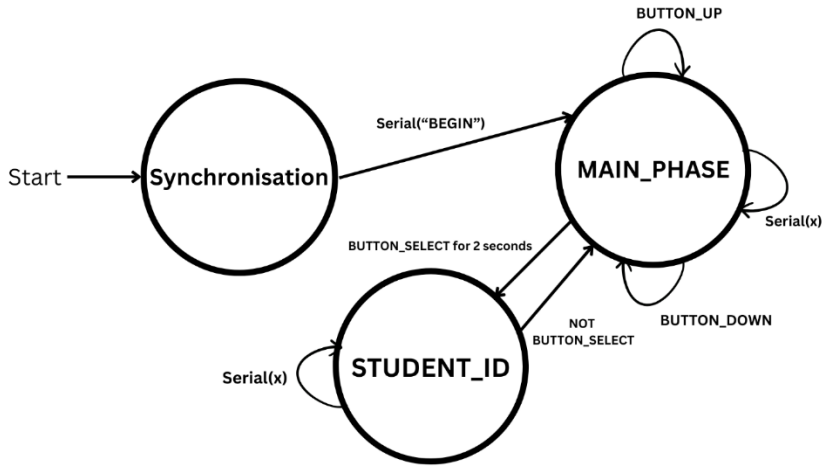
24COA202 Coursework

F422436

Semester 1

1 FSMs

FSM := $(\Sigma, S, s0, \delta, F)$
∑ := {BUTTON_UP, BUTTON_DOWN, BUTTON_SELECT, Serial(BEGIN), Serial(x)}
Where x is a given command from the allowed list {ADD, PST, GRD, SAL, CJT, DEL, ROM}
S := {SYNCHRONISATION, MAIN_PHASE, STUDENT_ID}
s0 = SYNCHRONISATION
F = ∅
O := ∅
FSM Diagram:



When transitioning through states, the backlight changes, the screen is cleared, and global variables are set to a state which would not affect the new state. For example, when entering "STUDENT_ID", the backlight goes to purple, the screen is cleared, the scroll is set to false, and the relevant data is displayed. While the state still listens for button inputs and serial input.

2 Data structures

The main data structure used in this system is a doubly linked list. This was chosen due to the simple implementation and limits wasted memory due to it being a dynamic solution. A doubly linked list was chosen to allow for the nodes to be traversed forward and backward in the "MAIN_PHASE" without the need for extensive processing. The data structure is mainly implemented using 2 structs and then several functions to handle operations regarding the LL. One struct is responsible for the nodes which make up the LL whilst the other holds the actual employee data (given below). The employee data is abstracted into a separate node to allow for better maintainability and to allow functions to only focus on the employee data if that is all the function needs (ie removes repetitive dereferencing). The functions operating on the LL include "createNode", "addNode", "findEmployee", "deleteNode". "createNode" and "addNode" are responsible for creating a node out of employee data and adding it to the LL (using malloc). "findEmployee" is responsible for finding and returning the pointer for a valid node. "deleteNode" is responsible for removing a node from the linked list and freeing the data. "updateNode" would update a node with new employee data but is not actually used within the system, instead updates to nodes are handled within "handleCommand" (using "findEmployee"). These functions make use of global pointers for "root", "current_node" and "active_node". "root" always points to the root of the LL (or null if no nodes in list). This is essential for operations to ensure the start of the list can be quickly and easily found to traverse through the list. "current_node" is used to point to the current node being inspected in a traversal linked with an operation on a node in the LL. This is a global pointer as it is used throughout the system and reduces the chance of an individual function reaching the memory limit (by limiting the amount of local variables used) – ie to delete a node only 2 bytes worth of memory is needed to store local variables during the call – this helps ensure memory can be made available even when low on memory. "active_node" points to the node currently being displayed on the lcd screen to ensure LL operations are independent of the lcd screen.

Structs:

```
// Total Storage Space for node w/data: 44bytes
struct employee {                          // 38 bytes
  int  job_grade;                          // 2 bytes
  bool on_pension;                         // 1 byte
  char salary[9];                          // 9 bytes
  char id[8];                              // 8 bytes
  char job_title[18];                      // 18 bytes
};
struct employee_node {                     // 6 bytes
  employee* employee_data;                 // 2 bytes.
  employee_node* next_employee;            // 2 bytes.
  employee_node* previous_employee;        // 2bytes.
};
employee_node* root = nullptr; // Always points to root of LL.
employee_node* current_node = nullptr;// Current node for LL operations.
employee_node* active_node = nullptr; // Currently viewed node on screen.
```

