School of Science

Computer Science

20COA202 Embedded Systems Programming

Coursework development documentation

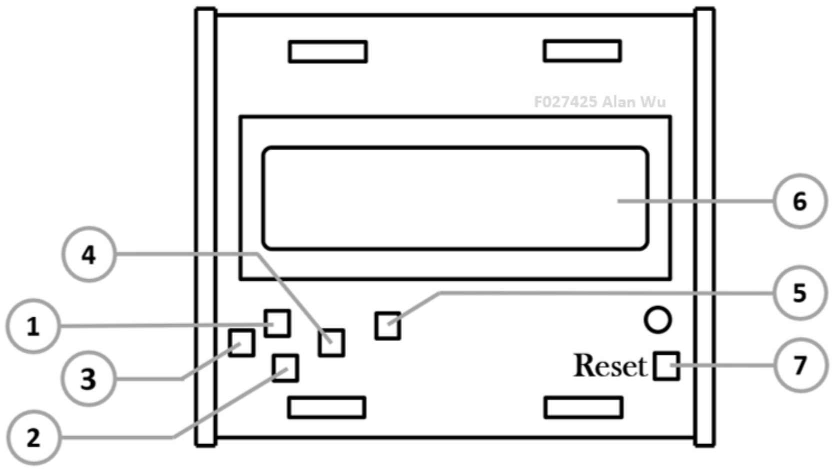
Alan Wu F027425 Semester 2 2020-2021

Contents

Introduction

A home control system manages and controls a variety of systems in a house, these systems include (but not limited to) the lighting and heating systems in a house. In this document, I will be describing my process of designing and developing a system which meets the requirements of the given specification. In the following sections, I will be explaining how to navigate the system, the implementation (design and testing) of my system as well as any extension features which I have included.

Navigating the system



Button (#)	Description
1	Up press to increase value
2	Down press to decrease value
3	Left press to cycle menu options
4	Right Press to cycle menu options
5	Select Press to confirm option selection
6	LCD screen
7	Reset Press to reset system

Figure 1: Diagram of Arduino unit and description of buttons.

When booting the system, the user will be greeted with a “Welcome” message. To begin using the system, the user presses the Right button to begin cycling the menu.

The user can press Select to confirm their selection of which gives them more options to cycle through. When Select is pressed, the user is prompted with an “OK.” on the LCD screen to notify them that the button press has gone through with the system.

Pressing Left will cause the menu to reset and will allow the user to begin cycling options from the start of the menu again. When Left is pressed, the user is prompted with “BACK.” on the LCD screen to notify them that they have gone back to the start of the menu.

Pressing the Reset button will cause the system to reset entirely; when this is done the system will forget all values it has updated and cause all values to revert to their default values.

When cycling the menu, the menu will display the last option chosen and the current option, i.e.

Ground/Hall	Lighting/Main
• Ground floor has been chosen. • The user can now cycle which room on the given floor they want to choose. Shorthand for: [Floor] / …	• Lighting has been chosen. • Devices for lighting in the given room can now be cycled. Shorthand for: [Floor] / [Room] / Lighting / …

When the user wants to set their device to on, off, or a level, the controls are slightly changed here.

	On/Off option chosen	**Level option chosen**
Up	Increases Hours/Minutes	Increases value of Level
Down	Decreases Hours/Minutes	Decreases value of Level
Left	Change Hours (if you are currently changing Minutes)	n/a
Right	Change Minutes (if you are currently changing Hours)	n/a
Select	Confirm selection	

Note: when choosing on/off times, the system makes you set the hours first by default.

After setting the user has set the level, or on/off times for a given device, pressing select will take them back to the main menu, where they will be able to set another change to the system. When the user presses select, they will be prompted on the LCD screen with a "SET." – meanwhile a message is displayed on the Serial monitor which will contain all information about the most recently updated room – including the changes the user has just set, i.e.

This is the serial monitor after three operations:

1. Turning off bedroom 2's heating at 21:00

2. Changing living room heating level to 65.

3. Turning on hall lights at 14:00.

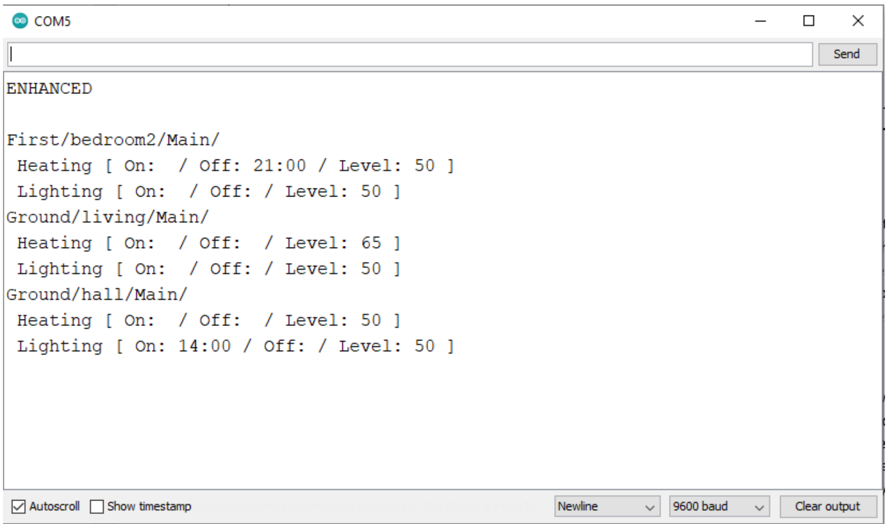


Figure 2: The serial monitor after three operations

Base implementation

Data structures

Storing data

In this section, I will be discussing the process of implementing the data structures I used for my system and their effectiveness.

In my implementation of the home control system, I experimented with an array of structures (on a global scope) to represent the rooms in the house. In my first implementation, I started off with a simple structure named “room_s” (see **Figure 3**).

```
struct room_s {              // rooms in the house are defined using a type definition
structure
    String room_floor, room_name, room_devices, on_time, off_time;
    int lighting, heating;
};

// Define my structures
room_s kitchen_s = {"Ground", "Kitchen", "Main", "12:00", "12:00", 50, 50};
. . .

room_s myHouse[6] = {kitchen_s, hall_s, living_s, bed1_s, bed2_s, bath_s};
```

Figure 3: version 1 of my structure room_s

This structure consisted of String and Integer values and worked reliably for the development of my system but unfortunately had some flaws to it; most notably that this structure meant that there would only be one on time and one off time for each room – this meant that all devices in a given room would share the same on/off times, of which would not be very ideal for the end user of this sort of control panel. Admittedly the idea of a shared on and off time for the lighting and heating devices were a result of my misunderstanding of the specification itself. As a result, I thought about alternative layouts for my structure, of which led me to create other versions of room_s.

Initially, I thought of using an array to represent the lighting and heating devices such as shown below (**Figure 4**).

```
struct room_s {       // rooms in the house are defined using a type definition structure
  String Floor, Room, Name, Lighting[3], Heating[3];
};

// Define my structures
room_s kitchen_s = {"Ground", "Kitchen", "Main", {"00:00", "00:00", "50"}, {"00:00",
"00:00", "50"}};
. . .

room_s myHouse[6] = {kitchen_s, hall_s, living_s, bed1_s, bed2_s, bath_s};
```

Figure 4: version 2 of my structure room_s, now containing two arrays.

Despite how useful this implementation would have been to use, and the added flexibility the altered structure gives, I found it to be very difficult to implement into my system despite overhauling the way in which my system interacts with the structure room_s. This was a result of how much memory the String type uses, and version 2 of room_s was built primarily of Strings.

