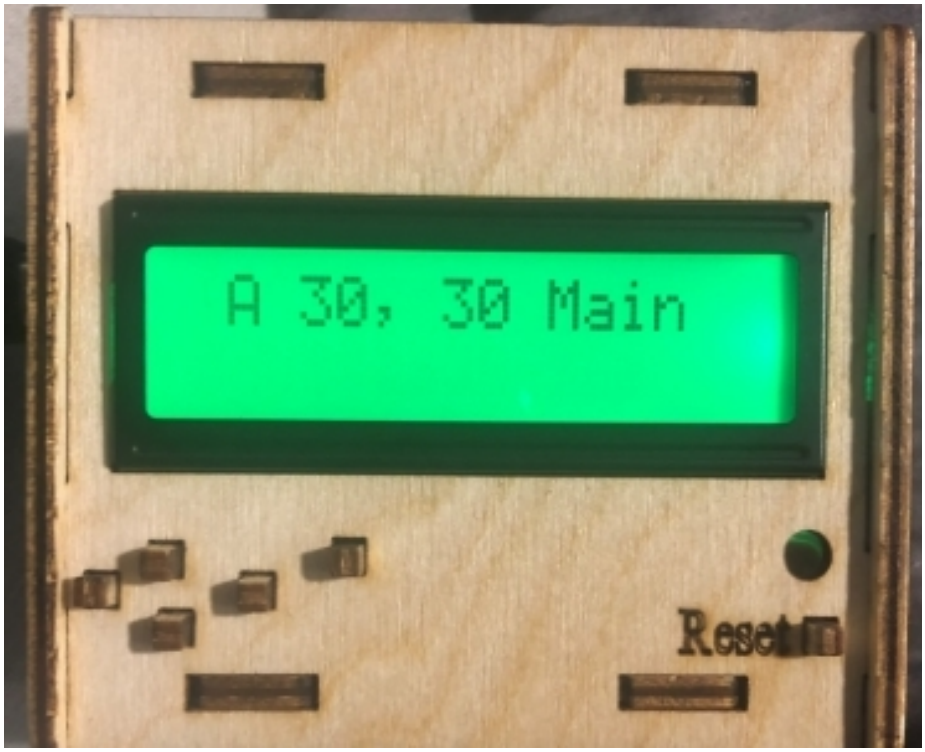

Figure 20: `LEFT` pressed

8 EEPROM

The namespace `_EEPROM` contains the code relating to the EEPROM extension.

Each channel occupies 26 continguous bytes in the EEPROM:

Table 4: EEPROM Channels

# of Bytes	Description
1	ID
1	Maximum value
1	Minimum Vaue
1	Description length
15	Description
7	My student ID

The beginning address for a channel is calculated using `(id - 'A') * 26`, creating a distance of 26 bytes between the beginning addresses of each channel (A: 0, B: 26, C: 52, ...).

I stored my student ID in order to be able to check that the values in the EEPROM were written by me, see §8.3.

8.1 Writing to EEPROM

Modifications to the EEPROM are only done using 'update' functions and not 'write' function to prolong the life of the EEPROM.

Each characteristic of a channel that needs to be written to the EEPROM (see table 4) has a specific offset from the beginning address of the channel. These are defined by the following macros (`idAddr` is the beginning address - the offset for the ID is 0):

- `maxOffset(idAddr)` `(idAddr + 1)`
- `minOffset(idAddr)` `(idAddr + 2)`
- `descOffset(idAddr)` `(idAddr + 3)`
- `studentIdOffset(idAddr)` `(idAddr + 19)`

Every time a channel's description, value, maximum or mimimum is updated, the EEPROM is updated using `_EEPROM::updateEEPROM(Channel*)`. This ensures the EEPROM is kept up to date.

