RMIT UNIVERSITY

EEET2490 – Embedded System: OS and Interfacing, Semester 2023-2

Assessment 2 – Individual Assignment Report

ADDITIONAL FEATURES FOR BARE METAL OS

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

Mr. Phuc Nguyen - phuc.nguyenhoangthien@rmit.edu.vn

Student name : Jaeheon Jeong

Student ID : s3821004

Date : 31.08.2023

TABLE OF CONTENTS

I. INTRODUCTION

In this project, the requirement is that Bare metal OS should be made. By implementing the command line interpreter and standard printf function, it will help to understand how the standard OS is running. It will strengthen our development skills by experiencing embeede OS.

In this report, it will show how the bare metal OS is made for this project. The background will be shown before starting the implementation. It contains about the brief backgound of command line interpreter, ANSI codes, and variable arguments handling functions. After the background, it will explains about how the functions is made, what is the result, and the limits of the function. At the end, it will show conclusion about final result and self-reflection.

II. ADDITIONAL FEATURES FOR BARE METAL OS

1. BACKGROUND

A command line interpreter (CLI) is a software interface that allows users to interact with a computer system or application by entering commands as text. It provides a way to control and manage a system or application without using a graphical user interface. It is used in the operating system and embedded system. It has a help system and auto-completion for users to control easily. For making a CLI, ANSI codes are needed. It is a series of control sequences used to format text and control cursor movement on a text-based terminal or console. For example, '\t' is a horizontal tab, and '\b' is a backspace. It will be used for the delete and auto-completion functions.

While making an OS, the printf function is essentially similar to C programming, and for making the printf function, it needs variable arguments handling functions[1] which is for detecting the arguments. It is helpful to handle the variable inputs. The 'va_list' type is used to hold the list of arguments and the 'va_arg' retrieves the next argument of a specified type. It will come up in the printf implementation.

2. IMPLEMENTATION

a) Welcome message and Command Line Interpreter (CLI)

When the OS is booted up, it should display the welcome message. The command line interpreter (CLI) should appear after the welcome message. It is an indispensable feature of every operating system. The window has 2 CLI called Command Prompt and Powershell. It will show the OS name while waiting for the user to type the command. Deleting the character is required when the user makes the typo and auto-completion, command history, and some commands should be made.

1. Welcome message

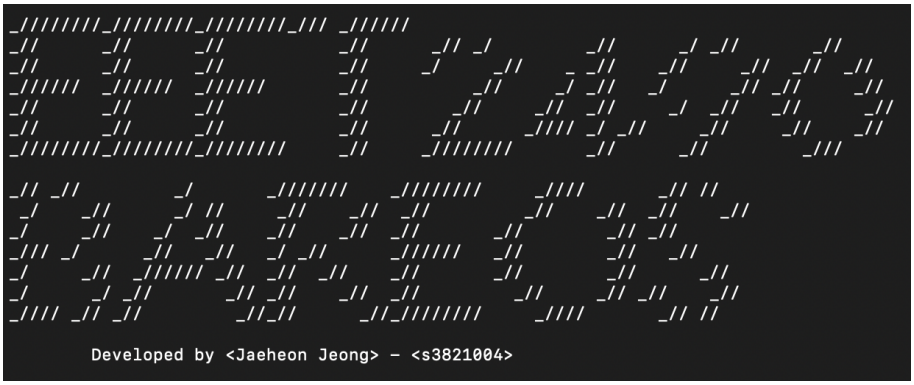


Figure 01: Weclome message when OS is started

When the OS is booted up, it needs welcome message with the student name and student id. The ‘welcomeMessage()’ is implemented with using ASCII art string[2]. The function is used before starting the while loop because it only needs to show once.

2. OS name

Figure 02: OS name while wating user typing the command

After the command is done or wating the user to type the command, it should keep showing the OS name. In the while loop of the main(), ‘cli()’ is keep running so that it should contain OS name in ‘cli()’ at the last.

3. Auto-completion

After the command receives '\t' which is the ANSI code of the tab key during the user types the command, it will operate the 'auto_completion' function. It has a 'command' array which has 'help', 'setcolor', 'clear', and 'showinfo'. When the user puts a word in the command line and if the command has the same order of letters with 'command' array, it will delete the command line and replace the completed 'command'.

```
char* command = auto_completion(cli_buffer);

//uart_puts(command);
custom_memset(cli_buffer, 0, custom_strlen(cli_buffer));
custom_strcpy(cli_buffer, command);
uart_puts(cli_buffer);
index = custom_strlen(cli_buffer);
```

Figure 03: Process of auto-completion function

It will return the completed command to 'command' in the main and it will replace the 'cli_buffer' to command and update the index number.

