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: Mai Chieu Thuy
Student ID: s3877746
Date: Aug 25th, 2023

Table of Contents

Introduction

The pursuit of innovation has no limitations in operating system development. This study goes into creating extra features for a bare-metal operating system, showing the efforts made to improve its functionality. As the demand for more sophisticated and versatile systems rises, the incorporation of new features inside a bare-metal environment offers exciting potential to expand its capabilities. This paper provides insight into the motivations motivating the discovery of new paths, the hurdles faced along the road, and the results obtained in the journey of coding more functionality for a bare-metal OS. We hope to discover the possibility of changing a minimalist operating system into a powerhouse with increased features by going on this route.

Additional Features for Bare Metal OS

1. Background

The Command Line Interpreter (CLI) has been seen as a necessary tool for interfacing with computer systems via textual instructions a long time ago. It provides a user-friendly interface that allows users to communicate directly with an operating system or program, making operations like file management, system configuration, and software execution easier. Users may submit commands and receive results quickly and efficiently using the CLI, making it an important component of traditional and modern computing environments.

ANSI escape codes, on the other hand, have played a critical role in improving the visual representation of text-based interfaces. By adding these codes to command line outputs, developers can modify text colors, cursor positions, and formatting, improving the user experience and enabling information comprehension.

Furthermore, variable argument handling using functions such as va_arg, va_copy, va_end, and va_start adds flexibility to function parameters. These routines allow developers to work with a variable number of parameters, which is useful in cases where the number of inputs varies from one function call to the next. This technique is handy when designing functions that must support various data kinds or numbers without the need to define distinct functions for each situation.

2. Implementation

a. Welcome message and Command Line Interpreter (CLI)

I. Requirement:

- When our Bare Metal OS opens, it's critical to greet users with a greeting that sets the tone for engagement. We accomplish this by creating an engaging text-based welcome message using ASCII graphics. The welcome message is not only a greeting, but it also improves the visual appeal of the operating system environment. The created ASCII art can be shown on the console, allowing for a more personalized and immersive user experience.

- The Command Line Interpreter (CLI) is an essential component of our Bare Metal OS, allowing users to communicate with the system using text-based commands. We employ the following features to improve the user experience and streamline command execution:
  - Displaying the OS name (e.g., MyOS) provides a consistent reference for users.
  - Auto-completion is implemented using the TAB key, which assists users by suggesting potential command completions as they type.
  - Command history navigation is facilitated using special keys (underscore for UP arrow and plus sign for DOWN arrow) to scroll through previously entered commands.
  - Commands such as "help," "clear," "setcolor," and "showinfo" are implemented to provide users with relevant information and control over the OS environment.

II. Implement Description:

- Welcome View:
  - Create an ASCII art depiction of the desired welcome message using an online ASCII art-generating tool.[1]
  - Convert the ASCII art string to a format that the programming language (C) understands.
  - Incorporate the ASCII art string into your Bare Metal OS code.
  - Using the uart_puts function to print the ASCII art string to the console displaying the welcome view.
  - Make a loop that reads user input from the console continually.
  - As the first input prompt, show the name of the operating system (for instance, "MyOS").

```
(base) maichieuthuy@ca-na MaiThuy_Assessment2 % make
rm -rf *.img .\build\*.elf .\build\*.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -DUART -c src/kernel.c -o build/ker
nel.o
aarch64-none-elf-ld -nostdlib  ./build/boot.o ./build/function.o ./build/kernel.o ./build/mbox.o ./bu
ild/printf.o ./build/uart.o  -T ./src/link.ld -o ./build/kernel8.elf
aarch64-none-elf-objcopy -O binary ./build/kernel8.elf kernel8.img
qemu-system-aarch64 -M raspi3b -kernel kernel8.img -serial stdio

d88888b d88888b d88888b d888888b .d888b.   j88D  .d888b.  .d88b.
88'     88'     88'     `~~88~~' VP  `8D  j8~88  88' `8D .8P  88.
88ooooo 88ooooo 88ooooo    88       odD' j8' 88  `V8o88' 88  d'88
88~~~~~ 88~~~~~ 88~~~~~    88     .88'   V88888D    d8'  88 d' 88
88.     88.     88.        88    j88.        88    d8'   `88  d8'
Y88888P Y88888P Y88888P    YP    888888D     VP   d8'     `Y88P'

d8888b.  .d8b.  d8888b. d88888b    .d88b.  .d8888.
88  `8D d8' `8b 88  `8D 88'       .8P  Y8. 88'  YP
88oooY' 88ooo88 88oobY' 88ooooo   88    88 `8bo.
88~~~b. 88~~~88 88`8b   88~~~~~   88    88   `Y8b.
88   8D 88   88 88 `88. 88.       `8b  d8' db   8D
Y8888P' YP   YP 88   YD Y88888P    `Y88P'  `8888Y'
        Developed by Thuy - s3877746
```