8.2 Reading from EEPROM

The change to the FSM was that in the `AFTER_SYNC` state, the stored channels will be read from the EEPROM:

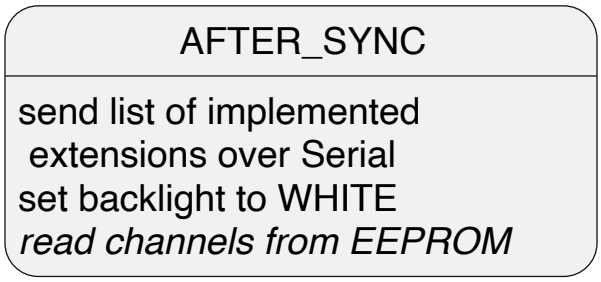

Figure 21: EEPROM FSM Change

In my code, this was realised by calling `_EEPROM::readEEPROM()`:

```
799     case AFTER_SYNC:
800       Serial.println(IMPLEMENTED_EXTENSIONS);
801
802       // EEPROM
803       topChannel = Channel::headChannel = _EEPROM::readEEPROM();
804       debug_print(F("DEBUG: Channels stored in EEPROM: "));
805       _printChannelsFull(topChannel);
806
807       state = MAIN_LOOP;
808       break;
```

Figure 22: EEPROM FSM Code Change

The function `_EEPROM::<unnamed>::readChannel(char)` returns a pointer to the channel with the provided ID, with its values initialised by values read from the EEPROM. `_EEPROM::readEEPROM()` is used to read all the channels from the EEPROM. This is done by iterating from A through to Z, getting the channel using `readChannel(char)`, and forming a linked list of channels, then returning the head of this linked list:

Figure 23: `_EEPROM::readEEPROM()`

8.3 Persistence Validation

The mechanism to determine whether the values were written by me is a 3 step process:

1. Check that the written student ID is equals to my student ID
2. Check that the written ID matches the ID of the channel that should be written in that address
3. Check that the written description length is between 1 and 15 inclusive

Therefore, it is simple to 'invalidate' any written channel: by modifying the values written such that they fail any step in the above process. For example setting the written ID to '@', similar to what is done by `_EEPROM::invalidateChannel(char)` and `_EEPROM::invalidateEEPROM()`. Please see lines 764 - 765 in the code.

Although writing my student ID with every channel feels a bit excessive, I think it is the best mechanism to be able to verify that the values were written by **me** without being complicated.

9 RECENT

As RECENT seems to be impossible without making any compromise, it is implemented in 3 different ways, which make different compromises:

1. using a queue with a maximum size
2. using a circular array
3. using an exponential moving average

You can choose which one the program uses by changing the macro `RECENT_MODE`. It is designed to be effectively an enum; its value should only be as one of the following defined macros:

- `LL` (linked list as queue)
- `ARRAY` (circular array)
- `EMA` (exponential moving average)

If `RECENT_MODE` is defined as any other value, the program will not compile.

9.1 Queue (Linked List)

The compromise made by this implementation is that not all 64 values can be stored, unless another compromise is made elsewhere, e.g. limiting the number of channels that can be created.

I use a queue (implemented with a singly-linked-list), with a maximum size defined by the macro `MAX_RECENT_SIZE`, which once exceeded will discard the head value. I thought this was a genius solution as it only allocated memory when needed, no memory would be 'wastefully' allocated.

`Channel#recentHead` stores the head of this linkedlist, and can be used for all list-related operations.

Table 5: `Channel` struct extra attributes

Type	Name	Description
RecentNode*	recentHead	the head of the RECENT linked list for this channel
RecentNode*	recentTail	the tail of the RECENT linked list for this channel
byte	recentLen	the number of recent values currently stored

9.2 Circular Array

The compromise made by this implementation is that not all 64 values can be stored, unless another compromise is made elsewhere, e.g. limiting the number of channels that can be created.

I use a circular byte array of size `RECENT_ARRAY_SIZE`, overwriting the oldest value once a new value has been entered (after more than `RECENT_ARRAY_SIZE` values have been entered).

Table 6: `Channel` struct extra attributes

Type	Name	Description
byte*	recents	The array
unsigned long	nRecents	The number of values entered

9.3 Queue vs Circular Array

As using a queue or circular array make the same compromise, it makes sense to compare them.

