Shape

Description automatically generated with medium confidence

EEET2490 – Embedded System: OS and Interfacing, Semester 2025-1

Group Assignment Report

**CLI, SCREEN DISPLAY, AND APPLICATION DEVELOPMENT FOR A BARE METAL OPERATING SYSTEM**

**Lecturer**: Dr Linh Tran – [linh.tranduc@rmit.edu.vn](mailto:linh.tranduc@rmit.edu.vn),

**Team Number: 6**

**Team Members:**

Huynh Ngoc Tai (s3978680)

Tran Quang Minh (s3988776)

Kim Nhat Anh (s3978831)

Vu Thien Minh Hao (s3988776)

**Date** : 26/05/2025

**TABLE OF CONTENT**

[I. INTRODUCTION 1](#_Toc199145491)

[II. WELCOME MESSAGE AND COMMAND LINE INTERPRETER (CLI) 1](#_Toc199145492)

[III. IMAGE, VIDEO, AND TEXT DISPLAY 2](#_Toc199145493)

[IV. APPLICATION DEVELOPMENT 2](#_Toc199145494)

[V. CONCLUSION 3](#_Toc199145495)

[VI. REFERENCES (USE IEEE STYLES) 3](#_Toc199145496)

# I. INTRODUCTION

This report documents the development of a bare-metal operating system running on Raspberry Pi. The assignment focuses on three major tasks: implementation of a welcome message and command line interface (CLI), display of images/videos/text on screen, and the development of an embedded game application. Our objective was to demonstrate understanding of OS-level features, low-level interfacing, framebuffer manipulation, and user interaction design.

# Engineering, particularly in the realm of embedded systems, is a discipline that blends theoretical knowledge with practical implementation to solve complex real-world challenges. Within this context, understanding how software interfaces directly with hardware is crucial. A strong grasp of operating systems, memory-mapped I/O, and low-level driver development empowers engineers to build responsive, efficient, and lightweight applications on resource-constrained platforms.

# This project expands upon core embedded systems concepts by exploring the design and implementation of a bare-metal operating system on the Raspberry Pi 3. Unlike traditional software environments that rely on OS-level abstractions, our system directly manipulates hardware registers to control UART communication, framebuffer display, and command-line processing. The absence of an operating system challenges students to understand and manage hardware initialization, interrupt handling, memory layout, and execution flow at a low level.

# The primary objective of this assignment is to design and implement a minimal OS that supports a custom Command-Line Interface (CLI), text/image/video rendering, and a fully interactive 2D game. Students are tasked with managing screen output via the Raspberry Pi’s framebuffer, implementing custom font rendering, and supporting multimedia interaction through UART.

# Our team designed and developed a game titled “Basketball Star”, a fast-paced 2D game in which players control a hoop to catch falling basketballs and avoid bombs. The gameplay incorporates stage progression, dynamic object spawning, and real-time feedback through UART logs. The game is built entirely using C and interacts with the Raspberry Pi’s graphics subsystem at the register level.

# This report provides a comprehensive overview of our implementation. It details each module's design, underlying logic, and the rationale behind key architectural decisions. Included are usage instructions, code explanations, testing outcomes, and result screenshots.

# II. WELCOME MESSAGE AND COMMAND LINE INTERPRETER (CLI)

**Requirement:** Implement a CLI system with features such as welcome message, command parsing, history, and execution for basic system commands like help, clear, showinfo, etc. Features like auto-completion and command history were to be added using special key handling.

**Background**

In any operating system, a Command Line Interpreter is fundamental to user interaction, serving as a bridge between the user and the underlying system functionality. A robust CLI provides efficiency through features like command auto-completion, which reduces typing errors and speeds up user interaction. Command history is equally critical, allowing quick recall of previously executed commands to enhance productivity and ease repetitive tasks.

