RMIT UNIVERSITY

EEET2490 – Embedded System:
OS and Interfacing, Semester 2024-2

Group Assignment Report

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

Lecturer: *Dr. Linh Tran*

Team Number: Team 6

Team Members:
Do Khoa Nguyen (s3978796)
Ngo Ngoc Thinh (s3879364)
Tran The Quang Minh (s3979562)
Nguyen Dinh Quoc Bao (s3938309)
Tran Kiem Phuc (s3927187)

Due Date: September 24th, 2024

Table of Contents

I. INTRODUCTION

As part of EEET2490 Embedded System: OS and Interfacing's assignment, the team had to implement essential operating system components, such as a command line interpreter (CLI), a welcome message while rendering images, videos, and an application development challenge that involved making a game of Tetris. Using a Raspberry Pi as the target platform, the project stresses practical expertise with low-level hardware interfacing, OS-level programming, and graphical interface creation.

The project objectives were split up into multiple important tasks:

- Implementing a user-friendly CLI with various commands and features.
- Creating methods to display images, text, and even video on the screen.
- Developing a Tetris Game

II. WELCOME MESSAGE AND COMMAND LINE INTERPRETER

Welcome message

The system welcome message is displayed using welcome() function which is a part of the command line interface. The function first prints a piece of welcome text that tells the user the name of the operating system, then shows the name of the group and the course that this project belongs to. The welcome text is a piece of ASCII art, stored in a character array.

```
int welcome() {
  println("");
  println("");
  print(welcome_text);
  println("");
  println("");
  print_color("Group ", CMD_COLOR_GRN);
  print_color(GROUP_NAME, CMD_COLOR_CYN);
  print_color(" - EEET2490 - Embedded System", CMD_COLOR_GRN);
  println("");
  return 0;
}
```

The ASCII art is acquired by a text to ASCII art tool on the internet called onlinetools.com in the font. After displaying the welcome text, the group name and the course name are printed out, using functions and macros defined in our own library.

It is displayed upon initialization of the operating system, which will use the default mode, command line interface (CLI). The welcome message will be printed only once during the running period of the OS and will not be shown again even if the user switched back to CLI mode from other available operating modes.

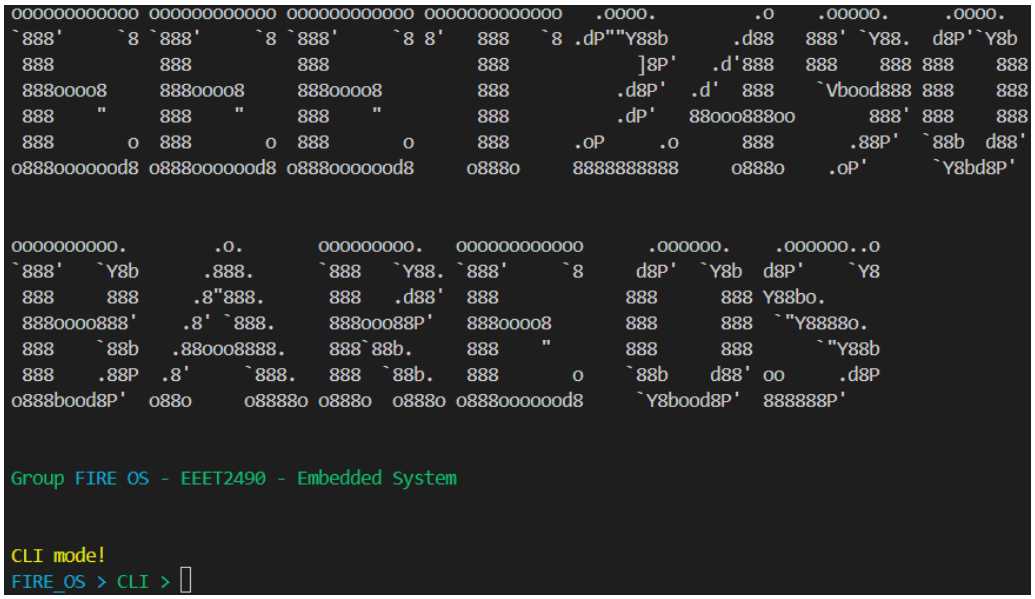

Figure 1. Welcome message display in the CLI.

CLI command structure

As commands are the backbone of our bare metal operating system – being the primary way to interface and interact with the Raspberry Pi, how commands are handled is an especially important aspect in system design. To handle how commands are called, named, executed, and providing relevant help message while maximizing code reusability, the team implemented each command as C structure stored inside a "global" commands array for reuse and ease of calling in subprocesses.

```
typedef struct {
  char *name;
  int length;
  char *help;
  char *full_help;
  void (*fn)();
} Command;
```

In detail, each structure stores the command name for knowing when to invoke the function, its name length, short and long help text, with the pointer to the function where the command executes. The system statically defines commands on compiles and stores all commands as an array for reference and invoking.

With this structure, each command detail is known to every part of the system as contexts, ease the implementation of auto-completion engine, help command, and execute history further in development.

Input buffer

Implemented using a stack, the main command input buffer allows user input as characters to be sequentially pushed to the stack then flush back to command handlers on enter (command submit).

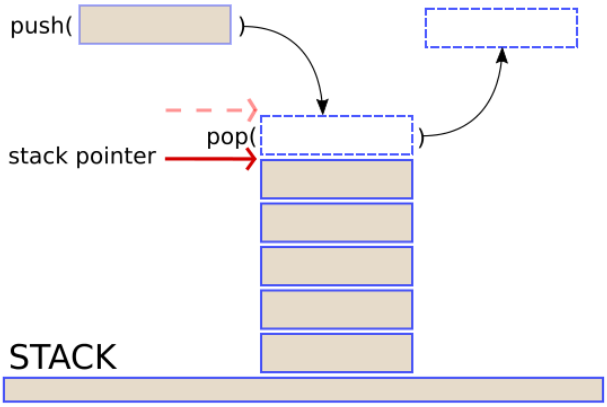

Figure 2. Stack buffer. (Humby, 2016)

This implementation allows flexibility in command building, as string concatenation and last character removal are easily implemented using stack push or pop methods.

ANSI Escape Sequences

In the context of communications between devices – hereby Raspberry Pi (the board) or emulated Raspberry Pi (using QEMU) to our own PC (host device) that mainly controls the operating system via UART protocol, our main "interface" of communication lies in the terminal emulator that the host machine uses to receive and transmit text to the remote device with ASCII character set and serial communication.

Introduced as a standard to control multiple aspects of terminal emulators such as the cursor, colors and font display, ANSI Escape Sequences are one of the main aspects that the team focuses on while developing the system command line system.

There are two main types of control codes and sequences the team will take advantage of using development, that is ANSI C0 control codes for basic control and ANSI Control Sequence Introducer (CSI) sequences for more advanced cursor manipulation and color formatting.

In more detail, instead of using one basic character – hence the name C0 control *code*, CSI sequences use multiple characters to build up a string, therefore called *sequence*. Often, CSI sequences start with `ESC [` combined with multiple characters as parameters and the command itself. In various programming languages, this could also be written as `\033` (octal representation), `\0x1b` or `\x1B` (hexadecimal representation) and `\e` (compiler-dependent C escape sequence) (GNU, n.d.).

Cursor control

Using ANSI Control Sequence Introducer (CSI) sequences, the team was able to control multiple properties of the terminal emulator, including cursor control for manipulating cursor location, implementing new line sequences, and supports character deletion on the command line interface.

Using known ANSI CSI sequences, the project implemented a core tty.c (short for *tele*type*writers*, named after inspiration from the Linux virtual terminal) utilities for wrapping common reusable code.

As the most used function of the project, `println()` was implemented for printing a string that ends with new line using ANSI C0 control codes. In more details, this function will first print the string to current cursor location, then prints C0 control code LF – line feed, also known as `\n` for moving the cursor to next line and prints CR – carriage return, known as `\r` for moving the cursor back to the beginning of the line.

```
void println(char *str) {
  print(str);
  print("\r\n");
}
```

The utilities library also provides a clear line function, `clrln()` used to remove the current line that the user was on, implemented using ANSI CSI sequence EL – erase in line, known as `\033[`*n*`K` with n as 0 for clearing from cursor to the end of the line, 1 for clearing from cursor to the beginning of the line, and 2 for clearing the whole line where the cursor is residing.

```
void clrln() {
  char clrln_sequence[] = "\033[2K";
  print(clrln_sequence);
  print("\r");
}
```

Take clearing characters off the screen further, the utilities library provides a clear entire terminal screen function `clrscr()` that was implemented with ANSI CSI sequence CUP – cursor position, known as `\033`*n*`;`*m*`H` with n and m can be left blank and default to 1 as the top-left position of the terminal, combined with ANSI CSI sequence ED – erase in display, known as `\033`*n*`J` for clearing from the cursor to the end of the screen

with n is default to 0. This will clear the whole terminal display and set the cursor position to the top-left of the terminal.

```
void clrscr() {
  char clr_sequence[] = "\033[H";
  print(clr_sequence);
  char home_sequence[] = "\033[J";
  print(home_sequence);
}
```

Moreover, ANSI C0 control code BS – backspace, known as \b was also used thoroughly in the system shell to provide character deletion feedback to the user.