3 Debugging

For debugging my system I used C++ macros, this was implemented in several different ways, using a “const_debug”, and 2 levels for debugging. “const_debug” is intended to be shown all the time to give more descriptive/specific messages to what is happening in the system to the user. Then 2 levels consist of one level for messages which would be sent very frequently (e.g. checking changes in millis) and the other is for less frequent yet still important messages (e.g. debugging a function handling a certain functionality which only gets fired occasionally), this ensures that if I am debugging a specific function there is no overload of debug messages so the messages I want to see don’t get lost. I also edited the testing python program to print serial responses even if they were not prepended by “DONE!”, “ERROR:” or “DEBUG:”. Furthermore, I added to the test messages and improved the program to change the terminal text colour for successful and unsuccessful tests as this made executing many tests a lot quicker and easier to identify any issues and what might be causing them as well as more thoroughly testing the system. I also made physical notes to work through algorithms to ensure they were working as I intended including producing truth tables for certain if statements to ensure they were firing in the correct scenarios.

4 Reflection

Upon reflection of my code, the main bit of code I would like to improve is "handleCommand". "handleCommand" was meant to be a smaller function that handled identifying commands, handling data validation, and then running the necessary doubly LL (Linked List) operation functions. However, the actual implementation of "handleCommand" is very inelegant and long. This was caused by backtracking my plan for implementing an abstracted "dataValidation" function mid-development, causing each command implementation to be inconsistent, including some redundant checks. I would fix this by refactoring to make all the "dataValidation" abstracted again and to have consistency between the commands. Steps towards this were taken towards the end of development as common data validation were conflated to reduce redundant code. To further improve, I could refactor "updateNode" (an oversight early on in development) to handle the updating of nodes into a separate function. More abstraction and modularity would allow for better modular testing to catch and debug bugs earlier before reaching functionality testing whilst making maintainability easier.

"deleteNode" is inconsistent with other LL operations as it handles error messages within the function instead of allowing the caller function to manage the errors. The calling function would handle the error messages to make it consistent, and specific outcome comments would become debug messages.

I would have liked to decrease the use of String; however, this would have more major refactoring requiring longer time for development. A step for this would be to reduce the amount of substring calls; this would improve the system's security and could lead to fewer fragmentation concerns.

There is an identical code for checking valid "job_title" in 2 places, which could be abstracted and would make debugging only necessary for one place.

I am happy with the implementation of the LL and consistency with variable and function naming.

Extension Features

5 UDCHARS

Creating custom characters for the system makes the system more user-friendly as the system uses more realistic and defined arrows instead of "^" & "v". To create UDCHARS I used the online matrix generator (https://omerk.github.io/lcdchargen/). Using the pixel selector produced the given byte matrices shown below. I decided to leave a gap in between the arrows to ensure it looked like separate arrows instead of one big arrow. Once the lcd screen had been initialised, I created custom characters from the character matrices (shown below). The arrows are then displayed conditionally during the "MAIN_PHASE" given there are nodes surrounding the "active_node".

Pixels

```
byte custom_up_arrow[8] = {
  0b00100,
  0b01110,
  0b11111,
  0b00100,
  0b00100,
  0b00100,
  0b00100,
  0b00000
};
```

Pixels

```
byte custom_down_arrow[8] = {
  0b00000,
  0b00100,
  0b00100,
  0b00100,
  0b00100,
  0b11111,
  0b01110,
  0b00100
};
```

```
// UDCHARS Initialisation
lcd.createChar(0, custom_up_arrow);
lcd.createChar(1, custom_down_arrow);
```

```
// Display main menu (test arrows -> dependent on if active_node pointers)
// up_arrow
if (active_node != NULL && active_node->previous_employee != NULL) {
  lcd.setCursor(0,0);
  lcd.write((uint8_t)0);
} else {
  lcd.setCursor(0,0);
  lcd.print(" ");
}
// down_arrow
if (active_node != NULL && active_node->next_employee != NULL) {
  lcd.setCursor(0,1);
  lcd.write((uint8_t)1);
} else {
  lcd.setCursor(0,1);
  lcd.print(" ");
}
```

6 FREERAM

The heavy lifting for this extension is done using the “MemoryFree.h” header file. This allows for this extension to be implemented simply by calling “freeMemory()”. Free memory is displayed when the system is in the “STUDENT_ID” state on the second row of the LCD screen to separate it from my student ID. However, “freeMemory()” is also used within the program as an added security feature. The maximum number of employees is set at 20, as the system should comfortably be able to hold this many employees (decided by the amount of free memory after global variables, ensures a sufficient buffer is still available for stack calls (for functions and local variables) to avoid memory collisions). However, as added security features, if there is less than 100 bytes of free RAM, then no new nodes will be added until more memory becomes available. As roughly 50 bytes would be needed to create a new node this buffer should still allow the system to run without memory collisions and run functions (including deleting nodes) – this buffer also assumes the memory may be slightly fragmented, meaning the useful memory is lower than this value (due to my use of String and other uses of memory throughout the system).