As can be seen in there are now two array of strings which represent the lighting and heating devices in each room respectively. I planned to make it so that:

- `Lighting[0]` and `Heating[0]` represent the on time of a device as a string
- `Lighting[1]` and `Heating[1]` represent the off time of a device as a string
- `Lighting[2]` and `Heating[2]` represent the level of a device as a string (altering this value would require that the string is converted to an integer, then converted from integer to string again to be stored in the system memory).

As a result of these findings, I thought that it would be a good idea to go with a simpler approach to fix this problem, **Figure 5**.

```
// Define structure of room
struct room_s {             // rooms in the house are defined using a type definition
structure
  String room_floor, room_name, room_devices, lighting_on, lighting_off, heating_on,
heating_off;
  int lighting, heating;
};

// Define my structures
room_s kitchen_s = {"Ground", "Kitchen", "Main", 50, 50, "12:00", "12:00"};
. . .

room_s myHouse[6] = {kitchen_s, hall_s, living_s, bed1_s, bed2_s, bath_s};
```

Figure 5: The third version of the structure room_s which now contains more String elements as a work around for representing on/off times.

Instead of using an array, I would add more elements to my structure which would mean that both lighting and heating devices would still be able to have their own on/off times – this however would be at the cost of the structure's flexibility/extensibility to have more devices per room. This however was not the only problem which I encountered with this approach – it would often cause the system to bug due to the amount of memory it used up. Hence, because of this, I decided to go for another approach as detailed in **Figure 6**.

My final implementation of the structure room_s involved nesting two structures together (**Figure 6**) of which I feel has adequately satisfied the minimum requirements of the specification.

```
struct device_s {
  String on_time, off_time;
  uint8_t level;
} deviceOnStartUp;

const uint8_t maxDevicesPerRoom = 2;    //Assume 2: lighting and heating
typedef struct room_s {
  String room_floor, room_name, device_name;
  device_s devices[maxDevicesPerRoom];
} room_t;
const int roomCount = 6;
room_t kitchen, hall, living, bed1, bed2, bath;
room_s arrHouse[roomCount] = {kitchen, hall, living, bed1, bed2, bath};
```

Figure 6: Nested structures to represent the rooms in my house.

In **Figure 6**, there are two structures; device_s, of which is used to hold the information regarding the individual lighting and heating devices in the house; and room_s, of which represents each room in the house. The structure device_s is held as part of an array within room_s, suggesting that each room has multiple devices; in the example above however, the number of devices has been limited to 2 per room as can be seen by the definition of the integer constant `maxDevicesPerRoom` – this value can be changed later on if more devices are to be supported in each room, however this may lead to a less reliable program as this approach uses quite a lot of dynamic memory by itself already.

In this version of room_s, and device_s, Strings and uint8_t data types have been used. Strings have been used to represent the names of room as well as the on/off times for each device. I have also decided to use uint8_t instead of regular integers as a measure to save memory in my system.

As with all other versions of the room_s structure, each room is then put into an array of room_s structures (or alternatively, in **Figure 6**, a room_t type). The rooms have been created on a global scope here into an array called `arrHouse[roomCount]`, and its elements are defined during system setup by a subroutine called `setupRooms()` – see **Figure 21**.

Updating the data structure

In my implementation of the home control system, the data structure is updated whenever the user is shown a “SET.” message on the LCD display. The way in which the system does this depends on what is being updated by the system. Within the main loop of the system, a switch case is used to navigate the different states (and substates) that the system can be in - when updating the data structure, the system is in the `chooseValue` state.

Within the `chooseValue` state, one of two things could happen; if the user is updating an on or off time, the function `changeTime` is called (with some parameters passed into it); if the user is updating the level of an appliance, the user directly manipulates the values of the data structure when making changes to the system level.

The `changeTime()` function

This function is called whenever the user updates an on or off time as previously mentioned. When this function is called, three integer (uint8_t, to save dynamic memory) variables and a Boolean variable is passed in as parameters. The integer parameters (are passed in as parameters to help the function to find and access the correct part of the data structure to assign a new on or off time for a given appliance. The Boolean parameter is used as a system check to determine if the user has updated the time or not; when a time is set the same Boolean value is returned to the main loop.

Updating appliance levels

When updating the level of a given device in the home control system, the system handles this request within the loop itself. The system will first begin by getting the value assigned to the device whose level is being changed from the data structure and display this to the LCD screen; then, depending on the user’s actions, the value stored in the data structure is increased or decreased – in turn, the value displayed on the LCD screen will increase or decrease.

Finite State Machine

In this section, I will be describing the finite state machine at the centre of my implementation of the home control system. **Figure 7** shows the FSM I have designed for my home control system; it consists of 6 possible main states, with one of them containing substates called *chooseValue*.

Figure 8 shows the state diagram of my implementation.

