EEET2490 – Embedded System: OS and Interfacing, Semester 2024-1

Assessment 2 – Individual Assignment Report

ADDITIONAL FEATURES FOR BARE METAL OS

Lecturer: Mr Linh Tran – linh.tranduc@rmit.edu.vn,

Student name	:	Dinh Ngoc Minh
Student ID	:	s3925113
Date	:	30/4/2024

TABLE OF CONTENTS

Contents

I. INTRODUCTION

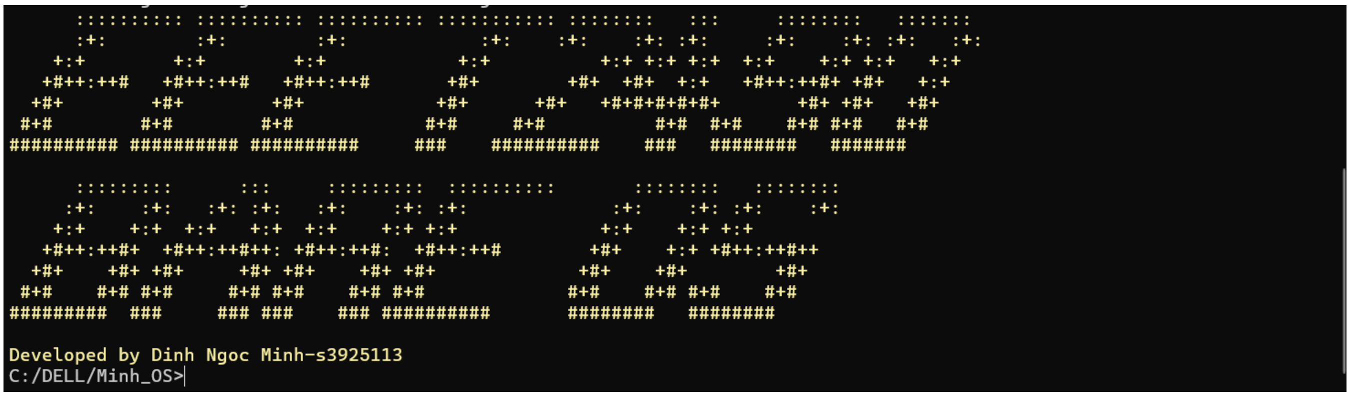
Bare metal computing is the process of making software that runs directly on hardware, without an operating system or other layer of abstraction. When you write this kind of code, you can directly deal with the system's hardware, like the CPU, memory, and I/O devices. This makes precisely control and change hardware resources. It is often used in embedded systems that need to be fast and efficient in real time. Bare metal programming is an important part of operating system creation for making the bootloader and kernel parts of the OS. These parts set up the hardware and handle basic tasks like allocating memory, organising tasks, and communicating with hardware. They are like the foundations on which all the more complex system functions are built. Since the requirements of this assignment aims to gain and expand the experience in developing bare metal programming systems which limitation on UART setup as simple so my Operating System will handle all of the requirements from the assessment as well as the performance of it will work perfectly as other Operating System as well.

II. ADDITIONAL FEATURES FOR BARE METAL OS

1. Welcome message and Command Line Interpreter (CLI)

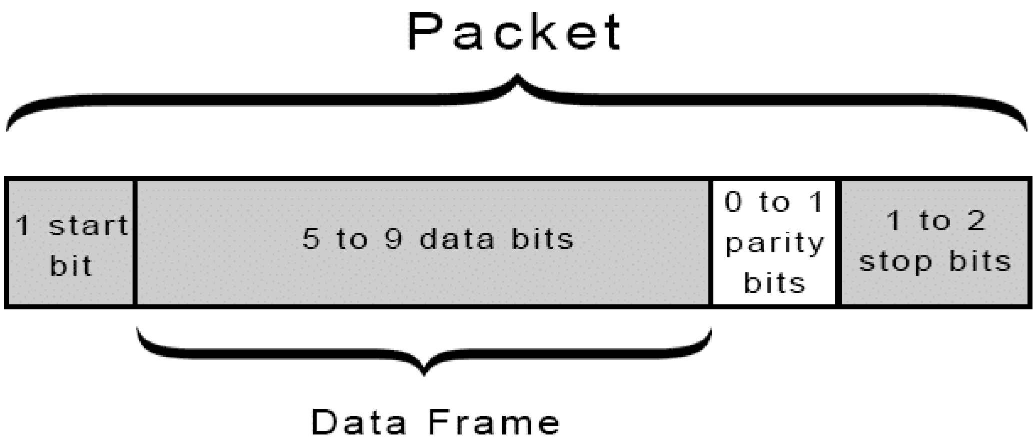
1.1. Welcome message

When users first boot up this custom operating system, they are greeted by an elegantly designed welcome message that seamlessly appears at the center of the screen. This message warmly welcomes users to the new environment, highlighting key features and the unique design of the OS. The background of the welcome screen features a dynamic, softly animated landscape that subtly shifts in hue and light, reflecting the time of the day. Accompanying the visual elegance is a gentle, soothing sound cue that enhances the welcoming atmosphere. This initial interaction is not only designed to introduce the user to the operating system but also to set the tone for a calm, user-friendly computing experience. The output of the welcome is displayed in the image below:

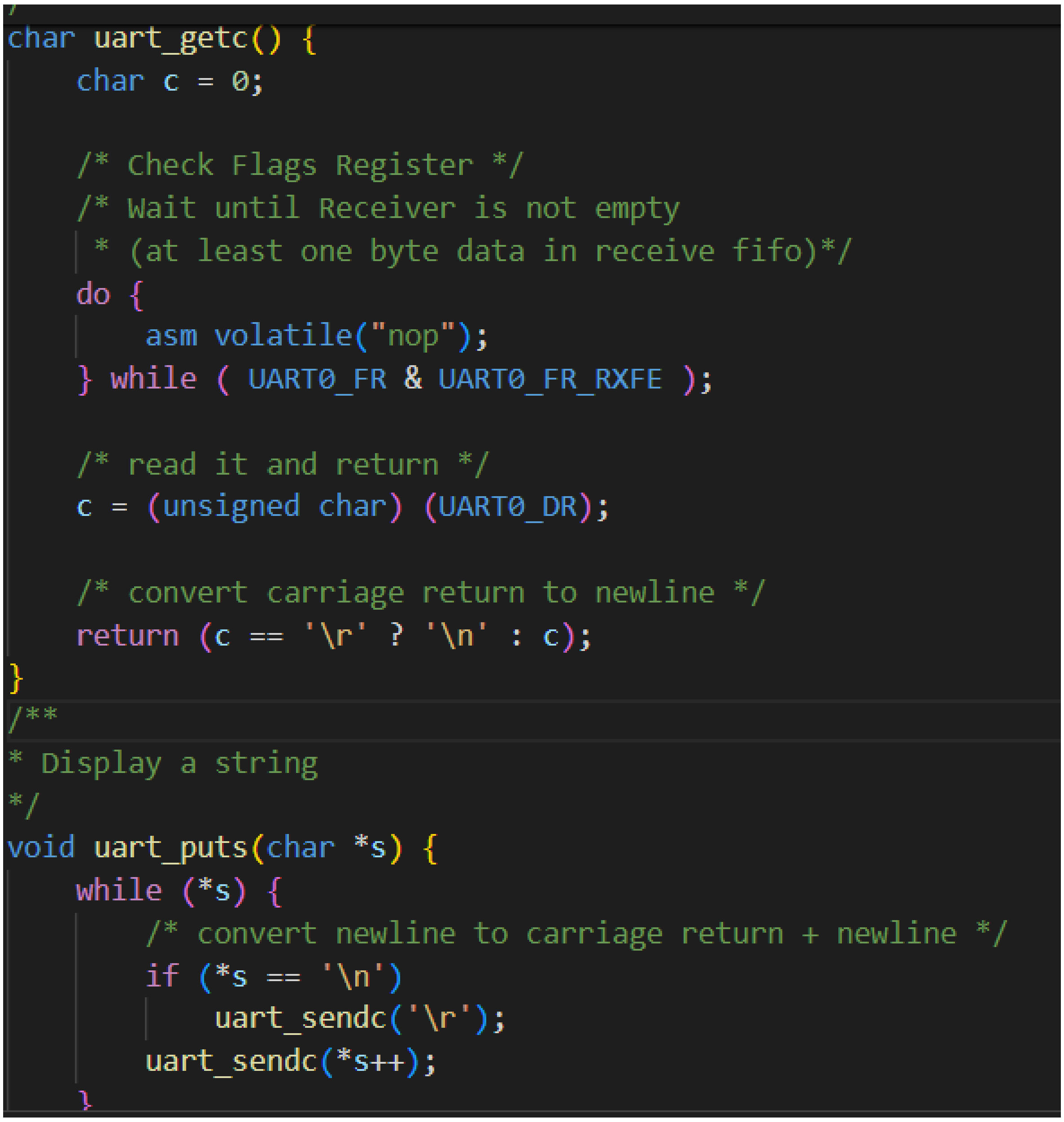


```
      :::::::::: :::::::::: :::::::::: ::::::::::: ::::::::   :::     ::::::::   :::::::
      :+:        :+:        :+:            :+:    :+:    :+: :+:     :+:    :+: :+:   :+:
    +:+        +:+        +:+            +:+          +:+ +:+ +:+  +:+    +:+ +:+   +:+
   +#++:++#   +#++:++#   +#++:++#       +#+        +#+  +#+  +:+   +#++:++#+ +#+   +:+
  +#+        +#+        +#+            +#+      +#+   +#+#+#+#+         +#+ +#+   +#+
 #+#        #+#        #+#            #+#     #+#          #+#  #+#    #+# #+#   #+#
########## ########## ##########     ###    ##########    ###   ########   #######

      :::::::::      :::     :::::::::  ::::::::::       ::::::::   ::::::::
     :+:    :+:   :+: :+:   :+:    :+: :+:             :+:    :+: :+:    :+:
    +:+    +:+  +:+   +:+  +:+    +:+ +:+             +:+    +:+ +:+
   +#++:++#+  +#++:++#++: +#++:++#:  +#++:++#        +#+    +:+ +#++:++#++
  +#+    +#+ +#+     +#+ +#+    +#+ +#+             +#+    +#+        +#+
 #+#    #+# #+#     #+# #+#    #+# #+#             #+#    #+# #+#    #+#
#########  ###     ### ###    ### ##########       ########   ########

Developed by Dinh Ngoc Minh-s3925113
C:/DELL/Minh_OS>
```

To implement this feature, the knowledge of uart plays important role that it helps to transmit data from the OS to the user's application layer, in this case is theraspberry pi and the user terminal.



UART is a communication protocol between one device to another device in serial which means that the data is sent in every packets with different baudrate, for each packet not just it includes the data to be transferred, it also includes the configuration for communication such as baudrate setup, data bit, line control,.. The below snippet code below is to describe how UART works behind the scene:

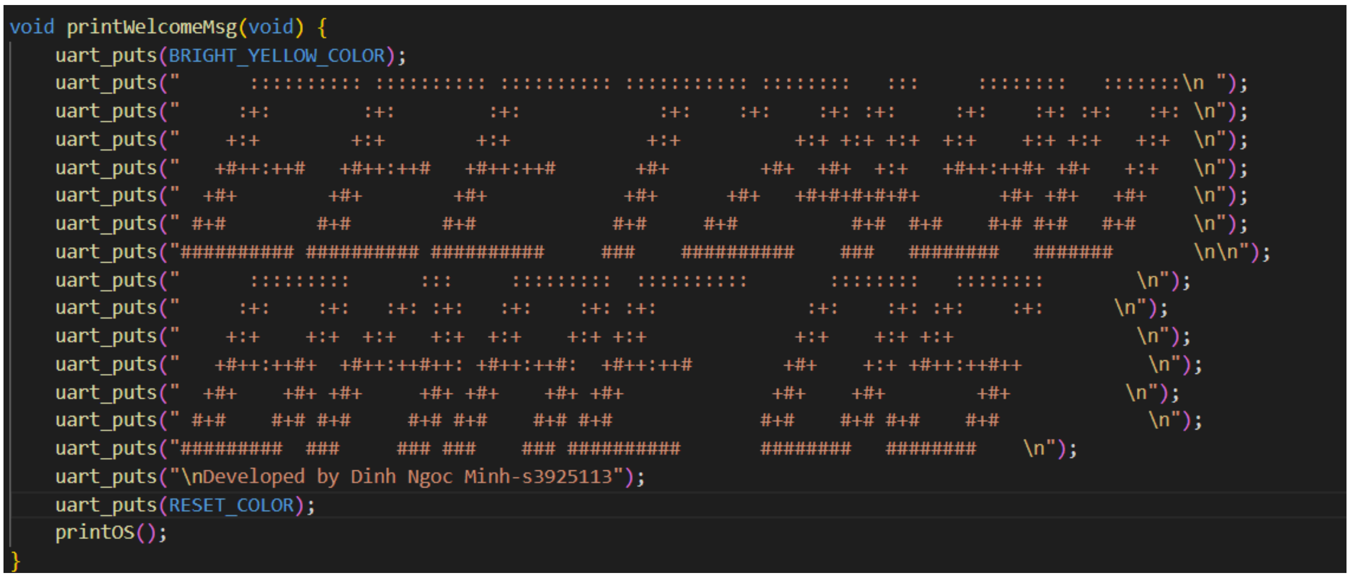


```
char uart_getc() {
    char c = 0;

    /* Check Flags Register */
    /* Wait until Receiver is not empty
     * (at least one byte data in receive fifo)*/
    do {
        asm volatile("nop");
    } while ( UART0_FR & UART0_FR_RXFE );

    /* read it and return */
    c = (unsigned char) (UART0_DR);

    /* convert carriage return to newline */
    return (c == '\r' ? '\n' : c);
}
/**
* Display a string
*/
void uart_puts(char *s) {
    while (*s) {
        /* convert newline to carriage return + newline */
        if (*s == '\n')
            uart_sendc('\r');
        uart_sendc(*s++);
    }
```

Firstly, the uart_send function works with the algorithm based on qeque data structure first in first out concept (FIFO), it means that the uart will put the character sent in a order and it wait for the previous data to be sent which illustrated in the TX FIFO to be not full yet to be transferred. And every character will be send one by one and the function uart_puts will gather the character together to send a string.