Colors and Font decorations

Using ANSI CSI sequence SGR – select graphic rendition, with the value of \033*n*m, there are multiple parameters that could be applied to customize the display of a block – similar to the width and height of the cursor blinker.

To keep the scope of the program small, the team decided on define only 3-bit color space of supported ANSI SGR colors (foreground for text color, attribute range from 30-37), bring the total to eight provided and reusable colors macros.

```
#define CMD_COLOR_RED "\x1B[31m"
#define CMD_COLOR_GRN "\x1B[32m"
#define CMD_COLOR_YEL "\x1B[33m"
#define CMD_COLOR_BLU "\x1B[34m"
#define CMD_COLOR_MAG "\x1B[35m"
#define CMD_COLOR_CYN "\x1B[36m"
#define CMD_COLOR_WHT "\x1B[37m"
```

In case of the need to reset current parameters, an SGR reset macro was also defined with n being 0 for turning off all attribute at cursor location.

```
#define CMD_COLOR_RESET "\x1B[0m"
```

For later use to display and preview of command autocompletion, a macro of yellow text with underline and italic was also defined. This introduce chained SGR attribute that is 33 for yellow foreground, 4 for underline and 3 for italic, all being separated by semicolon character.

```
#define CMD_COLOR_YEL_UNDER_LINE "\x1b[33;4;3m"
```

Advanced keys input

In supported terminal emulator (the team tested on Tera Term, WezTerm and Alacritty), some extra keys could also be handled with ANSI CSI sequences, such as the four arrow keys. Using CUU, CUD, CUF and CUB sequence ranging from \033[A to \033[D.

```
#define CSI_CUU "[A" // up
#define CSI_CUD "[B" // down
#define CSI_CUF "[C" // right
#define CSI_CUB "[D" // left
```

Help command

On help received, the system will display all available commands, and its short help message gathered in the command struct above. Using colored text and padding between elements, the help command aims to provide quick reference while making it easier for the user to identify the needed command.

When help is received with an argument, such as help showinfo, the system will separate showinfo token to get full_help message, then show it to the user for that particular command.

Clear command

Using tty.c provided clrscr() function, on invoke, the clear command will immediately clear the user terminal emulator, similar to Linux implementation of clear.

Auto completion

The auto-completion functionality in our project improves user experience by offering command recommendations upon user input followed by the Tab key to activate the suggestion. Upon activation, the functionality assists users in swiftly identifying commands without the necessity of typing the complete command text.

The auto-completion technique operates in the following manner:

Command Parsing: Upon the user entering a partial command and pressing Tab, the command parser interprets the input string. The parser recognizes and retrieves the final token of the command. For instance, if the command is "help show", the final token is "show".

Matching Process: Upon getting the final token, the application compares it to a predetermined set of commands stored in “command.c”. The commands are organized alphabetically to facilitate efficient matching.

Prefix Matching: The program thereafter verifies whether any command in the predefined list possesses a prefix that corresponds to the last token. Upon finding a match, the first corresponding command is proposed to the user with an italic yellow text.

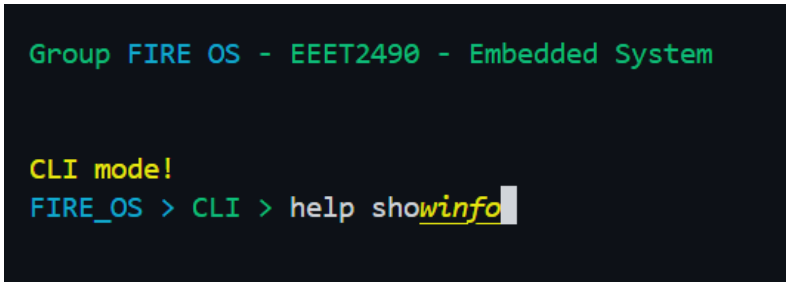

Figure 3. Auto-completion suggestion.

If user accept the suggestion, user can hit Tab/Enter:

Figure 4. Auto-completion accept.

If user do not accept the suggestion, user can continue to type or hit Backspace:

Figure 5. Auto-completion reject.

Sorted Command List: The predefined list of commands must be sorted to guarantee that the initial match is consistently the lexicographically smallest corresponding command. This arrangement enhances user experience by delivering consistent and rapid recommendations.

Command history

Implementation

With limited space and memory allocating capabilities, the team decided on using a ring buffer to store command history and traversing it to get previous commands. The buffer works by tracking the head (write position) to current push operation, meaning on first push (initialize with head = 0), the operation will write to rbuf[head] as rbuf[0] and advance head to the next position, then continuously advance the head counter as write operation are happening.

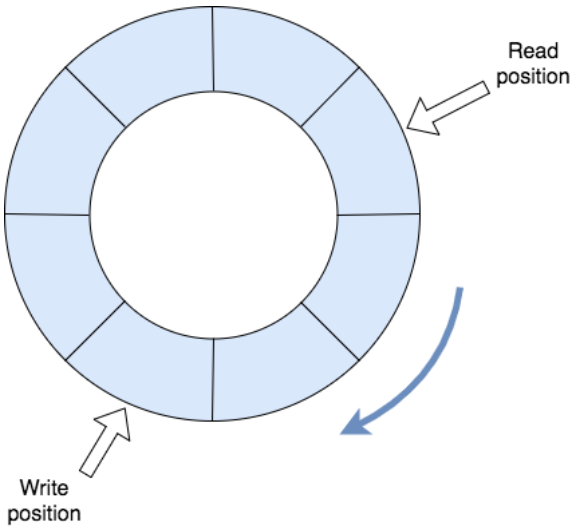

Figure 6. Circular buffer. (Mirek, 2017)

Using a ring (or circular) buffer, on overflow – when the maximum defined number of entries has been reached, latest command that were pushed to the buffer could be written back to the oldest entry (simply overriding the value) as the head (write position) advances back to the first position of the loop.

As an advantage with this data type, if traverse backwards from current head, the command given will guarantee to be in correct (reverse) order as it being pushed, therefore previous commands could be query with ease. It also provides simple check on when current oldest entry has been reached, using the current index of head.

This gives the user the ability to cycle through saved history without dynamically allocating memory.

Usage

Inside command line interface, on arrow up or underscore key press, the system will enter a shallow "history mode" to display and push previous command depending on current index for the user to edit or execute. Inside that "history mode", on arrow down or plus key press, the system will get latest commands based on current index, and our implementation will cycle back through the circular buffer if head is reached.

OS name

After every command register in the command line interface, the system will always display a prompt to give user information on current mode and the OS name (host name). This was implemented with prompt design taking some level of inspiration from Zsh – a UNIX command interpreter (shell), with elements such as color coding and the clean symbolic look (Zsh, n.d.).

Baud rate and Stop Bit settings

Baud rate

The baud rate determines the speed of data transmission in bits per second (bps) for UART communication. Fire OS supports configuring both UART0 and UART1 in both Raspberry PI 3 and 4. The formula for calculating the baud rate in the UART is:

+ UART1 baud rate register value formula:

$$uart1\_baud\_register\_value = \frac{systemclock\_frequency}{8 \cdot (baudrate\_require + 1)}$$

+ System clock frequency used = 250MHz.

+ UART0 IBRD and FBRD:

$$uart0\_IBRD = \frac{UART\_clock}{16 \cdot baudrate\_require}$$

$$uart0\_FBRD = (Fractional\ Part\ of\ uart0\_IBRD \cdot 64) + 0.5$$

+ UART clock used = 48Mhz (default UART clock frequency).

The table below shows the Baud rate values that the project currently supports with the mentioned clock frequency setup (for more valid baud rate values please refer to this handmade excel sheet):

Baudrate Value	UART1_BAUD	UART0_IBRD	UART0_FBRD
600	52082	5000	1
1200	26041	2500	1
2400	13020	1250	1
4800	6509	625	1
9600	3254	313	33
14400	2169	208	22
19200	1627	156	17
38400	813	78	9
57600	542	52	6
115200	270	26	3
230400	135	13	2

460800	67	7	33
921600	33	3	17

Table 1. List of valid baud rates.