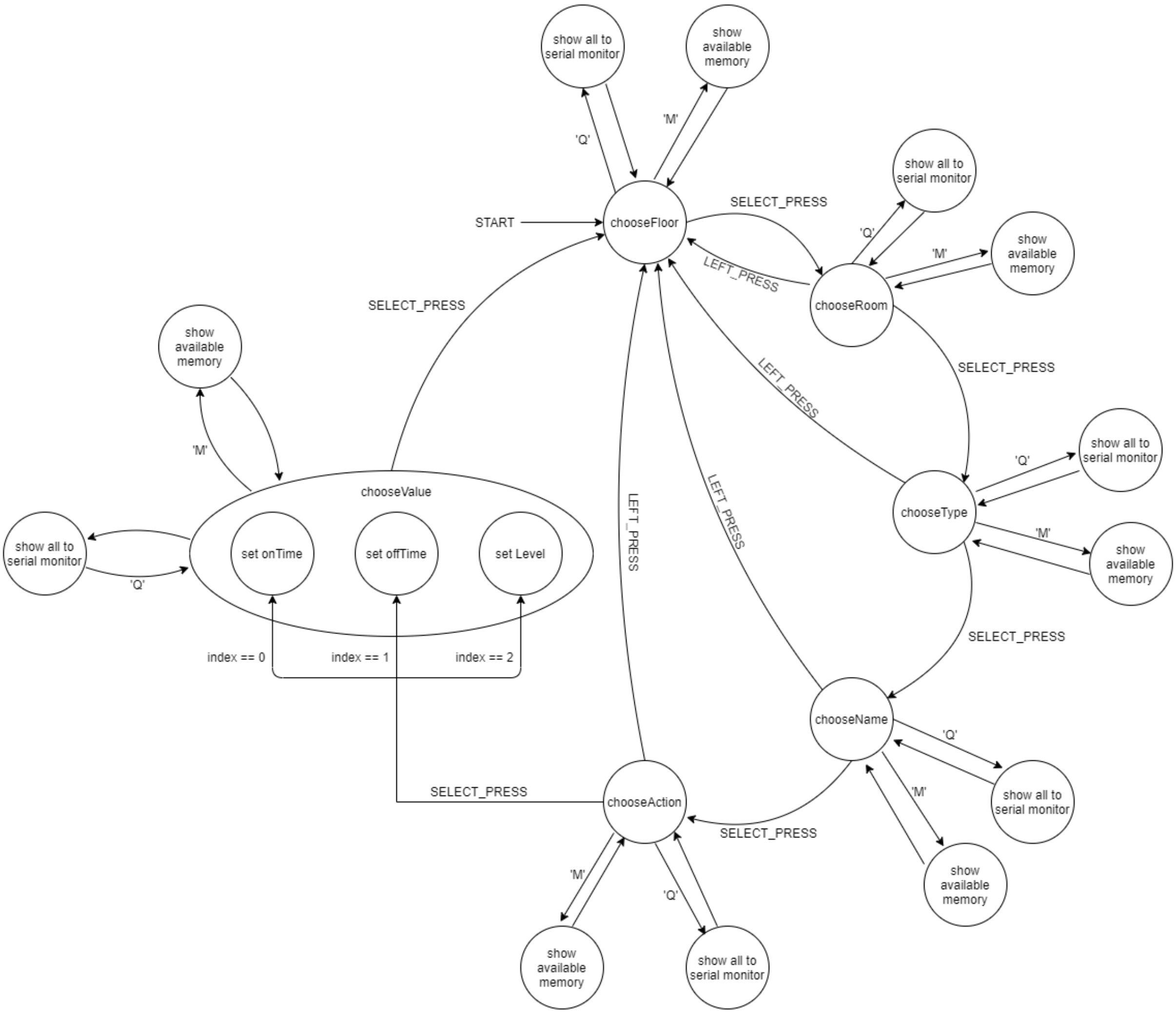


Figure 7: The FSM at the centre of my implementation

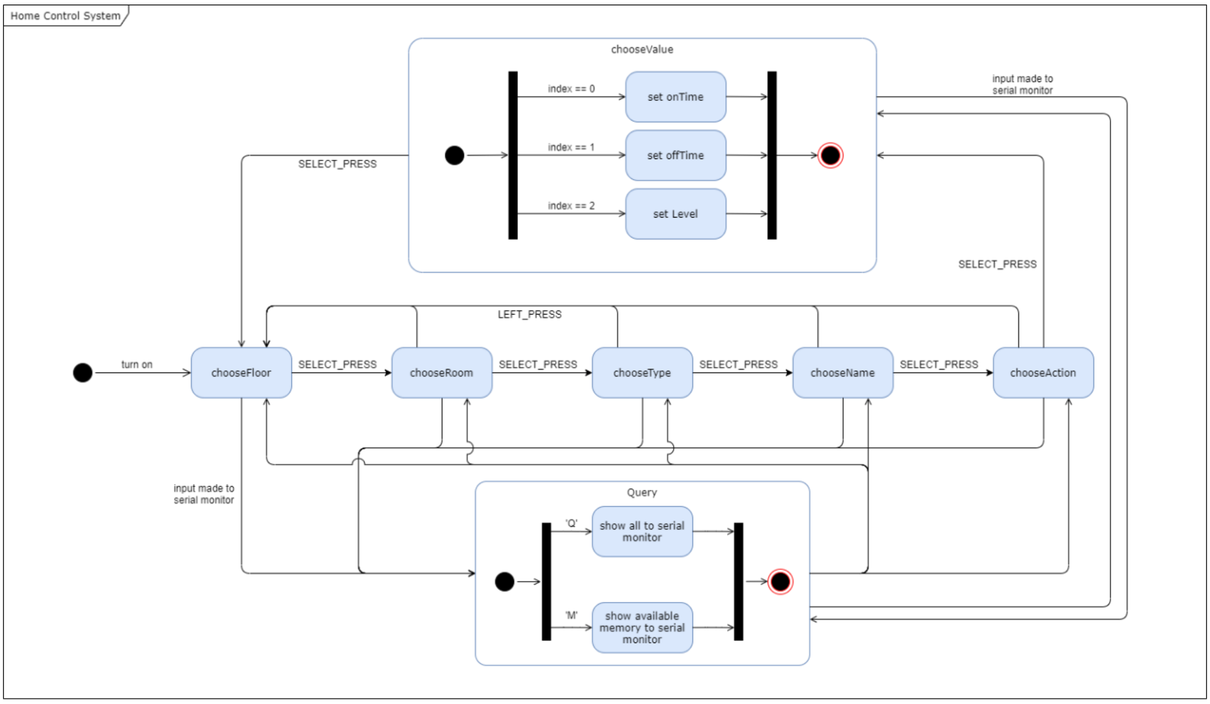


Figure 8: State diagram of my system

I decided to use the states above as I thought it was fitting to have a state for each time the user makes a selection on the Arduino shield; for example, when the user chooses a floor – by pressing SELECT – the state will move from chooseFloor to chooseRoom. Additionally, when the user presses LEFT, the system will cause the menu to reset and go back to the first state, chooseFloor. The state chooseValue is a substate itself; within the chooseValue state, there are three different states the system can go into: set onTime, set offTime and set Level. The system decides which substate to go into by using an index variable in the code itself – this index tells the system what value the user wanted to change in the data structure, the index variable is assigned in the chooseAction state. A more fitting name could have been used to describe this index variable, but index this variable is reused throughout the program as a measure to save memory.

Inputs (taken from the Arduino shield):

- Floor: user chosen floor in the house
- Room: user chosen room in the house
- Type: chosen type in which the user wants to change i.e., temperature or lighting
- Device: chosen device name the user wants to change
- Action: either on time, off time or level
- Value: value that the user wants to change the temperature or lighting to

Outputs (printed to the serial monitor):

- Update value: tells the user the lighting/heating level in the house has changed.

The user will have to select a room before they can set the lighting/heating value of a given room. Once a room has been chosen, the user can then choose between heating or lighting and then update the values of this in a given range (0 to 100). The user must submit their value changes before they can update any other values in the house.

Testing

Test/debugging code

To aid the development of my system, I wrote lines of code which wrote to the Serial monitor to allow me to see what variables were being used by the system itself. Upon developing the data structure, a lot of experimentation was done to also make sure the structure worked as intended – this was done again by writing to the serial monitor.

However, now that the system is finalised, I had to keep some lines for debugging in the case of maintenance or updating. To accommodate this, I defined `DEBUG` which can be commented in or out for debugging (**Figure 9**); along with this, I have used a number of #ifdef derivatives (for example in **Figure 10** and **Figure 12**) to write to the Serial Monitor.

v4_nested_struc | Arduino 1.8.13 (Windows Store 1.8.42.0)

File Edit Sketch Tools Help

v4_nested_struc

```
#define DEBUG

#include <Wire.h>
```

Figure 9: The `DEBUG` is defined at the top of my code.

v4_nested_struc | Arduino 1.8.13 (Windows Store 1.8.42.0)