**Requirement**

The primary requirement for this task was to design a welcoming and interactive Command Line Interpreter (CLI) for our Bare Metal Operating System on the Raspberry Pi. Essential features include a visually appealing ASCII-art welcome message, command auto-completion, navigation through command history using custom keys ('\_' for UP and '+' for DOWN), and implementation of core commands such as help, clear, showinfo, baudrate, and handshake.

**Implementation:**

* **Welcome Message:**

An ASCII art was generated using <https://onlineasciitools.com/convert-text-to-ascii-art> and printed to UART during boot.

* **CLI Prompt:**

The prompt always shows “FixingGoodOS>” before each command.

* **Command Handling:**

Input is collected using a character buffer commandBuffer, and processed when Enter is pressed.

Autocompletion is triggered when the Tab key (\t) is pressed. The implementation checks for prefix matches among available commands:

History is stored in a ring buffer. Pressing “–” navigates back, while “=” navigates forward. The buffer is managed to overwrite the oldest command if full. (We don’t use “-” and “+” explicitly because it would require an extra Shift on keyboard for it)

Backspace (127 or \b) removes the last character and redraws the line.

* **Commands Implemented:**

-help: Lists all commands or detailed info if followed by a command name.

-clear: Clears terminal screen using ANSI escape codes.

-showinfo: Reads MAC and board revision from specific memory locations.

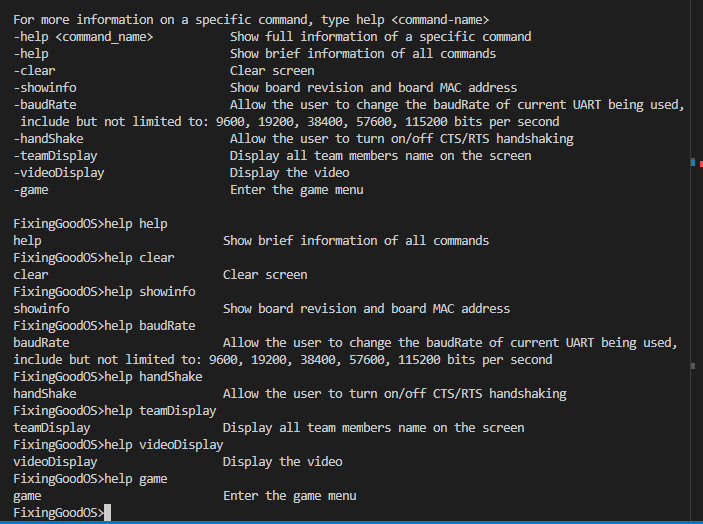
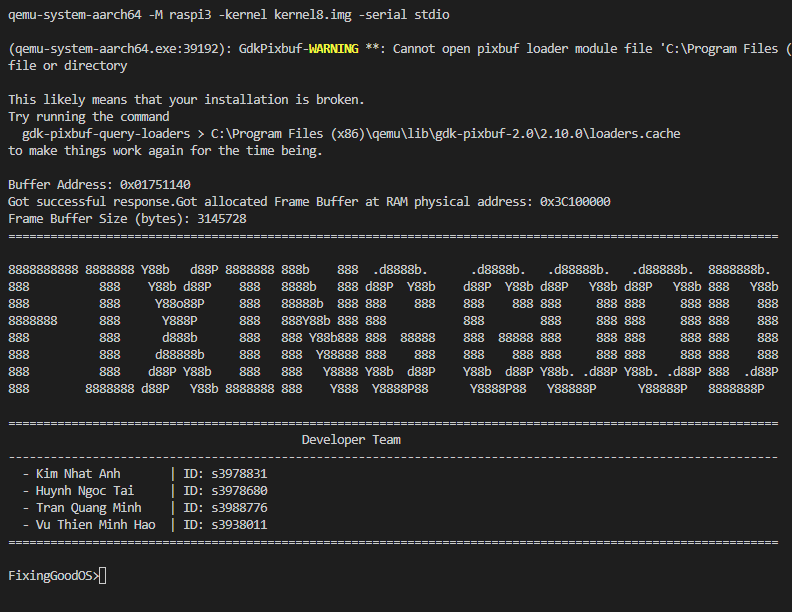
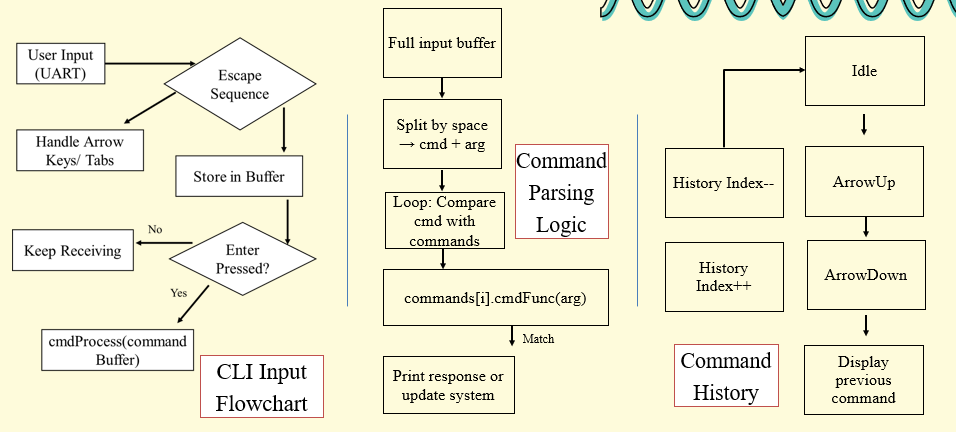
-baudrate: Accepts integer input and sets UART baudrate via divider calculation.

