Shape

Description automatically generated with medium confidence

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](mailto:linh.tranduc@rmit.edu.vn),

Mr. Phuc Nguyen - [phuc.nguyenhoangthien@rmit.edu.vn](mailto:phuc.nguyenhoangthien@rmit.edu.vn)

Student name : Park Anh Kiet

Student ID : s3681475

Date : 31/8/2023

**TABLE OF CONTENTS**

Contents

[I. INTRODUCTION 1](#_Toc144416512)

[II. ADDITIONAL FEATURES FOR BARE METAL OS 1](#_Toc144416513)

[**1.** **BACKGROUND** 1](#_Toc144416514)

[**2. IMPLEMENTATION** 1](#_Toc144416515)

[III. CONCLUSION 11](#_Toc144416516)

[IV. REFERENCES (USE IEEE STYLES) 11](#_Toc144416517)

# I. INTRODUCTION

The "EEET2490 - Embedded System: OS and Interfacing " provides an opportunity for students to get exposure to the Bare Metal operating system. To enhance skills in implementing important features such as CLI, standard printf functionality to handle digital transformation. This exercise encourages students to review basic concepts such as ANSI Regression Codes, color codes, and develop an understanding of terminal controls. In addition, implementing the Raspberry Pi mailbox setup functions helps students gain more experience in practicing, designing and developing operating systems.

# II. ADDITIONAL FEATURES FOR BARE METAL OS

## **BACKGROUND**

1. Command Line Interpreter (CLI):

This is the interface that allows the user to interact with the system by entering commands. It allows users to control the system and perform tasks without the need for a user interface. CLI is a fundamental component of operating systems such as Linux and Windows, where users can enter commands and wait for responses and results.

1. ANSI Escape Codes and Color Codes:

This is a sequence of characters used to control the color and formatting of text in the terminal. These codes provide a way to change text color, background color, cursor position, and other properties. By using ANSI code, developers can improve the visual apprearance of the terminal to make it easier to see and use.

1. Variable Arguments Handling Functions:

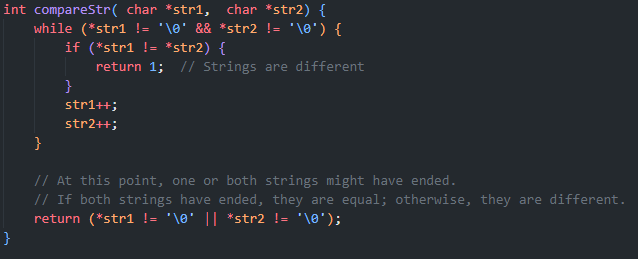
These are functions that handle variable numbers. They are used in the C and C++ programming languages to manage the amount of numeric changes in functions. Functions provide access to variables added to the function to implement other functions. In it, the number and type of arguments can be different. va\_arg retrieves each argument while va\_copy, va\_end, va\_start manage that argument list