File Edit Sketch Tools Help

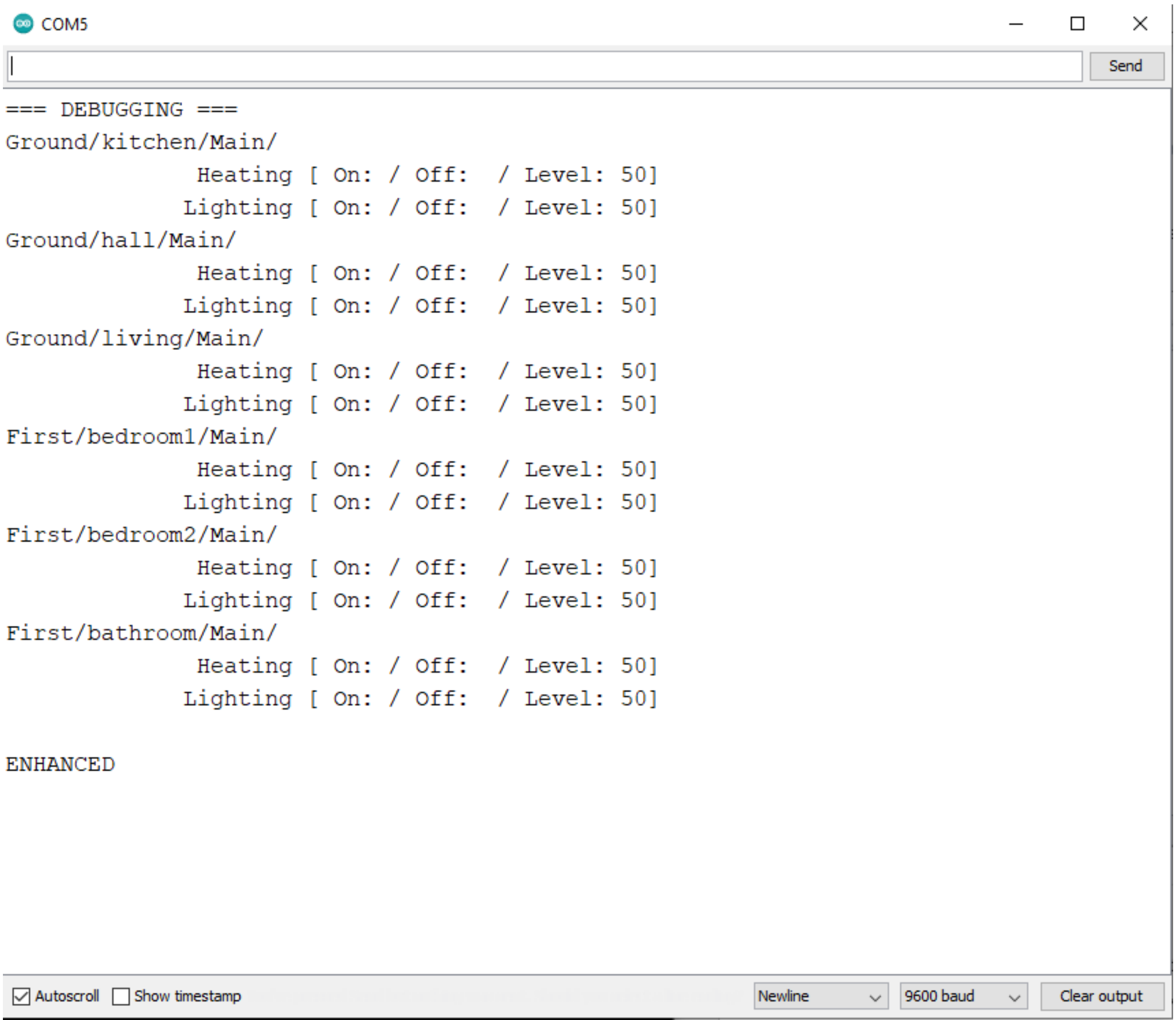
v4_nested_struc

```
void setup() {
  // put your setup code here, to run once:
  lcd.begin(16,2);
  Serial.begin(9600);

  setupRooms();
  #ifdef DEBUG
  Serial.println(F("=== DEBUGGING ==="));
  showAll();
  #endif
  Serial.println(F("ENHANCED"));
  Serial.println();
  lcd.print("Welcome");
}
```

Figure 10: An example of the #ifdef derivative being used in my system's setup code.

In my debugging code, the system only writes to the serial monitor when the user has advanced in the system's Menu and when the system is first booted (this will show the entire data structure - this is to show the data structure is working as intended, shown in **Figure 11**).




```
=== DEBUGGING ===
Ground/kitchen/Main/
              Heating [ On: / Off:  / Level: 50]
             Lighting [ On: / Off:  / Level: 50]
Ground/hall/Main/
              Heating [ On: / Off:  / Level: 50]
             Lighting [ On: / Off:  / Level: 50]
Ground/living/Main/
              Heating [ On: / Off:  / Level: 50]
             Lighting [ On: / Off:  / Level: 50]
First/bedroom1/Main/
              Heating [ On: / Off:  / Level: 50]
             Lighting [ On: / Off:  / Level: 50]
First/bedroom2/Main/
              Heating [ On: / Off:  / Level: 50]
             Lighting [ On: / Off:  / Level: 50]
First/bathroom/Main/
              Heating [ On: / Off:  / Level: 50]
             Lighting [ On: / Off:  / Level: 50]

ENHANCED
```

Figure 11: DEBUG upon system start up.

Debugging code has also been written the extension activities I have attempted; the debugging code written print to the Serial Monitor, as shown in **Figure 12**.

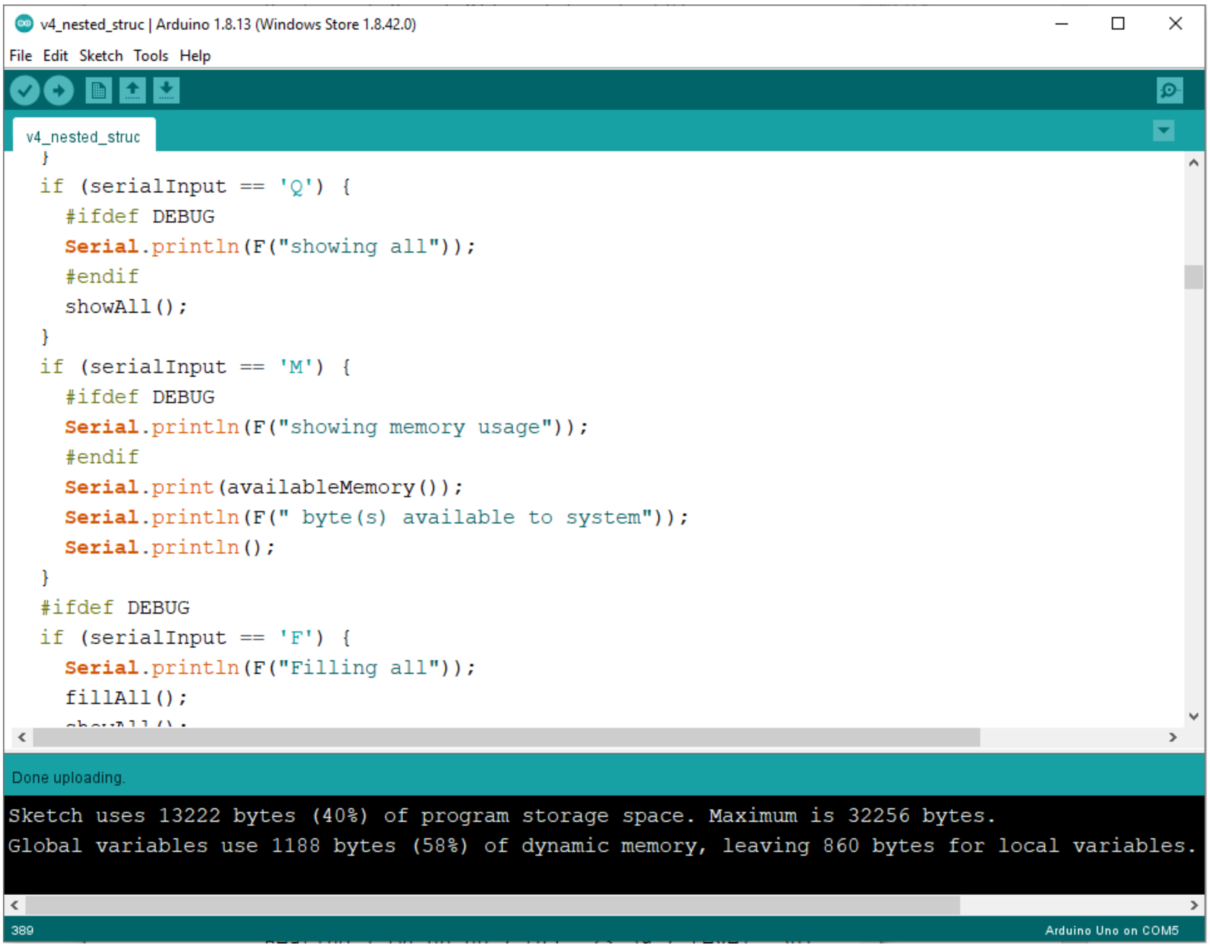


Figure 12: The debugging code written to assist debugging for the extension activities.