It can not show the result clearly by showing before and after when the tab key is pressed, Therefore, it will be shown in the demo video.

4. Delete function

When the 'uart_get()' gets '\b', the delete function will activated. If the 'index' is over 0 which means there is character in 'cli_buffer', it will use 'deleteCharacter()'.

```
for (int i = index; cli_buffer[i] != '\0'; i++) {
    cli_buffer[i] = cli_buffer[i + 1];
}

uart_sendc('\b'); // Move cursor back one position
uart_sendc(' '); // Display a space to erase the character
uart_sendc('\b'); // Move cursor back again
```

Figure 04: overwrite deleted character in 'cli_buffer' and delete in the terminal

By using for loop, it will Shift the characters in 'cli_buffer' to the left to overwrite the deleted character. After that, it will remove a letter from the terminal by moving the cursor back to one position replacing it with blank, and moving cursor back again.

Figure 05: Result after pressing delete key

5. Command history

To show history command when pressing '_' or '+', 'history' array of 2 dimension is created to store the command that user has typed.

```
while (index > 0) {
    uart_sendc('\b'); // Move cursor back
    uart_sendc(' ');  // Clear character
    uart_sendc('\b'); // Move cursor back again
    index--;
}

// Display the new command
uart_puts(command);
index = custom_strlen(command);


custom_memset(cli_buffer, 0, custom_strlen(cli_buffer));
custom_strcpy(cli_buffer, command);
```

Figure 06: Process of showing previous command in terminal

When '_' is pressed, it will check 'history' is not empty. After that, it will delete the letters from that current command line. 'command' is the command of the previous command from the 'history' array. It will show the previous command until it reaches the first command. 'custom_memset' is used to delete the current 'cli_buffer' and it will fill with 'command' in 'cli_buffer'.

when '+' is pressed, it will increase the 'history' index to show next command from the 'history' array. It will works until the 'history' reaches the end of the command. The process of deleting the command line is same when '_' is pressed.

The result does not show clearly from the screenshot, therefore, it will show in the demo video.

6. Command features

To compare the command has the right format. It has 'custom_strcpy()', 'custom_strncmp()', 'custom_strlen', and 'custom_strstr' which are same function that "string.h' library has in C programming.

```
if (custom_strncmp(cli_buffer, "setcolor -t\n", 8) == 1) {
    changeBackground(cli_buffer);
    changeText(cli_buffer);
} else if (custom_strncmp(cli_buffer, "clear\n", 5) == 1) {
    clearScreen();
} else if (custom_strncmp(cli_buffer, "help\n", 4) == 1) {
    help(cli_buffer);
} else if (custom_strncmp(cli_buffer, "showinfo\n", 8) == 1) {
    showinfo();
}
```

Figure 07: Compare ‘cli_buffer’ with string

When ‘\n’ is detected, it means user type is ended so that ‘cli_buffer’ will compare the string to find the right command. If the command it true, it will return 1.

A. Command help

After comparing string “help” with ‘cli_buffer’, it will execute the ‘help()’ function.

```
const char *commands[] = {
    "help\t\t\t\tShow brief information of all commands",
    "help <command_name>\t\tShow full information of the command",
    "clear\t\t\t\tClear screen",
    "setcolor -t <text color>\tSet text color",
    "setcolor -b <background color\tSet back color",
    "showinfo\t\t\tShow board revision and board MAC address "
};
```

Figure 08: List of the command information

It has an array that shows the command and the short information about the command. it will use for loop until the array ends when then help function is executed.

```
MyOS> help
For more information on a specific command, type help command-name
help                            Show brief information of all commands
help <command_name>             Show full information of the command
clear                           Clear screen
setcolor -t <text color>        Set text color
setcolor -b <background color   Set back color
showinfo                        Show board revision and board MAC address

MyOS>
```

Figure 09: Result of ‘help’ command

In the terminal, it shows successfully when it types ‘help’ in the command. After printing all information about the command, it will show OS name in the new line.

If the user wants to see details of the command information, it could show information by typing ‘help <command_name>’. For identifying that the command has <command_name> after ‘help’, it will use ‘custom_strlen’ to check if the string length is over 5. If it is over 5

characters, it will use ‘custom_strncmp’ to check whether there is the same command name or not.

```
MyOS> help showinfo
Show board revision and board MAC address in correct format/ meaningful information

MyOS> help clear
Clear screen (in our terminal it will scroll down to current position of the cursor).

MyOS>
```

Figure 10: Result of ‘help <command_name>’ command

By typing different command_name with ‘help’, it shows different information and more detail than the result of ‘help’ command shows.