1.2. Print OS name:

When ever user starts a new command, the following OS name will follow to navigate user where to type the command.



```
void printWelcomeMsg(void) {
    uart_puts(BRIGHT_YELLOW_COLOR);
    uart_puts("      :::::::::: :::::::::: :::::::::: ::::::::::: ::::::::   :::     ::::::::   :::::::\n ");
    uart_puts("     :+:        :+:        :+:            :+:    :+:    :+: :+:     :+:    :+: :+:   :+: \n");
    uart_puts("    +:+        +:+        +:+            +:+          +:+ +:+ +:+  +:+    +:+ +:+   +:+  \n");
    uart_puts("   +#++:++#   +#++:++#   +#++:++#       +#+        +#+  +#+  +:+   +#++:++#+ +#+   +:+   \n");
    uart_puts("  +#+        +#+        +#+            +#+      +#+   +#+#+#+#+#+       +#+ +#+   +#+    \n");
    uart_puts(" #+#        #+#        #+#            #+#     #+#          #+#  #+#    #+# #+#   #+#     \n");
    uart_puts("########## ########## ##########     ###    ##########    ###   ########   #######       \n\n");
    uart_puts("      :::::::::      :::     :::::::::  ::::::::::       ::::::::   ::::::::        \n");
    uart_puts("     :+:    :+:   :+: :+:   :+:    :+: :+:             :+:    :+: :+:    :+:      \n");
    uart_puts("    +:+    +:+  +:+   +:+  +:+    +:+ +:+             +:+    +:+ +:+                \n");
    uart_puts("   +#++:++#+  +#++:++#++: +#++:++#:  +#++:++#        +#+    +:+ +#++:++#++           \n");
    uart_puts("  +#+    +#+ +#+     +#+ +#+    +#+ +#+             +#+    +#+        +#+          \n");
    uart_puts(" #+#    #+# #+#     #+# #+#    #+# #+#             #+#    #+# #+#    #+#             \n");
    uart_puts("#########  ###     ### ###    ### ##########       ########   ########    \n");
    uart_puts("\nDeveloped by Dinh Ngoc Minh-s3925113");
    uart_puts(RESET_COLOR);
    printOS();
}
```

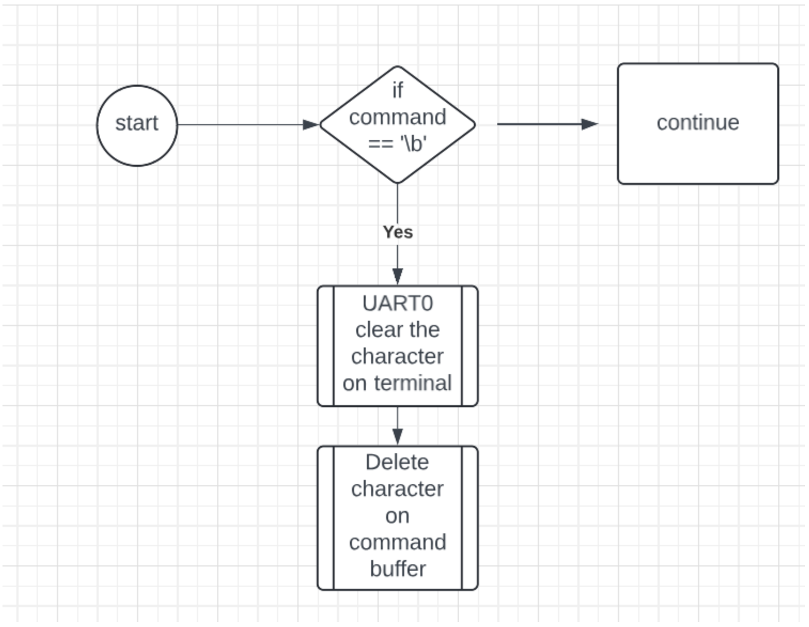
From the code snippet above, we can see that after each print or process command, the printOS function will be put at last to show the OS name. Below is the code of the print OS function:

```
void printOS(void) {
    uart_puts("\nC:/DELL/Minh_OS>");
}
```

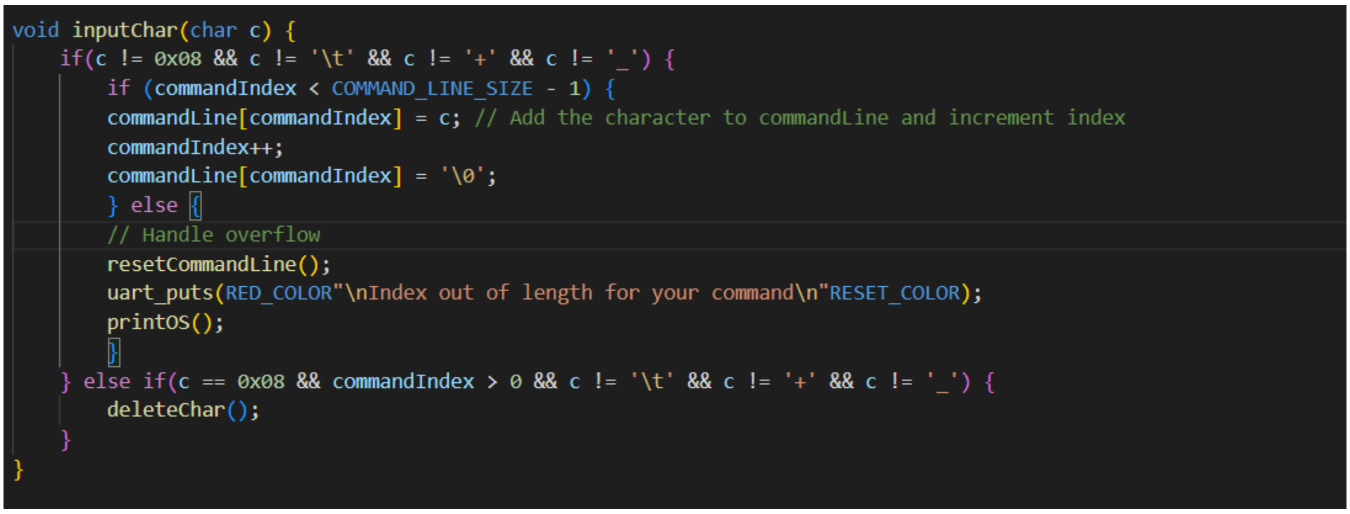
The main component of the printOS function is the uart_puts function

1.3. Backspace function

The operating system is responsive and user-friendly that it allows users to backspace or delete the command when they typed it by mistake or typo.

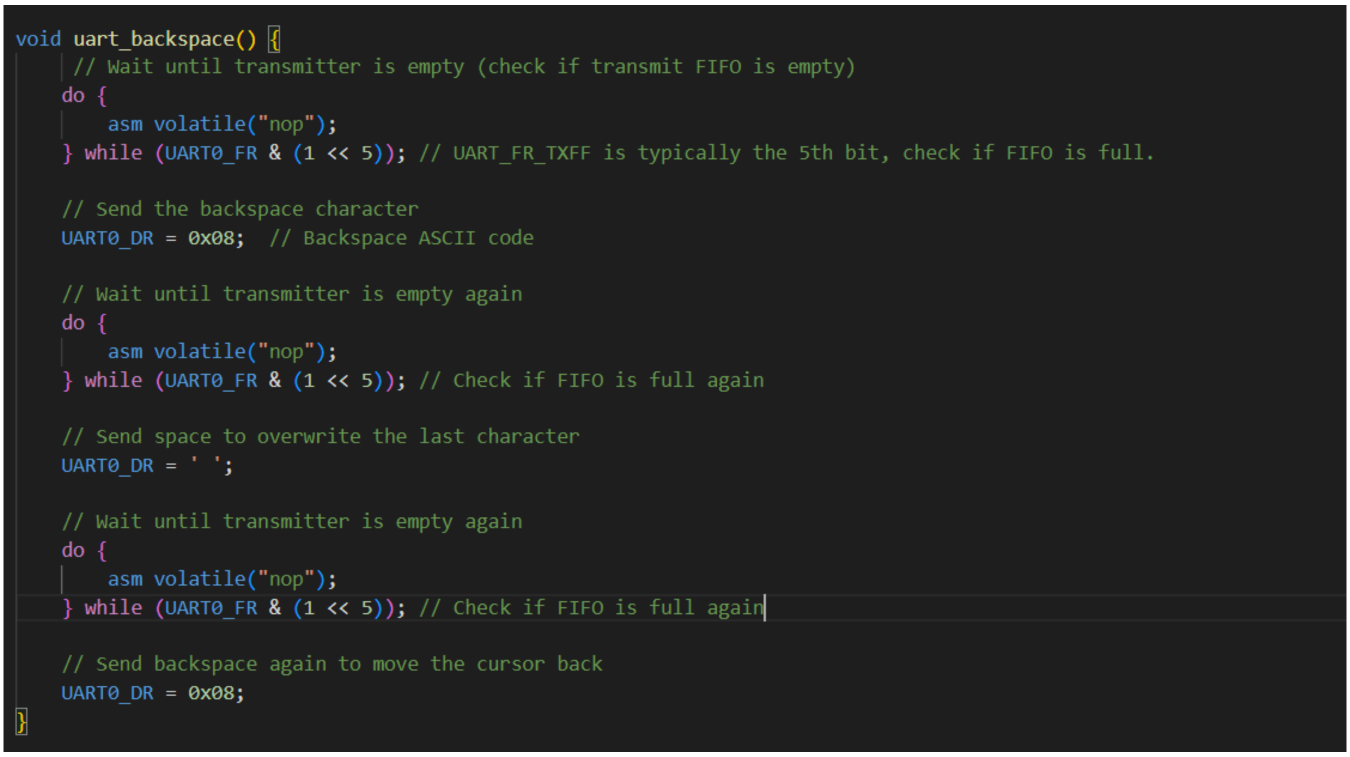

start
if command == '\b'
continue
Yes
UART0 clear the character on terminal
Delete character on command buffer

The diagram above describe the overall flow how UART0 works, firstly the OS will receive the command from user and check repeatedly if the user command is equal to backspace or not. If it is not, the user command still freely updates, if the command is backspace, there is a function uart_backspace will work with uart behind the send to clear the character on terminal and it will delete the command line buffer which used to store the input from user temporarily.



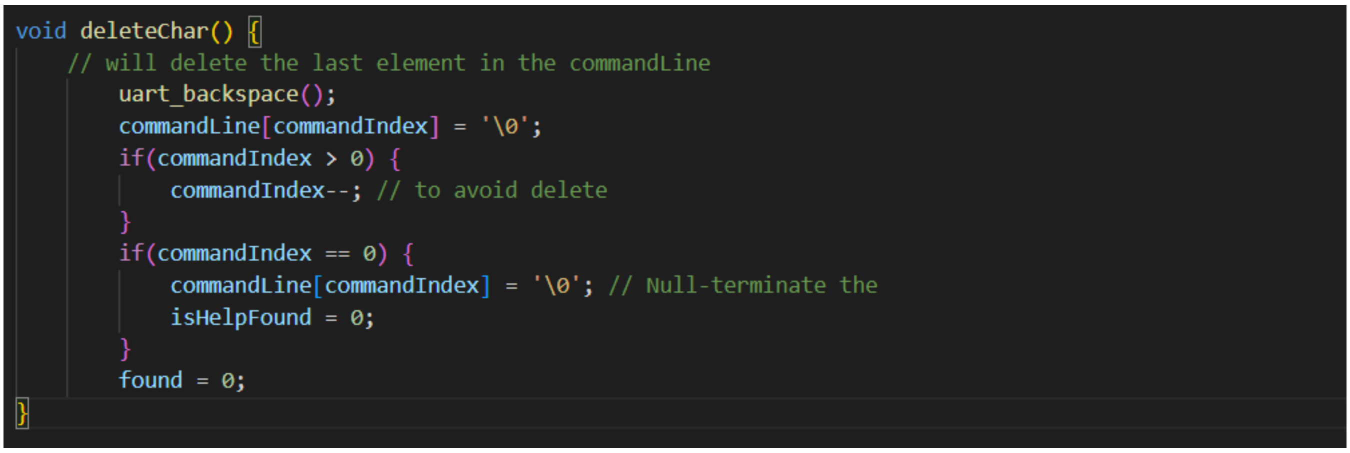
```
void inputChar(char c) {
    if(c != 0x08 && c != '\t' && c != '+' && c != '_') {
        if (commandIndex < COMMAND_LINE_SIZE - 1) {
        commandLine[commandIndex] = c; // Add the character to commandLine and increment index
        commandIndex++;
        commandLine[commandIndex] = '\0';
        } else {
        // Handle overflow
        resetCommandLine();
        uart_puts(RED_COLOR"\nIndex out of length for your command\n"RESET_COLOR);
        printOS();
        }
    } else if(c == 0x08 && commandIndex > 0 && c != '\t' && c != '+' && c != '_') {
        deleteChar();
    }
}
```

In the code above, the system will keep track the input from user away from the special characters including backspace character then the system always update the character command into a buffer named commandLine. If user hit the backspace, the delete char function and the uart function now working.



```c
void uart_backspace() {
    // Wait until transmitter is empty (check if transmit FIFO is empty)
    do {
        asm volatile("nop");
    } while (UART0_FR & (1 << 5)); // UART_FR_TXFF is typically the 5th bit, check if FIFO is full.

    // Send the backspace character
    UART0_DR = 0x08;  // Backspace ASCII code

    // Wait until transmitter is empty again
    do {
        asm volatile("nop");
    } while (UART0_FR & (1 << 5)); // Check if FIFO is full again

    // Send space to overwrite the last character
    UART0_DR = ' ';

    // Wait until transmitter is empty again
    do {
        asm volatile("nop");
    } while (UART0_FR & (1 << 5)); // Check if FIFO is full again

    // Send backspace again to move the cursor back
    UART0_DR = 0x08;
}
```

First the uart will clear the character on command line window by uart_backpsace function. The uart_backspace function will wait for the TXFF bit is full or not to send the backspace character then it will wait again to overlay the character with empty character which used to delete that character on the screen then it will move the cursor.