While using a linked list will start off using less memory than an aray, because each node will use 5 bytes (1 for the value and 4 for the pointer of the next node), its memory usage will increase very quickly. Assuming `MAX_RECENT_SIZE` = 64, once 64 values have been entered, the linked list will take 320 bytes which is a lot more than the 64 bytes an array will take.

As the array uses less memory, it allows for an array of a larger size than the maximum size of the linked list, which *should* give a more accurate average of the last 64 values.

Their memory usages can be seen below (assuming all 64 values are stored):

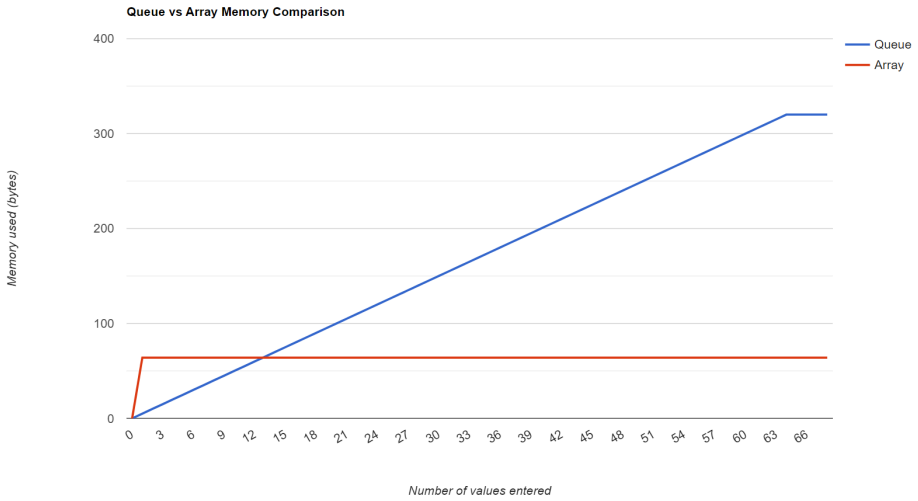

Figure 24: Memory Usage Comparison

9.4 Exponential Moving Average

The compromise made by this implementation is that the most recent values are not actually stored, and it is an estimation (though the other implementations are also estimations for the average of the most recent 64 values). However, this *should* give a closer estimate to the average of the most recent 64 values than the other two implementations as this takes the last 64 values into account, whereas the others can only take into account a very limited number of values.

I had to make a choice between using the average of all values entered and using an exponential moving average (EMA). I decided to implement an EMA as it is more accurate for the average of the last 64 values. More specifically, as the numbers of values entered increases, the more accurate an EMA is over a total average.

Table 7: `Channel` struct extra attributes

Type	Name	Description
byte	data	The most recent value
double	runningAvrg	The current estimated average
unsigned long	nRecents	The number of values entered

The EMA implementation is based upon the recursive formula (from the Wikipedia page):

$$y[n] = \alpha x[n] + (1 - \alpha)y[n - 1] \tag{1}$$

Where:

- α is the weighting for the new value
- $y[n]$ is the new average
- $y[n - 1]$ is the old average
- $x[n]$ is the most recent value

A drawback of using an EMA is that the early results are unreliable as the spin-up interval is not completed. In this implementation, the spin-up interval would be the first 64 values being entered.

I was able to overcome this drawback by constantly changing α while less than 64 values have been entered. Thus making the average of the first 64 values is very nearly exact.

Once more than 64 values have been entered, I used an α value of $\frac{1}{47}$. I created a Kotlin script (see Appendix) to calculate what the best α value would be (see comments in `Channel::addRecent` when `RECENT_MODE == LL`), the results were:

- α values roughly between $\frac{1}{53}$ and $\frac{1}{57}$ were best for when roughly 100 values were entered
- α values roughly between $\frac{1}{44}$ and $\frac{1}{48}$ were best for when more than 300 values were entered
- As the number of values increased, a higher α value was better, until about $\frac{1}{45}$, whence the average estimate would be further from the exact average

10 NAMES

SCROLL and NAMES were implemented together as they go hand-in-hand, in the namespace `NAMES_SCROLL`.