How the program handle baud rate config command: In the CLI mode, the program constantly listen for new command and when the new command is started with keyword baudrate which is checked with function str_start_with_2(), it will execute the function as highlighted below.

```
// handle command only for cli mode
int _handle_cli_internal() {
  int is_handled = 0;
  if (str_equal(command, CMD_SHOW_IMAGE)) {
    displayWelcomeImage();
    is_handled = 1;
  } else if (str_start_with_2(command, CMD_BAUDRATE_PREFIX)) {
    _handle_baudrate_config();
    is_handled = 1;
  } else if (str_start_with_2(command, CMD_STOPBIT_PREFIX)) {
    _handle_stopbit_config();
    is_handled = 1;
  }
  return is_handled;
}
```

Basically, _handle_baudrate_config() function will extract the second token of the command and validate if that token is a valid baud rate number within a predefined list. In case, either the second token is invalid, or user does not specify the second token, the program will print out an error message with the valid baud rate list [Figure 7].

```
FIRE_OS > CLI > baudrate 123
Command received: baudrate 123
Invalid baudrate!
Valid baudrate list:
- 600
- 1200
- 2400
- 4800
- 9600
- 14400
- 19200
- 38400
- 57600
- 115200
- 230400
- 460800
- 921600
```

Figure 7. Valid baud rate list.

In case the baud rate is valid, the program will configure the appropriate baud rate number by passing the parsed number (second token) into the uart_init_with_config() function of the UART library. The function then prints a message showing the baud rate plus stop bit value and the values respectively.

```
void _handle_baudrate_config() {
 TokenIndex idx = get_nth_token_indices(command, 1, ' ');
 if (idx.start == -1) {
  uart_puts("Please specify a baudrate number!\n");
  _print_list_of_valid_baudrates();
 } else {
  char *baudNumStr = tmp_buffer;
  TokenIndex idx = get_nth_token_indices(command, 1, ' ');
  unsigned int token_len = idx.end - idx.start + 1;
  for (unsigned int i = 0; i < token_len; i++) {
   baudNumStr[i] = command[idx.start + i];
  }
  baudNumStr[token_len] = '\0';

  Baudrate *inputBaudrate = get_baudrate(baudNumStr);
  if (!is_valid_baudrate(inputBaudrate)) {
   uart_puts("Invalid baudrate!\n");
   _print_list_of_valid_baudrates();
  } else {
   uart_puts("Configuring UART");
#ifdef UART0
```

```
    uart_puts("0");
#else // RPI4
    uart_puts("1");
#endif
    uart_puts(" with baudrate = ");
    uart_puts(baudNumStr);
    uart_puts(" now...\n");

    uart_init_with_config(inputBaudrate, current_stopbits);

    uart_puts("Done configuring baudrate!\n");
    _print_current_baudrate_and_stopbit();
  }
 }
}
```

Note: Both UART0 (u0) and UART1 (u1) can handle these baud rates. The implementation uses system clock and default UART clock settings to calculate the required value for the baud rate register, ensuring accurate transmission rates.

Stop Bit

Stop bits signal the end of data transmission in UART communication. Depending on the UART used, the number of stop bits can be configured vary:

UART0 (u0) supports both 1 stop bit and 2 stop bits. This flexibility allows for compatibility with various serial communication standards, depending on the connected device's requirements.

UART1 (u1), however, only supports 1 stop bit, as the mini UART lacks the capability to handle 2 stop bits. This is a limitation specific to the hardware design of the mini UART on Raspberry Pi, which simplifies the communication interface.

Similar to the baud rate settings mechanism, the program also listens for command with the prefix “stopbit” and extracts the second token (if have) of this command. Depending on the current compiled UART (0 or 1), the program will check the second, if it is not either 0 or 1, the program will reject the command and print out error.

```
void _handle_stopbit_config() {
  TokenIndex idx = get_nth_token_indices(command, 1, ' ');
  if (idx.start == -1 || idx.end == -1) {
    uart_puts(
        "Please specify a stopbit number for UART either:\n+ 1 or 2 for "
        "UART0\n+ 1 for UART1\n");
  } else {
    char *stopbitNumsStr = tmp_buffer;
    TokenIndex idx = get_nth_token_indices(command, 1, ' ');
```

```
    unsigned int token_len = idx.end - idx.start + 1;
    for (unsigned int i = 0; i < token_len; i++) {
      stopbitNumsStr[i] = command[idx.start + i];
    }
    stopbitNumsStr[token_len] = '\0';

#ifdef UART0
    if (!str_equal(stopbitNumsStr, "1") && !str_equal(stopbitNumsStr, "2")) {
      uart_puts("Invalid a stopbit for UART0 number must be either: 1 or 2\n");
    } else {
      uart_puts("Configuring UART0 with stopbit = ");
      uart_puts(stopbitNumsStr);
      uart_puts(" now...\n");

      uart_init_with_config(current_baudrate,
                            str_equal(stopbitNumsStr, "1") ? 1 : 2);

      uart_puts("Done configuring stopbit!\n");
      _print_current_baudrate_and_stopbit();
    }
#else  // UART1
    if (!str_equal(stopbitNumsStr, "1\0")) {
      uart_puts("Invalid a stopbit for UART1 number must be only: 1\n");
    } else {
      uart_puts("Configuring UART1 with stopbit = ");
      uart_puts(stopbitNumsStr);
      uart_puts(" now...\n");

      uart_init_with_config(current_baudrate, 1);

      uart_puts("Done configuring stopbit!\n");
      _print_current_baudrate_and_stopbit();
    }
#endif
  }
}
```

In case the stop bit value is valid, the program again calls uart_init_with_config() and prints the success messages. A special not the function uart_init_with_config() will be implemented differently with UART0 and UART1 depending on the compiled target (use UART0 and UART1) then the appropriate implementation will be compiled.

Show board revision and MAC address

Initialization

To get the board revision code and MAC address, we use channel 8 of mailbox 0. The response data for MAC address and board revision from mailbox are displayed in the console and then processed to get more detailed information of the board.

```
print_color("INFO  ", CMD_COLOR_GRN);
print("Response code for mailbox message: ");
uart_hex(mBuf[1]);
println("");

print_color("INFO  ", CMD_COLOR_GRN);
print("Response code in GET board revision message: ");
uart_hex(mBuf[4]);
println("");
print_color("INFO  ", CMD_COLOR_GRN);
print("Board revision data: ");
uart_hex(mBuf[5]);
println("");

print_color("INFO  ", CMD_COLOR_GRN);
print("Response code in GET MAC address message: ");
uart_hex(mBuf[8]);
println("");
print_color("INFO  ", CMD_COLOR_GRN);
print("MAC address data: ");
uart_hex(mBuf[9]);
print(" & ");
uart_hex(mBuf[10]);
println("");
println("");
```

Figure 8. Mailbox messages to get MAC address and board revision.

Board revision process and display

The information that can be obtained about a board revision can be found in the official document from Raspberry Pi foundation on GitHub (GitHub, 2014). We divide the information retrieved from board revision code into two sections: hardware information and diagnostic information, distinguishing them by printing them out in different colors.

By analyzing segments of revision code and comparing them with official docs, we are able to display correct information of the Raspberry Pi 3 and Pi 4 that are used in this course. The information about hardware, which includes information about model, revision, processor, manufacturer, and memory size is shown first.

```
println_color("Hardware", CMD_COLOR_BLU);

print_color("Model", CMD_COLOR_BLU);
print(": Raspberry Pi ");

if (type_bits == 0x11) {
    println("4B");
} else if (type_bits == 0xD) {
    println("3B+");
} else if (type_bits == 0xE){
    println("3A+");
} else if (type_bits == 0x8){
    println("3B");
}
```

Figure 9: handling of different Pi 3 and 4 models

```
print_color("Revision", CMD_COLOR_BLU);
print(": 1.");
uart_dec(revision_bits);
println("");

print_color("Processor", CMD_COLOR_BLU);
print(": ");
if (processor_bits == 0x0) {
    println("Broadcom BCM2835");
} else if (processor_bits == 0x1) {
    println("Broadcom BCM2836");
} else if (processor_bits == 0x2) {
    println("Broadcom BCM2837");
} else if (processor_bits == 0x3) {
    println("Broadcom BCM2711");
} else if (processor_bits == 0x4) {
    println("Broadcom BCM2712");
}
```

Figure 10: processing and displaying revision and processor information

```
print_color("Memory", CMD_COLOR_BLU);
print(": ");
if (memory_size_bits == 0x0) {
    println("256MB");
} else if (memory_size_bits == 0x1) {
    println("512MB");
} else if (memory_size_bits == 2) {
    println("1GB");
} else if (memory_size_bits == 3) {
    println("2GB");
} else if (memory_size_bits == 4) {
    println("4GB");
} else if (memory_size_bits == 5) {
    println("8GB");
}
```