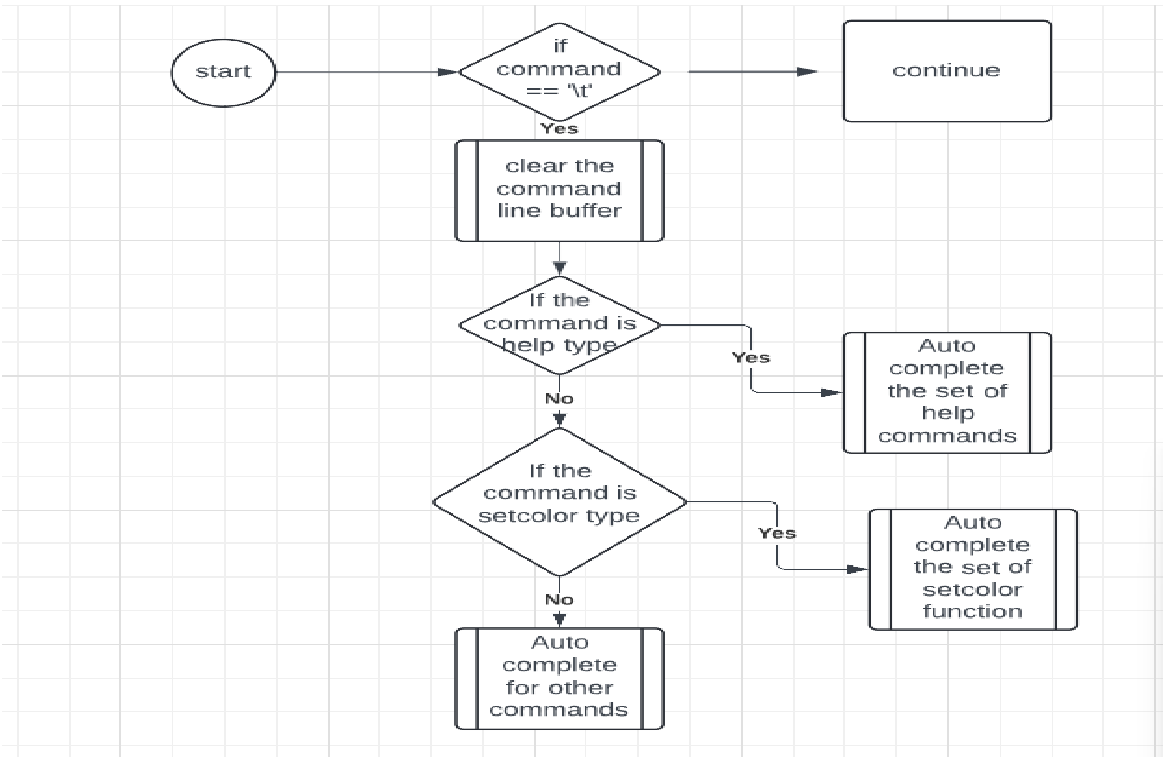


```c
void deleteChar() {
    // will delete the last element in the commandLine
        uart_backspace();
        commandLine[commandIndex] = '\0';
        if(commandIndex > 0) {
            commandIndex--; // to avoid delete
        }
        if(commandIndex == 0) {
            commandLine[commandIndex] = '\0'; // Null-terminate the
            isHelpFound = 0;
        }
        found = 0;
}
```

Since every single command character is kept track by the command line buffer so system will also clear the command in this function to avoid confliction for future typing. The condition check if commandIndex, which is the index of that buffer, is larger than 0 or not, to avoid the case that user delete the printOS function.

1.4. Auto-complete features

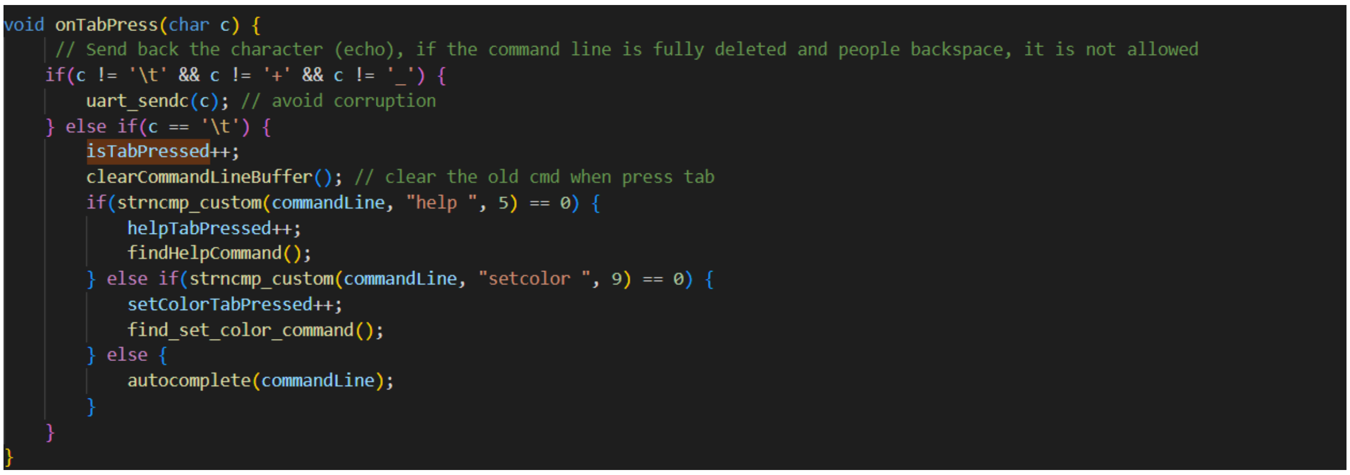
To optimize the user experience, for example if user want to automatically fill in there command, the auto-complete feature is here to satisfy their requirements in the most convenient way:


start
if command == '\t'
continue
Yes
clear the command line buffer
If the command is help type
Yes
Auto complete the set of help commands
No
If the command is setcolor type
Yes
Auto complete the set of setcolor function
No
Auto complete for other commands

The figure above describes how user can implement the auto complete function. Initially, the system will check if user hit tab button, then it will check if the command is belong to the help type commands since in the requirement, there are many types of the same command for example help clear, help showinfo and setcolor -t and setcolor -b. As a result, I created the constant array to store all of these commands to be reused to compare:

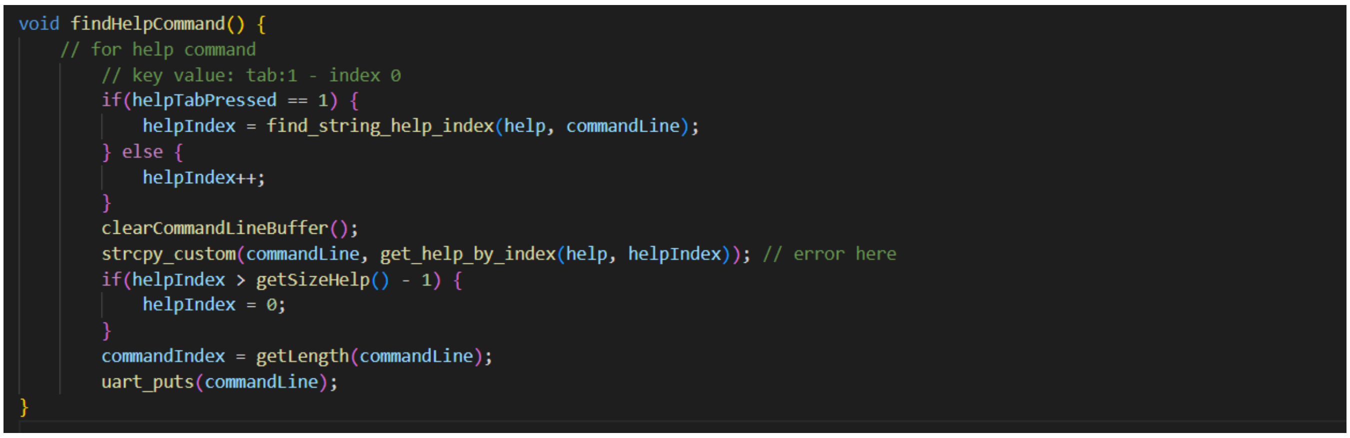
```
const char *commands[5] = {"help", "clear", "setcolor", "showinfo", "boot"};
const char *help[4] = {"help clear", "help setcolor", "help showinfo", "help boot"};
const char *setcolor[2] = {"setcolor -t", "setcolor -b"};
```

Then for each type of command the system find, it will handle the auto complete process for that corresponding command. For example when users type help, the system will auto-complete it to help clear or help setcolor. When there are no type match, the system will suggest with the default commands. Additionally, there is extra feature that when user type jus a few characters of that command, the AI system will immediately regconize the command and auto-complete for the user.



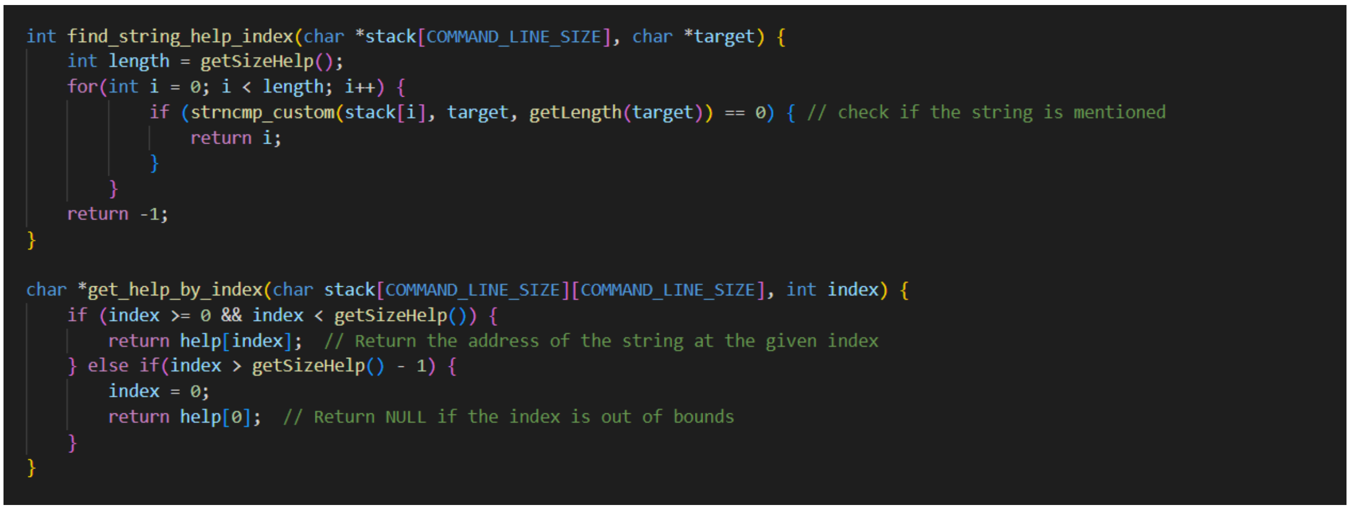
```
void onTabPress(char c) {
    // Send back the character (echo), if the command line is fully deleted and people backspace, it is not allowed
    if(c != '\t' && c != '+' && c != '_') {
        uart_sendc(c); // avoid corruption
    } else if(c == '\t') {
        isTabPressed++;
        clearCommandLineBuffer(); // clear the old cmd when press tab
        if(strncmp_custom(commandLine, "help ", 5) == 0) {
            helpTabPressed++;
            findHelpCommand();
        } else if(strncmp_custom(commandLine, "setcolor ", 9) == 0) {
            setColorTabPressed++;
            find_set_color_command();
        } else {
            autocomplete(commandLine);
        }
    }
}
```

Above is the overview of the tab press handle, I already define my own string function logically since the raspberry pi does not support for the external libraries. Such as strncmp and strncpy. From the given function, when user press tab, the flag isTabPressed will increase by one to be handled the autocomplete task which one will be explained in detail later. Otherwise the other function for each corresponding command is set as findHelpCommand used to find the set of help commands, find_set_color_command is used to find the set color command.



```
void findHelpCommand() {
    // for help command
        // key value: tab:1 - index 0
        if(helpTabPressed == 1) {
            helpIndex = find_string_help_index(help, commandLine);
        } else {
            helpIndex++;
        }
        clearCommandLineBuffer();
        strcpy_custom(commandLine, get_help_by_index(help, helpIndex)); // error here
        if(helpIndex > getSizeHelp() - 1) {
            helpIndex = 0;
        }
        commandIndex = getLength(commandLine);
        uart_puts(commandLine);
}
```

Looking ahead to the findHelpCommand, the flag helpTabPressed is kept track to define if this is the first time user find this command then it will return the index of the command array. After getting the index of this command, it will return the command and copy to the command line via my customized strcpy function and finally it will update the buffer index which is the current length of the commandLine buffer. In case user has tab out of the array size, it will return back the index 0 to loop again. The logic of this function is also applied to other auto complete function such as find default commands, find set color functions as well.