Initially, I stored the channel's description as a `String`, but in the end I chose to use a `const char*` as this allowed me to allocate the exact amount of memory needed to store the description.

The channel description is printed to the display using `lcd.print` with padded spaces on the end to overwrite the previously displayed description:

```
733  void displayChannelName(int row, Channel *ch) {
734    // ch->desc SHOULD NOT be nullptr
735
736    const byte dLen = ch->descLen;
737    byte &si = ch->scrollIndex;
738
739    // NAMES
740    lcd.setCursor(DESC_POSITION, row);
741    for (int i = si; i < si + DESC_DISPLAY_LEN; i++) {
742      // display space(s) at end to overwrite old displayed value
743      char c = (i < dLen) ? ch->desc[i] : ' ';
744      lcd.print(c);
745    }
```

Figure 25: Displaying the channel description

At first, I copied the description to a separate buffer, with padded spaces to the end (to overwrite the previously displayed description), then that buffer would be printed to the LCD:

```
741  char buf[DESC_DISPLAY_LEN + 1];
742  strncpy(buf, ch->desc + si, min(DESC_DISPLAY_LEN, dLen - si));
743  buf[DESC_DISPLAY_LEN] = '\0';
744
745  // pad spaces
746  if (dLen < DESC_DISPLAY_LEN) {
747    memset(buf + dLen, ' ', DESC_DISPLAY_LEN - dLen);
748  }
749
750  lcd.print(buf);
```

But I chose to direcly access the string buffer as it does not involve creating a temporary buffer.

10.1 Example

For example, after a `CAMain` message, the Arduino should look like:

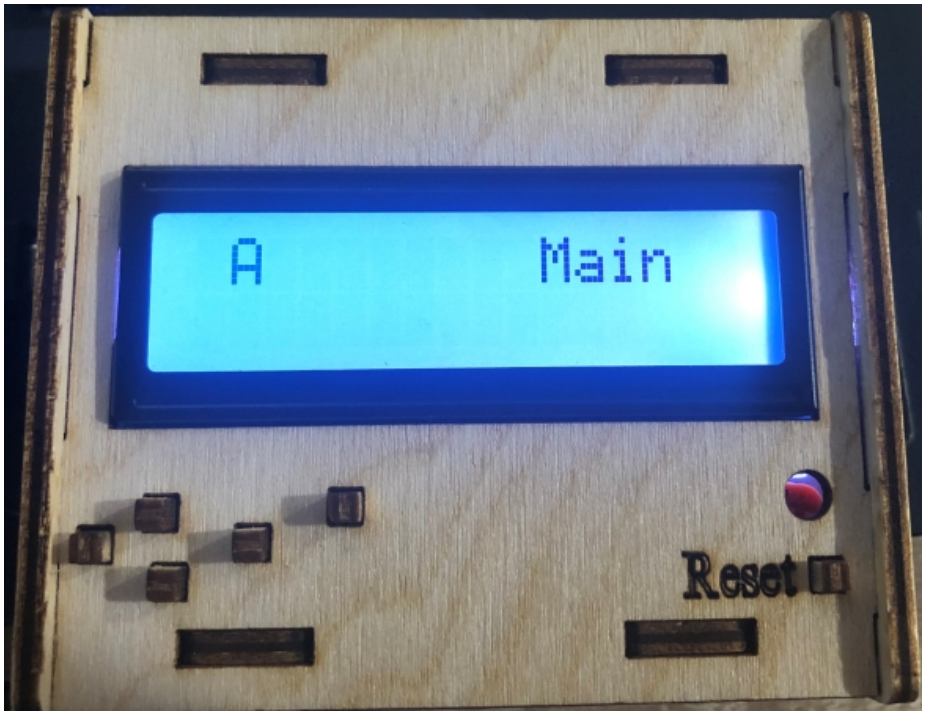

Figure 26: NAMES Arduino Display

11 SCROLL

SCROLL and NAMES were implemented together as they go hand-in-hand, in the namespace `NAMES_SCROLL`.

SCROLL is implemented by essentially dispalying a moving substring of the channel description.