Figure 11: processing and displaying memory size

```
print_color("Manufacturer", CMD_COLOR_BLU);
print(": ");
if (manufracturer_bits == 0x0) {
    println("Sony UK");
} else if (manufracturer_bits == 0x1) {
    println("Egoman");
} else if (manufracturer_bits == 0x2) {
    println("Embest");
} else if (manufracturer_bits == 0x3) {
    println("Sony Japan");
} else if (manufracturer_bits == 0x4) {
    println("Embest");
} else if (manufracturer_bits == 0x5) {
    println("Stadium");
}
println("");
```

Figure 12: processing and displaying manufacturer name

After showing hardware information, show_boardrev() will show diagnostic information of the board as it currently is.

```
    println_color("Diagnostic", CMD_COLOR_BLU);

    if (style_flag_bit == 0x0) {
        print_color("Old", CMD_COLOR_CYN);
    } else if (style_flag_bit == 0x1) {
        print_color("New", CMD_COLOR_CYN);
    }
    println(" revision code style");

    print("Warranty ");
    if (warranty_bit == 0x0) {
        println_color("intact", CMD_COLOR_CYN);
    } else if (warranty_bit == 0x1) {
        println_color("voided", CMD_COLOR_CYN);
    }

    print("OTP reading ");
    if (otp_read_bit == 0x0) {
        println_color("allowed", CMD_COLOR_CYN);
    } else if (otp_read_bit == 0x1) {
        println_color("not allowed", CMD_COLOR_CYN);
    }

    print("OTP programming ");
    if (otp_program == 0x0) {
        println_color("allowed", CMD_COLOR_CYN);
    } else if (otp_program == 0x1) {
        println_color("not allowed", CMD_COLOR_CYN);
    }

    print("Overvoltage capabilities ");
    if (overvoltage == 0x0) {
        println_color("allowed", CMD_COLOR_CYN);
    } else if (overvoltage == 0x1) {
        println_color("not allowed", CMD_COLOR_CYN);
    }
}
```

Figure 13: processing and displaying diagnostics information.

Displaying the MAC address

After information from revision code is printed out to the console, the MAC address is formatted from mailbox response value and printed out in the 6 pairs of digits.

```
println_color("Networking", CMD_COLOR_BLU);
print_color("MAC address", CMD_COLOR_BLU);
print(": ");
unsigned char *byte_ptr = (unsigned char *)(&mBuf[9]);
for (int i = 0; i <= 5; i++) {
  uart_hex_no_base(*byte_ptr, 2);
  if (i != 5) {
    print(":");
  }
  byte_ptr++;
}
println("");
println("");
```

Figure 14: handling and showing MAC address.

Error handling and end of function

In the case the mailbox call fails to get a valid response, it will display an error message.

Results

The results when showinfo() successfully show board information for Pi 3, Pi 4 are shown in figures below:

```
CLI mode!
FIRE_OS > CLI > showinfo
Command received: showinfo
INFO  Getting system information...
INFO  Buffer address: 0x06EAABD0
INFO  Got successful response!
INFO  Response code for mailbox message: 0x80000000
INFO  Response code in GET board revision message: 0x80000004
INFO  Board revision data: 0x00B03115
INFO  Response code in GET MAC address message: 0x80000006
INFO  MAC address data: 0x12005452 & 0x00005734

Hardware
Model: Raspberry Pi 4B
Revision: 1.5
Processor: Broadcom BCM2711
Memory: 2GB
Manufacturer: Sony UK

Diagnostic
New revision code style
Warranty intact
OTP reading allowed
OTP programming allowed
Over voltage capabilities allowed

Networking
MAC address: 52:54:00:12:34:57
```

Figure 15: Showing Raspberry Pi 4 information.

```
CLI mode!
FIRE OS > CLI > showinfo
Command received: showinfo
INFO  Getting system information...
INFO  Buffer address: 0x06EAAC50
INFO  Got successful response!
INFO  Response code for mailbox message: 0x80000000
INFO  Response code in GET board revision message: 0x80000004
INFO  Board revision data: 0x00A02082
INFO  Response code in GET MAC address message: 0x80000006
INFO  MAC address data: 0x12005452 & 0x00005734

Hardware
Model: Raspberry Pi 3B
Revision: 1.2
Processor: Broadcom BCM2837
Memory: 1GB
Manufacturer: Sony UK

Diagnostic
New revision code style
Warranty intact
OTP reading allowed
OTP programming allowed
Over voltage capabilities allowed

Networking
MAC address: 52:54:00:12:34:57
```

Figure 16: Showing Raspberry Pi 3 information

III. IMAGE, VIDEO, AND TEXT DISPLAY

Image and text display

For this task, we needed to implement a method to display an image as a background, coupled with developing our own font to display text on the screen.

We started by developing a function to display an image by its color bitmap.

```
void displayWelcomeImage() {
  int pixel_index = 0;
  for (int y = 0; y < image_height; y++) {
    for (int x = 0; x < image_width; x++) {
      // Extract the ARGB32 color from the bitmap array
      unsigned int color = imageBitmap[pixel_index];

      // Draw the pixel at that coordinates
      drawPixelARGB32(x, y, color);

      pixel_index++;  // Move to the next pixel
    }
  }
```

```
  // omitted
}
```

The function processes each pixel of the image by iterating through its bitmap, extracting the color data for each pixel, and displaying it on the screen using the drawPixelARGB32() function. The bitmap, which contains the image's pixel color values in hexadecimal format, was generated using the image2cpp tool. Each value in the array corresponds to the color of a single pixel.

```
static unsigned int image_width = 1024;
static unsigned int image_height = 768;

//Comment out to lower compile time
 static unsigned long imageBitmap [] = {
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000,

// omitted
```

With the image done, we move on with the font, something that defines the pattern for each individual character glyph.

The pixel data for these character glyphs is also stored as a bitmap, where each pixel is represented by a single bit. If the bit is 'ON' (value 1), the pixel is drawn in the foreground color; if 'OFF' (value 0), the pixel remains in the background color.

For example, the letter "A" in an 8x8 font can be represented as follows:

```
0 0 0 0 1 1 0 0 = 0x0C
0 0 0 1 1 1 1 0 = 0x1E
0 0 1 1 0 0 1 1 = 0x33
0 0 1 1 0 0 1 1 = 0x33
0 0 1 1 1 1 1 1 = 0x3F
0 0 1 1 0 0 1 1 = 0x33
0 0 1 1 0 0 1 1 = 0x33
0 0 0 0 0 0 0 0 = 0x00
```

Figure 17. Letter "A" in 8x8 font.

This representation of the letter "A" can be stored as an array of 8 bytes: {0x0C, 0x1E, 0x33, 0x33, 0x3F, 0x33, 0x33, 0x00}.

Originally, we started working on our own 8x8 font glyphs for all alphanumeric characters. After testing, we deemed that 8x8 was way too restrictive and plain for

the project, especially for the game we planned to develop later. Therefore, the group decided to upgrade the font glyphs to 16x16 which would allow us way more breathing room to style our font.

```
unsigned char font[FONT_NUMGLYPHS][FONT_BPG] = {
     0x1f, 0xf8, 0x1f, 0xf8, 0x78, 0x1c, 0x78, 0x1c, 0x7c, 0x3c, 0x7c, 0x3c, 0x7c, 0x3c, 0x00, 0x00}, // U+0041 (A)
    {0x00, 0x00, 0x00, 0x00, 0x3e, 0xf8, 0x3e, 0xf8, 0x3f, 0xfe, 0x1e, 0x1e, 0x1e, 0x1e, 0x1f, 0xf8,
     0x1e, 0x1c, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x3f, 0xfe, 0x3e, 0xf8, 0x3e, 0xf8, 0x00, 0x00}, // U+0042 (B)
    {0x00, 0x00, 0x00, 0x00, 0x0f, 0xdc, 0x0f, 0xdc, 0x0f, 0xfc, 0x3e, 0x1c, 0x3e, 0x1c, 0x38, 0x00,
     0x38, 0x00, 0x38, 0x00, 0x3e, 0x1c, 0x0f, 0xfc, 0x0f, 0xfc, 0x0f, 0xdc, 0x00, 0x00, 0x00, 0x00}, // U+0043 (C)
    {0x00, 0x00, 0x00, 0x00, 0x1e, 0x78, 0x1e, 0x78, 0x1f, 0xfe, 0x0e, 0x0e, 0x00, 0x00, 0x0e, 0x0e,
     0x0e, 0x0e, 0x00, 0x00, 0x0e, 0x0e, 0x1f, 0xfe, 0x1e, 0x78, 0x1e, 0x78, 0x00, 0x00, 0x00, 0x00}, // U+0044 (D)
    {0x00, 0x00, 0x3f, 0xdc, 0x3f, 0xdc, 0x3f, 0xfc, 0x0e, 0x00, 0x0e, 0x00, 0x0e, 0xf8, 0x0f, 0xf8,
     0x0f, 0xf8, 0x0e, 0x00, 0x0e, 0x00, 0x3f, 0xfc, 0x3f, 0xdc, 0x3f, 0xdc, 0x00, 0x00, 0x00, 0x00}, // U+0045 (E)
```

Figure 18. Glyphs.