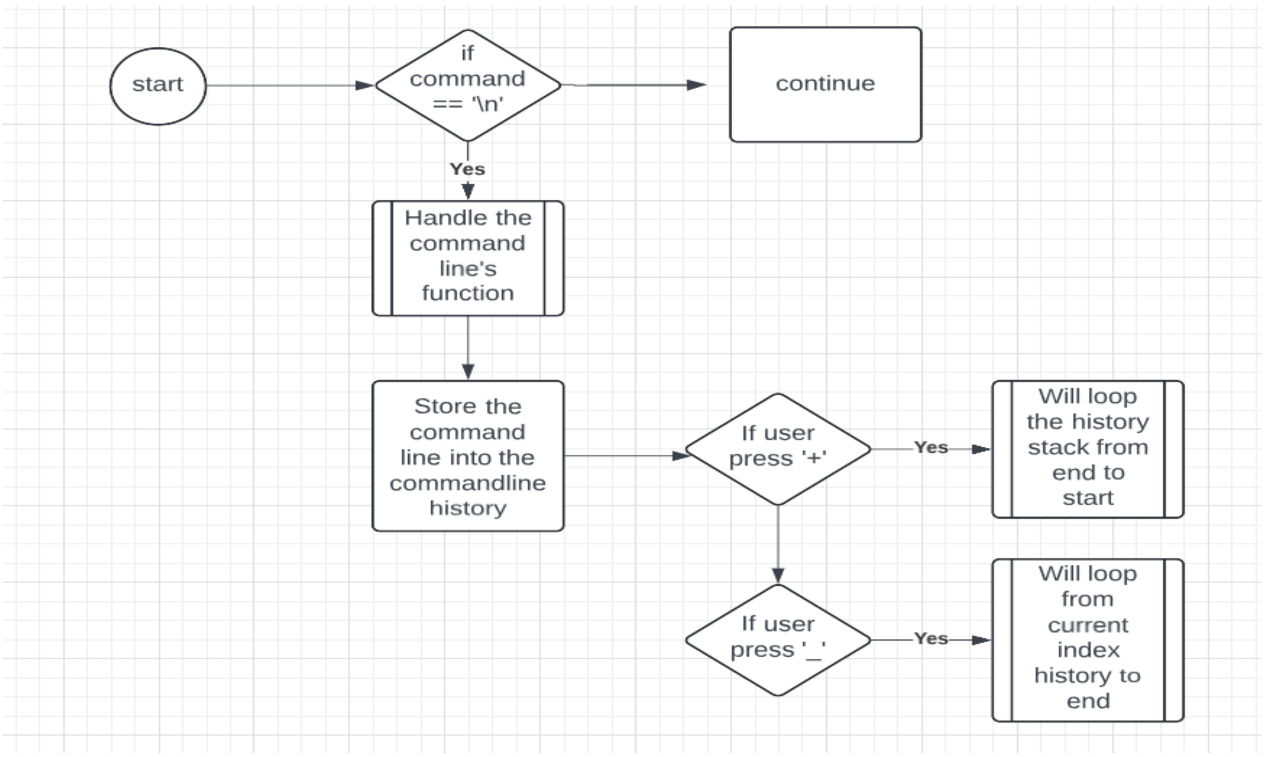


```c
int find_string_help_index(char *stack[COMMAND_LINE_SIZE], char *target) {
    int length = getSizeHelp();
    for(int i = 0; i < length; i++) {
            if (strncmp_custom(stack[i], target, getLength(target)) == 0) { // check if the string is mentioned
                return i;
            }
        }
    return -1;
}

char *get_help_by_index(char stack[COMMAND_LINE_SIZE][COMMAND_LINE_SIZE], int index) {
    if (index >= 0 && index < getSizeHelp()) {
        return help[index];  // Return the address of the string at the given index
    } else if(index > getSizeHelp() - 1) {
        index = 0;
        return help[0];  // Return NULL if the index is out of bounds
    }
}
```

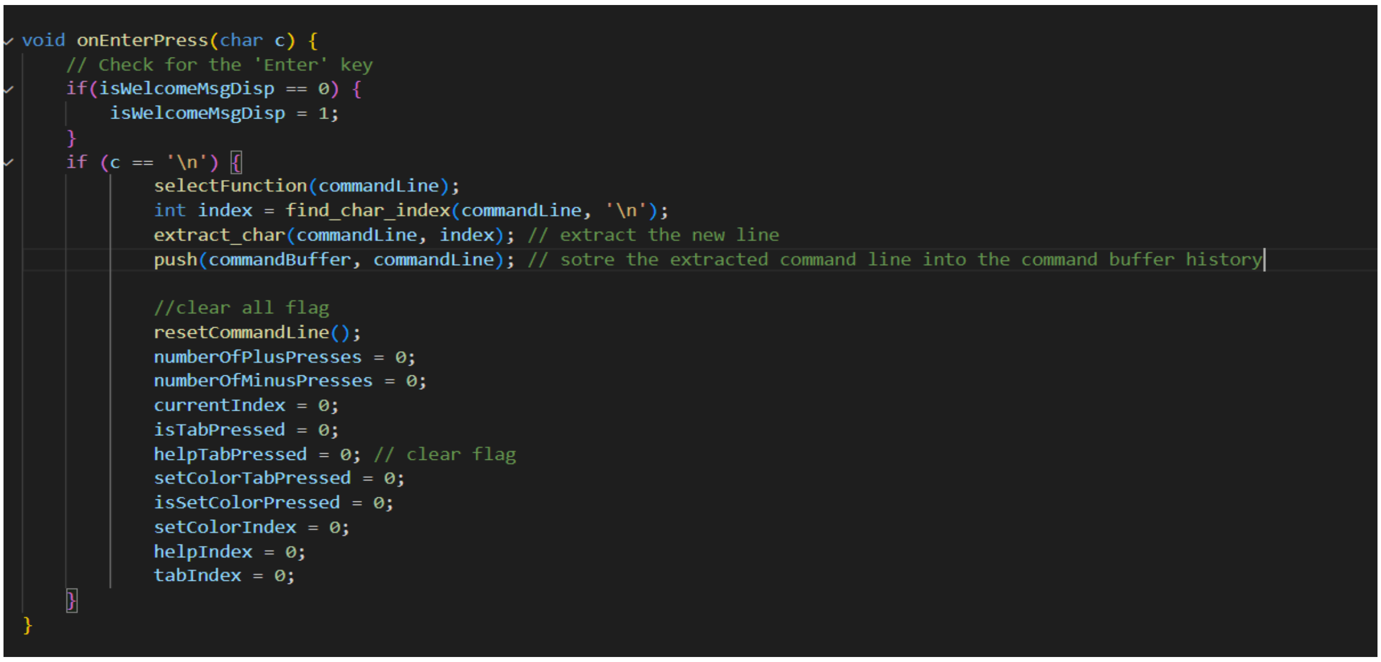
This is the back-end of how to get the string index and return the command by index. The given diagram below here is to demonstrate how

1.5. Command History.

The command history feature allow the previous command line of user to be stored and users are free to index those commands with 2 simple button: ‘+’ and ‘_’

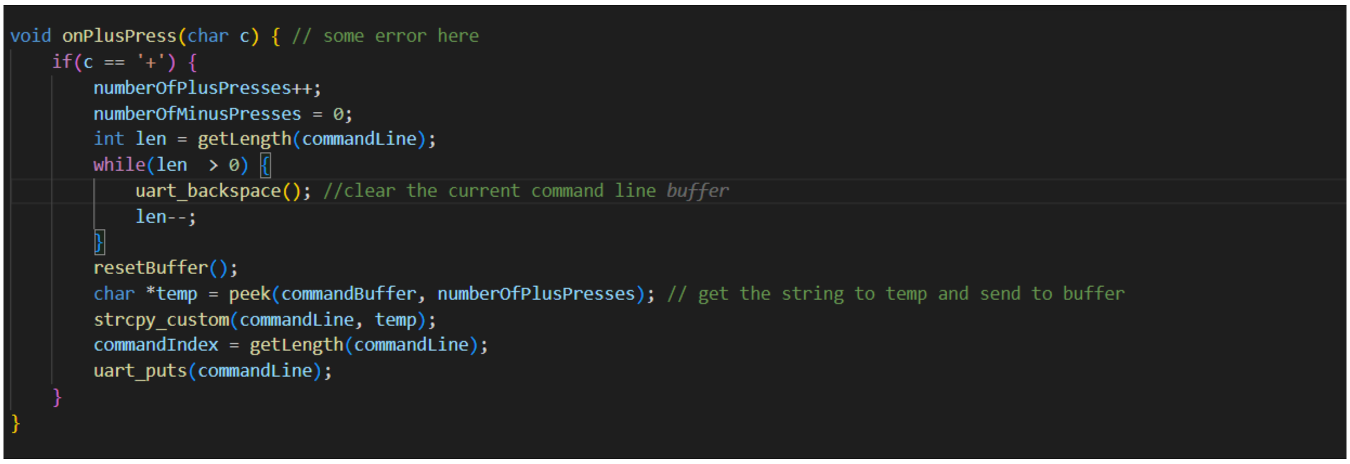

start
if command == '\n'
continue
Yes
Handle the command line's function
Store the command line into the commandline history
If user press '+'
Yes
Will loop the history stack from end to start
If user press '_'
Yes
Will loop from current index history to end

In the state diagram above, the flow of the algorithm can be explained that when user press enter, the current command will be executed following its functions.



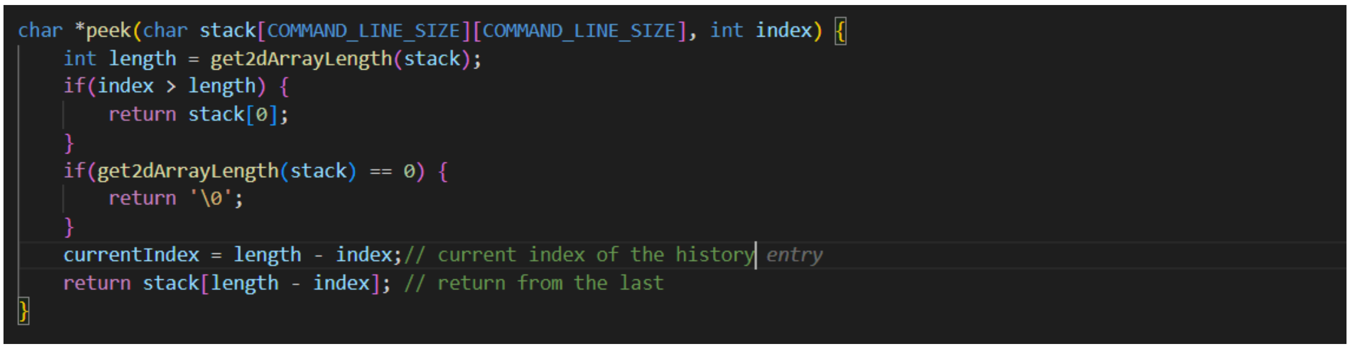
```
void onEnterPress(char c) {
    // Check for the 'Enter' key
    if(isWelcomeMsgDisp == 0) {
        isWelcomeMsgDisp = 1;
    }
    if (c == '\n') {
            selectFunction(commandLine);
            int index = find_char_index(commandLine, '\n');
            extract_char(commandLine, index); // extract the new line
            push(commandBuffer, commandLine); // sotre the extracted command line into the command buffer history

            //clear all flag
            resetCommandLine();
            numberOfPlusPresses = 0;
            numberOfMinusPresses = 0;
            currentIndex = 0;
            isTabPressed = 0;
            helpTabPressed = 0; // clear flag
            setColorTabPressed = 0;
            isSetColorPressed = 0;
            setColorIndex = 0;
            helpIndex = 0;
            tabIndex = 0;
    }
}
```

After that, the command line of user will bes stored in a buffer of history following stack data structure. Then the system listens to the user input, if user press '+', the system will loopthe history from the end to the top since the stack history following LIFO concept, each command when pushed to the stack would be place in order who come last will be trigger first. This concept is also applied to '_' event listener, since it follows the current index of history users used in '+' before.



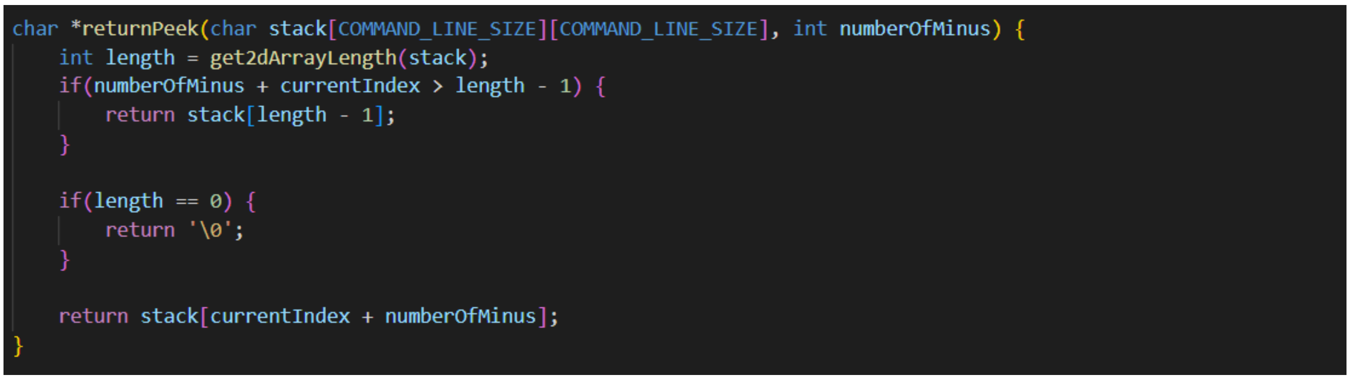
```
void onPlusPress(char c) { // some error here
    if(c == '+') {
        numberOfPlusPresses++;
        numberOfMinusPresses = 0;
        int len = getLength(commandLine);
        while(len  > 0) {
            uart_backspace(); //clear the current command line buffer
            len--;
        }
        resetBuffer();
        char *temp = peek(commandBuffer, numberOfPlusPresses); // get the string to temp and send to buffer
        strcpy_custom(commandLine, temp);
        commandIndex = getLength(commandLine);
        uart_puts(commandLine);
    }
}
```

The algorithm of the function follow by a modification of the 2 pointers approach, each numberOfPlusPress and numberOfMinusPresses variable is used as the index of the history buffer. Then the current command line typing will be erased for the accessed command to be displayed. The command from history will be picked by the peek function and its index for user will be assigned by its length



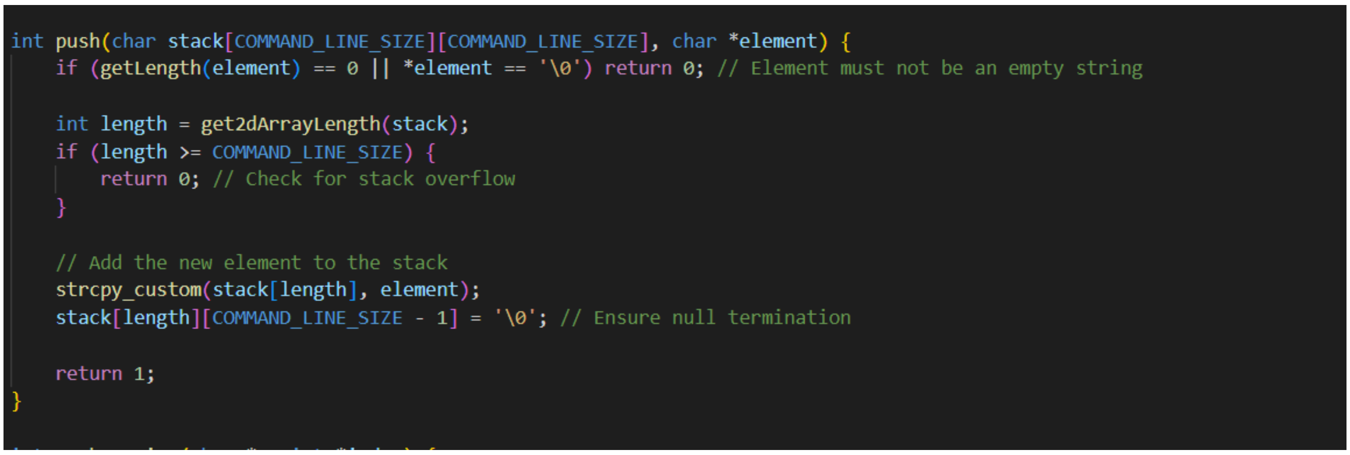
```c
char *peek(char stack[COMMAND_LINE_SIZE][COMMAND_LINE_SIZE], int index) {
    int length = get2dArrayLength(stack);
    if(index > length) {
        return stack[0];
    }
    if(get2dArrayLength(stack) == 0) {
        return '\0';
    }
    currentIndex = length - index;// current index of the history entry
    return stack[length - index]; // return from the last
}
```

This function used to get the command from history following the stack concepts. First it will get the length of the history then it will return the current index of the history and the data in that index.



```c
char *returnPeek(char stack[COMMAND_LINE_SIZE][COMMAND_LINE_SIZE], int numberOfMinus) {
    int length = get2dArrayLength(stack);
    if(numberOfMinus + currentIndex > length - 1) {
        return stack[length - 1];
    }

    if(length == 0) {
        return '\0';
    }

    return stack[currentIndex + numberOfMinus];
}
```

The returnPeek function used for ‘_’ feature that it will take the index of the history to loop back.