## **2. IMPLEMENTATION**

1. **Welcome message and Command Line Interpreter (CLI)**
   * **Requirement**

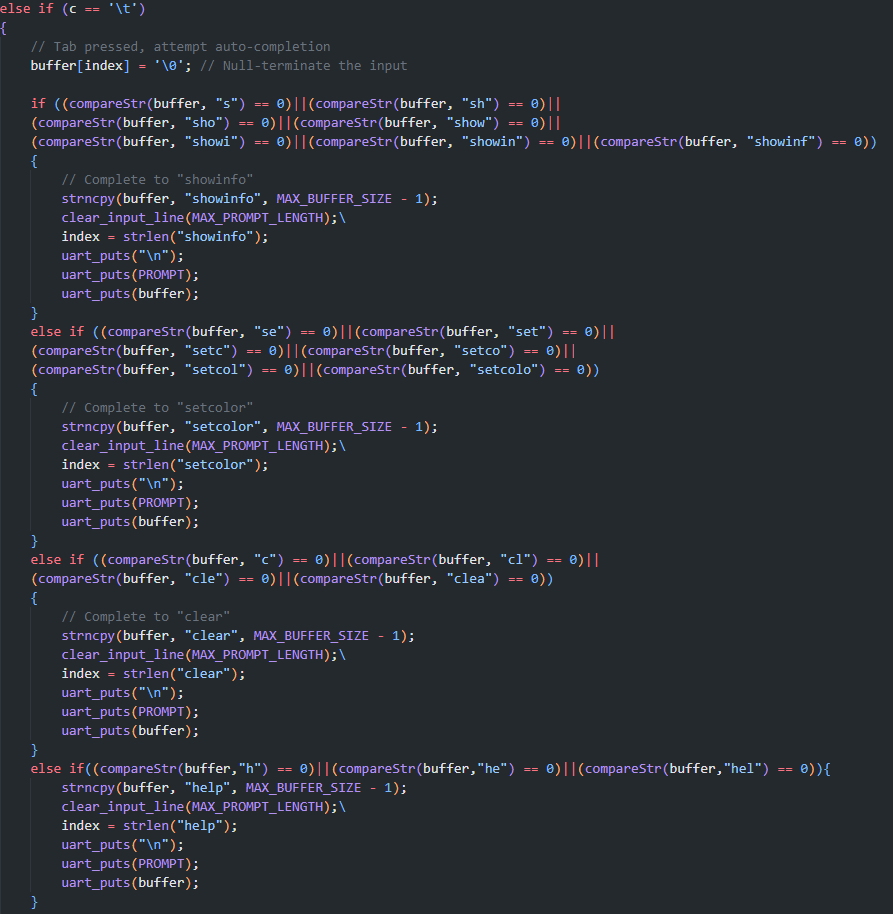
Display a welcome message and a CLI that reads keypresses into a buffer and executes the command when the key “Enter” is pressed. The CLI also have these Feature

* An OS Name (e.g. MyOS) as an initial text that always display in the console while waiting for user to type in command
* Auto-completion: User use TAB for auto-complete of the command
* Command history: to browse within the command history.
  + **Implementation**
    - Welcome Text
    - Compare String

The ‘compareStr’ function is crucial for determining the commands entered by the user in the CLI. It is used to check if the input buffer (‘buffer’) against a command or a partial command such as ‘help’, ‘clear’, ‘setcolor’, ‘showinfo’. The comparision allow the program to execute the function corresponds to the command. The function is also used for auto-completion of command names when the user press the ’Tab’ key. Overall, the ‘compareStr’ function plays a crucial role in the kernel.c by enabling command recognition, execution, and auto-completion. Contributing significantly to the functionality and user interaction of the terminal.  


* + - Auto completion

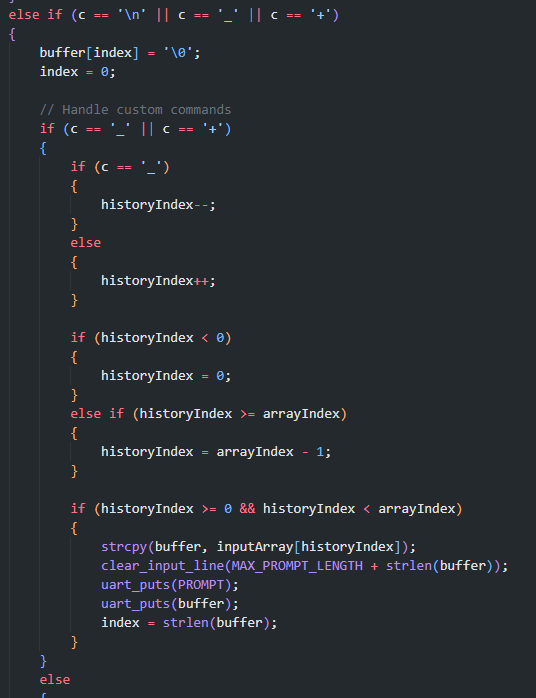
In this implementation, the entered buffer is compared against multiple partial variations of the command. Each variation represents a step towards completing the full command.