This is carried out in `NAMES_SCROLL::displayChannelName(int, Channel*` by using `Channel#scrollIndex` to keep track of the start index of this substring and `Channel#lastScrollTime` to keep track of the last time a scroll happened (to scroll over every 500ms).

It is implemented using a flowchart:

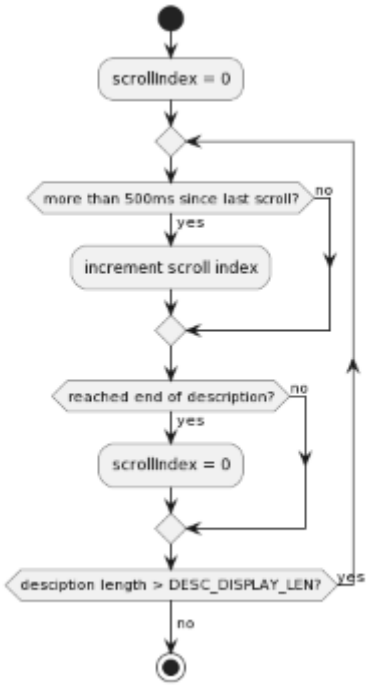

Figure 27: SCROLL Flowchart

11.1 Example

For example, the message `CA0123456789` is sent.

The display should initially look like:

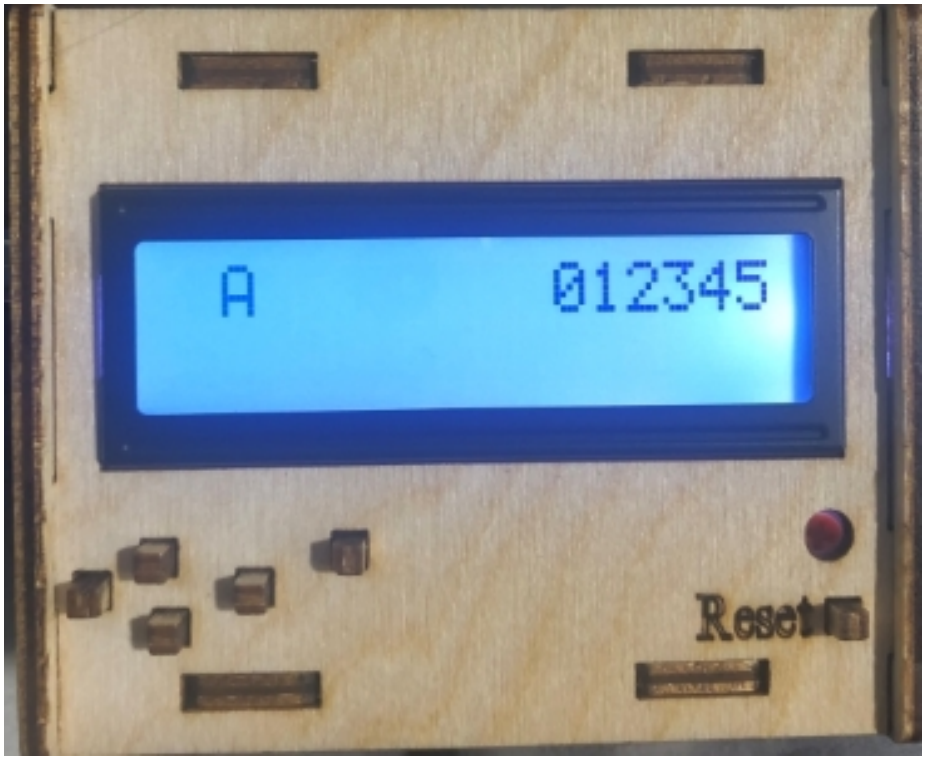

Figure 28: Start of Channel Description

Then after 500ms, it should scroll a character, and look like:

Once the end of the channel description has been reached:

Scrolling should then start again, and the display should look like 28.