B. Command clear

After comparing string “clear” with ‘cli_buffer’, it will execute the ‘clearScreen()’ function.

```
void clearScreen() {
    uart_puts("\033[2J");  // Clear the screen
    uart_puts("\033[H");  // Move cursor to top left corner
}
```

Figure 11: Process of ‘clearScreen()’

"\033[2J" will clear the screen first, and "\033[H" will make to move cursor on the top left cursor.

```
MyOS> help showinfo
Show board revision and board MAC address in correct format/ meaningful information

MyOS> help clear
Clear screen (in our terminal it will scroll down to current position of the cursor).

MyOS> clear

MyOS>
```

Figure 12: Scroll up after using ‘clear’ command

‘clearScreen’ works successfully. It clears the screen and only OS name appears on the terminal. When the user scroll up the terminal, it can see the previous command as the Figure 12 shows.

C. Command setcolor

```
if (custom_strncmp(cli_buffer, "setcolor\n", 8) == 1) {
    changeBackground(cli_buffer);
    changeText(cli_buffer);
}
```

Figure 13: 2 functions when ‘cli_buffer’ has string “setcolor”

After comparing string “clear” with ‘cli_buffer’, it will execute the ‘changeBackground()’ and ‘changeText’ because after ‘setcolor’, it needs ‘ -t’ or ‘ -b’ to change color of text or background.

After ‘ -t’ or ‘ -b’ is detected, it needs color that user wants to change.

```
if (custom_strncmp(color_t, "black", 5) == 1) {
    // Set text color to yellow
    uart_puts("\033[30m");
} else if (custom_strncmp(color_t, "yellow", 6) == 1) {
    // Set text color to yellow
    uart_puts("\033[33m");
} else if (custom_strncmp(color_t, "red", 3) == 1) {
    // Set text color to red
    uart_puts("\033[31m");
} else if (custom_strncmp(color_t, "blue", 4) == 1) {
    // Set text color to black
    uart_puts("\033[34m");
} else if (custom_strncmp(color_t, "green", 5) == 1) {
    // Set text color to black
    uart_puts("\033[32m");
} else if (custom_strncmp(color_t, "purple", 6) == 1) {
    // Set text color to black
    uart_puts("\033[35m");
} else if (custom_strncmp(color_t, "cyan", 4) == 1) {
    // Set text color to black
    uart_puts("\033[36m");
} else if (custom_strncmp(color_t, "white", 5) == 1) {
    // Set text color to black
    uart_puts("\033[37m");
}
```

Figure 14: Compare color and set text color

It will use ‘custom_strncmp’ to compare color and if the color matches with the ‘color_t’, it will set text color as the Figure 14 shows. The background color set will have similar function but different code for setting background.

```
MyOS> setcolor -t yellow
MyOS> setcolor -b blue
MyOS> setcolor -t red
MyOS> setcolor -b green -t white
MyOS> help
For more information on a specific command, type help command-name
help                            Show brief information of all commands
help <command_name>             Show full information of the command
clear                           Clear screen
setcolor -t <text color>        Set text color
setcolor -b <background color   Set back color
showinfo                        Show board revision and board MAC address
```

Figure 15: Result of ‘setcolor -t’ and ‘setcolor -b’ in terminal.

On the first row, it will change the text to yellow. It will show yellow text from the second row. Second row, the background as blue, and it shows a background with blue and text with yellow in the third row. The third row command sets the text color as red and the fourth row changes successfully. On the fourth, it changes text and background at once. After the fourth row, it successfully changes the text to white and the background to green.

D. Command showinfo

After comparing string “showinfo” with ‘cli_buffer’, it will execute the ‘showinfo()’.