With the glyphs down, all we need now is a function that would allow us to render these glyphs.

The drawChar() function renders a single character on the screen using glyph data from the font array. The character (ch) is used as an index to retrieve the corresponding glyph. The function iterates through each pixel of the character, drawing it pixel by pixel at the specified position (x, y) using the given color attribute (attr) and scaling factor (zoom). For each pixel, the appropriate byte from the glyph data is extracted and bitwise operations are used to determine whether the pixel should be set. If the pixel is active, the function calls drawPixelARGB32 to render it on the screen. The glyph pointer is updated after each line of the character, ensuring the correct part is used as the function accounts for zoom by repeating pixel rows and columns accordingly. The result is a scaled and colored representation of the character drawn at the desired location.

```
/* Functions to display text on the screen, 16x16 */
// NOTE: zoom = 0 will not display the character
void drawChar(unsigned char ch, int x, int y, unsigned int attr, int zoom) {
  unsigned char *glyph =
     (unsigned char *)&font + (ch < FONT_NUMGLYPHS ? ch : 0) * FONT_BPG;

  for (int i = 0; i < FONT_HEIGHT * zoom; i++) {
    for (int j = 0; j < FONT_WIDTH * zoom; j++) {
      unsigned char byte = glyph[(j / zoom) / 8];
      unsigned char mask = 1 << (7 - ((j / zoom) % 8));

      if (byte & mask) {
        drawPixelARGB32(x + j, y + i, attr);
      }
    }
    if ((i + 1) % zoom == 0) {
      glyph += FONT_BPL;
    }
```

```
  }
}
```

The drawString() function is responsible for rendering a sequence of characters (a string) on the screen starting at the specified position (x, y). It processes each character in the string and calls the drawChar() function to render it. Special handling is provided for carriage returns (\r), which reset the x-coordinate to the start of the line, and newline characters (\n), which reset the x-coordinate and move the y-coordinate down by the character height to the next line. For regular characters, drawChar() draws the character, and then the x-coordinate is incremented by the character's width (scaled by the zoom factor) to position the next character. This process repeats until the entire string is rendered.

```
void drawString(int x, int y, char *str, unsigned int attr, int zoom) {
  while (*str) {
    if (*str == '\r') {
      x = 0;
    } else if (*str == '\n') {
      x = 0;
      y += (FONT_HEIGHT * zoom);
    } else {
      drawChar(*str, x, y, attr, zoom);
      x += (FONT_WIDTH * zoom);
    }
    str++;
  }
}
```

With all of that done, we can finally complete our function to display our image with custom text to introduce our group:

```
void displayWelcomeImage() {
  int pixel_index = 0;
  for (int y = 0; y < image_height; y++) {
    for (int x = 0; x < image_width; x++) {
      // Extract the ARGB32 color from the bitmap array
      unsigned int color = imageBitmap[pixel_index];

      // Draw the pixel at that coordinates
      drawPixelARGB32(x, y, color);

      pixel_index++;  // Move to the next pixel
    }
  }
  drawString(100, 100, "Team member:", 0xFF0000, 3);
  drawString(100, 200, "Tran The Quang Minh", 0x2702F5, 2);
  drawString(100, 300, "Do Khoa Nguyen", 0xFFFF00, 2);
  drawString(100, 400, "Ngo Ngoc Thinh", 0xFFA500, 2);
  drawString(100, 500, "Nguyen Dinh Quoc Bao", 0xFFFF00, 2);
```

```
  drawString(100, 600, "Tran Kiem Phuc", 0xE502F5, 2);
}
```

And finally, the result:

Figure 19. Render image with team member names from customized font.

Video

To display a video, it is essentially just displaying still images (frames) in very quick succession. With the functions to display images down, all we needed left to complete this task was to develop a function to do the timing between each frame.

First, we recorded a very short video and used an online tool to extract 25 frames out of it, each frame is represented by a bitmap similar to the one mentioned in the Image section above. We put all the bitmap into an array (video_array) to iterate through later.

```
    0x0071585b, 0x0071585b, 0x0071585b, 0x0071585b, 0x0071585b, 0
    0x006e5558, 0x006e5558, 0x006e5558, 0x006e5558, 0x006e5558, 0
    0x006c5356, 0x006c5356, 0x006c5356, 0x006c5356, 0x006c5356, 0
    0x006a5154, 0x006a5154, 0x006a5154, 0x006a5154, 0x006a5154, 0
};

const int video_array_len = 25;
const int video_pixels_width = 640;
const int video_pixels_height = 480;
const unsigned long* video_array[25] = {
    frame_Image1,
    frame_Image2,
    frame_Image3,
    frame_Image4,
    frame_Image5,
    frame_Image6,
    frame_Image7,
    frame_Image8,
    frame_Image9,
    frame_Image10,
    frame_Image11,
    frame_Image12,
    frame_Image13,
    frame_Image14,
    frame_Image15,
    frame_Image16,
    frame_Image17,
    frame_Image18,
    frame_Image19,
    frame_Image20,
    frame_Image21,
    frame_Image22,
    frame_Image23,
    frame_Image24,
    frame_Image25
};
```

Figure 20. Video frames represented by array.

Now, the function to render them.

The resume_video function is used to resume video playback by setting the is_playing flag to 1. This flag indicates that the video is currently playing, while the pause_video function does the opposite.

```
void resume_video() { is_playing = 1; }

void pause_video() { is_playing = 0; }
```

The _draw_next_frame_if_is_playing function handles the rendering of the next video frame, but only if the is_playing flag is set to 1. When playback is active, the function draws the current frame from the video_array onto the screen at coordinates calculated by the functions _get_cinema_background_fit_x and _get_cinema_background_fit_y, which ensure the video is correctly positioned on the background image. After rendering the frame, the function increments the current_frame_idx to move to the next frame in the video or loops back to the beginning frame after reaching the end.

```
void _draw_next_frame_if_is_playing() {
  if (is_playing) {
    drawImage(video_array[current_frame_idx], _get_cinema_background_fit_x(),
              _get_cinema_background_fit_y(), video_pixels_width,
              video_pixels_height);
    current_frame_idx++;
    if (current_frame_idx == video_array_len) {
      current_frame_idx = 0;
    }
  }
}

int _get_cinema_background_fit_x() {
  return (width - video_pixels_width) / 2 + 18;
}

int _get_cinema_background_fit_y() { return 48; }
```

Putting all of that together into a final function which is called right after the user enter the video player mode via the CLI

```
void handle_video_player_mode() {
  print_color("\n\nVideo mode!\n", CMD_COLOR_YEL);
  print_prefix();
  should_exit_video_mode = 0;
  current_frame_idx = 0;
  is_playing = 1;

  // draw background
  drawImage(img_cinema, 0, 0, img_pixels_width, img_pixels_height);

  while (is_video_player_mode() && !should_exit_video_mode) {
    // frame_time_ms = 0.04s, video_len = 1s => 25 fps
    set_wait_timer_cb1(1, frame_time_ms, _uart_scanning_call_back);

    _draw_next_frame_if_is_playing();  // draw next video frame

    set_wait_timer_cb1(0, frame_time_ms, _uart_scanning_call_back);
  }
```

```
}
```

The background of a movie theater is rendered first (img_cinema), then the _draw_next_frame_if_is_playing function is called to render the video on the screen while the set_wait_timer_cb1 handles the timing of 0.04s (more details in the Timer section below) between the frames during a total of 1 second period. The result is a video that plays at 25 frames per second.