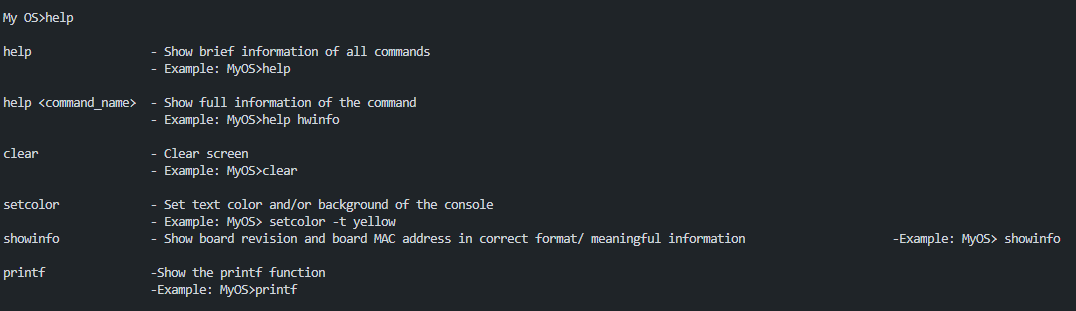
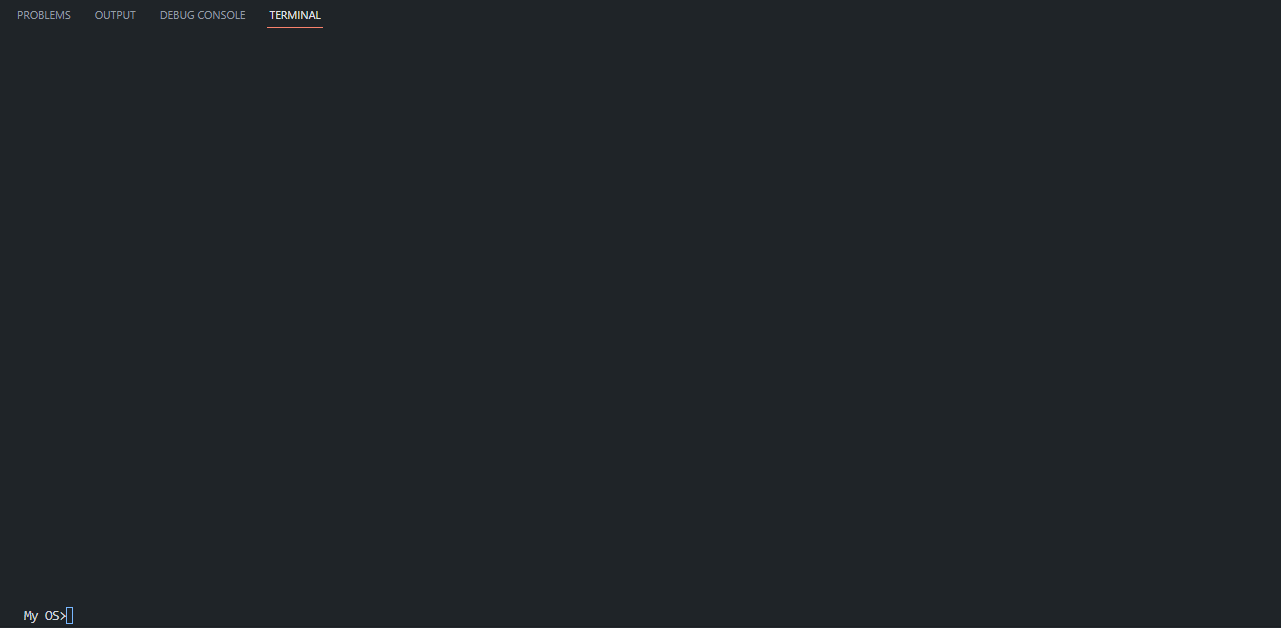
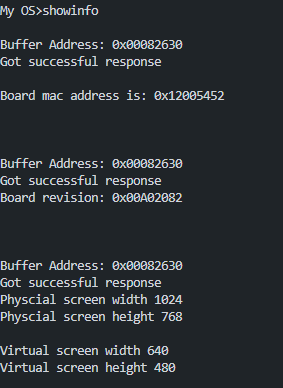
If a partial command match is found, the CLI will complete the command by replacing the partial input with the full command. After completing the command in the buffer, the CLI clears the current input line to make space for the completed command. Then it updates the console display with the new buffer content.  