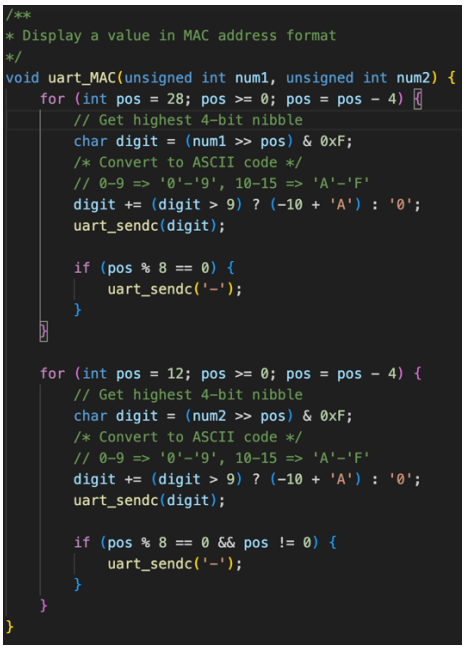
```
mBuf[0] = 12 * 4; // Message Buffer Size in bytes (12 elements * 4 bytes (32 bit) each)
mBuf[1] = MBOX_REQUEST; // Message Request Code (this is a request message)

mBuf[2] = 0x00010002; // TAG Identifier:board revision
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

mBuf[6] = 0x00010003; // TAG Identifier: MAC address
mBuf[7] = 6; // Value buffer size in bytes (max of request and response lengths)
mBuf[8] = 0; // REQUEST CODE = 0
mBuf[9] = 0; // clear output buffer
mBuf[10] = 0; // clear output buffer

mBuf[11] = MBOX_TAG_LAST;
```

Figure 16: set mailbox to get board revision and MAC address

To display board revision and MAC address in the terminal, it should set the mailbox. It needs 12 elements and the value for board revision will store in ‘mBuf[5]’ and MAC address will store in ‘mBuf[9]’ nad ‘mBuf[10]’.

```
MyOS> showinfo
Board Revision: 0x00A02082 : rpi-3B BCM2837 1GiB Sony UK
MAC Address: 57-34-12-00-54-52
```

Figure 17: Result of command ‘showinfo’



```c
/**
* Display a value in MAC address format
*/
void uart_MAC(unsigned int num1, unsigned int num2) {
    for (int pos = 28; pos >= 0; pos = pos - 4) {
        // Get highest 4-bit nibble
        char digit = (num1 >> pos) & 0xF;
        /* Convert to ASCII code */
        // 0-9 => '0'-'9', 10-15 => 'A'-'F'
        digit += (digit > 9) ? (-10 + 'A') : '0';
        uart_sendc(digit);

        if (pos % 8 == 0) {
            uart_sendc('-');
        }
    }

    for (int pos = 12; pos >= 0; pos = pos - 4) {
        // Get highest 4-bit nibble
        char digit = (num2 >> pos) & 0xF;
        /* Convert to ASCII code */
        // 0-9 => '0'-'9', 10-15 => 'A'-'F'
        digit += (digit > 9) ? (-10 + 'A') : '0';
        uart_sendc(digit);

        if (pos % 8 == 0 && pos != 0) {
            uart_sendc('-');
        }
    }
}
```

Figure 18: ‘uart_MAC’ function to convert in MAC address format

By showing board revision, it will use ‘checkBoardRevision()’ to chechk the value from ‘mBuf[5]’ and print the board name. For the MAC address, the format should setted. It needs ‘-‘ between after 2 digits appear. Therefore, ‘uart_MAC’ function is made in the uart c file.

The limit of the CLI is that it also detects 1 more character when the user types some commands. ‘helps’ also works although ‘strncmp’ does for 4 letters. Besides, it has an error when using ‘_’ or ‘+’. It sometimes removes the last letter of the OS name.

b) printf function

For an OS, when printing the data to the console, it uses the printf function. Therefore, in this project, the printf function should be developed similarly to the printf in C programming. It needs different functions to print out different formats such as string, integer, hexa, or character. In this section, it will explain about the function of different formats.

When using the printf function, it will state the character array ‘buffer’ and it will clear with 0 value at the beginning. Since the printf function has unlimited arguments, it requires variable arguments handling libraries (stddef.h, stdint.h, stdarg.h) during the codes are implemented. The libraries are included in the ‘printf.h’ file.

1. d specifier

After detecting '%' from '*string', d specifier function occurs when it also detects 'd'. It will get the integer value by using 'va_arg()'.

```
int isNegative = 0;
if (x < 0) {
    x = -x; // Make x positive for the conversion
    isNegative = 1;
}

if (isNegative) {
    buffer[buffer_index] = '-';
    buffer_index++;
}
```

Figure 19: Add '-' when the given integer is minus

When the given integer is negative number, it should be printed with '-' in the front. Therefore, when 'isNeative' is 1, buffer stores the '-' and increase the index number.

```
do {
    temp_buffer[temp_index] = (x % 10) + '0';
    temp_index--;
    x /= 10;
} while(x != 0);
```

Figure 20: Add decimal integer in temp_buffer

It uses the do while loop to store a decimal integer in 'temp_buffer'. It will store the value which is the remainder of 10 and cobine with character '0' because it can convert integer to character when storing in the string. It is based on the ASCII table. After all digits are stored in the 'temp_buffer', it will restore in the 'buffer' and printed out by 'uart_puts()'.

```
printf("\n\ndecimal: %d\n", 5);

printf("decimal: %d\n\n", -11);
```

Figure 21: Test d specifier in kernel.c file

```
decimal: 5
decimal: -11
```

Figure 22: Result of the d specifier

By testing the positivie integer and negative integer, it display successfully in the terminal as the Figure 22 showed.