Figure 1: Welcome View

- Auto Completion:
  - Apply logic if - else and check each character matching with command syntax to deal with auto-completion when the TAB key is used:[2]

- Compare the user's input with the names of accessible commands and offer completions.

```
//handle tab for auto completion
else if(c == '\t') {
    cli_buffer[index] = '\0';

    //case help
    if((compare_cli(cli_buffer, "h") == 0) ||
    (compare_cli(cli_buffer, "he") == 0) ||
    (compare_cli(cli_buffer, "hel") == 0)) {
        copynString(cli_buffer, "help", INPUT_MAX_SIZE - 1);
        clear_cli(20);
        index = count_length("help");
        uart_puts("\n");
        uart_puts("thuyiswater> ");
        uart_puts(cli_buffer);
    }
```

Figure 2: Logic if else to handle tab auto completion

- History Command:
  - Navigating Command History:
    - If the user presses the up (_) or down (+) arrow key, the code performs the following steps:
      1. Terminates the current command in the cli_buffer by adding a null terminator at the current index and resetting the index to zero.
      2. Updates the history_index based on whether the user pressed the up or down arrow key. Ensures that the index remains within the valid history range.
      3. If the updated history_index is within valid bounds ([0, arr_index-1]), the code does the following:
      4. Copies the command from the history array at the history_index to the cli_buffer.
      5. Clears the CLI screen and prints the updated command.
      6. Updates the index to reflect the length of the copied command.

  - Handling Enter Key (Storing Commands):
    - When the user hits the "Enter" key, the code:[2]
      1. Adds the current command in the cli_buffer to the command history array using the copynString function, which respects a maximum size limit.

2. Increments arr_index and ensures it doesn't exceed the maximum history limit.
3. Sets the history_index to the last command entered (latest index).

- copynString() Function:
  - The copynString function safely copies strings while considering the specified maxSize limit.
  - It returns a pointer [9] to the beginning of the destination string (origin).
  - The function iterates through the source string, copying characters to the destination until the size limit is reached or the source is exhausted.
  - Any remaining space in the destination is filled with null terminators to ensure proper termination.

```
char* copynString(char *destination, const char *source, int maxSize) {
    char *origin = destination;

    while(maxSize > 0 && *source != '\0') {
        *destination++ = *source++;
        maxSize--;
    }

    while(maxSize > 0) {
        *destination++ = '\0';
        maxSize--;
    }
    return origin;
}
```

Figure 3: copynString function

- compare_cli() function: this function applies to all case in help menu
  - It initializes a flag to 0, which will remain 0 if the syntax is correct.
  - The function iterates through both str1 (user input) and str2 (reference syntax) simultaneously.
  - If characters match, it continues iterating; if there's a mismatch or one of the strings ends prematurely, the flag is set to 1.
  - After comparison, it resets the str1 pointer [9] to its initial position (necessary for reuse in case of future processing).
  - The function returns the flag, where 0 indicates correct syntax, and 1 indicates incorrect syntax.

```
//compare input from user if it in correct syntax
int compare_cli(char* str1, char* str2) {
    int flag = 0;

    while (*str1 != '\0' || *str2 != '\0') {
        if (*str1 == *str2) {
            str1++;
            str2++;
        } else if ((*str1 == '\0' && *str2 != '\0') || (*str1 != '\0' && *str2 == '\0') || *str1 != *str2) {
            flag = 1;
            break;
        }
    }
    reset_str(str1);  //reset the pointer here
    return flag;
}
```

Figure 4: compare_cli() function

- Help Command:
  - The execute_command function displays a list of available commands along with their usage information.
  - The displayed information helps users understand the available commands and their correct usage.