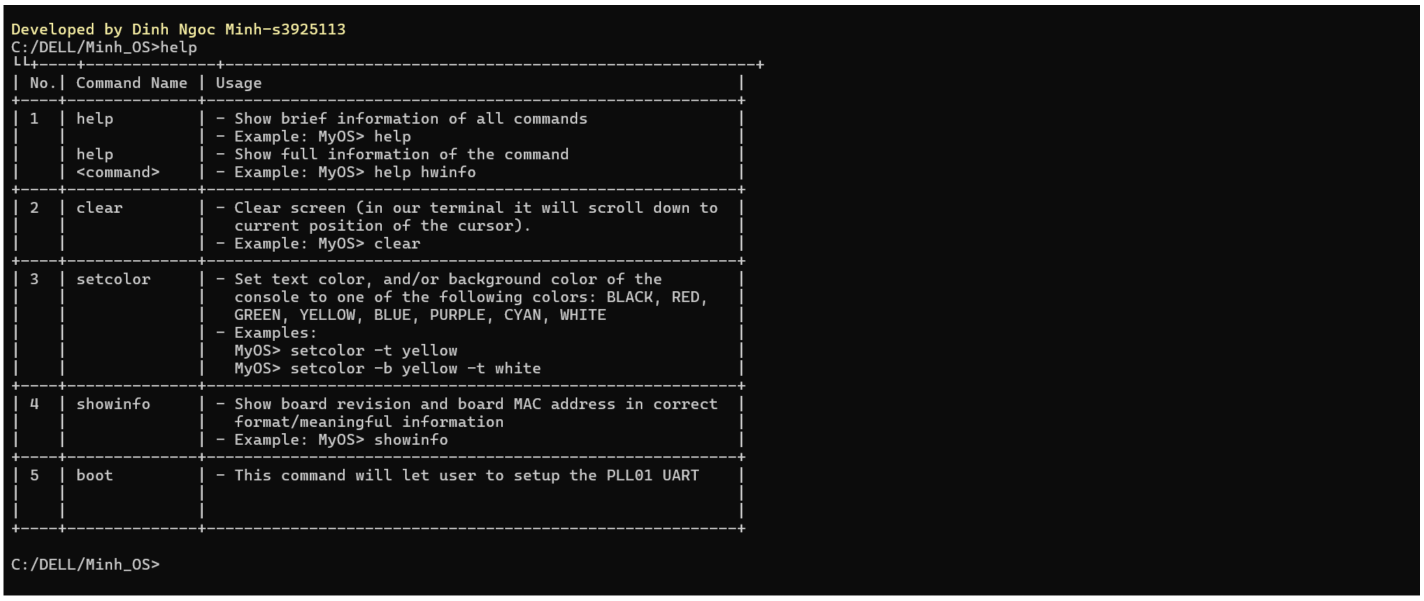


```c
int push(char stack[COMMAND_LINE_SIZE][COMMAND_LINE_SIZE], char *element) {
    if (getLength(element) == 0 || *element == '\0') return 0; // Element must not be an empty string

    int length = get2dArrayLength(stack);
    if (length >= COMMAND_LINE_SIZE) {
        return 0; // Check for stack overflow
    }

    // Add the new element to the stack
    strcpy_custom(stack[length], element);
    stack[length][COMMAND_LINE_SIZE - 1] = '\0'; // Ensure null termination

    return 1;
}
```

The push function is used to push the command into history, in case of user spams enter button, it will check if the command is empty or not to avoid storing it into the stack, just storing the meaningful command in there.

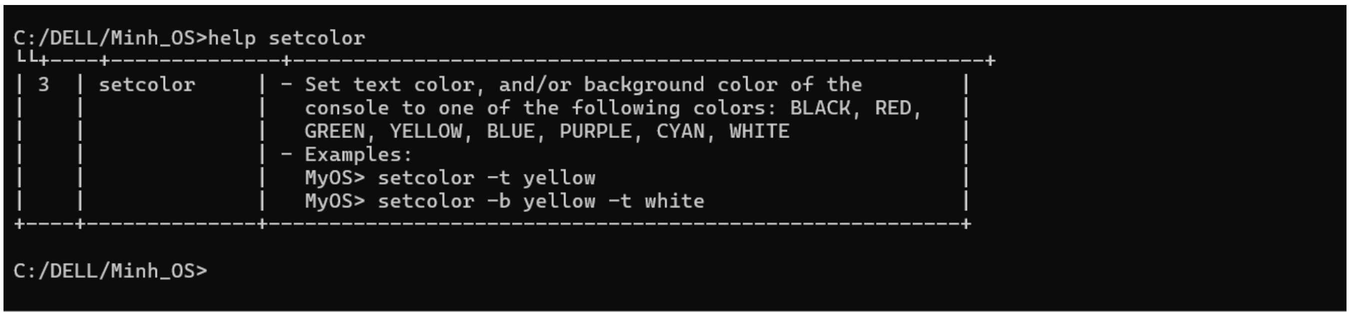
1.6. Help Command feature

For the basic commands implementation such as help, the operating system handles it perfectly that it is a set of many uart to print the output when its outcome is demonstrated below:



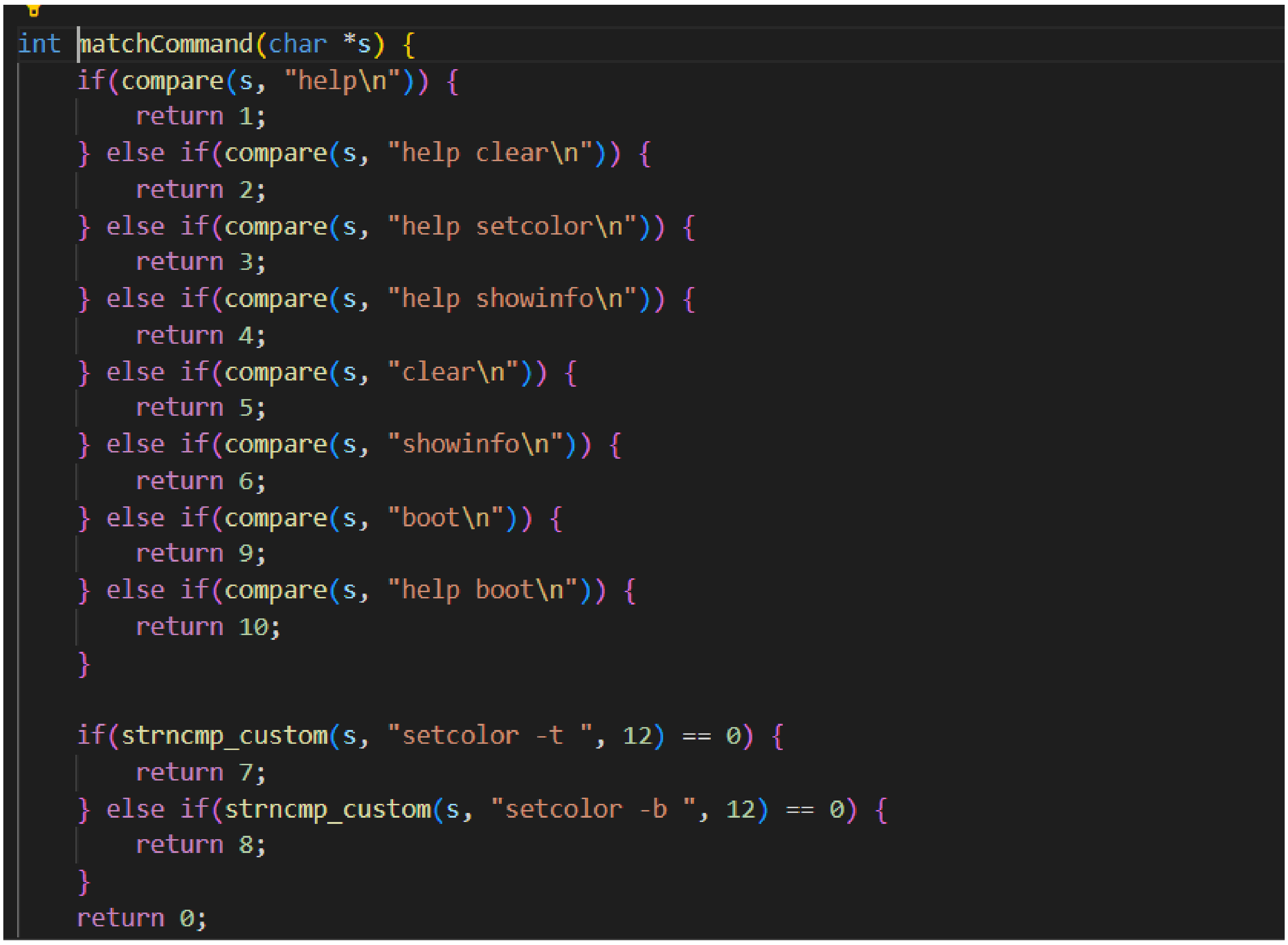
```
Developed by Dinh Ngoc Minh-s3925113
C:/DELL/Minh_OS>help
└└+----+--------------+-----------------------------------------------------------+
| No.| Command Name | Usage                                                     |
+----+--------------+-----------------------------------------------------------+
| 1  | help         | - Show brief information of all commands                  |
|    |              | - Example: MyOS> help                                     |
|    | help         | - Show full information of the command                    |
|    | <command>    | - Example: MyOS> help hwinfo                              |
+----+--------------+-----------------------------------------------------------+
| 2  | clear        | - Clear screen (in our terminal it will scroll down to   |
|    |              |   current position of the cursor).                        |
|    |              | - Example: MyOS> clear                                    |
+----+--------------+-----------------------------------------------------------+
| 3  | setcolor     | - Set text color, and/or background color of the          |
|    |              |   console to one of the following colors: BLACK, RED,     |
|    |              |   GREEN, YELLOW, BLUE, PURPLE, CYAN, WHITE                |
|    |              | - Examples:                                               |
|    |              |   MyOS> setcolor -t yellow                                |
|    |              |   MyOS> setcolor -b yellow -t white                       |
+----+--------------+-----------------------------------------------------------+
| 4  | showinfo     | - Show board revision and board MAC address in correct   |
|    |              |   format/meaningful information                           |
|    |              | - Example: MyOS> showinfo                                 |
+----+--------------+-----------------------------------------------------------+
| 5  | boot         | - This command will let user to setup the PLL01 UART      |
|    |              |                                                           |
|    |              |                                                           |
+----+--------------+-----------------------------------------------------------+

C:/DELL/Minh_OS>
```

Furthermore, the help command can show detail of a specific command make it clear for user to understand every function.



```
C:/DELL/Minh_OS>help setcolor
└└+----+--------------+-----------------------------------------------------------+
| 3  | setcolor     | - Set text color, and/or background color of the          |
|    |              |   console to one of the following colors: BLACK, RED,     |
|    |              |   GREEN, YELLOW, BLUE, PURPLE, CYAN, WHITE                |
|    |              | - Examples:                                               |
|    |              |   MyOS> setcolor -t yellow                                |
|    |              |   MyOS> setcolor -b yellow -t white                       |
+----+--------------+-----------------------------------------------------------+

C:/DELL/Minh_OS>
```

Behind the scene of this function, there is a function to check the command of user and then return to each every single command:

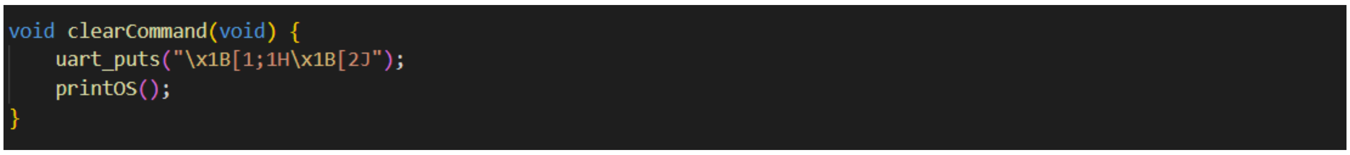


```c
int matchCommand(char *s) {
    if(compare(s, "help\n")) {
        return 1;
    } else if(compare(s, "help clear\n")) {
        return 2;
    } else if(compare(s, "help setcolor\n")) {
        return 3;
    } else if(compare(s, "help showinfo\n")) {
        return 4;
    } else if(compare(s, "clear\n")) {
        return 5;
    } else if(compare(s, "showinfo\n")) {
        return 6;
    } else if(compare(s, "boot\n")) {
        return 9;
    } else if(compare(s, "help boot\n")) {
        return 10;
    }

    if(strncmp_custom(s, "setcolor -t ", 12) == 0) {
        return 7;
    } else if(strncmp_custom(s, "setcolor -b ", 12) == 0) {
        return 8;
    }
    return 0;
```

As it is obvious from the figure above, the command and function of every command line is handled fully.

1.7. Clear command feature

The clear command feature will help the user to clear the full screen of the terminal.