-handshake: Enables/disables CTS/RTS using GPIO setup.

-teamDisplay and videoDisplay are integration commands used to demonstrate Task 2.

**Result Discussion:** All CLI features function correctly on UART0. The auto-completion significantly improves usability. Command history is intuitive. Limitations include no fuzzy-matching or command arguments parsing.

Our CLI successfully delivered on all specified requirements, providing a robust, user-friendly interface that significantly improves the interaction experience with our Bare Metal OS. Users benefit from efficient command entry with auto-completion, easily manageable command history, and comprehensive built-in command support. This level of interaction greatly facilitates the troubleshooting and management of the system, establishing a solid foundation for the further tasks in this project.

**Summarized program Flowchart/Diagrams:**

# 

# III. IMAGE, VIDEO, AND TEXT DISPLAY

**Requirement:**

a) Show names of team members with colored text over a background image. Custom font must be used.

b) Display a short video as sequence of images.

**Background**

Displaying multimedia content such as images, videos, and text is a critical capability for embedded systems, especially those involving user interactions and graphical interfaces. It enhances user experience significantly by providing clear visual feedback and dynamic content presentation. Proper multimedia handling requires careful management of hardware resources, such as frame buffer memory and graphics processing, to ensure smooth and efficient rendering.

**Requirement**

The primary requirement for this task was to develop robust capabilities within our Bare Metal Operating System for rendering multimedia content. Specific features included displaying static images clearly with accurate colors, rendering short video clips smoothly with optimized frame rates, and managing custom text rendering using bitmap fonts.

**Implementation:**

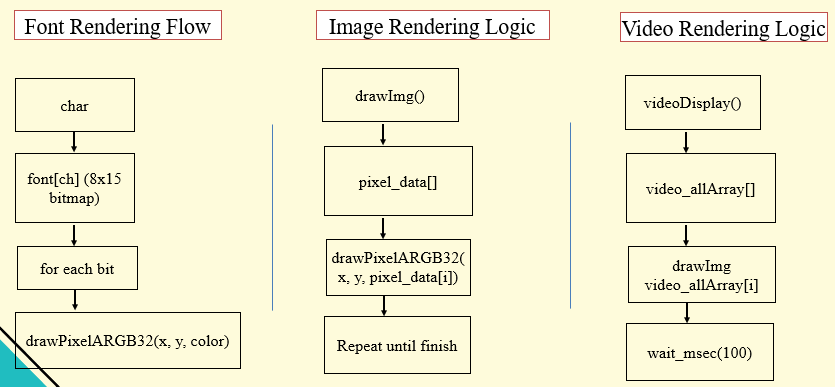
* **Font:**

We implemented a custom font system where each character glyph is represented as an 8x15 bitmap. These bitmaps were extracted from the Tamzen8x15.bdf file in the Tamzen font repository - <https://github.com/sunaku/tamzen-font/tree/master/bdf>.