2. c specifier

c specifier function occurs when the string has '%c'.It will get character by 'va_arg()' and it will directly store in the buffer because it only displays 1 character.

```
printf("\n\nchracter: %c\n", 'A');

printf("chracter: %c\n\n", 'c');
```

Figure 23: Test c specifier in kernel.c file

```
chracter: A
chracter: c
```

Figure 24: Result of the c specifier

By testing upper case character and lower case character, it display successfully in the terminal as the Figure 24 showed.

3. s specifier

s specifier function occurs when the string contains '%s'. By getting the value with character pointer with 'va_arg()' in 's'. it will store character until the 's' has '\0' which means it ends the string.

```
s_index = 0;

while (s[s_index] != '\0') {
    buffer[buffer_index] = s[s_index];
    buffer_index++;
    s_index++;
}
```

Figure 25: String the character in 'buffer'

It uses while loop until checking the '\0' in the array of 's'. Inside the loop, it will store character and increase index for 'buffer_index' and 's_index'.

```
printf("\n\nstring: %s\n\n", "Hello World");
```

Figure 26: Test s specifier in kernel.c file

```
string: Hello World
```

Figure 27: Result of the s specifier

The result of the s specifier shows the string successfully in the terminal.

4. f specifier

f specifier function occurs when the string contains '%f'. The given value is gotten by the double using 'va_arg()'. It will store '-' in buffer if the given value is negative. The process is same with the d specifier.

```
int int_part = (int)f;
double frac_part = f - int_part;
```

Figure 28: Divide float to 2 section

As the Figure 28 shows, it will store the given value with 2 section. Integer is stored in 'int_part' and floating point is sotred in 'frac_part'. After stroing the integer part in 'buffer' by while loop, it will add '.' to display that after '.' shows 'frac_part'.

```
// Print fractional part (up to 6 digits)
for (int i = 0; i < 6; i++) {
    frac_part *= 10;
    int digit = (int)frac_part;
    buffer[buffer_index] = digit + '0';
    buffer_index++;
    frac_part -= digit;
}
```

Figure 29: Process of storing fractional part

The f specifier shows up to 6 digits as default so that it uses for loop to rotating only 6 times. Each time, it will multiply to get the 1 digit of integer from 'frac_part' and store in the 'buffer'.

```
printf("float: %f\n", 3.1235678);

printf("float: %f\n", -10.25);
```

Figure 30: Test f specifier in kernel.c file

```
float: 3.123456
float: -10.250000
```

Figure 31: Result of the f specifier

When the given number has 8 digits in the fractional part, it only displays 6 digits in the terminal. The second test shows a negative number and the number has only 2 digits, the rest of the fractional part is shown as '0' because the default is up to 6 digits. Figure 31 shows that the f specifier works successfully.

5. % specifier

```
else if (*string == '%') {
    buffer[buffer_index] = '%';
    buffer_index++;
    string++;

}
```

Figure 32: Process of % specifier

When specifier use '%' in the string, % specifier is needed to display '%'. When '%' is detected in the string continuously, it will store character '%' in 'buffer'.

```
printf("\n\n%% is used in string\n\n");
```

Figure 33: Test % specifier in kernel.c file

```
% is used in string
```

Figure 34: Result of the % specifier

As the Figrue 34 shows, % specifier works successfully.

6. x specifier

To convert the integer to hexadecimal number. It should know number of digit dividing by 16 firstly. It will use do while loop.

When the given number is negative, it should apply 2's complement to the function that the value starts at 0xFFFFFFFF with decreasing order so that the given number is added with 0xFFFFFFFF when checking if the number is negative.

```
do {
    int remainder = x % 16;
    if (remainder < 10) {
        temp_buffer[temp_index] = remainder + '0';
    } else {
        temp_buffer[temp_index] = remainder - 10 + 'a';
    }
    temp_index--;
    x /= 16;
} while (x != 0);
```

Figure 35: Process of converting integer to hexa

It will get the remainder divided by 16 and compared with the value of 10. If the remainder is under 10, it will add '0' to store in the character. If it is not, it will subtract 10 and add 'a' because the hexa decimal number includes 'a' to 'f' after the number 9. Since it completes storing value in 'temp_buffer', it will store in the 'buffer'.

```
printf("\n\nhexa: %x\n", 160);

printf("hexa: %x\n\n", -177);
```

Figure 36: Test x specifier in kernel.c file

```
hexa: a0
hexa: ffffff4f
```

Figure 37: Result of the x specifier

As th Figure 37 shows, it display hexa decimal number successfully with positive and negative number.