* + - Command history

When the user presses ‘Enter’ or custom navigation keys ‘\_’ and ‘+’, the input buffer is processed. The current input in the buffer will be null and the index is reset to 0. If the ‘\_’ is pressed, the history index is decremented to nagivate back through the previous entered command. If ‘+’ is pressed, the history index is incremented to navigate forward. The history index is bounded to prevent going out of the valid history range. If the navigation commands are used and the history index is within the valid range, the corresponding command from the input history is coppied to the buffer. The input line is cleared and updated to display the restored command.

If the user has entered a complete command, it will be compare to the known commands and if it matches a command (e.g. ‘help’, ‘clear’, ‘showinfo’, …, the correspoding action is executed. If the user enter an unknown command, the CLI will inform the user that the command is unknown.

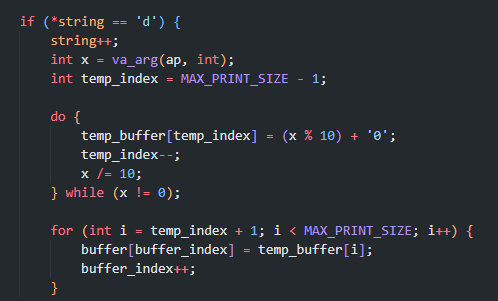

* + Command processing  
    Help command: Show brief information of all commands  
      
    Clear command: Clear screen  
      
    setcolor: Set text color, and/or background color of the console  
    showinfo: Show board revision, and board MAC address   
    
  + **Result discussion**

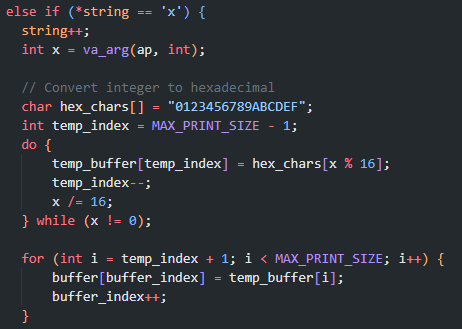
The three aspect of the implementation above create a functional and user-friendly command line interpreter. The user can input commands, navigate through the command history and interact with the CLI’s features. While the welcome message helps the user to realize the name of the OS, the user can use ‘help’ to know all of the commands available to use and help the user to know it’s usage and how to use it, ‘setcolor’ makes the CLI more colorful, allows users change the color and the text according to the user’s will, the ‘clear’ helps user clear out the terminal for easy tracking of where or what the user is doing, ‘showinfo’ helps the user know the specifications of the hardware the users are using.

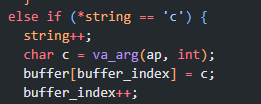
1. **printf function**

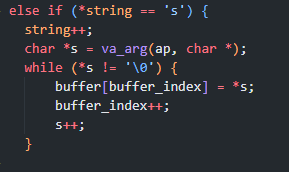
The printf function implemented here provides a simplified version of the standard C library printf function. It handles basic formatting directives (%d, %x, %c, %s) and constructs a formatted output in the buffer, which is then printed using the uart\_puts function. The implementation lacks certain features and optimizations present in the standard printf function, but it serves as a functional example of how a basic printf-like formatting can be achieved.

The code checks for the presence of % characters to identify formatting directives.

%d: For integer formatting, the code uses a division-and-modulus approach to extract digits one by one and convert them to characters.  