The BDF file was parsed and converted into a C-style header file containing a two-dimensional array, where each glyph is stored as a sequence of bytes. During rendering, each bit in the glyph data is read and drawn to the screen pixel-by-pixel using drawPixelARGB32().

* **Background Image:** An image was converted to ARGB32 using https://javl.github.io/image2cpp/ and displayed full-screen on qemu.
* **Text Overlay:** Each team member's name is drawn in a different color at fixed positions with the background image that’s draw beforehand using drawPixelARGB32() function.
* **Video Display:** A short video was converted to 31 BMP frames using FFmpeg, then each frame converted to ARGB32 arrays. These arrays are shown sequentially using a videoDisplay() loop.

**Summarized program Flowchart/Diagrams:**

****

**Result Discussion:** Text and images render as expected. Video playback is smooth but limited to 20 FPS due to processing overhead.

**Summary of features implemented in both Tasks 1 & 2**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature Group** | **Command/ Feature** | **Implementation** | **Testing (any issues/limitations)** |
| **CLI Basic Features** | welcome screen | complete | Displays on UART boot |
| help | complete | Shows usage with and without args |
| clear | complete | Clears screen using ANSI codes |
| showinfo | complete | Displays MAC and board revision |
| baudrate | complete | Changes UART0 baudrate live |
| **CLI Enhancement** | OS name in CLI | complete | Static prompt FixingGoodOS> |
| Auto-completion in CLI | complete | Matches prefix and fills input |
| Command history in CLI | complete | Works with “–“ (going back) and “=” (going forward)  (We don’t use “-” and ”+” because it would require an extra shift on keyboard) |
| **Image, Video, and Text Display** | Background image and text display | teamdisplay | Shows names of 4 members in color with the RMIT background. |
| Video display | videodisplay | Plays 31 frames @ ~20 FPS |

# IV. APPLICATION DEVELOPMENT

**Game: Basketball Star**

**Introduction**

**Catch Game is an engaging interactive game specifically designed for our Bare Metal Operating System running on the Raspberry Pi. In this game, players control a basket to catch falling items within a set time limit, demonstrating real-time responsiveness, graphical capabilities, and efficient resource management inherent in embedded systems.**

**Objective and Gameplay**

**The primary objective of Catch Game is to control a basket to catch various falling items, aiming to accumulate points within the given timeframe. Players navigate the basket horizontally using keyboard inputs ('a' to move left and 'd' to move right) to catch items descending from the top of the screen. Different items yield different scores or effects, requiring strategic gameplay.**

**The game comprises three progressive stages, each with unique challenges and item spawn rates. Players must achieve specific score thresholds to progress through stages, each having progressively longer time limits. Players must skillfully manage their time and catch beneficial items while avoiding detrimental ones.**

**Gameplay Mechanics and Rules**

**Basket Control**

**Players use simple keyboard inputs to control horizontal movement of the basket. Responsive input handling is crucial to successfully catch items and avoid hazards.**

**Items and Effects**

**Several types of items fall from the top of the screen:**

* **Normal Ball: Grants standard points.**
* **Special Ball: Grants higher points.**
* **Bomb: Deducts significant points.**
* **Enlarge Item: Temporarily increases the basket's size.**
* **Score Multiplier: Temporarily doubles the points earned.**

**Strategically catching these items greatly impacts gameplay, requiring players to balance risk and reward effectively.**