7. 0 flag

When '0' appears after the '%', it will start 0 flag. After '0', it will give integer between 'd' which is the number of '0' to display.

```
int zeroPadding = 0;
while (*string >= '0' && *string <= '9') {
    zeroPadding = zeroPadding * 10 + (*string - '0');
    string++;
}
```

Figure 38: Getting number for displaying zero

Displaying 0 number will be saved in ‘zeroPadding’. When the number character is detected in the string, it will store the number in ‘zeroPadding’.

```
// Add zero padding
while (buffer_index <= temp_index - MAX_PRINT_SIZE + zeroPadding + cur_index - isNegative) {
    buffer[buffer_index] = '0';
    buffer_index++;
}
```

Figure 39: Store ‘0’ in ‘buffer’

When displaying ‘0’. It should include the digit of the number and ‘-‘. Therefore, while loop only rotates ‘zeroPadding” times excluding the digit of the number and ‘-‘.

```
printf("\n\nPositive decimal: %010d\nNegative decimal: %010d\n\n", 25, -25);
printf("Positive float: %010f\nNegative float: %010f\n\n", 10.25, -3.123);
printf("Positive hexa: %010x\nNegative hexa: %010x\n\n", 25, -25);
```

Figure 40: Test 0 flag in kernel.c file

```
Positive decimal: 0000000025
Negative decimal: -000000025

Positive float: 010.250000
Negative float: -03.123000

Positive hexa: 0000000019
Negative hexa: 00fffffe7
```

Figure 41: Result of the 0 flag

As the Figure 41 shows, 0 flag works successfully. When printing integer, it includes ‘-‘ and display ‘0’. For the decimal floating point, ‘.’ Is also included before printing ‘0’.

8. Width

```
int widthPadding = 0;

if (*string == '*') {
    string++;

    widthPadding = va_arg(ap, int);
}
```

Figure 42: Result of the 0 flag

For using width, it has '*' between the specifier, for example, '%*d' or '%*x'. when '*' is detected in the string, 'widthPadding' will get integer by 'va_arg()'.

```
if (widthPadding != 0) {
    int cur_index = buffer_index;

    while (buffer_index <= temp_index - MAX_PRINT_SIZE + widthPadding + cur_index - isNegative) {

        buffer[buffer_index] = ' ';
        buffer_index++;
    }
}
```

```
for (int i = 0; i < widthPadding - s_index; i++) {
    buffer[buffer_index] = ' ';
    buffer_index++;
}
```

Figure 43 & 44: Process of adding width in integer and string

For d, x, and f specifiers, it will have a similar process to make blank space. Similar to the 0 flag, it should exclude the digit of the number and '-'. For the string, it uses for loop that rotates 'widthPadding – s_index' times. It also excludes the length of the string. ' ' is stored in the 'buffer' when showing the blank. It also works with a c specifier.

```
printf("\n\ndecimal: %*d\nfloat: %*f\n", 5, -110, 12, 3.25);
printf("string: %*s\ncharacter: %*c\n", 8, "Hello", 3, 'c');
printf("hexa: %*x\n\n", 5, 16);
```

Figure 45: Test width in kernel.c file

```
decimal:  -110
float:     3.250000
string:    Hello
character:   c
hexa:    10
```

Figure 46: Result of the width

The width specifier works successfully in the terminal. It excludes '-' when the number is negative. For decimal floating point, it also includes the '.' and fractional part when showing blank. Therefore, only 4 blanks are displayed in the second row of Figure 46.