An additional bit of code I added for testing purposes is my `fillAll()` function (**Figure 14**); this function fills up the data structure with times so that the tester does not need to manually /physically fill out each on/off time via the Arduino. To access this function, `DEBUG` has to be defined – if already defined, the user only has to enter a single 'F' character to the Serial monitor (code for this can be seen in **Figure 13**) and all empty fields should be filled. This was included to make sure that the system could handle a full data structure. Previous experimentations with the data structure showed that my program (during development) would corrupt the data if the program used too much memory.

A running of this feature can be seen in Figure 20.

```
v4_nested_struc | Arduino 1.8.13 (Windows Store 1.8.42.0)
File Edit Sketch Tools Help
v4_nested_struc

  }
  #ifdef DEBUG
  if (serialInput == 'F') {
    Serial.println(F("Filling all"));
    fillAll();
    showAll();
  }
  #endif
```

Figure 13: code added to handle the 'F' input from the serial monitor. This is encapsulated in an `#ifdef` to make sure it is only available when debugging; after calling `fillAll()` the `showAll()` function is also called so the user can see that all devices have now got on and off times.

```
v4_nested_struc | Arduino 1.8.13 (Windows Store 1.8.42.0)
File Edit Sketch Tools Help
v4_nested_struc

}

void fillAll() {
  for (uint8_t i=0; i<roomCount; i++) {
    for (uint8_t j=0; j<maxDevicesPerRoom; j++) {
      arrHouse[i].devices[j].on_time = "00:00";
      arrHouse[i].devices[j].off_time = "23:59";
    }
  }
}
```

Figure 14: my function `fillAll()` which uses a nested for loop to assign on and off times to all devices in the house.

Test Plan

In this section I will be detailing the tests I carried out to ensure all parts of my system work. I have colour coded the Actual outcome column for each test section to make clear which bits work and do not work, a key has been made below.

Key colour	What the colour means
(green)	Works as expected – nice
(blue)	Works as expected but some bugs can occur from this
(orange)	Does not work as expected at all

Using the LCD screen interface - menu

In this section, I will be detailing some tests that can be carried out to ensure that the LCD screen menu works as intended.

What is meant by *menu cycling*? This is when the user is picking the Floor, Room, Type, Name or Action. **This does section does not include testing for setting the on time, off time or level for a device.**

Test description	Expected outcome	Actual outcome
Pressing RIGHT while cycling menu	Floor/Room/Type/Name/Action state should only let the user cycle the different Floor/Room/Type/Name/Action in the house.	Same as expected outcome
Pressing SELECT when cycling the menu	System should move into the next state and this should be reflected on the LCD screen. "OK." Should display on the LCD screen to notify the user the button press has been acknowledged by the system. The LCD screen should show: [Last chosen option] / [available options]	Same as expected outcome BUT holding down the SELECT button can advance states but not refresh the LCD screen
Pressing LEFT while cycling menu	When LEFT is pressed, the system should make the user start again with the menu process by making them choose starting from the floor. The LEFT button should only work when choosing the Room, Type, Name or Action.	Same as expected outcome
Pressing DOWN while cycling menu	Nothing should happen	Same as expected outcome
Pressing UP while cycling menu	Nothing should happen	Same as expected outcome

Using the LCD screen interface – choosing on/off times

This section will detail the tests associated with applying on or off times to an appliance.

Test description	Expected outcome	Actual outcome
Pressing RIGHT while setting Hours	Should move cursor right and allow the user to alter the minutes	Same as expected outcome
Pressing LEFT while setting Hours	Nothing should happen	Same as expected outcome
Pressing RIGHT while setting Minutes	Nothing should happen	Same as expected outcome
Pressing RIGHT while setting Minutes	Should move cursor left and allow the user to alter the hours	Same as expected outcome
Pressing UP while setting Hours	Should increment the hours by 1, hours should not exceed 23	Same as expected outcome
Holding UP while setting Hours	Should increment the hours until the user lets go, hours should not exceed 23	Same as expected outcome
Pressing DOWN while setting Hours	Should decrement the hours by 1, hours should not go below 0	Same as expected outcome
Holding DOWN while setting Hours	Should decrement the hours until the user lets go, hours should not go below 0	Same as expected outcome
Pressing UP while setting Minutes	Should increment the hours by 1, minutes should not exceed 59	Same as expected outcome
Holding UP while setting Minutes	Should increment the minutes until the user lets go, minutes should not exceed 59	Same as expected outcome
Pressing DOWN while setting Minutes	Should decrement the minutes by 1, minutes should not go below 0	Same as expected outcome
Holding DOWN while setting Minutes	Should decrement the hours until the user lets go, minutes should not go below 0	Same as expected outcome
Pressing SELECT	System should reset the menu to allow the user to set another value for the home controller. The system should also update the data structure appropriately to match the user's actions. "SET." Should display on the LCD screen to notify the user the value has been set	Same as expected outcome However, holding down SELECT can cause the menu to continue cycling without refreshing the LCD screen

Using the LCD screen interface – choosing a Level

This section will detail the tests associated with applying levels to an appliance.

Test description	Expected outcome	Actual outcome
Pressing RIGHT while setting level	Nothing should happen	Same as expected outcome
Pressing LEFT while setting level	Nothing should happen	Same as expected outcome
Pressing UP while setting level	Should increment level by 1, the level should not exceed 100	Same as expected outcome
Holding UP while setting level	Should increment level until the user lets go, the level should not exceed 100	Same as expected outcome
Pressing DOWN while setting level	Should decrement level by 1, the level should not go below 0	Same as expected outcome
Holding UP while setting level	Should increment level until the user lets go, the level should not go below 0	Same as expected outcome
Pressing SELECT	System should reset the menu to allow the user to set another value for the home controller. The system should also update the data structure appropriately to match the user's actions. "SET." Should display on the LCD screen to notify the user the value has been set	Same as expected outcome However, holding down SELECT can cause the menu to continue cycling without refreshing the LCD screen

Using the Serial Monitor – DEBUGGING MODE

This section will detail how to test the Serial Monitor inputs **in debugging mode**.

Test description	Expected outcome	Actual outcome
Entering 'Q' into the Serial Monitor	"`Showing all`" should be written to the Serial monitor followed by a display of all values assigned to all devices in the house.	Same as expected outcome
Entering 'q' into the Serial Monitor	Nothing should happen.	Same as expected outcome
Entering 'M' into the Serial Monitor	"`Showing memory usage`" should be written to the Serial monitor, followed by a message which tells the user how many bytes of memory is available to the system.	Same as expected outcome
Entering 'm' into the Serial Monitor	Nothing should happen.	Same as expected outcome
Entering 'F' into the Serial Monitor	"`Filling all`" should be written to the Serial Monitor, then the serial monitor will display all values assigned to all devices in the house.	Same as expected outcome
Entering 'f' into the Serial Monitor	Nothing should happen	Same as expected outcome
Entering anything but {'Q', 'M', 'F'} into the Serial monitor	Nothing should happen	Same as expected outcome
Serial monitor can take input regardless of system state	The system should be able to handle a query regardless of the system's state (i.e. chooseRoom, chooseFloor, etc)	Same as expected outcome

Using the Serial Monitor – outside of debugging mode

This section will detail how to use the Serial monitor outside of debugging mode.