**Time and Score Management**

**Each stage has a specific time limit: 30 seconds for Stage 1, 40 seconds for Stage 2, and 60 seconds for Stage 3. Players must meet or exceed predefined score goals within these time limits to advance. Failure occurs either by not achieving the required score within the given time or if the player's score drops below a critical negative threshold due to catching bombs.**

**Win and Lose Criteria**

**Winning is achieved by successfully progressing through all three stages by meeting the specified score goals within the designated times. Losing conditions include running out of time without achieving the required score or hitting a negative score threshold due to bomb interactions.**

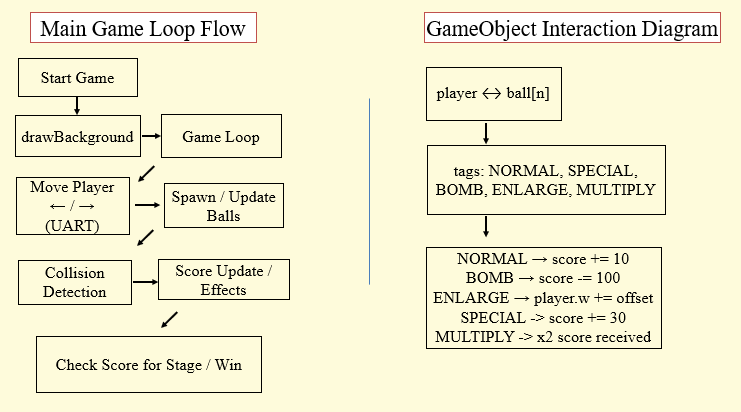
**Gameplay:**

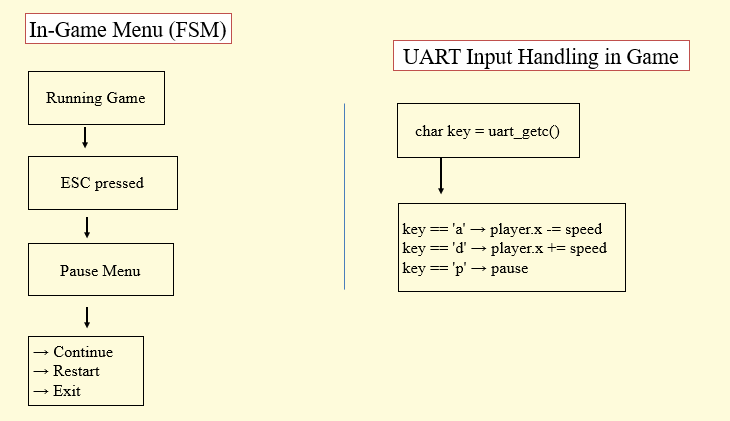
* Player controls a basketball hoop at the bottom of the screen.
* Balls fall from the top. Player must move left/right to catch them.
* Different ball types: normal, special (bonus), bomb (penalty).

**Design:**

* **Background:** Static image displayed once.
* **Sprites:** Hoops and balls are drawn using pixel data.
* **Physics:** Balls fall vertically; collision is checked against the hoop.
* **Controls:** Arrow keys are mapped through UART for player movement.
* **Game Logic:** Score increments or decrements based on ball type.
* **Communication:**
  + UART logs each command received.
  + Acknowledges valid commands with ACK.
  + Logs current score and number of balls caught.

**Summarized program Flowchart/Diagrams:**





**Result Discussion:** Game is functional with responsive controls and smooth visuals. Limitation: no pause/resume functionality, and fixed difficulty.

Catch Game successfully demonstrates the responsive interaction and graphical capabilities achievable through our Bare Metal OS implementation. The gameplay mechanics were highly effective, with precise input handling and real-time graphical updates providing a seamless user experience. The strategic complexity introduced by different item effects added depth, enhancing engagement and replayability. Accurate collision detection and efficient timer management ensured robust gameplay mechanics, effectively demonstrating the embedded system’s capabilities in handling real-time graphical interactions.