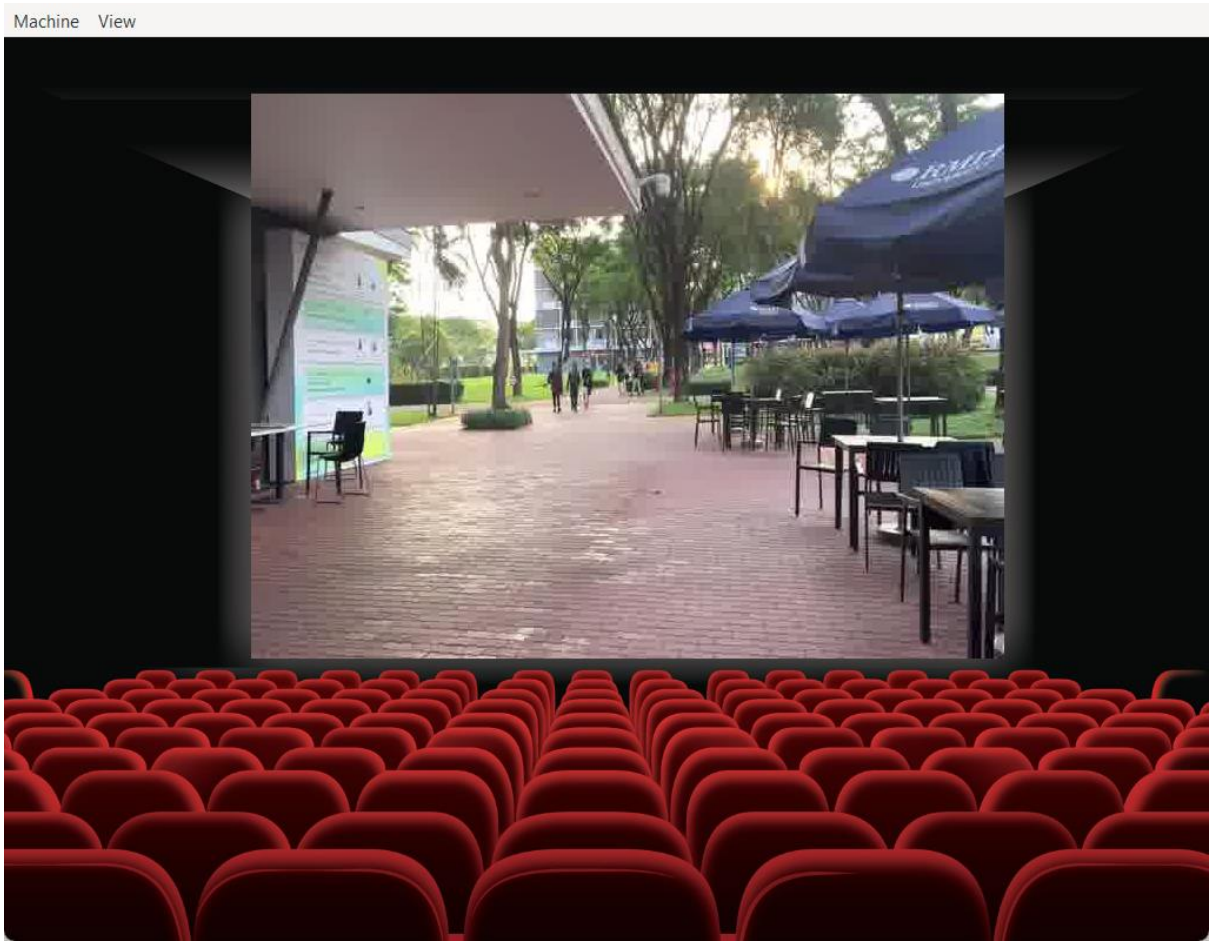

Figure 21. Video play mode.

During the video mode, the user can use the **"pause"** and **"resume"** commands to stop and play the video accordingly.

Summary of features implemented in both Tasks 1 & 2

Feature Group	Command/Feature	Implementation	Testing
CLI Basic Features	welcome screen, welcome text	**Completed**	No issue
	help: show help for all commands help <name>: show detail information for a specific command	**Completed**	No issue
	clear: clear CLI	**Completed**	No issue
	showinfo: show board detail information	**Completed**	No issue
	baudrate <number>: change baudrate settings	**Completed**	No issue
	stopbit + <number>: change stopbit settings	**Completed**	No issue
CLI Enhancement	OS name in CLI	**Completed**	No issue
	Auto-completion in CLI	**Completed**	No issue
	Command history in CLI	**Completed**	No issue
Image, Video, and Text Display	displayimage: Background image and text with customized font (team member's name)	**Completed**	No issue
	playvideo: Play video in 25fps pause: Pause video resume: Resume video	**Completed**	No issue
Extra command/feature	exit: Exit from current mode (CLI or Video)	**Completed**	No issue
	currentmode: Print current program's mode	**Completed**	No issue
	Automatically beautify commands, for example: " help showinfo " will be interpreted as "help showinfo"	**Completed**	No issue

IV. APPLICATION DEVELOPMENT

Game start and game controls

1. Game start

The game available for this operating system is a variant of the well-known game Tetris. To starts the game, the player types "playgame" command, which will put the console to game mode and simultaneously show the main menu in the graphical user interface:

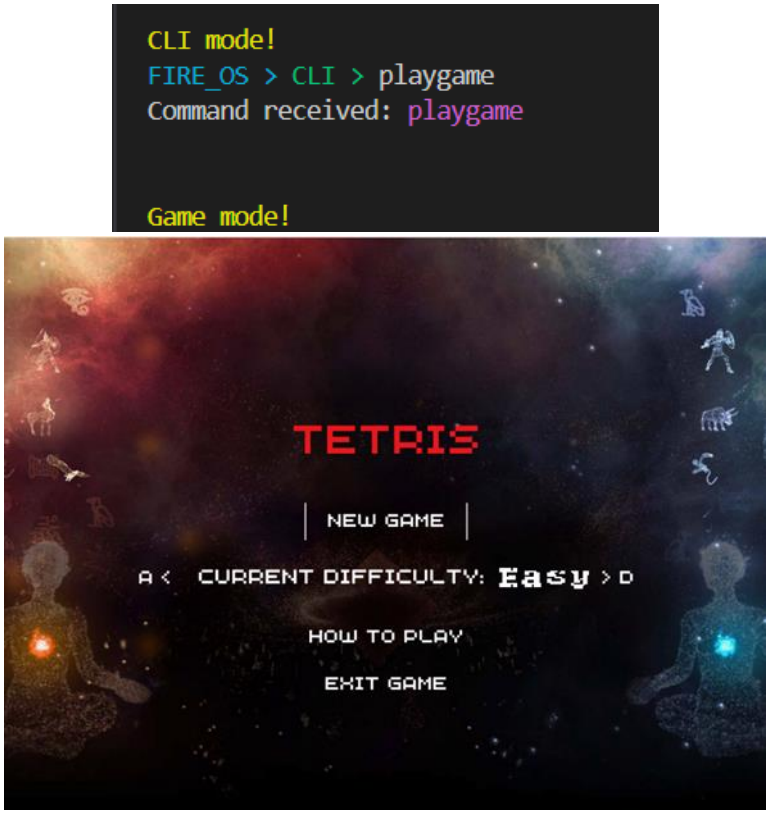

Figure 22. CLI enter game mode and welcome game screen.

2. Basic of game controls

Unlike other modes, every character typed is treated as a command and does not require confirmation by pressing ENTER, except for the backtick to prevent accidental exit or pausing in game mode. If a user types in an invalid command, the CLI displays an "unknown command" message, acknowledging that the command is unknown in game mode, otherwise, it confirms user's key press.

Figure 23. ACK and NAK in game mode.

In the game, the user can use a variety of keys: W/A/S/D or arrow keys, enter/space and backtick (`) in the CLI. Using these keys, the user can navigate through the game, play it, exit the current game or the application, and perform other actions available to them. Next part will show the flow of the game, and how to navigate it:

3. Game flow

3.1 Menu navigation

Menu controls

At the start of the game, the player is shown the main menu, awaiting their input. The player can us the following controls to navigate the menu and make selections:

Key	**Effect**
W/Arrow UP	move up to another item in the menu, until the top item is reached
S/Arrow DOWN	move down to another item in the menu, until the bottom item is reached
D/Arrow RIGHT	move right in difficulty selection (easy to medium, medium to hard).
A/Arrow LEFT	move left in difficulty selection (hard to medium, medium to easy).
SPACE/ENTER	Confirm the current selection/ cycle through game levels
Backtick (`)	Exit the application, return to CLI mode

Table 2. Game menu controls.