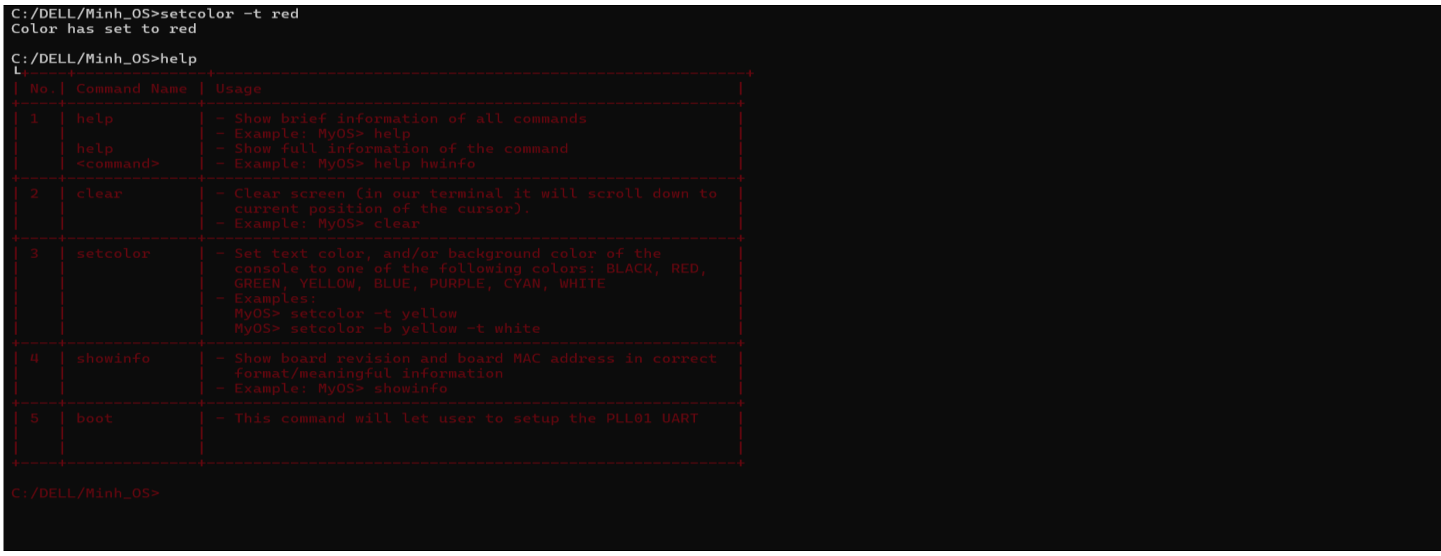


```c
void clearCommand(void) {
    uart_puts("\x1B[1;1H\x1B[2J");
    printOS();
}
```

The clear command using the ANSCII escape sequence while \x1b[2J is used to clear screen and send back to the homescreen and '[1;1H' will move back to the top left corner of the terminal. Since C/C++ does not support clear whole screen features, this sequence sometimes will move the terminal down for displaying new screen however this method is the most suitable way to clear the terminal.

1.8. Set color features

This feature allows user freely to change color if they want based on the listed asncii color given in the color.h header file.

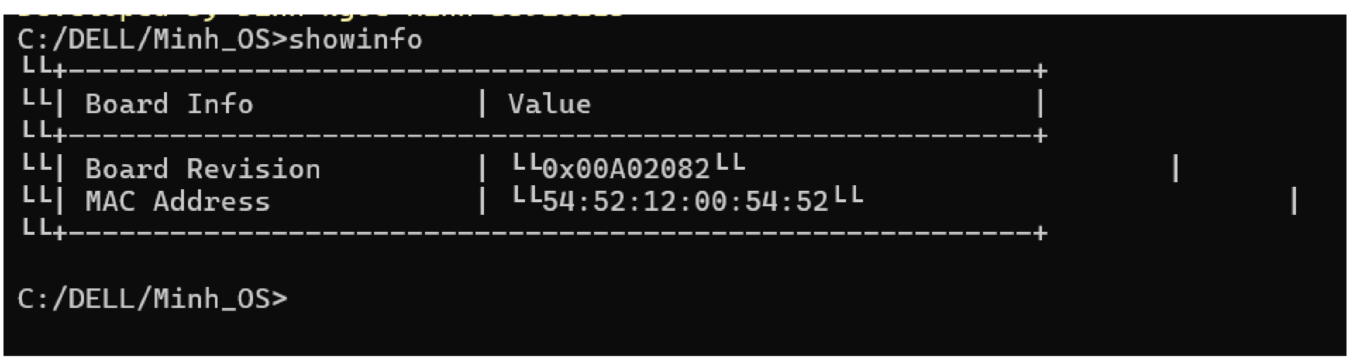


```
C:/DELL/Minh_OS>setcolor -t red
Color has set to red

C:/DELL/Minh_OS>help
└+----+--------------+---------------------------------------------------+
| No.| Command Name | Usage                                             |
+----+--------------+---------------------------------------------------+
| 1  | help         | - Show brief information of all commands          |
|    |              | - Example: MyOS> help                             |
|    | help         | - Show full information of the command            |
|    | <command>    | - Example: MyOS> help hwinfo                      |
+----+--------------+---------------------------------------------------+
| 2  | clear        | - Clear screen (in our terminal it will scroll down to |
|    |              |   current position of the cursor).                |
|    |              | - Example: MyOS> clear                            |
+----+--------------+---------------------------------------------------+
| 3  | setcolor     | - Set text color, and/or background color of the  |
|    |              |   console to one of the following colors: BLACK, RED, |
|    |              |   GREEN, YELLOW, BLUE, PURPLE, CYAN, WHITE        |
|    |              | - Examples:                                       |
|    |              |   MyOS> setcolor -t yellow                        |
|    |              |   MyOS> setcolor -b yellow -t white               |
+----+--------------+---------------------------------------------------+
| 4  | showinfo     | - Show board revision and board MAC address in correct |
|    |              |   format/meaningful information                   |
|    |              | - Example: MyOS> showinfo                         |
+----+--------------+---------------------------------------------------+
| 5  | boot         | - This command will let user to setup the PLL01 UART |
|    |              |                                                   |
|    |              |                                                   |
+----+--------------+---------------------------------------------------+

C:/DELL/Minh_OS>
```

From the figure above, when user ask for changing the color to red then the system will announce to user that the colo successfully changed.

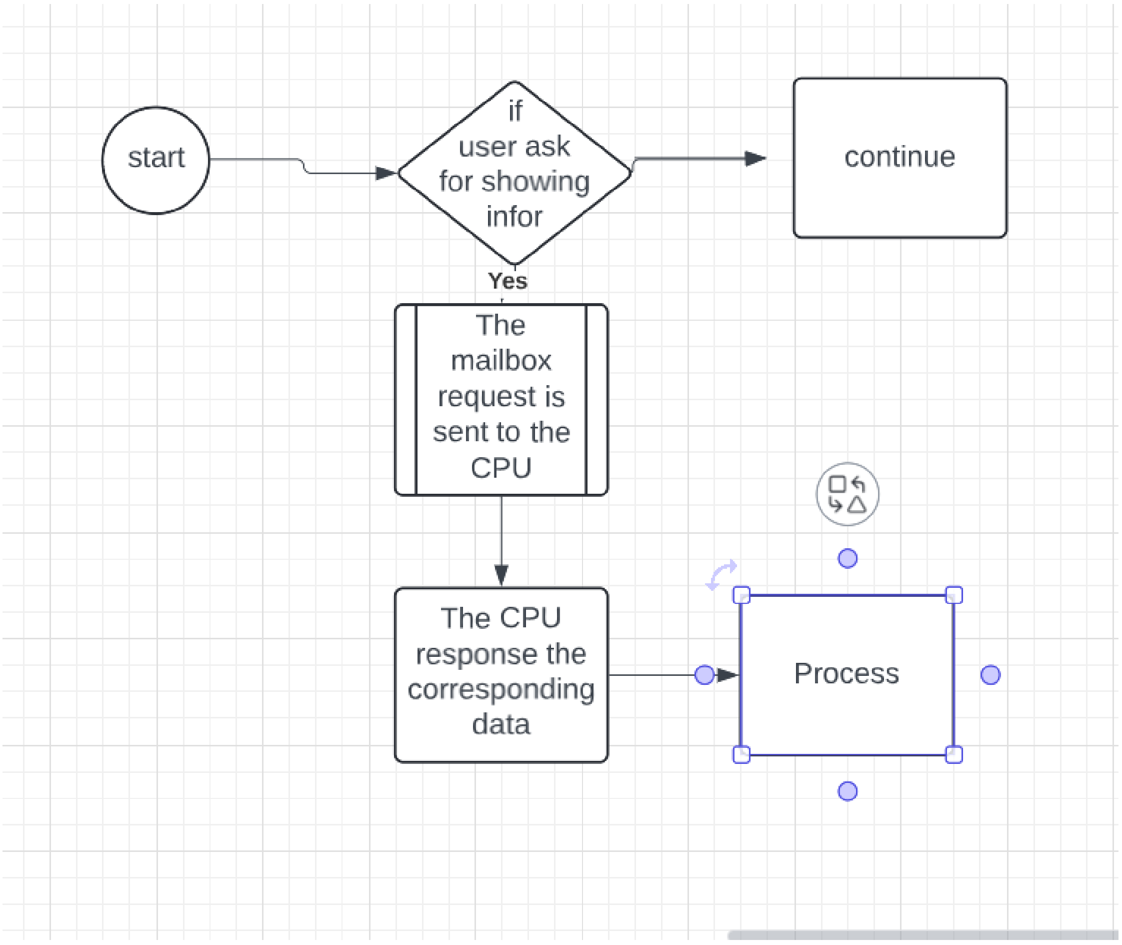
1.9. Show Information command

This command will help user to know all of the information of the device they are working on for example, users can know more about the clock frequency, board revision, the output of the the commad is shown in the figure below:

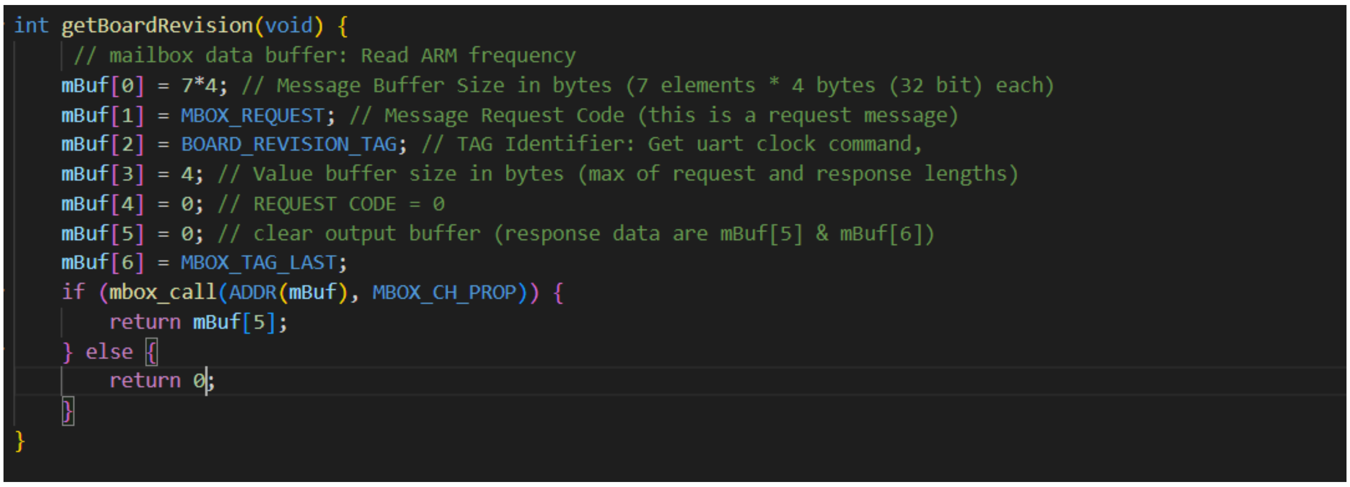


```
C:/DELL/Minh_OS>showinfo
└└+----------------------------------------------------+
└└| Board Info              | Value                    |
└└+----------------------------------------------------+
└└| Board Revision          | └└0x00A02082└└                |
└└| MAC Address             | └└54:52:12:00:54:52└└               |
└└+----------------------------------------------------+

C:/DELL/Minh_OS>
```

This feature works based on the mailbox, to communicate to the hardware layer of the MCU so the knowledge of the mailbox is required to implement the main logic of this task. Firstly and foremost, the definition of mailbox is straightforward, mailbox, literally like its name is a communication between CPU and GPU, Each mailbox is an 8-deep FIFO of 32-bit words, which can be read (popped)/written (pushed) by the ARM and VC. In our Operating System, the mailbox will send the request to CPU for data and will send back to the terminal.

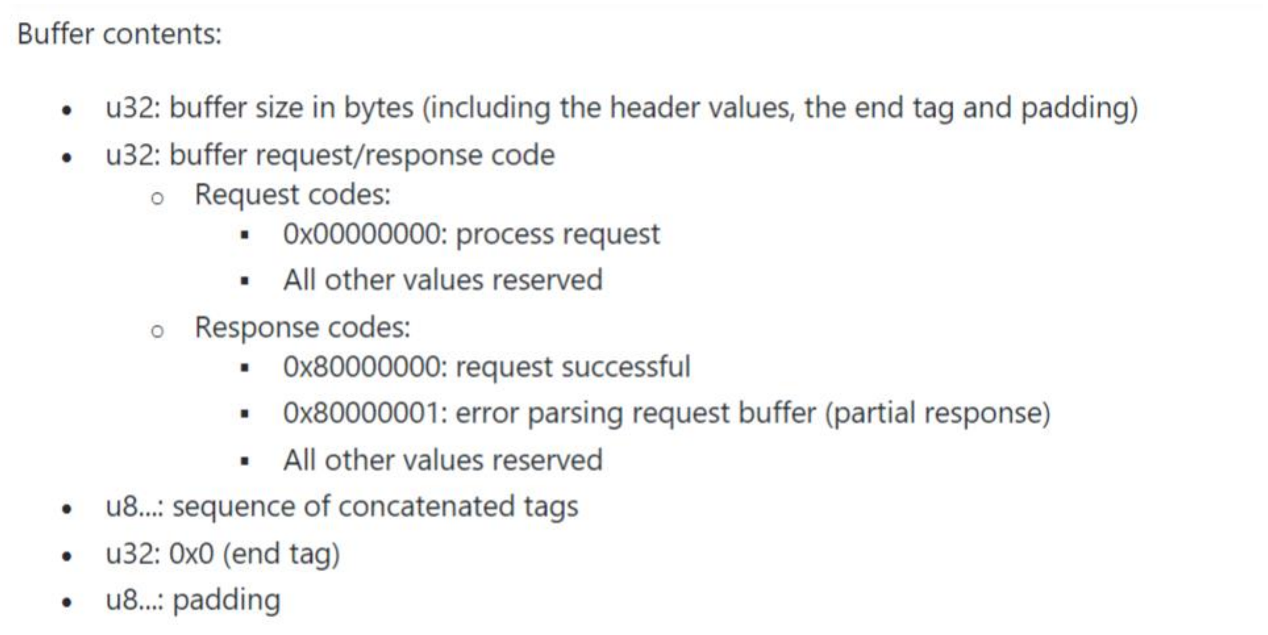

start
if user ask for showing infor
continue
Yes
The mailbox request is sent to the CPU
The CPU response the corresponding data
Process