9. Precision

```
int precisionPadding = 0;
while (*string >= '0' && *string <= '9') {
    precisionPadding = precisionPadding * 10 + (*string - '0');
    string++;
}
```

Figure 47: Getting number for ‘precisionPadding’

Similar to the 0 flag, it also get precisionPadding after ‘.’ Is detected in the specifier such as ‘%.3d’ or ‘%.5s’. it will store number when ‘string’ has character number.

```
int cur_index = buffer_index;

// Add precision padding
while (buffer_index <= temp_index - MAX_PRINT_SIZE + precisionPadding + cur_index) {

    buffer[buffer_index] = '0';
    buffer_index++;
}
```

Figure 48: Store ‘0’ in ‘buffer’

When store ‘0’ with ‘precisionPadding’ times. It should exclude the digits of the number.

```
printf("\n\ndecimal: %.5d\nfloat: %.12f\n", 15, -10.25);
printf("string: %.3s\nhexa: %.6x\n\n", "Hello world", 255);
```

Figure 49: Test precision in kernel.c file

```
decimal: 00015
float: -10.250000000000
string: Hel
hexa: 0000ff
```

Figure 50: Result of the precision

In Figure 50, the first row prints three ‘0’ because it already has 2 digits. The second row prints ‘0’ ten times because when precision is used for decimal floating point, it will start printing ‘0’ after the fractional part. However, the fractional part already contains 2 digits, thus, it will only show 10 ‘0’. For the string, it only requires 3 characters so that it only prints ‘Hel’.

The difference with the 0 flag and precision specifier appears when it uses negative number.

```
printf("\n\n0 flag integer: %05d\nprecision integer: %.5d\n\n", -25, -25);
```

Figure 51: Test for the difference between 0 flag and precision

```
0 flag integer: -0025
precision integer: -00025
```

Figure 52: Result of the difference between 0 flag and precision

When using the 0 flag, the total digits are 5 including the '-'. However, it ignores '-' when the precision specifier is used. It counts after '-' is appeared. Although it has a similarity that shows '0', some kinds are different.

The limit of the printf function in this project is that it cannot use width and precision at once. When using the 'stdio.h' library and printing with width and precision, it successfully prints the string. However, self-made printf only consider width when the string has width and precision specifier.

c) function for mailbox setup

The requirement of this section is that making the code cleaner for the setup of the mailbox. The function is given and it should be implemented for different TAG setup.

```
void mbox_buffer_setup(unsigned int buffer_addr, unsigned int tag_identifier,
                       unsigned int **res_data, unsigned int res_length,
                       unsigned int req_length, unsigned int [list of request values])
```

Figure 53: Requirement to develop the function 'mbox_buffer_setup()'

```
void mbox_buffer_setup(unsigned int buffer_addr, unsigned int tag_identifier,
unsigned int **res_data, unsigned int res_length, unsigned int req_length, ...);
```

Figure 54: 'mbox_buffer_setup' in 'custom.h' file

Its arguments contain the address of the mailbox buffer, TAG identifier value, response data, response value, request data, and list of the request values. For the list of request values, each tag

has different request values. Some do not need request value and some have 1 or 2 values. Therefore, variable arguments handling functions such as ‘va_arg’, ‘va_end’, and ‘va_start’ are used and the function above in Figure 54 is implemented in the ‘custom.h’ file.

```
/* tags that are used in setup function*/
#define MBOX_TAG_GETCLOCKRATE 0x00030002
#define MBOX_TAG_GETFIRMWARE 0x00000001
#define MBOX_TAG_GETBOARDREVISION 0x00010002
#define MBOX_TAG_SETPHYWH 0x00048003
#define MBOX_TAG_GETMODEL 0x00010001
```

Figure 55: 5 different TAGs that are used in setup function

In the assignment detail, it only needs to support 5 different TAGs so that 5 TAGs are selected as Figure 55 shows.

At the first and the end, it should contain va_start and va_end to get the value when the request value is needed. Some TAGs have 2 response data and some have only 1 so the size of the message buffer is different and the index of the ‘MBOX_TAG_LAST’ is also different. Clock rate and setting physical width/height return 2 response data and the rest return 1 data. Hence, if statement is used to separate the TAGs to set different sizes and TAG last. For returning 2 response data, it will contain 8 elements and the rest will have 7 elements.

```
if (res_length > req_length) {
    mBuf[3] = res_length;
} else {
    mBuf[3] = req_length;
}
```

Figure 56: Compare size of the response size and request size

When setting the value buffer size, res_length and req_length should be compared because the value buffer size should be setted by the bigger value.

There are 3 cases to set the response data because of different request data and response data. Above 5 TAGs, it is separated by no request value 1 response data, 1 request value 2 response datas, and 2 request values 2 response datas.

1. No request value 1 response data

   Getting firmware revision, getting board revision, and getting board model has to send back only 1 response data.

```
} else if (tag_identifier == MBOX_TAG_GETFIRMWARE
        || tag_identifier == MBOX_TAG_GETBOARDREVISION
        || tag_identifier == MBOX_TAG_GETMODEL) {
    mBuf[5] = 0;
    *(volatile unsigned int **)res_data = &mBuf[5];
}
```

Figure 57: To set 1 response data

mBuf[5] is cleared because it will store the response data. After clearing, 'res_data' double pointer[3] points the same address of the mBuf[5] so that it can be manipulated by external code.