To play the game, the player can follow these steps:

1. Navigate the menu:
    - To move up in the menu: The player presses W or UP arrow to move up one menu item each time, focusing on the newly chosen item. If the top item is already highlighted in the menu, the next move up action does not cycle to the bottom item but stays at the same place.
    - To move down in the menu: The player presses S or DOWN to move down one menu item each time, focusing on the newly chosen item. If the bottom item is already highlighted in the menu, the next move down action does not cycle to the top item but stays at the same place.
2. Selecting a menu option
    - To confirm a menu selection: The player press SPACE or ENTER to confirm the current option available in the menu. Different options have different effects:
        - New game: Starts a new game.
        - Current difficulty: Adjust the difficulty of the game.
        - How to play: show a short game controls guide.
        - Exit: Quit the game and return to CLI mode.
3. Adjust game difficulty
    - The default difficulty is easy. To change the game difficulty, the player can:
        - Move right: Press D or arrow RIGHT to increase difficulty in the order of easy -> medium -> hard.
        - Move left: Press A or Arrow LEFT to reduce difficulty in the order of hard-> medium -> easy.
        - Cycling through difficulties; Press ENTER or SPACE to cycle from left to right, from easy to hard as described above.
        - Confirm selection: Press SPACE or ENTER to confirm the desired difficulty.

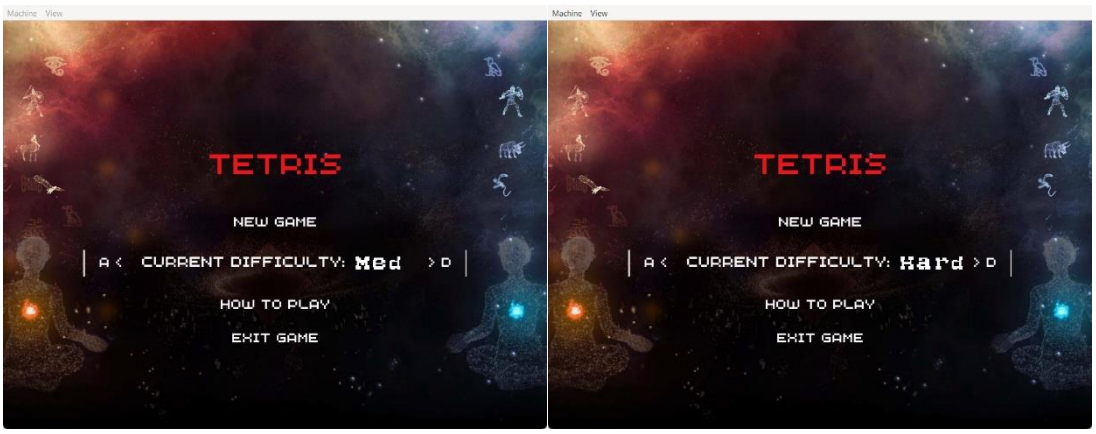

Figure 24. Medium and hard game difficulty adjustment.

4. View how to play.
    - When how to play is selected, pressing ENTER or SPACE will open the guide. The guide provides information on the control commands of the game.
    - To exit the guide screen, press enter again.

Figure 25. How to play view.

5. Application exit
    - To leave the application, the user moves to the exit option and confirm by pressing SPACE or ENTER.

3.2 Game play

Game play controls

During gameplay, the controls are used for moving the bricks and interact with the game as follows:

Key	**Effect**
W/Up arrow	No action, only available in the menu
S/Down arrow	Increase the brick falling speed, for faster placement.

D/Right arrow	Move the current brick to the right of the screen/Move to exit option in game over screen
A/Left arrow	Move the current brick to the left of the screen/ Move to new game option in game over screen
Space/Enter	Rotate the current brick 90 degrees/ Confirm option in game over screen
Backtick (`)	Exit the current game, return to main menu

Table 3. Game controls.

Game rule and game play

In a Tetris game, the player drops the various blocks, called "bricks" that stack up in the playfield, a rectangular grid of 24 rows and 11 columns. On the right of the game screen, the scoreboard displays the score of the current game, while the next brick display helps the user in devising strategy for the upcoming blocks.

The goal is to arrange the bricks to fill in rows completely to clear it and increase the player's score. The more rows cleared at once, the higher the score is, and the player receives bonus points for clearing multiple rows at the same time.

The game goes on until the bricks stack reaches the top of the playfield. Reaching that point, the game ends, and a game over screen is displayed, showing the final score. Here, the player can try again or return to the main menu.

Difficulty levels

The three difficulties available in this game are:

- **Easy:** the default level, for inexperienced players or who just want a relaxing game. The game speed is the slowest here (the piece will be moved down each **1.2s**).
- **Medium:** this level offers a moderate increase in game speed, giving the player a little challenge (the piece will be moved down each **0.7s**).
- **Hard:** This level has the highest game speed, requires rapid response from the player to keep up with the falling speed of the bricks (the piece will be moved down each **0.4s**). Also, the playfield will be occupied by immovable, frozen bricks as the game goes on, reducing available room for the player's brick to stack up.

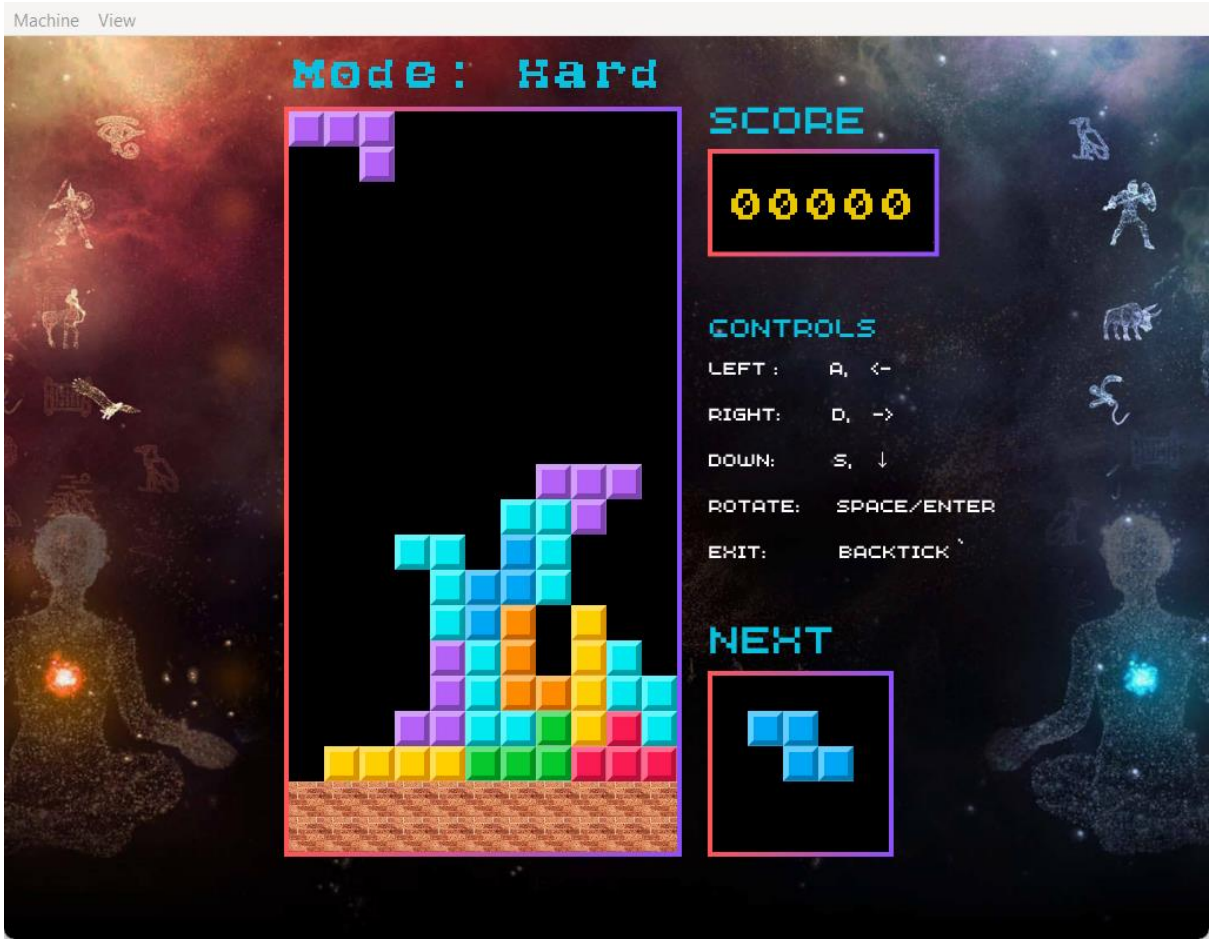

Figure 26. Hard game mode with frozen bricks.

Score formula

```
int _calculate_score_change(int number_of_row_cleared_at_once) {
  return score_step*number_of_row_cleared_at_once*number_of_row_cleared_at_once;
}
```

Figure 27. Score formula.

+ Each time a dropping piece is settled down, if there is any completed row the score will be changed. The number of completed rows at a time also effect the score change by the formula:

$$score\_changes = score\_step * completed\_rows^2$$

$$+ score\_step = 10$$

This formula will encourage players to accumulate completed rows and clear them at once, which makes the game more challenging and interesting.