%x: For hexadecimal formatting, a similar approach is used, where the integer is repeatedly divided by 16 and the remainder is mapped to the appropriate hexadecimal character.  


%c: For character formatting, the character is directly extracted from the variable arguments and added to the buffer.  


%s: For string formatting, the function iterates over the provided string and adds its characters to the buffer.  


* Result discussion  
  The printf function implemented here provides a simplified version of the standard C library printf function. It handles basic formatting directives (%d, %x, %c, %s) and constructs a formatted output in the buffer, which is then printed using the uart\_puts function. The implementation lacks certain features and optimizations present in the standard printf function, but it serves as a functional example of how a basic printf-like formatting can be achieved.
* Considerations and Improvements:

The code should handle edge cases and error scenarios more gracefully, such as invalid format specifiers. The use of fixed-size buffers may lead to buffer overflow or inefficiency when handling large inputs. Dynamically allocating memory for the buffer might be a better approach.

More format specifiers and formatting options (e.g., precision, field width) could be added to enhance the function's versatility. Some comments explaining the code's functionality and different parts of the code would improve its readability and maintainability.

1. **function for mailbox setup**

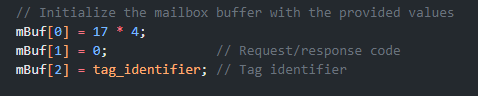
Function Signature:

The function signature indicates that the function takes multiple arguments, including a buffer address, a tag identifier, pointers to result data and its length, request length, and variable arguments using the ellipsis (...) syntax.

Buffer Initialization:

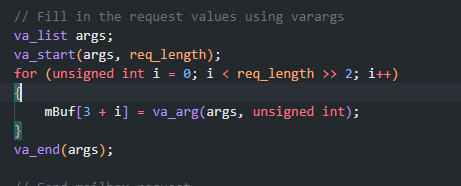
The function starts by initializing a buffer (mBuf) with some initial values. The first two entries in the buffer are of particular importance:

mBuf[0]: The total length of the buffer (17 \* 4 bytes).

mBuf[1]: Request/response code. It is initially set to 0, and later updated to signal a request or response.  


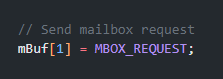
Request Construction:

The code then proceeds to fill in the buffer with the provided tag identifier and request values using variable arguments. The va\_list structure is used to handle variable arguments. The loop iterates over the request values and populates the buffer accordingly.

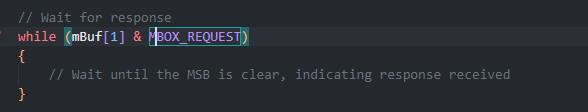


Sending the Request:

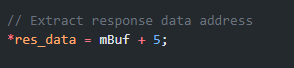
After constructing the request, the request code is set to MBOX\_REQUEST, indicating that a request is being sent.



Waiting for Response:

The code enters a while loop that waits until the most significant bit (MSB) of mBuf[1] is clear. This is a common technique to wait for a response to arrive, as the MSB is often used as a busy flag. 

Response Handling:

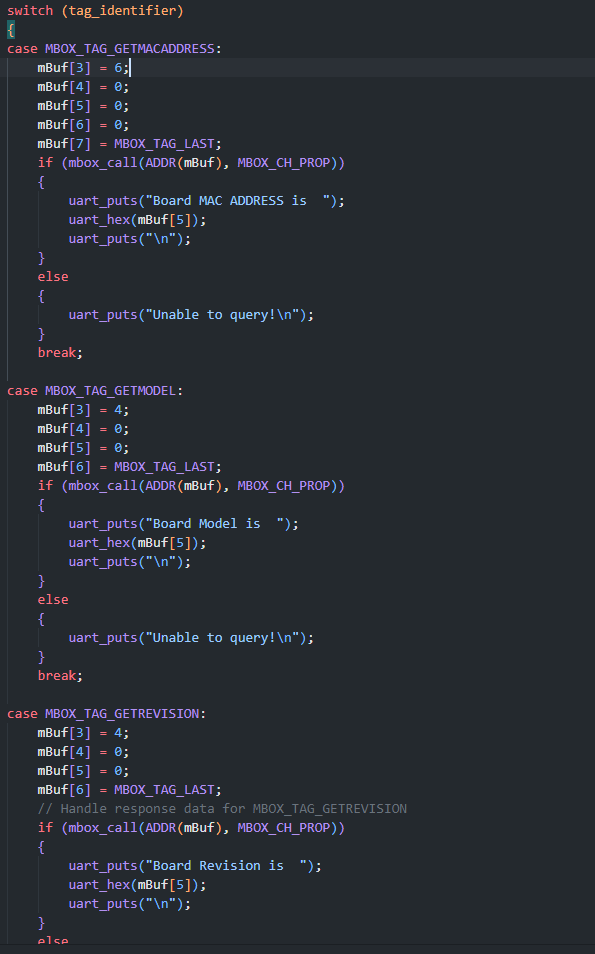
Once the response is received, the code extracts the response data address by setting the pointer res\_data to point to the appropriate location in mBuf. The code then enters a switch statement based on the tag\_identifier to handle different types of responses.  


Switch Cases:

The switch statement handles different cases based on the tag\_identifier. It constructs specific requests for certain tags and sends them using the mbox\_call function. It then prints information related to the response or indicates an error if the response couldn't be obtained.

Variable Arguments and Case Handling:

The function uses variable arguments for certain cases. For example, the MBOX\_TAG\_SETPHYWH case uses variable arguments to set width and height values in the request.





# III. CONCLUSION

In this project, I have developed an operating system-like program that offers a command-line interface for user interaction. The program is designed to provide various functionalities, such as system information retrieval, command history, auto-completion, and interaction with built-in commands. While the project encompasses multiple aspects of programming, including hardware communication, user input handling, and function integration, there is an area that remains incomplete: the implementation of the "setcolor" command.

Individual Reflection:

Throughout the development of this project, I have gained valuable insights into various programming concepts and practices. I successfully implemented key features like the command-line interface, history tracking, and system information retrieval using mailbox communication. However, the unfinished "setcolor" command serves as a reminder of the challenges and complexities that can arise during software development.

Reflecting on this project, I realize the importance of proper planning and time management. The incomplete feature highlights the need for setting realistic goals and prioritizing tasks. Additionally, I've learned the significance of breaking down complex tasks into smaller, manageable steps to ensure steady progress.

Despite the setback with the "setcolor" command, I consider this project a valuable learning experience. It has reinforced the importance of thorough testing, debugging, and seeking help when facing challenges. Furthermore, this project has provided me with hands-on experience in building interactive applications, handling user input, and integrating different components to create a cohesive program.

As I move forward, I plan to revisit the "setcolor" implementation and address the challenges that hindered its completion. This project has encouraged me to continue honing my programming skills and to approach future projects with enhanced planning and perseverance.

Overall, while the "setcolor" command remains unfinished, I am proud of the progress made in developing a functional and interactive program. This project has deepened my understanding of software development and has motivated me to continue exploring and improving my programming abilities.

# IV. REFERENCES (USE IEEE STYLES)

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

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

[3] fnky. “ANSI Escape Codes,” Gist. https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797 (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] “PRINTF,” cplusplus.com, https://cplusplus.com/reference/cstdio/printf/ (accessed Aug. 31, 2023).

[6] 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. 31, 2023).

[7] Va\_arg(), va\_copy(), va\_end(), va\_start() - access function arguments, https://www.ibm.com/docs/en/zos/2.3.0?topic=lf-va-arg-va-copy-va-end-va-start-access-function-arguments (accessed Aug. 31, 2023).

[8] “C - pointer to pointer,” Tutorialspoint, https://www.tutorialspoint.com/cprogramming/c\_pointer\_to\_pointer.htm (accessed Aug. 31, 2023).