2. 1 request value 2 response datas

Getting clock rate has 1 request and 2 response datas.

```
if (tag_identifier == MBOX_TAG_GETCLOCKRATE) {
    for (int i = 5; i < 5 + (res_length / 4 - 1); i++) {
        int x = va_arg(ap, int);

        mBuf[i] = x;
        *(volatile unsigned int **)res_data = &mBuf[i];
    }

    mBuf[6] = 0; // clear output buffer
    *(volatile unsigned int **)(res_data + 1) = &mBuf[6];
}
```

Figure 58: To set 1 request 2 response datas

By getting 'res_length', it could easily know that it has list of the request value. However, it only contains 1 request which is clock id. The value will be stored inside of the for loop. By having 2 response, 'res_data' should store 2 values. Therefore, second value should be store in 'res_data + 1'. It points to address of the mBuf and it can be gotten by using array.

3. 2 request values 2 response datas

Setting physical width/height needs2 requests and 2 response datas.

```
else if (tag_identifier == MBOX_TAG_SETPHYWH) {
    for (int i = 5; i < 5 + (res_length / 4); i++) {
        int x = va_arg(ap, int);

        mBuf[i] = x;
        *(volatile unsigned int **)(res_data + i - 5) = &mBuf[i];
    }
}
```

Figure 59: To set 2 requests 2 response datas

In this case, 2 request can be known by dividing the res_length with 4. Setting physical width/height size is 8 and it needs width value and height value for request. Thus, each response data is should be store in 'res_data' that points the address of the mBuf.

After the implementation, the function is tested in the kernel.c.

```
unsigned int *physize = 0;
mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_SETPHYWH, &physize, 8, 8, 1024, 768);
mbox_call(ADDR(mBuf), MBOX_CH_PROP);
uart_puts("\nGot Actual Physical Width: ");
uart_dec(physize[0]);
uart_puts("\nGot Actual Physical Height: ");
uart_dec(physize[1]);
```

Figure 60: setup for setting physical width/height

```
Got Actual Physical Width: 1024
Got Actual Physical Height: 768
```

Figure 61: Result of physical width and height

Before using the 'mbox_buffer_setup()', it should initialize the uart first and it should be inside of the main(). For 'physize', bot response length and request length are 8 and it needs 2 request values (1024, 768). After mbox_call, it will get data from GPU to CPU so that the response data can be found in the physize[0] and physize[1]. The value will be shown as Figure 61

```
unsigned int *clock_rate = 0;
mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_GETCLOCKRATE, &clock_rate, 8, 4, 3);
mbox_call(ADDR(mBuf), MBOX_CH_PROP);
uart_puts("\nGot ARM clock rate: ");
uart_dec(clock_rate[1]);
```

Figure 62: setup for getting ARM clock rate

Figure 63: Result of ARM clock rate

For checking that the function works with the different TAG, clock rate is tested. It only has 1 request value which is clock id and the rate is stored in the second response so that the rate is stored in clock_rate[1].

The limitation of the implemented code is that it only works with the 5 TAGs. If the request length is 12, then the code should add 1 more case for setting up the TAG.

III. CONCLUSION

In conclusion, this project shows the CLI, self-made printf, and function for mailbox setup. CLI works with some commands, auto-completion, delete, and command history. 'printf' has some specifier functions by using variable arguments handling functions and the mailbox function works well to easily set up the mailbox.

By making the Bare OS, it shows how the OS is made such as how the command prompt or terminal is worked. Besides, by making 'printf' by myself, it gives details about how 'printf' works in C programming by comparing with self-made 'printf' and helpful to use specifier in C programming. Setup mailbox gives me how the mailbox works and learned what is the response value by setting different TAGs.

IV. REFERENCES

[1] TylerMSFT, “Va_arg, va_copy, va_end, va_start,” Microsoft Learn, https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/va-arg-va-copy-va-end-va-start?view=msvc-160 (accessed Aug. 29, 2023).

[2] “Convert text to ASCII ART,” Online Tools, https://onlinetools.com/ascii/convert-text-to-ascii-art (accessed Aug. 28, 2023).

[3] “C - pointer to pointer,” Tutorialspoint, https://www.tutorialspoint.com/cprogramming/c_pointer_to_pointer.htm (accessed Aug. 27, 2023).