The flowchart above indicates the process of handling showinfo function, first the system will check if user request to know the board or not. Then the system will send the request data with mail via mailbox buffer, which holds the address of that request. It is known that all the communications in embedded system is the address of the data transfer



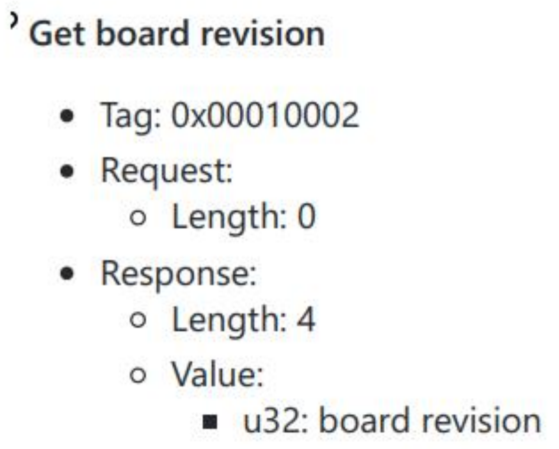
```
int getBoardRevision(void) {
    // mailbox data buffer: Read ARM frequency
    mBuf[0] = 7*4; // Message Buffer Size in bytes (7 elements * 4 bytes (32 bit) each)
    mBuf[1] = MBOX_REQUEST; // Message Request Code (this is a request message)
    mBuf[2] = BOARD_REVISION_TAG; // TAG Identifier: Get uart clock command,
    mBuf[3] = 4; // Value buffer size in bytes (max of request and response lengths)
    mBuf[4] = 0; // REQUEST CODE = 0
    mBuf[5] = 0; // clear output buffer (response data are mBuf[5] & mBuf[6])
    mBuf[6] = MBOX_TAG_LAST;
    if (mbox_call(ADDR(mBuf), MBOX_CH_PROP)) {
        return mBuf[5];
    } else {
        return 0;
    }
}
```

The snipet code above is to get the data of the board revision, behind the scene, the mailbox use the set of buffers to setup the request message to send. The use of each buffers can be briefly described as below:


Buffer contents:

- u32: buffer size in bytes (including the header values, the end tag and padding)
- u32: buffer request/response code
    - Request codes:
        - 0x00000000: process request
        - All other values reserved
    - Response codes:
        - 0x80000000: request successful
        - 0x80000001: error parsing request buffer (partial response)
        - All other values reserved
- u8...: sequence of concatenated tags
- u32: 0x0 (end tag)
- u8...: padding

Combine the buffer contents to the snipet code above, we can determine that mbuf[0] is the size of all elements here which is referred as total number of mbuffer is 7 and the size of response here is 4 byte referred to the TAG datasheet. Mbuffer[1] is the request code for mailbox, by default it is 0x0. Mbuffer[2[is the tag id to distinguish the different tags.


Get board revision

- Tag: 0x00010002
- Request:
    - Length: 0
- Response:
    - Length: 4
    - Value:
        - u32: board revision

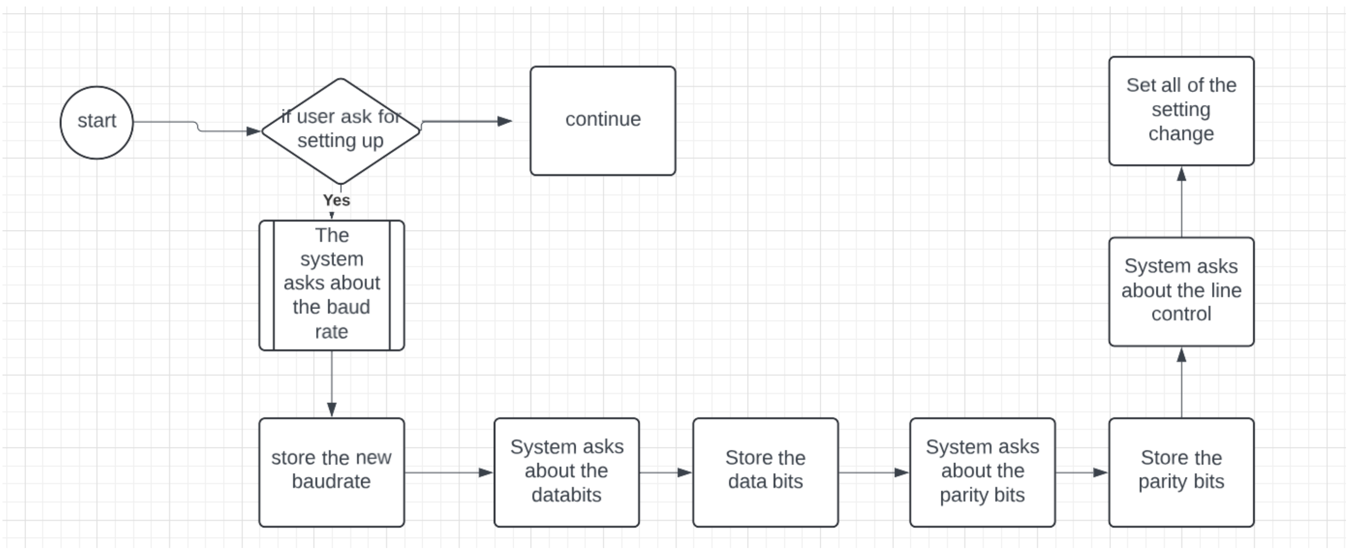
The mbuffer[3] is the maximum size of the value buffer, in this case it is length for 4. The data will be parsed in mbuffer[5] to be display by uart. The mac address of the board works as the same with the revision board function, however, the mac address has specific format type to indicate following the format XX:XX:XX:XX:XX and the return type of the function is integer so I defined a function to convert the integer type into the correct format hexa type

....
....

2. Further Development of UART Driver

....
....

In case user want to change the uart configuration during the system running, just like when booting up the phone, the operating system also has a command which supports that needs of the users:


start
if user ask for setting up
continue
Yes
The system asks about the baud rate
store the new baudrate
System asks about the databits
Store the data bits
System asks about the parity bits
Store the parity bits
System asks about the line control
Set all of the setting change

From the flowchart above describes that first when users hit the boot command, the system will ask users about the corresponding configuration value such as baudrate, data bits, stop bits,… to store in setting variables

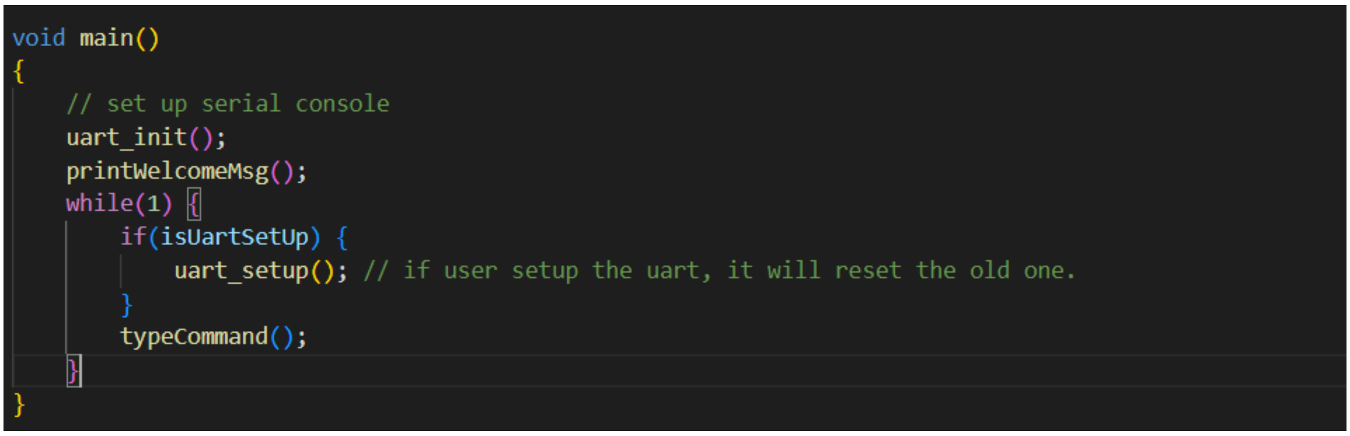
```
/**@brief Function is to setup the uart config
 */
void uart_setup() {
   uint32_t integer_baud, fractional_baud, line_control, control_reg;
   UART0_CR = 0x0; // reset all registers

   /* Calculate baud rate divisor */
```

```
    integer_baud = UART_CLOCK / (16 * custom_baudrate);
    fractional_baud = ((UART_CLOCK % (16 * custom_baudrate)) * 64 + custom_baudrate / 2) /
custom_baudrate;

    UART0_IBRD = integer_baud;
    UART0_FBRD = fractional_baud;

    /* Configure data bits */
    line_control = 0;
    switch (custom_data_bit) {
        case 5:
            line_control |= UART0_LCRH_WLEN_5BIT;
            break;
        case 6:
            line_control |= UART0_LCRH_WLEN_6BIT;
            break;
        case 7:
            line_control |= UART0_LCRH_WLEN_7BIT;
            break;
        case 8:
            line_control |= UART0_LCRH_WLEN_8BIT;
            break;
        default:
            line_control |= UART0_LCRH_WLEN_8BIT; // Default to 8 bits if invalid
            break;
    }

    /* Configure stop bits */
    if (custom_stop_bit == 2) {
        line_control |= UART0_LCRH_STP2; // if 2 for 2 stop bits, else for 1
    }

    /* Configure parity */
    if (parity_bit == EVEN_PARITY_BIT) {  // Even parity
```

```
        line_control |= (UART0_LCRH_PEN | UART0_LCRH_EPS);
    } else if (parity_bit == ODD_PARITY_BIT) {  // Odd parity
        line_control |= UART0_LCRH_PEN;
    }

    /* Enable FIFO */
    line_control |= UART0_LCRH_FEN;

    UART0_LCRH = line_control;

   // Assuming control_reg is declared and appropriately scoped
    control_reg = UART0_CR_TXE | UART0_CR_RXE; // Enable transmit and receive

    // Configure hardware flow control (RTS/CTS)
    // Finalize UART configuration and enable it
    control_reg = UART0_CR_TXE | UART0_CR_RXE | UART0_CR_UARTEN;
    if (hand_shaking == RTS_CTS_EN) {
        control_reg |= (UART0_CR_CTSEN | UART0_CR_RTSEN);
    }
    UART0_CR = control_reg;
    // UART0_CR = control_reg;

    /* Mask all interrupts and clear pending ones */
    UART0_IMSC = 0;
    UART0_ICR = 0x7FF;
}
```

The code snippet above describes the setting of the new uart configuration, compare to the default uart, the new setting uart also need to handle multiple aspects including control register, line control,… So it seems little bit complicated. First, the system will setup the baudrate for both integration part and partion part along with the uart clock of 48 MHZ and the formula I noted in the command above.

Then the length of data bit also be configured within the LCRH register which has multiple options length from 5 to 8. The stop bit is also set to enabled or left by default. The control register will check the set up for handshaking in which to RTS/CTS or none by default.


```
void main()
{
    // set up serial console
    uart_init();
    printWelcomeMsg();
    while(1) {
        if(isUartSetUp) {
            uart_setup(); // if user setup the uart, it will reset the old one.
        }
        typeCommand();
    }
}
```

After setting up, the isUartSetUp flag will be set so the system will use new uart config to run the program.

Summary of features implemented in both Task 1 & 2

Feature Group		**Command/ Feature**	**Implementation**	**Testing (any issues/limitations)**
Basic Commands		help	complete	Pass
		clear	complete	Pass
		setcolor	complete	Pass
		showinfo	complete	Pass
CLI enhancement		OS name in CLI	complete	Pass
		Auto-completion in CLI	complete	Pass
		Command history in CLI	complete	Pass
UART settings	**Baud rate**	… insert command name…	complete	Pass
	Data bits	… insert command name…	complete	Pass
	Stop bits	… insert command name…	complete	Pass
	Parity	… insert command name…	complete	Pass

	Hansharking	… insert command name…	complete	Pass

3. Some Common Sensors

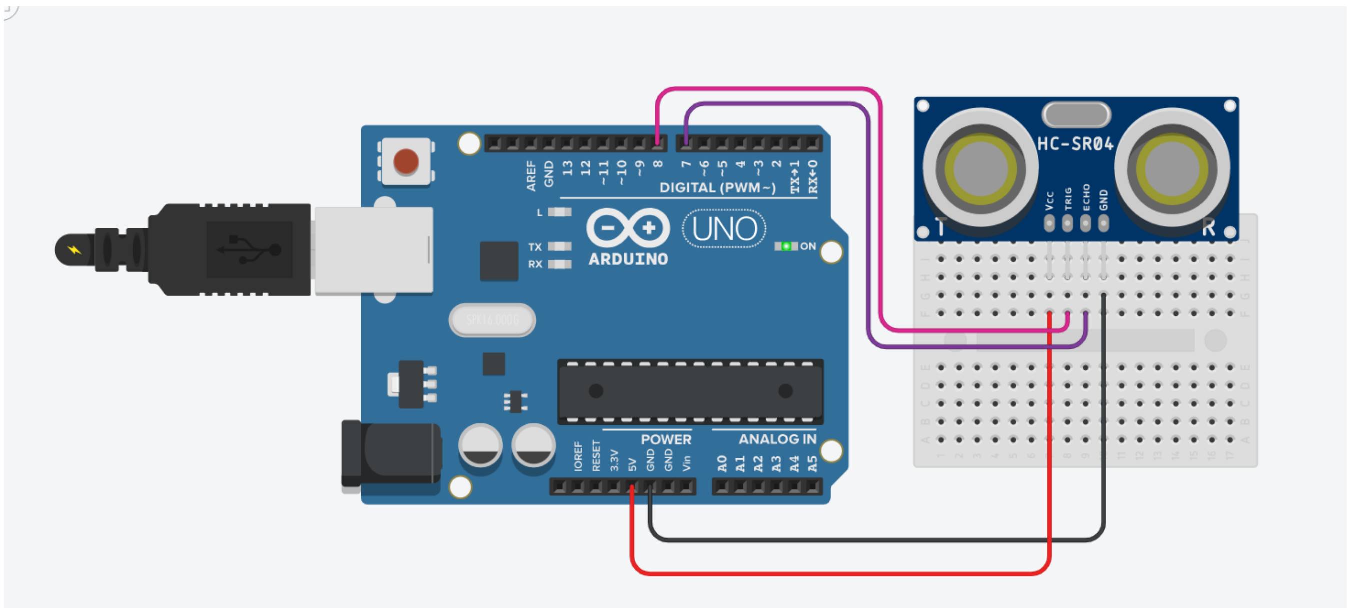
a. Ultra sonic sensor (HC-SR04)