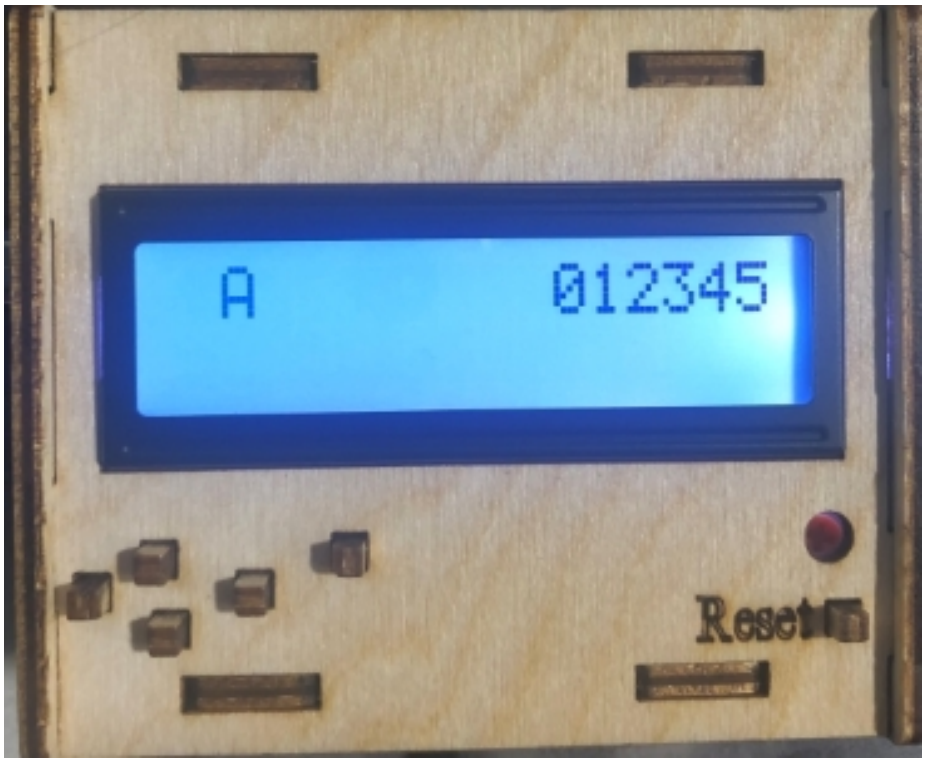

Figure 29: ‘Middle’ of Channel Description

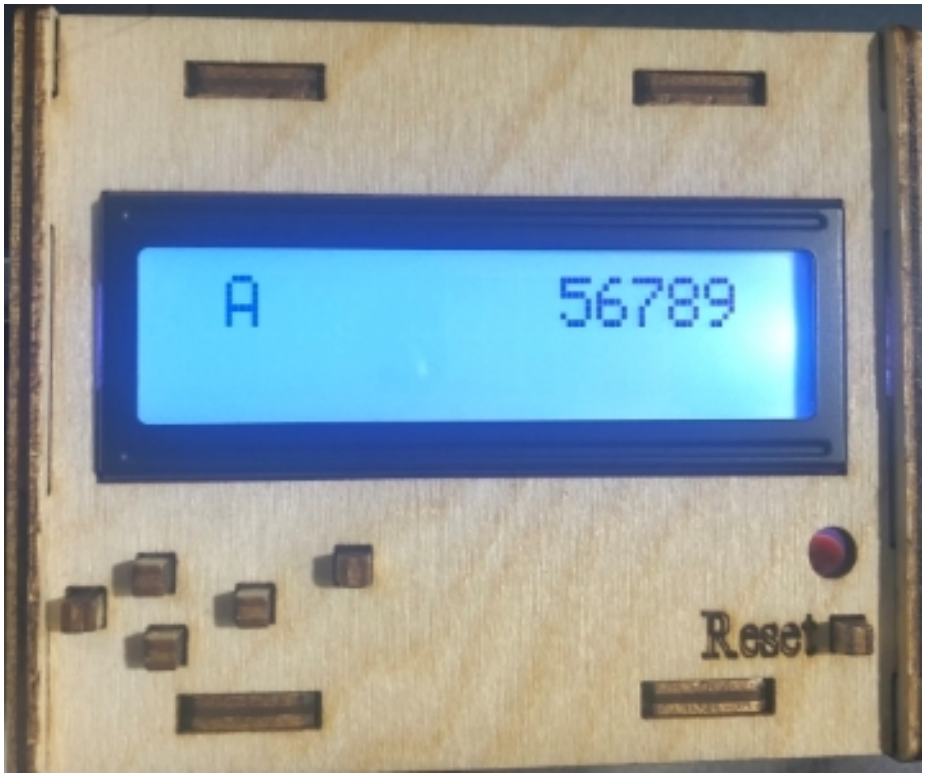

Figure 30: End of Channel Description

Appendix

SCROLL Kotlin Script

```kotlin
import kotlin.math.absoluteValue

// to see difference when using total average
const val TOTAL_AVERAGE = 0

fun main(vararg args: String) {
    val results = mutableMapOf<Int, MutableList<Double>>()

    repeat(1_000) {
        val a = Alpha(300)
        a()
        a.avrgDiffResults.forEach { (alpha, diff) ->
            results.computeIfAbsent(alpha) { mutableListOf() }
                .add(diff)
        }
    }

    results.mapValues { (alpha, diffs) -> diffs.average() }
        .apply {
            toSortedMap { d1, d2 ->
                // sort by ascending avrg (best avrg diff first)
                val a1 = this[d1]!!
                val a2 = this[d2]!!
                a1.compareTo(a2)
            }.forEach { (alpha, avrgDiff) ->
                if (alpha == TOTAL_AVERAGE)
                    println(" TOTAL  =  $avrgDiff".format(alpha))
                else
                    println("1 / %-2d  =  $avrgDiff".format(alpha))
            }
        }
}

class Alpha(val nRecents: Int) {
    val alphas = (2..64) + TOTAL_AVERAGE

    val alphasAvrgs = mutableMapOf<Int, Double>().apply {
        alphas.forEach {
            this[it] = 0.0
        }
    }
    val results = mutableMapOf<Int, MutableList<Double>>()
```

```
43      val avrgDiffResults = mutableMapOf<Int, Double>()
44
45      private val randomByte: Int
46          get() = (0..255).random()
47
48      val values = mutableListOf<Int>().apply {
49          repeat(64) {
50              add(randomByte)
51          }
52      }
53
54      operator fun invoke() {