*Use of malloc also has null checkers to stop adding nodes when there is insufficient free memory - informs the user to delete some nodes first.

7 EEPROM

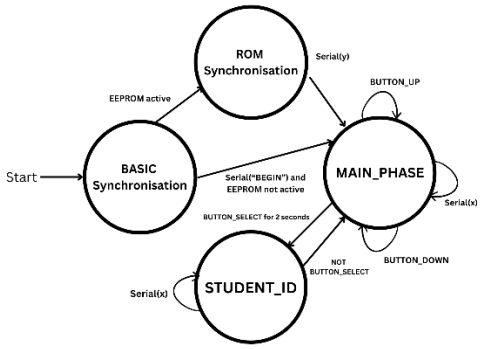
EEPROM Extension uses the "EEPROM.h" header file to read and write from the EEPROM. When the Arduino first starts up, the first byte is read to see if the EEPROM is actively being used to store employee data. If it is, then the command to complete the sync phase is read from EEPROM 1-10 addresses. When this sync message is entered, the employee stored count is read from the EEPROM, and subsequently, that amount * 38 (data size for employee data) is read from the EEPROM and made into nodes to be added to the LL.

When the ROM command is used in the serial monitor given it passes the validation. If EEPROM is already active, then it will deactivate it and write the deactivated state value to the EEPROM (then all the data will get overwritten the next time it is activated anyway). When the EEPROM is not active, and a valid ROM command is used. The eeprom_active is written as true. The command is written to the EEPROM (so the sync phase can be passed on the next boot-up), then the employee count, and then all the employee data is written in its current state.

EEPROM DESIGN= {0 : eeprom_active, 1-10: eeprom_command, 12: current_employee_count, 13->: employees (in blocks of 38 for 38 bytes per node}.

Due to the implementation of adding nodes, there should never be more than 20 employee nodes. This should comfortably be stored in the EEPROM as 1023 -12 = 1011. 1011/38 > 25. This means 20 employees should comfortably be stored in the EEPROM. This should ensure that all the data can be held in the SRAM when the EEPROM is being read back.

This approach only writes to the EEPROM when the eeprom_activated is true. The RAM should comfortably be able to read all the data back into memory from EEPROM. The memory usage for global variables has slightly increased for this extension as it now stores data for the EEPROM active state and the command (12 bytes extra). Due to the limit on employees, the max memory address' should also not be reached. The only concern is reaching the limit of read/writes for EEPROM if extensive use of the Arduino and system is used.



During the ROM Synchronisation state, the synchronisation command {y} is being waited for. Then all the data will be read from EEPROM.

8 SCROLL

This extension makes use the “TimerOne.h” header file. This allows for the “scrollHandler” function to create an interrupt at a given interval. This allows for the system to appear to simultaneously handle a scrolling job title and handle commands , states, displays etc.

When displaying a user, “displayEmpoyee” checks the length of the job title. If the job title cannot be displayed in it’s entirety at once, then the global variable “scroll” is set to true. This will allow scrolling on the next interrupt by “scrollHandler”. “scrollHandler” has an interval of 1 second, this ensures it does not interfere with other system functionality by always interrupting. As a result of this, “displayEmployee” will always just display the job title as normal and assume no scrolling (this will cause the first 7 characters to be displayed for a job title longer then 7 characters long). This ensures that something is displayed before “scrollHandler” is ran the next time. Scrolling is turned off whenever there is no active node or the state switches.

As the given rate of scrolling is 2 characters per second, the scrolling functionality is handled by “scrollHandler”. In this function, if scrolling is on and there is a valid job title which is longer then 7 characters, then a buffer is used to store the currently displayed 7 characters. This is determined by the offset which is updated every call of “scrollHandler” to give the sense of scrolling. An edge case is also defined in scroll as if the job title is an even number of characters, due to the offset changing by 2 each time and 7 characters being available to show each time. The last character could never be shown. So, by adding a space to the end of the job title ensures the last character is shown (with the space) in the last iteration of the offset before being reset.

The actual updating of the screen however is handled in the main loop, this is because the system would break if the lcd screen was updated from “scrollHandler”, so each run through of the main loop, if scroll is activated then the current state of the scroll buffer will be displayed.

A drawback to this implementation is that if the main loop takes longer then a second to run, then the display would skip 2 characters from the job title.

*I acknowledge the use of **Grammarly** to check Grammar and Spelling Errors (Proofreading). I confirm that all use of AI content is acknowledged and referenced appropriately to the best of my knowledge.*

Grammarly. (2024). Grammarly: Online writing assistant. Available at: https://www.grammarly.com Last Used: 04/12/2024 08:32