Ultra sonic sensor is a sensor to detects object via the ultrasonic radio, this type of sensor is mostly applied in the marine, radar or specifically in the electric car in order to detect the object all around the car. The ultra sonic sensor has 4 pin which can be explained detail:

- VCC: it is the voltage supplier which is used to power up and run the sensor
- Trigger Pulse Input: This pin is used to initiate the detection first by sendinga short 10uS pulse to the trigger input and it will send a 8 cylcles burst and ultrasound to scan
- Echo pulse output: When the ultrasound detect object and bounce back to the sensors, the echo output pulse to indicate the time of the wave travel from object to sensors
- GND: ground pin

The range can be calculated as below:

$$range = echo * velocity\ of\ sound/2$$



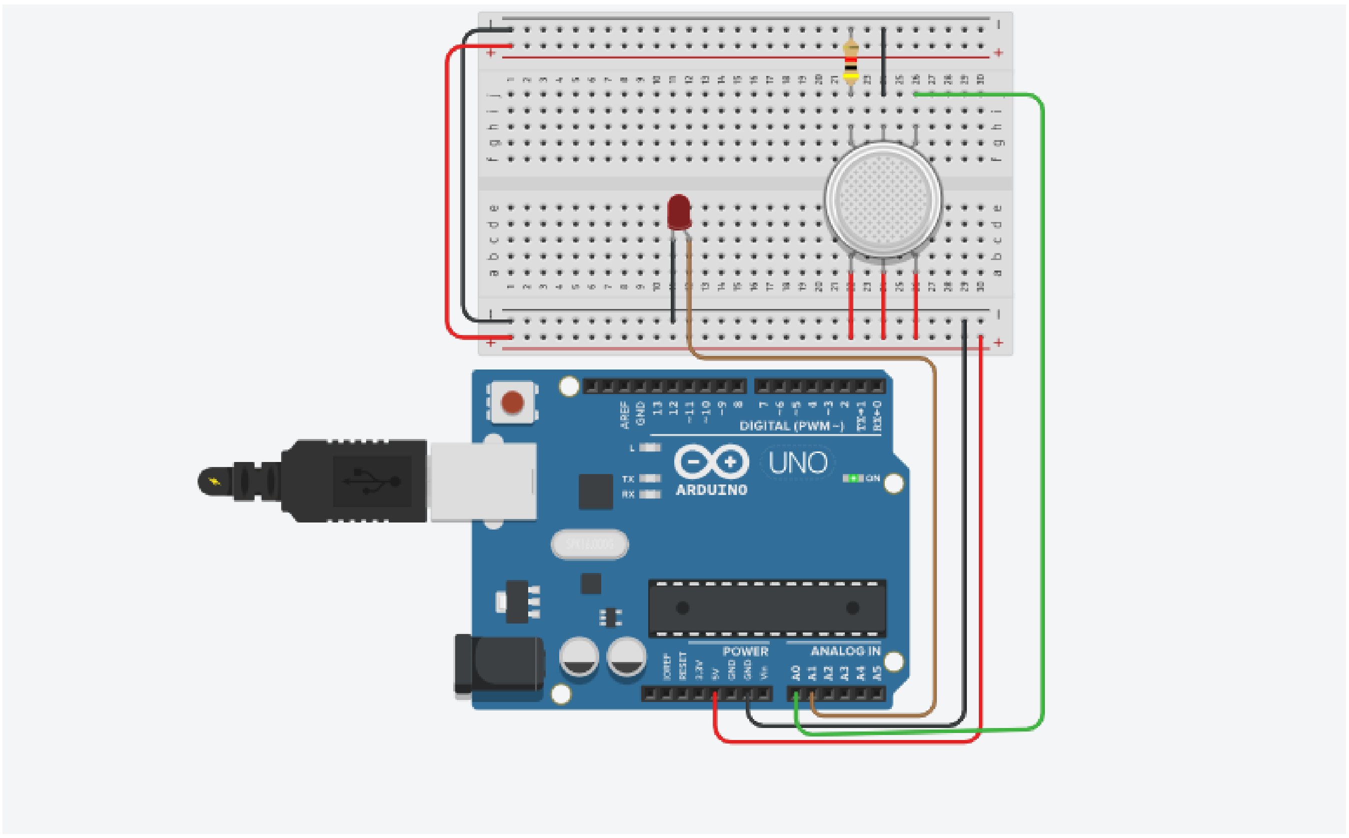
b. Gas sensor

The gas sensor is used to detect gas, and the sensors that is most commonly used is MQ2 gas sensor. The MQ-2 type smoke sensor is made of a tin dioxide semiconductor gas sensing material. When it works at a temperature of 200 to 300 °C, the tin dioxide adsorbs the oxygen in the air, which reduces the density of electrons on the semiconductor and thus increasing the resistance. Inside the sensor, there is a heating element that heats the tin oxide to a high temperature. This is necessary for the chemical reaction that allows the sensor to detect gases. hen the target gases are present in the environment, they react with the heated tin oxide, changing its conductivity. The change in conductivity increases with higher gas concentration.

Totally, the MQ2 has 4 pins to interface with the controller:

- VCC: the power supply for the sensor
- Digital Output: This pin can be used to detect gas if the gas data is over a threshold we set or not
- Analog Ouput: working based on the ADC concept, this pin repeatedly measer the gas and then convert it to the ADC value and send to the microcontroller to process the data, the different ADC value will be depend on the voltage reference of the controller, in Arduino it could be up to 1023 meanwhile in raspberry pi with 3.3 V it could be up to 4095
- GND: the ground pin

The application of MQ2 is used to detect the gas leak that is flameable such as methan, buthane or even measure the air quality if it has the poisonous gas or not. In chemical industry or in chemical lab, this sensor is well used to prevent any bad occurrences.



In the sketch diagram on tinkercad, the LED will turn on slightly if the MQ2 detect any gas or not

c. Light ambident sensor

The light sensors are commonly used to detect the light level, in this report I will discuss about the LDR module light detect. The way it works is based on the resistance of light when landing on it, for example when it is light, the resistance decreases and when it is dark, the resistance will increase and these values then transmitted as the ADC value for micro controller to read

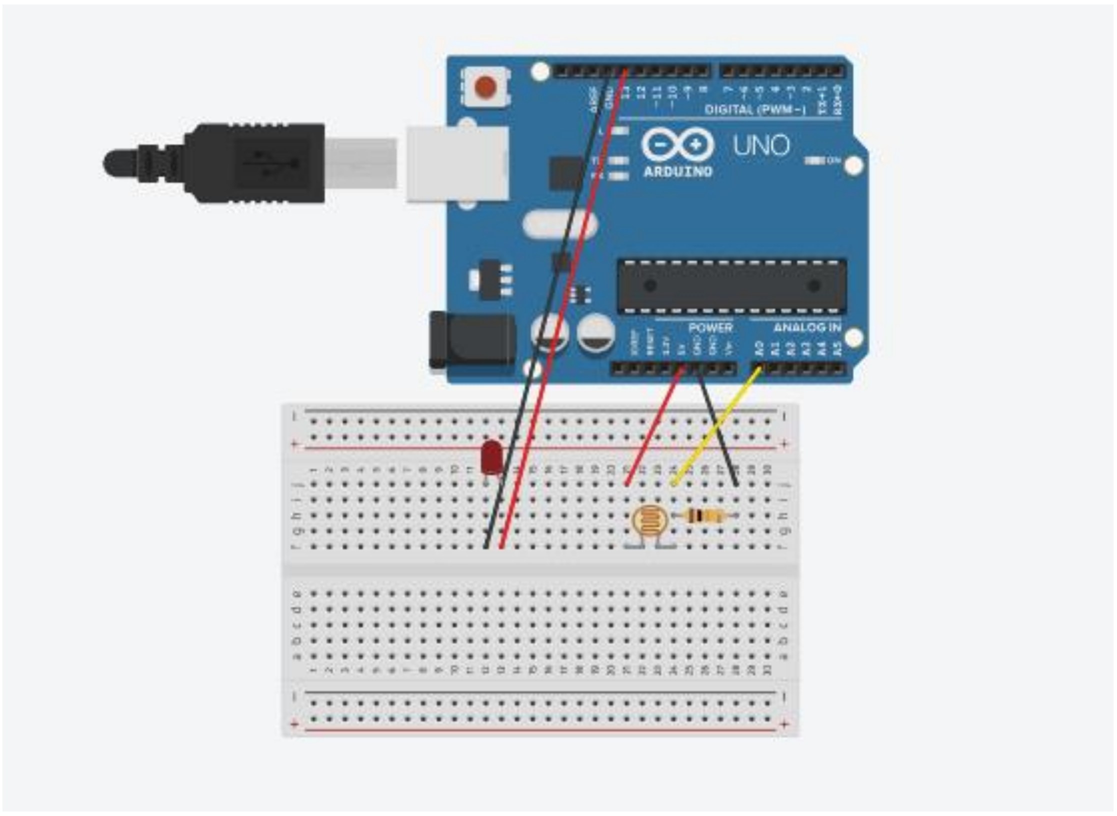
There are 3 pins:

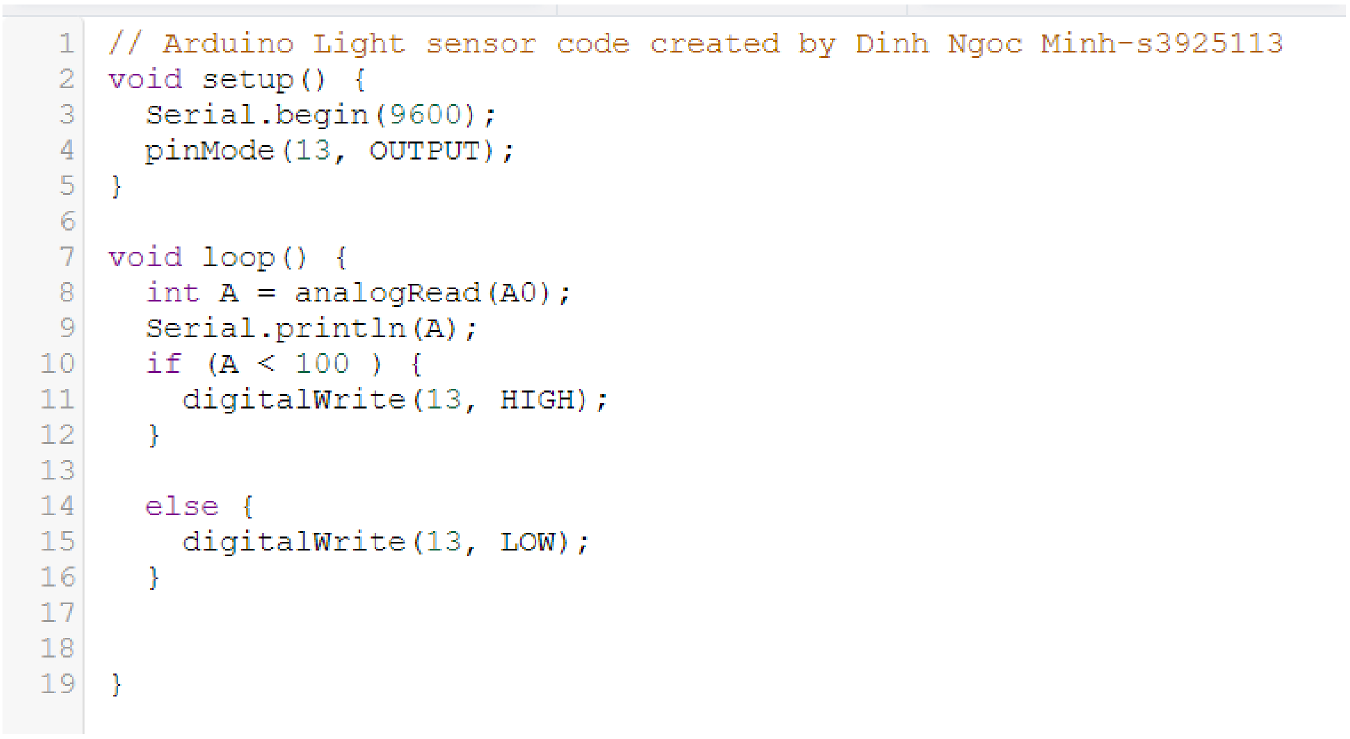
VCC: power supply

GND: ground pin

Analog Output: the analog value read from the sensor to send to micro controller

The light sensor is mostly used in the IoT system that it will automatically turn on the light when it darker than the set threshold and vice versa, which help much in saving the energy





```
// Arduino Light sensor code created by Dinh Ngoc Minh-s3925113
void setup() {
  Serial.begin(9600);
  pinMode(13, OUTPUT);
}

void loop() {
  int A = analogRead(A0);
  Serial.println(A);
  if (A < 100 ) {
    digitalWrite(13, HIGH);
  }

  else {
    digitalWrite(13, LOW);
  }


}
```

In the tinkercad above, the LED will turn on when the ADC value from light detector goes down below 100.

III. Reflection & Conclusion

Even though the system works seems perfectly, therefore there is some piece of codes that not being cleaned properly. As the result, this is the first time I developed my own operating system, so the scope of it can be improved in the future to meet other features as well. Behine the system works well, I spent most of the time to logically align the functionality and debug on both qemu and raspberry pi so after taking this course assignment I ensure that my problem solving skill and debugging skills are on the new level.

IV. REFERENCES (USE IEEE STYLES)

[1] Elec Freaks, 'HC-SR04 data sheet', **https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf**

[2] MakeBlock Education 'MQ2 Gas sensor', https://education.makeblock.com/help/mbuild-mq2-gas-sensor/#:~:text=in%20the%20air.-,Working%20principle,and%20thus%20increasing%20the%20resistance.

[3] Component 101, 'MQ2 Gas sensor pinout', https://components101.com/sensors/mq2-gas-sensor.

[4] Spurr.K, 'How an LDR (Light Dependent Resistor) Works', https://kitronik.co.uk/blogs/resources/how-an-ldr-light-dependent-resistor-works