55          val startAvrg = values.average()
56          alphas.forEach {
57              alphasAvrgs[it] = startAvrg
58          }
59          alphasAvrgs[TOTAL_AVERAGE] = values.sum().toDouble()
60
61          for (i in 65..nRecents) {
62              val added = randomByte.also { values.add(it) }
63              val exact = values.averageLast(64)
64
65              alphas.forEach {
66                  val diff: Double
67                  if (it == TOTAL_AVERAGE) {
68                      val sum: Double = alphasAvrgs.getValue(it) + added
69                      alphasAvrgs[it] = sum
70
71                      val avrg: Double = sum / values.size
72
73                       diff = (exact - avrg).absoluteValue
74                  } else {
75                      val alpha: Double = 1.0 / it
76
77                      var runningAvrg = alphasAvrgs.getValue(it)
78                      runningAvrg = alpha * added + (1 - alpha) * runningAvrg
79                      alphasAvrgs[it] = runningAvrg
80
81                      diff = (exact - runningAvrg).absoluteValue
82                  }
83
84                  results.computeIfAbsent(it) { mutableListOf() }
85                      .add(diff)
86              }
87          }
88
```

```
89          // calculate avrgDiffResults
90          results.forEach { (alpha, diffs) ->
91              avrgDiffResults[alpha] = diffs.average()
92          }
93      }
94  }
95
96  fun List<Int>.averageLast(n: Int): Double {
97      var sum: Double = 0.0
98      var count: Int = 0
99
100     for (i in this.lastIndex downTo this.lastIndex - n + 1) {
101         sum += this[i]
102         count++
103     }
104     return if (count == 0) Double.NaN else sum / count
105 }
```