- Full Information Command:
  - Checking for Help Command Syntax (check_help):
    - check_help() function: assesses whether the user-inputted command matches any predefined command syntax for requesting help with full information.
    - Takes the user's input (ar) as a parameter.

```
//check help command to show full info
int check_help(char* ar) {
    int n = -1;
    char *list[] = {"help help",
    "help hwinfo",
    "help clear",
    "help setcolor",
    "help showinfo"};

    for(int i = 0; i < 4; i++) {
        if(compare_cli(ar, list[i]) == 0) {
            n = i;
            break;
        }
    }
    return n;
}
```

Figure 5: check_help() function

- Displaying Full Command Information (help_info):
  - help_info() function: presents the complete information for a specific command based on the provided help_type.
  - Takes an integer help_type as a parameter, which indicates the index of the command's information within the predefined list.
  - The function uses a series of if statements to determine the specific command indicated by help_type and then displays its full information.
  - For each iteration in the loop, it calls the compare_cli function to check if the user's input matches the current predefined syntax.
  - If a match is found (i.e., compare_cli() returns 0), the loop breaks, and the index of the matched syntax in the list is returned. If no match is found, -1 is returned.

```
//full info of each command
void help_info(int help_type) {
    if(help_type == 0) {
        uart_puts("\nCommand: help\n");
        uart_puts("Show brief information of all commands\n");
        uart_puts("Example: thuyiswater> help\n");
    }
    if(help_type == 1) {
        uart_puts("\nCommand: help <command_name>\n");
        uart_puts("Show full information of the command\n");
        uart_puts("Example: thuyiswater> help hwinfo\n");
    }
```

Figure 6: help_info() function prints out full information of each command

- Clear Command:
  - With function compare_cli() check input from user if correct then uart_puts will be used with "\033[2J" and "\033[H" to clear screen and scroll to current position

- Showinfo Command:
  - board_revision() function[4]:
    1. Store the relevant information in indices 0 to 7 of mBuf, with index 5 reserved for the board revision value.
    2. Call the mbox_call() function with the appropriate parameters to communicate with the mailbox channel responsible for board properties.
    3. If the call to mbox_call() is successful:
    4. Extract the board revision value from mBuf[5].
    5. Display the board revision value using the uart_hex() function, preceded by the text "Board revision: "

```
void board_revision() {
    mBuf[0] = 7 * 4;
    mBuf[1] = MBOX_REQUEST;
    mBuf[2] = 0x00010002;
    mBuf[3] = 4;
    mBuf[5] = 0;
    mBuf[6] = 0;
    mBuf[7] = MBOX_TAG_LAST;

    if(mbox_call(ADDR(mBuf), MBOX_CH_PROP)) {
        uart_puts("\nBoard revision: ");
        uart_hex(mBuf[5]);
        uart_puts("\n");
    } else {
        uart_puts("INVALID\n");
    }
}
```

Figure 7: board_revision() function

- board_mac_address() function[5]:
  1. Initialize an array called mBuf to store the data.
  2. Set specific values in mBuf to form a mailbox request for the board's MAC address.
  3. Store the MAC address information in mBuf[5].
  4. Call the mbox_call() function with the appropriate parameters to communicate with the mailbox channel responsible for board properties.
  5. If the call to mbox_call() is successful:
  6. Extract the MAC address value from mBuf[5].
  7. Display the MAC address value using the uart_hex() function, preceded by the text "Board MAC address: ".
  8. If the call to mbox_call() fails, output the message "INVALID" using uart_puts().

```
void board_mac_address() {
    mBuf[0] = 7 * 4;
    mBuf[1] = MBOX_REQUEST;
    mBuf[2] = MBOX_TAG_GETMACADD;
    mBuf[3] = 6;
    mBuf[4] = 0;
    mBuf[5] = 0;
    mBuf[6] = MBOX_TAG_LAST;

    if(mbox_call(ADDR(mBuf), MBOX_CH_PROP)) {
        uart_puts("\nBoard MAC address: ");
        uart_hex(mBuf[5]);
    } else {
        uart_puts("INVALID\n");
    }
}
```