**Limitations**

Despite successful implementation and overall smooth gameplay, the project exhibits a few limitations. Notably, the single-threaded approach used in the game led to occasional performance degradation, particularly noticeable when multiple objects were simultaneously on-screen. This issue was due to our decision to utilize software-based timers and interrupts without leveraging the multicore capabilities of the Raspberry Pi, resulting in periodic CPU overload and occasional lag.

Additionally, the absence of a persistent state management system or database limited our ability to store player progress or reload game states effectively, thus impacting user experience, especially when transitioning between game stages or sessions.

Lastly, extensive graphical rendering and collision detection calculations contributed to minor graphical delays under specific scenarios, primarily during object-heavy sequences.

These limitations provide valuable insights for future improvements. Implementing multicore processing to distribute computational load, incorporating persistent storage solutions for state management, and further optimizing graphical and computational efficiencies would substantially enhance gameplay responsiveness and overall user experience.

# V. CONCLUSION

This assignment provided valuable experience in low-level OS development, framebuffer rendering, and interactive application design. We deepened our understanding of UART, CLI parsing, framebuffer structure, and embedded graphics handling. Working with the Raspberry Pi in a bare-metal environment helped bridge theory and practical implementation. Looking forward, we aim to explore real-time scheduling and multi-threading for more advanced systems.

This project has been a deeply rewarding experience, transforming our understanding of embedded systems from theoretical knowledge into practical capability. Through the design and development of a bare-metal operating system and the implementation of a complete interactive game, we were able to explore core concepts such as hardware interfacing, frame buffer manipulation, timer-based control, and UART communication in a meaningful and applied context.

Working with low-level graphics operations to render images, text, and game objects on screen gave us valuable insight into the challenges of direct hardware communication and memory management. Furthermore, developing a real-time game with precise timing and collision detection honed our ability to implement and debug time-sensitive routines using system timers and frame counting.

Beyond the technical scope, this project cultivated strong teamwork and project coordination. Regular collaboration, code reviews, and problem-solving sessions helped each member grow while ensuring consistent progress and quality. Logging mechanisms and in-game feedback systems played a crucial role in our debugging efforts and performance evaluation.

Despite the challenges encountered—such as CPU limitations and the absence of persistent state management—our final product delivers a smooth, responsive, and engaging user experience. We are proud of what we have accomplished and confident that the knowledge and skills gained will empower us to tackle more advanced embedded systems projects in the future. The Catch Game represents not only a technical milestone but also a testament to our growth as aspiring engineers.

# VI. REFERENCES (USE IEEE STYLES)

[1] Raspberry Pi Board Revision Info, https://www.raspberrypi-spy.co.uk/2012/09/checking-your-raspberry-pi-board-version/  
[2] MAC Address Info, https://www.javatpoint.com/what-is-mac-address  
[3] ASCII Art Generator, https://onlineasciitools.com/convert-text-to-ascii-art  
[4] Image to C Array Converter, https://javl.github.io/image2cpp/  
[5] Scratch Game Tutorials, www.youtube.com/watch?v=jFVJdRLZoQ4, www.youtube.com/watch?v=QXru0rSV2ZQ  
[6] PCS Font Tutorial, https://github.com/bztsrc/raspi3-tutorial/tree/master/0A\_pcscreenfont  
[7] Framebuffer Tutorial, <https://github.com/babbleberry/rpi4-osdev/tree/master/part5-framebuffer>

https://github.com/sunaku/tamzen-font/tree/master/bdf

*For each part of the assignment, please introduce the* ***requirement*** *(what will be implemented), then following by* ***implementation*** *(clearly explain how you implemented it, including important information for readers to understand), and finally* ***result discussion*** *(result and any limitation if it has).*

*For the code explanation, you don't need to go line by line, but should explain the way you do it (could be done in similar way of the lab guide). Flowcharts or diagrams could be used to support/illustrate your explanation if necessary.*