Test description	Expected outcome	Actual outcome
Entering 'Q' into the Serial Monitor	Serial monitor should display all values assigned to all devices in the house.	Same as expected outcome
Entering 'q' into the Serial Monitor	Nothing should happen.	Same as expected outcome
Entering 'M' into the Serial Monitor	Serial monitor should display a message which tells the user how many bytes of memory is available to the system.	Same as expected outcome
Entering 'm' into the Serial Monitor	Nothing should happen.	Same as expected outcome
Entering 'F' into the Serial Monitor	Nothing should happen	Same as expected outcome
Entering 'f' into the Serial Monitor	Nothing should happen	Same as expected outcome
Entering anything but {'Q', 'M'} into the Serial monitor	Nothing should happen	Same as expected outcome
Serial monitor can take input regardless of system state	The system should be able to handle a query regardless of the system's state (i.e., chooseRoom, chooseFloor, etc)	Same as expected outcome

Extension features

Query

To accommodate the Query extension, several lines were added to the main loop of my code, this is shown below in **Figure 15**.

```
static char serialInput;

  if (Serial.available() > 0) {
    serialInput = Serial.read();
  }
  if (serialInput == 'Q') {
    #ifdef DEBUG
    Serial.println(F("showing all"));
    #endif
    showAll();
  }
```

Figure 15: The additional code needed to meet the Query extension requirements.

I decided to use a char to represent the input from the serial monitor, this was done to use as little memory as possible. When the system detects that the user has entered ‘Q’, the system will call the subroutine `showAll()`, of which shows all the on times, off times and levels of all appliances in the house – the use of this function can be seen in **Figure 16** and **Figure 17**.

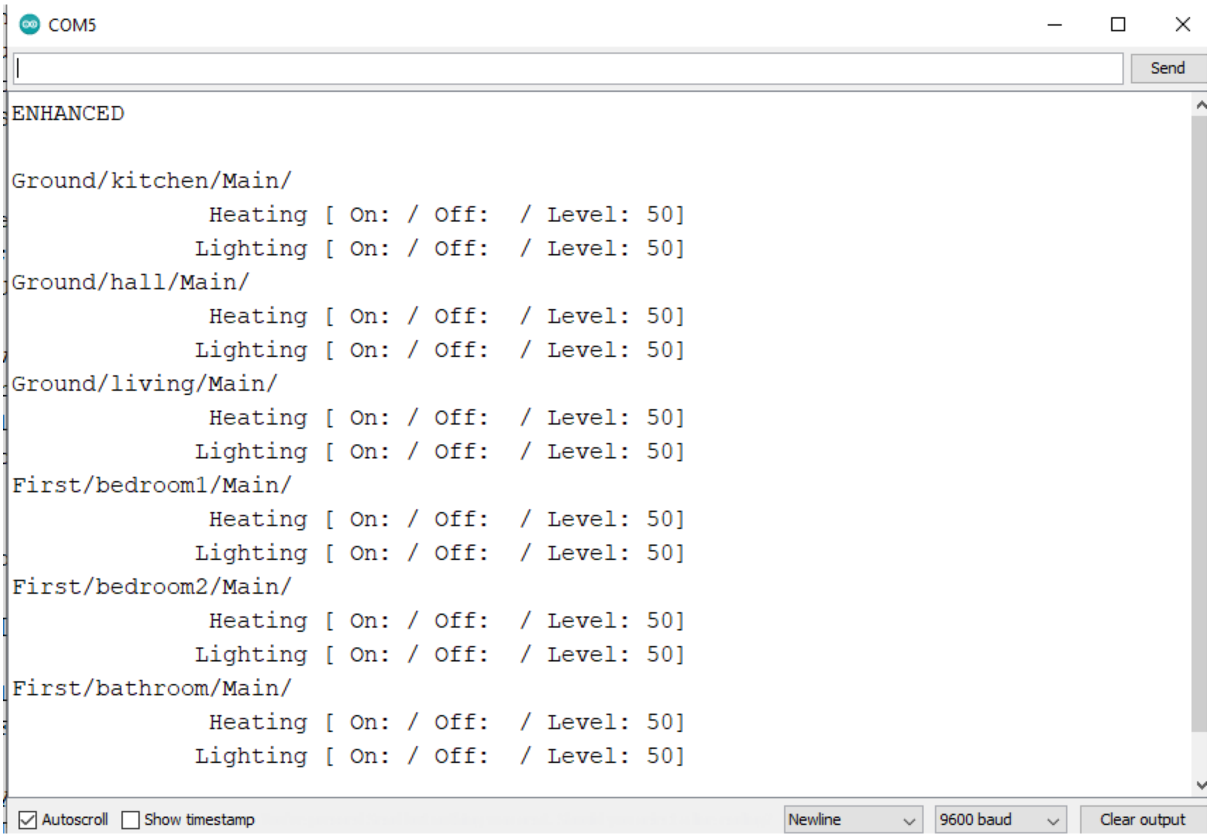


Figure 16: When the user enters char 'Q' to the Serial monitor.

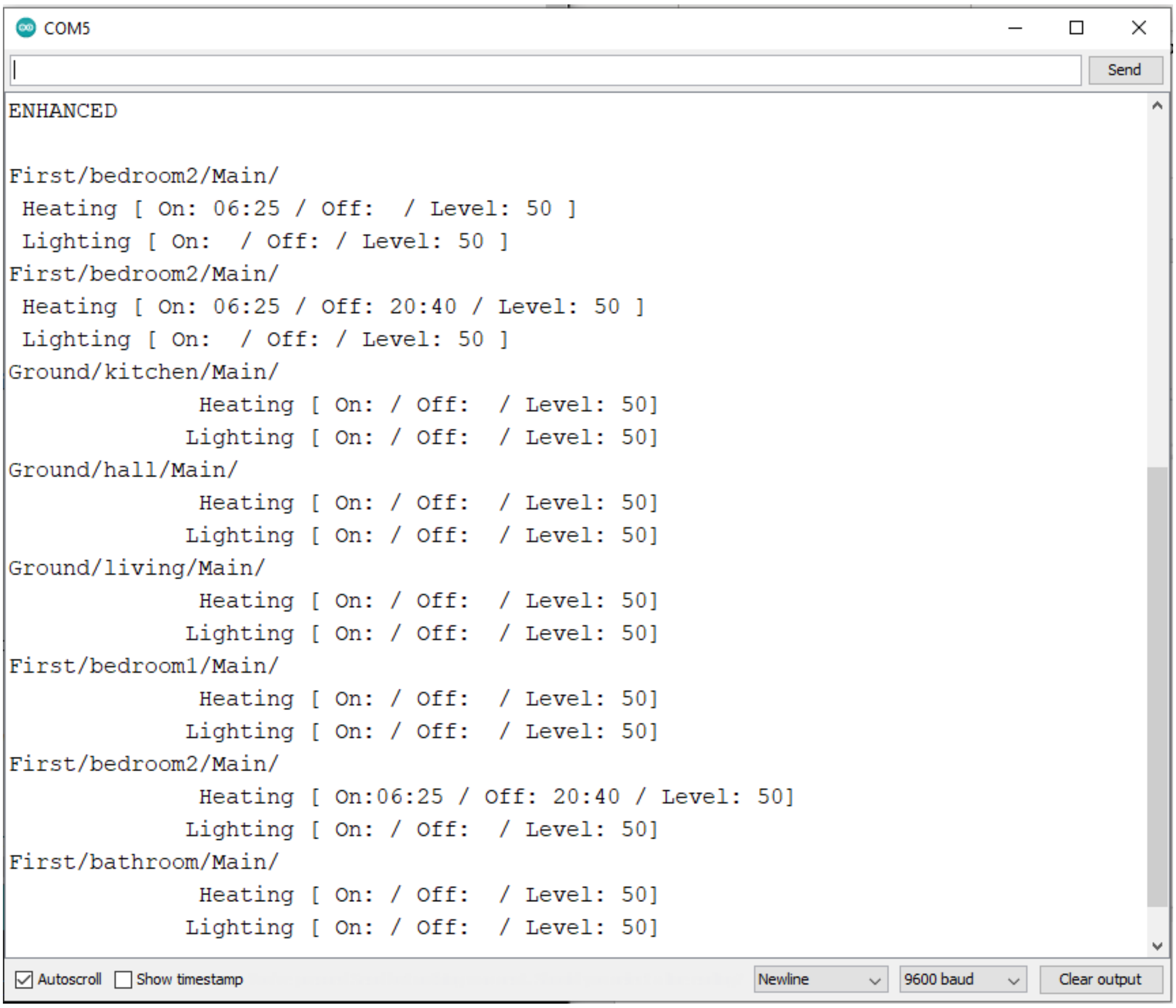


Figure 17: Entering char 'Q' after updating the on and off time for bedroom2's heating.