Figure 8: board_mac_address function

- Setcolor Command:
  - compare_input_color() function: compares two strings, str1 and str2, character by character. It increases the clAr counter for each matching character and returns a flag indicating if the strings match or not. If they don't match, the function resets str1 and sets clAr to 0.

```
//compare input color against the expected color string
//and track the number of matching characters
int compare_input_color(char *str1, char *str2, int *clAr) {
    int flag = 0;
    //if match, increase the clAR counter
    while(*str2 != '\0') {
        if(*str1 == *str2) {
            *clAr = *clAr + 1;
            str1++; str2++;
        }
        //if not match, reset the pointer
        else {
            flag = 2;
            reset_str(str1);
            *clAr = 0;
            break;
        }
    }
    return flag;
}
```

Figure 9: compare_input_color() function

  - compare_color() function[3]: also compares two strings, str1 and str2, character by character. It returns a flag indicating if the strings match or not. If they don't match, the function resets str1 to its original position before the mismatch occurred.

```c
//compare two color strings
//returning a flag indicating if they match or not
int compare_color(char* str1, char* str2) {
    int i = 0;
    int flag = 0;
    while (*str2 != '\0') {
        if (*str1 == *str2) {
            i++; str1++; str2++;
        } else {
            flag = 1;
            str1 = str1 - i;
            break;
        }
    }
    return flag;
}
```

Figure 10: compare_color() function

- get_colorset() function[3]: extracts the color set index from the input string str1. It skips the first character, checks for the end of input, and then iterates through color options to determine the corresponding color index.

```c
//get the color set based on the input string
int get_colorset(char* str1) {
    str1++;
    int flag = 0;

    if(*str1 == '\n') {
        flag = -1; //indicates end of input
        return flag;
    }

    if (*str1 == ' '){
        str1++;
        int color = -1;

        //iterate through color options and determine the color index
        for(int i = 0; i <= 8; i++) {
            if (compare_color(str1,"black") == 0){
                color = 0;
            } else if(compare_color(str1, "red") == 0) {
                color = 1;
            } else if (compare_color(str1, "green") == 0) {
                color = 2;
            } else if (compare_color(str1, "yellow") == 0) {
                color = 3;
            } else if (compare_color(str1, "blue") == 0) {
                color = 4;
            } else if (compare_color(str1, "purple") == 0) {
                color = 5;
            } else if(compare_color(str1, "cyan") == 0) {
                color = 6;
            } else if(compare_color(str1, "white") == 0) {
                color = 7;
            }
        }
        return color; //return the color index
    } else {
        flag = -1; //invalid input
        return flag;
    }
}
```

Figure 11: get_colorset() function

- set_color() function[3]: sets the color according to the input string str. It iterates through the string, handles the background color if the character is 'b', and handles the text color if the character is 't'. It calls the get_colorset() function to retrieve the color index, and based on that, updates the flag variable accordingly.

```
//set color according to the input
int set_color (char* str) {
    int flag = -1;

    while(*str != '\0') {
        if(*str == '-') {
            str++;
            //handle background color
            if(*str == 'b') {
                int color = get_colorset(str);
                if(color < 0) {
                    return -1;
                } else {
                    if(flag == -1) {
                        flag = 8;
                        flag = flag * 10 + color;
                    } else if (flag % 10 == 8) {
                        flag = flag - 8;
                        flag = flag + color;
                    }
                }
            }
            //handle text color
            else if (*str == 't') {
                int color = get_colorset(str);
                if(color < 0) {
                    return -1;
                } else {
                    if(flag == -1) {
                        flag = 8;
                        flag = color * 10 + flag;
                    } else if (flag / 10 == 8) {
                        flag = flag - 80;
                        flag =  color * 10 + flag;
                    }
                }
            } else {
                flag = -1;
                break;
            }
        }
        str++;
    }
    return flag;
}
```

Figure 12: set_color() function

- color_arr() function[2]: takes a color_index as input and sets the appropriate color for both the text and background. It uses arrays of strings text and background, where each index corresponds to a specific color. It extracts the text color and background color indices from color_index and outputs the corresponding ANSI escape codes using uart_puts().

```
void color_arr(int color_index) {
    char text[8][10] = {"\x1b[30m",
                        "\x1b[31m",
                        "\x1b[32m",
                        "\x1b[33m",
                        "\x1b[34m",
                        "\x1b[35m",
                        "\x1b[36m",
                        "\x1b[37m"};

    char background[8][10] = {"\x1b[40m",
                              "\x1b[41m",
                              "\x1b[42m",
                              "\x1b[43m",
                              "\x1b[44m",
                              "\x1b[45m",
                              "\x1b[46m",
                              "\x1b[47m"};

    int text_index = color_index / 10;
    if(text_index != 8) {
        uart_puts(text[text_index]);
    }

    int background_index = color_index % 10;
    if(background_index != 8) {
        uart_puts(background[background_index]);
    }
}
```

Figure 13: color_arr() function

III. Result Discussion: the addition of the welcome message to our Bare Metal OS adds an attractive and engaging touch. The OS efficiently communicates its identity and purpose to users using bespoke ASCII graphics, boosting the entire user experience. This strategy generates a distinct brand identity for our operating system and sets a pleasant tone for user interaction.

b. Printf Function

I. Requirement: support the following format specifiers: %d %c %s %f %x %%, 0 flag, width, and precision

II. Implementation Description

- Handling %d Specifier[6]:
  - Use va_arg [7][8] to get the integer argument from the variable argument list.
  - Uses the is_negative flag to handle negative values before converting them to positive.
  - Repeatedly divides the number by 10 to transform it into a string representation.
  - Depending on the precision, prepends the leading zeros.
  - Handles width by padding the left with zeros or spaces.
  - Copies the number's format from the buff_container to the main buffer.

- Handling %x Specifier[6]:
  - Use va_arg [7][8] to get the integer argument from the variable argument list.
  - Iteratively divides the integer by 16 to get a hexadecimal representation.
  - Depending on the precision, prepends the leading zeros.
  - By including padding spaces to the left, width is handled.
  - Copies the hexadecimal number in its format from the buff_container to the main buffer.

- Handling %c Specifier[6]:
  - Use va_arg [7][8] to pull the character argument from the list of variable arguments.
  - Adds the character as a formatting append to the buff_container.
  - Handles width by, if necessary, adding padding spaces to the left.
  - Copies the character in its formatted form from the buff_container to the main buffer.

- Handling %s Specifier[6]:
  - Use va_arg [7][8] to pull the string argument from the list of variable arguments.
  - Up to the precision or string end, copies the string's characters.
  - Handles width by, if necessary, adding padding spaces to the right.
  - Copies the characters that have been prepared from the buff_container to the main buffer.

- Handling %f Specifier[6]:
  - Use va_arg [7][8] to extract the double argument from the variable argument list.
  - Uses the is_negative flag to handle negative values before converting them to positive.
  - Depending on whether a precision is present, handles two formatting situations:
    1) Without specificity: formatted similarly to %d after converting the double to an integer (x_int).
    2) Separates the double's integer and fractional components, if precision is supplied.
       ➔ Converts the fractional portion to a formattable integer (fl_int).

➔ Depending on the precision, prepends the leading zeros.

- Handles width by padding the left with zeros or spaces.
- Hopies the number's format from the buff_container to the main buffer.

- Handling %% Specifier (Literal %)[6]:
  - Checks if the next character is also a %.
  - If yes, advances the string pointer [9] and adds a single % character to the buffer.

```
thuyiswater> printlist

6 is higher than 4
 Hello 16

Character: d

Hex number: 399
Hex number: 200
Hex number: 0

Float number: 87857.398476
Float number:     35.800
Float number:    -35.800

String: thuyiswater

Hex number: 504

Using %f: 987.654321
```

Figure 14: Result for every printf() case

```c
void print_list() {
    printf("\n");
        //print integer
        printf("%d is higher than %d\n ", 6, 4);
        printf("Hello %0d\n", 16);

        //print character
        printf("\nCharacter: %c\n", 'd');

        //print hex
        printf("\nHex number: %x\n", 921);
        printf("Hex number: %d\n", -200);
        printf("Hex number: %d\n", -24.17);

        //print float
        printf("\nFloat number: %f\n", 87857.3984756);

        //float and flag and precision and width
        printf("Float number: %10.3f\n", 35.8 );
        printf("Float number: %10.3f\n", -35.8 );

        //Print string and flag and width
        printf("\nString: %s\n", "thuyiswater");

        //print hexadecimal and flag and width
        printf ("\nHex number: %x \n", 1284);

        //print %
        printf ("\nUsing %%f: %f\n", 987.654321);
}
```

Figure 15: print_list() function to test printf() function

III. Result Discussion: The implementation of the printf function empowers the Bare Metal OS to output formatted data to the console, mirroring the behavior of the standard printf function in C programming. By supporting a wide range of format specifiers and formatting options, we enhance the flexibility of data representation and improve the overall usability of the OS.

c. Mailbox Setup Function:

I. Requirement: to simplify the mailbox setup process, we introduce the mbox_buffer_setup function. This function streamlines the mailbox configuration by encapsulating the necessary steps and arguments. The function takes parameters such as buffer address, tag identifier, response data pointers [9], and request values to set up the mailbox for specific purposes.