Memory

To implement memory querying, I added a new function and a few more lines under those written for the query extension task, this can be seen in **Figure 18** and **Figure 19** below.

When doing this task, I used the internet for help and what I found was that there is a library called MemoryFree which I could use but this was not listed in the allowed libraries in the coursework specification. As a result of this, another method had to be used. What I did find was a forum post which detailed an alternative way to find the system's memory usage (Esben, 2012) which I have followed to implement this extension task.

```
  if (serialInput == 'M') {
    #ifdef DEBUG
    Serial.println(F("showing memory usage"));
    #endif
    Serial.print(availableMemory());
    Serial.println(F(" byte(s) available to system"));
  }
```

Figure 18: Additional code added to handle the serial monitor user input 'M' which will then call the function `availableMemory()`.

```
int availableMemory() {
  /*
   *  taken from: https://stackoverflow.com/questions/8649174/checking-memory-footprint-in-
arduino
   *  user Esben ~~ answered Jan 2 '12 at 20:16
   */
  int size = 2048;  // use 2kb or 2048 bytes as the arduino uno is ATmega328P ~ 2KB of SRAM
  byte *buf;
  while ((buf = (byte *) malloc(--size)) == NULL);
      free(buf);
  return size;
}
```

Figure 19: function implemented to find the amount of free memory left in the system (Esben, 2012). This function will return the number of unused bytes by the system.

The function `availableMemory()` uses a two data types.

- `int size = 2048;`
    - This integer represents the size of the Arduino's memory. It is set to 2048 as the Arduino has 2048 bytes of memory, or alternatively, 2Kb of memory. The function will return this integer at the end, after it has run a while loop to check for any unused space.
    - **The integer returned is the amount of memory (in bytes) that is free to use by the system.**
- `byte *buf;`
    - Declaration of a pointer variable (named buf) which points to a byte variable. This will be used in conjunction with a while loop to find the amount of free memory in the system.

The while loop in `availableMemory()` goes through each of the 2048 memory addresses of Arduino's memory and checks whether or not the memory address is empty or not by memory allocation – this will cycle each address in descending order (from 2048 to 0). If the given memory address is empty, the memory address being pointed to is deallocated back to the system.

Running the memory extension can be seen in Figure 20, after filling the data structure, it can be observed that the number of bytes available to the system decreases.

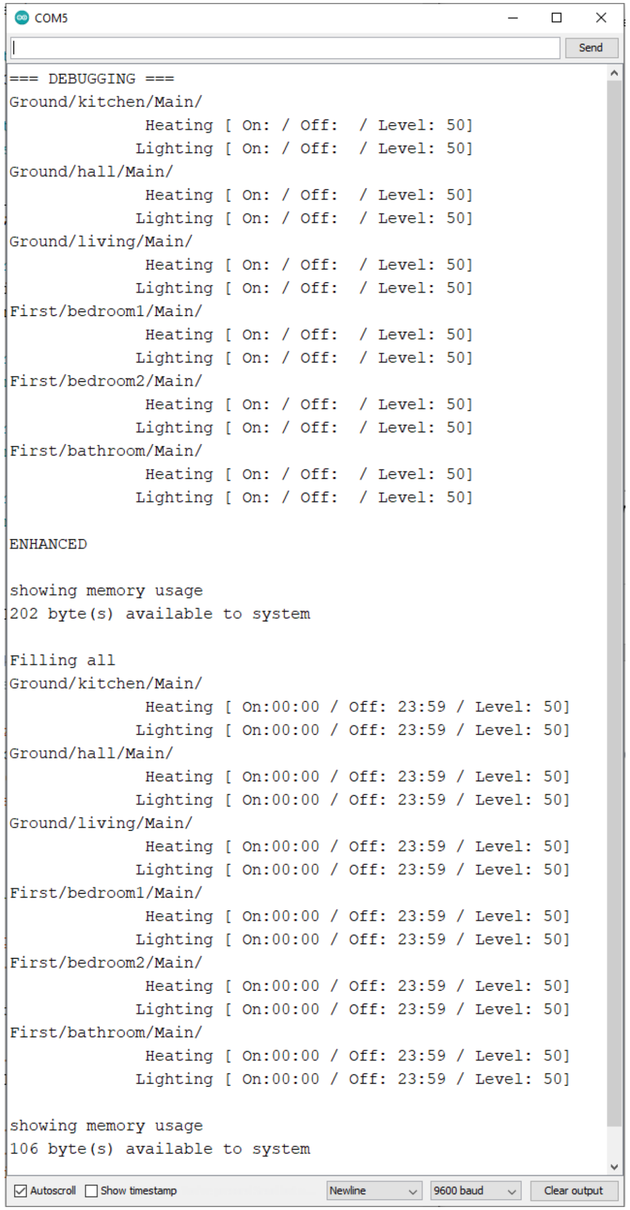


Figure 20: Running the memory extension in debugging mode which allows me to fill the data structure with on/off times automatically (Image on right).

LAMP
Not implemented

Outside
Not implemented

SOFT
Not implemented

EEPROM
Not implemented

Conclusions

Overall, I am quite satisfied with my implementation of the home control system although I know that there are several improvements I could make. I am pleased to say that my program works as intended and that I have met some of the extension tasks.

I quite enjoyed this exercise, especially during the development of my data structure as can be seen in the first section of this development documentation. I found it quite challenging to make everything work with the Arduino's limited memory, but this worked out in the end by using alternatives, for example, instead of using the `int` data type, I used `unit8_t` instead; by doing this I was able to save a byte of memory every time I did this (as the `int` data type is 16 bits where the `uint_8t` data type is 8 bits long). Additionally, writing to flash memory helped to cut down my code's memory usage, of which helped my code run more reliably and smoothly. Overall, the memory optimisation (although could be better, detailed below) was something I am proud to talk about in my implementation of the home control system.

Further improvements could have been made to further reduce my RAM usage however, this is most notable for my data structure itself. I am aware that it was possible for me to reduce the memory usage if I had used an array of chars instead of the String data type, but I found that using char array instead of Strings was very buggy, the outcomes of this can be seen in Figure 23 and Figure 24. If I had got this to work successfully, I think it would have been possible for me to add more extension features (such as the lamp and outside extensions) to my code as there might have been sufficient memory to allow for everything to run smoothly. I do personally feel however that I have created my data structure in a way such that it should not be hard to more elements to my data structure – if there were no memory constraints.

Another part of this exercise worth mentioning is the idea that holding the SELECT button down while cycling the menu will cause the menu to advance without the change being reflected on the LCD screen. Attempts were made to fix this issue, but all attempts made resulted in the system not being able to detect a button press. I know that there is a fix to this, but it would probably mean an overhaul of how my code transitions through states.