II. Implementation Description

- Set the buffer address, tag identification, response data pointers [9], and request values as parameters for the mbox_buffer_setup function.
- Set up the index of mBuf from 0 to 2 [5] because all cases are the same except index 2 of the ARM Clock frequency case. This will avoid the situation where mBuf is overwritten.
- Use the switch case algorithm to set up the remaining indexes of mBuf [5], print the results based on each tag_identifier that the user enters in the kernel file.
- Update the response data pointers [9] that were provided as arguments after processing the response data that was acquired from the mailbox.

```
Buffer Address: 0x00083F30
Got successful response
Board Model:  0x00000000

Buffer Address: 0x00083F30
Got successful response
Board Serial:  0x00000000

Buffer Address: 0x00083F30
Got successful response
Board MAC Address: 0x12005452

Buffer Address: 0x00083F30
Got successful response
ARM frequency clock rate: 700000000


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

Figure 16: Result of mbox_buffer_setup() function for 5 different tags

```
unsigned int *physize = 0; // Pointer to response data
unsigned int *get_model = 0;
unsigned int *get_serial = 0;
unsigned int *mac_addr = 0;
unsigned int *clock_rate = 0;

mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_GETMODEL, &get_model, 4, 0, 0);

mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_GETSERIAL, &get_serial, 8, 0, 0);

mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_GETMACADD, &mac_addr, 6, 0, 0);

mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_SETCLKRATE, &clock_rate, 6, 0, 0);

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

Figure 17: Syntax to call the mbox_buffer_setup() in kernel file

III. Result Discussion: the mbox_buffer_setup() function simplifies the process of configuring the mailbox for various tasks in our Bare Metal OS. By encapsulating the necessary configuration steps, developers can utilize this function to efficiently set up mailboxes and manage data exchange with peripheral devices.

Conclusion

This project is focused on the bare metal development of an operating system, specifically targeting embedded systems. The objective is to enhance practical skills by implementing additional features such as a command line interpreter (CLI), ANSI code for terminal formatting, standard printf function, and variable arguments handling.

Overall, this project aims to strengthen development skills and provide practical experience with embedded operating systems through the implementation of essential features like CLI, printf function, and mailbox setup.

Reference

[1] “Convert Text to ASCII Art,” onlinetools.com. https://onlinetools.com/ascii/convert-text-to-ascii-art (accessed Aug. 18, 2023).

[2] fnky. “ANSI Escape Codes,” Gist. https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797 (accessed Aug.19, 2023).

[3] C. Yeh. “Terminal Colors”. Chris Yeh. https://chrisyeh96.github.io/2020/03/28/terminal-colors.html (accessed Aug.19, 2023).

[4] Mat. “Checking Raspberry Pi Revision Number & Board Version”. RasBerryPi. https://www.raspberrypi-spy.co.uk/2012/09/checking-your-raspberry-pi-board-version/ (accessed Aug.21, 2023).

[5] L. Tran. “Mailbox property interface - Message TAGS.pdf”. Instructure. https://rmit.instructure.com/courses/121602/pages/w6-in-class-materials?module_item_id=5220599 (accessed Aug. 24, 2023).

[6] cpluscplus. “printf”. cpluscplus.com. https://cplusplus.com/reference/cstdio/printf/ (accessed Aug. 26, 2023).

[7] cpluscplus. “printf”. cpluscplus.com. https://cplusplus.com/reference/cstdarg/ (accessed Aug. 26, 2023).

[8] Microsoft. “va_arg, va_copy, va_end, va_start”. Microsoft. https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/va-arg-va-copy-va-end-va-start?view=msvc-160 (accessed Aug. 26, 2023).

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