Internal References

Here lies any references I have made to any bits of my code along with an explanation to why/how it was used.

```
void setupRooms() {
  deviceOnStartUp = {"", "", 50};
  //             {floor, room, name, [HEATING, LIGHTING]}
  arrHouse[0] = {myFloors[0], "kitchen", "Main", {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[1] = {myFloors[0], "hall", "Main", {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[2] = {myFloors[0], "living", "Main", {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[3] = {myFloors[1], "bedroom1", "Main", {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[4] = {myFloors[1], "bedroom2", "Main", {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[5] = {myFloors[1], "bathroom", "Main", {deviceOnStartUp, deviceOnStartUp}};
}
```

```
const uint8_t floorCount = 2;
const String myFloors[floorCount] = {"Ground", "First"};
```

Figure 21: The subroutine `setupRooms()` which I used to assign values to every room in the house represented by the system, along with the constant values which are used in the setup of the rooms. The on and off times for `deviceOnStartUp` have been left blank as I felt that a system should not have default on/off times, instead the user sets them themselves.

```
void showAll() {
    for (uint8_t i=0; i<roomCount; i++) {
    Serial.println(arrHouse[i].room_floor + "/" + arrHouse[i].room_name + "/" +
arrHouse[i].device_name + "/");
    for (uint8_t j=0; j<maxDevicesPerRoom; j++) {
      if (j==0) {
        Serial.print(F("               Heating [ On:"));
      }
      if (j==1) {
        Serial.print(F("              Lighting [ On:"));
      }
      Serial.print(arrHouse[i].devices[j].on_time);
      Serial.print(F(" / Off: "));
      Serial.print(arrHouse[i].devices[j].off_time);
      Serial.print(F(" / Level: "));
      Serial.print(arrHouse[i].devices[j].level);
      Serial.println(F("]"));
    }
  }
  Serial.println();
}
```

Figure 22: the subroutine `showAll()` which is used in debugging and in the QUERY extension, uses a nested for loop to cycle through my nested structure data structure

```
struct device_s {
  char on_time[5], off_time[5];
  uint8_t level;
} deviceOnStartUp;

const uint8_t maxDevicesPerRoom = 2;
typedef struct room_s {
  char room_floor[6], room_name[8], device_name[4];
  device_s devices[maxDevicesPerRoom];
} room_t;

const uint8_t roomCount = 6;
room_t kitchen, hall, living, bed1, bed2, bath;
room_s arrHouse[roomCount] = {kitchen, hall, living, bed1, bed2, bath};

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  setupRooms();
  showAll();
}

void loop() {
  // put your main code here, to run repeatedly:

}

void setupRooms() {
  deviceOnStartUp = {{'0', '0', ':', '0', '0'}, {'2', '3', ':', '5', '9'}, 50};
  //            {floor, room, name, [HEATING, LIGHTING]}
  arrHouse[0] = {{'G', 'r', 'o', 'u', 'n', 'd'}, {'k', 'i', 't', 'c', 'h', 'e', 'n', '\0'},
{'M', 'a', 'i', 'n'}, {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[1] = {{'G', 'r', 'o', 'u', 'n', 'd'}, {'h', 'a', 'l', 'l', '\0'}, {'M', 'a',
'i', 'n'}, {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[2] = {{'G', 'r', 'o', 'u', 'n', 'd'}, {'l', 'i', 'v', 'i', 'n', 'g', '\0'},
{'M', 'a', 'i', 'n'}, {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[3] = {{'F', 'i', 'r', 's', 't', '\0'}, {'b', 'e', 'd', 'r', 'o', 'o', 'm', '1'},
{'M', 'a', 'i', 'n'}, {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[4] = {{'F', 'i', 'r', 's', 't', '\0'}, {'b', 'e', 'd', 'r', 'o', 'o', 'm', '2'},
{'M', 'a', 'i', 'n'}, {deviceOnStartUp, deviceOnStartUp}};
  arrHouse[5] = {{'F', 'i', 'r', 's', 't', '\0'}, {'b', 'a', 't', 'h', 'r', 'o', 'o', 'm'},
{'M', 'a', 'i', 'n'}, {deviceOnStartUp, deviceOnStartUp}};
}

void showAll() {
    for (uint8_t i=0; i<roomCount; i++) {
      Serial.print(arrHouse[i].room_floor);
      Serial.print(F("/"));
      Serial.print(arrHouse[i].room_name);
      Serial.print(F("/"));
      Serial.print(arrHouse[i].device_name);
      Serial.println(F("/"));
    }
}
```

Figure 23: My attempt at using char array instead of string for my data structure, which does not work as intended

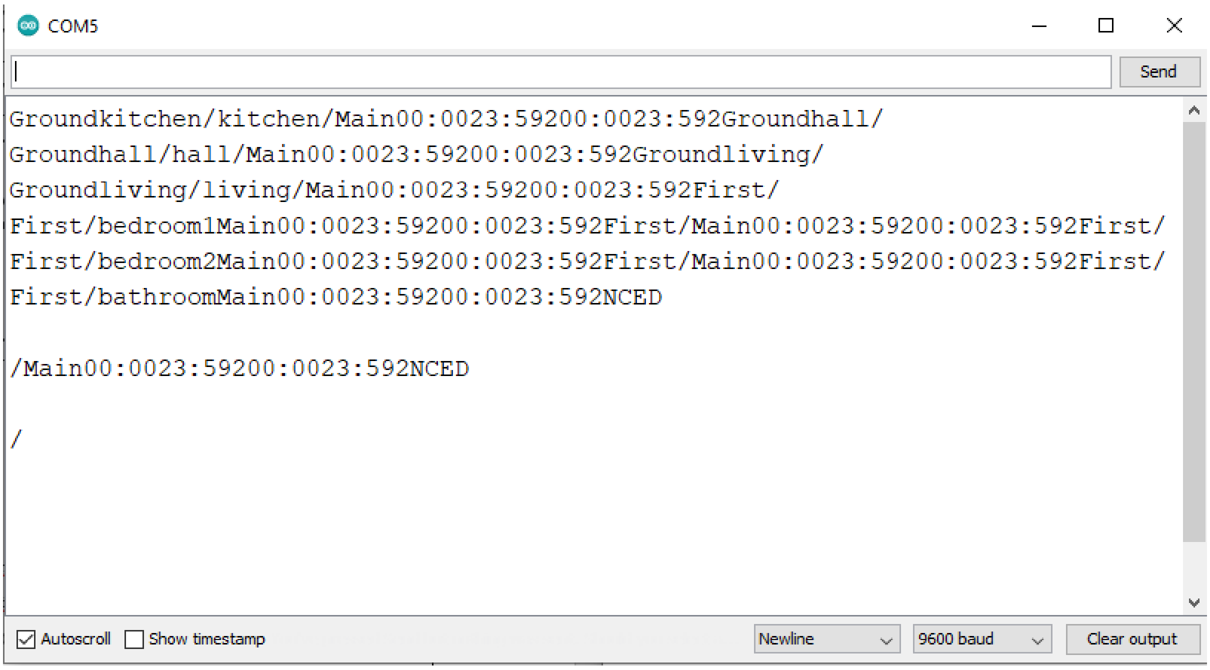


Figure 24: output of my char array structure – does not seem to be working as intended

External References

Esben, 2012. *Checking memory footprint in Arduino: stackoverflow.* [Online]
Available at: https://stackoverflow.com/questions/8649174/checking-memory-footprint-in-arduino
[Accessed 5 April 2021].