Game over

When the player loses, the application displays a game over screen, showing the final score and two options: return to the main menu or continue playing with a new game. The player can choose one of the two options, by using the same controls used in the main menu. The game also displays game statistics for troubleshooting on the console, letting users know the final scores, the game difficulty, number of rows cleared, total received commands and other information in the command line interface.

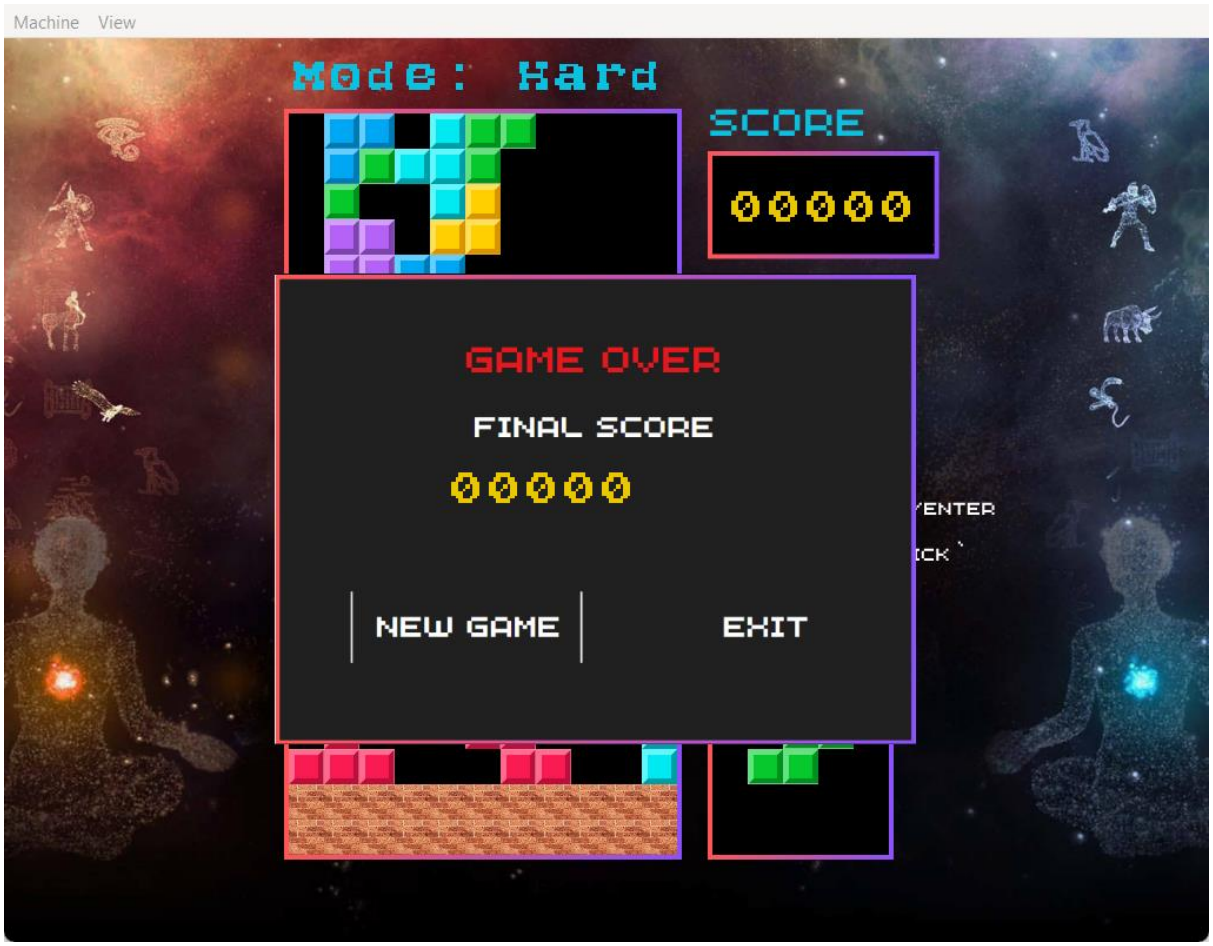

Figure 28. Game over screen.

Game Design

Timer

Before talking about the game mechanism, we will talk about how we set up the timer to generate different events within Fire OS. Remind about the function **"set_wait_timer"** provided by our lecturer Mr. Linh Tran, we have made a tricky adjustment to make this function never CPU-blocking again. But first, let explore how the original function work:

```c
void set_wait_timer(int set, unsigned int msVal) {
  static unsigned long expiredTime = 0;  // declare static to keep value
  register unsigned long r, f, t;

  if (set) { /* SET TIMER */
    // Get the current counter frequency (Hz)
    asm volatile("mrs %0, cntfrq_el0" : "=r"(f));

    // Read the current counter
    asm volatile("mrs %0, cntpct_el0" : "=r"(t));

    // Calculate expired time:
    expiredTime = t + f * msVal / 1000;
  } else { /* WAIT FOR TIMER TO EXPIRE */
    do {
      asm volatile("mrs %0, cntpct_el0" : "=r"(r));
    } while (r < expiredTime);
  }
}
```

```c
//set timer with 100ms
set_wait_timer(1, 100);

// go doing other tasks (which must take less than 100ms)
// ..........

//come back and wait for the timer to expire
set_wait_timer(0, 100);
```

Figure 29. Blocking timer function.

Basically, the original function will solve a problem to timing exactly between two events, which in the above figure is 100ms. But what if we "doing other tasks" take less than 100ms? Then we need to wait for exactly 100ms because the second do-while loop when we unset the timer only release when the interval has been passed, although "doing other tasks" takes much less than that in many cases. This disadvantage also led to an issue when we need to time for a longer interval for example 1 second, then for sure the CPU will be blocked in 1 second before the next user's interaction can be handled. According to our lecturer, the maximum response time for a good user experience should be 100ms. So, to overcome this disadvantage of the original function we have made a small adjustment that we pass in a callback when we call the timer function:

```c
void set_wait_timer_cb1(int set, unsigned int msVal, void (*cb)()) {
  static unsigned long expiredTime_1 = 0;  // declare static to keep value
  register unsigned long r, f, t;

  if (set) { /* SET TIMER */
    // Get the current counter frequency (Hz)
    asm volatile("mrs %0, cntfrq_el0" : "=r"(f));

    // Read the current counter
    asm volatile("mrs %0, cntpct_el0" : "=r"(t));

    // Calculate expired time:
    expiredTime_1 = t + f * msVal / 1000;
  } else { /* WAIT FOR TIMER TO EXPIRE */
    do {
      cb();
      asm volatile("mrs %0, cntpct_el0" : "=r"(r));
    } while (r < expiredTime_1);
  }
}
```

```c
while (is_game_mode() && !should_exit_game_mode) {
  set_wait_timer_cb1(1, smallest_interval_ms, _uart_scanning_callback);
  _handle_timing_events();
  set_wait_timer_cb1(0, smallest_interval_ms, _uart_scanning_callback);
}
```

Figure 30. Non-blocking timer function.

By this approach, we can choose a **"smallest wait interval"** act like a timer step (for example 10ms in our program) and when the CPU is blocked by that wait interval, we continuously call the callback function (in this case **"_uart_scanning_callback"** function) to listen to user's interaction (for example: user type somethings,..), until the wait interval has been passed. The **"_uart_scanning_callback**" function was built exactly the same as with description in the section II (command buffer) that can help to detect user's command, and we continuously call this function when we are blocked by the timer to ensure that any user's interaction can be detected and delegate to appropriate handler function immediately.

Video playing: Please note that the same technique is used when we render different frames of the video to prevent CPU's unresponsiveness to UART, more specifically, to generate exact frames per second we need to time exactly between frames rendering, if we just use the original function for timing, we will achieve the desired frames per second but the CPU will be busy either rendering (copy bytes to frame buffer) or blocking by the timing function, hence any user interaction related to UART signal will be unresponsive (CPU is too busy to read from UART buffer). But with this new adjustment the CPU can listen to user's interactions when it is blocked by the wait time interval between frames rendering or as soon as the next frame is transferred completely the program can detect immediately any user input. That explains why when the video is rendered continuously, if we type something the OS will immediately response to that interaction.

Timer Events

In additional, between wait interval of new timer function we also call **"_handle_timing_events"** to keep track of how long has been passed, the function looks like this:

```
typedef struct {
  int call_every_ms; // should be mutiple of smallest wait interval
  void (*handler)();
  volatile int counter;
} Event;
```

```
/* Events: call_every_ms should be multiple of 10 */
volatile Event events[] = {
    {.call_every_ms = 50,
     .handler = _handle_events_call_every_50ms,
     .counter = 0},
    {.call_every_ms = 100,
     .handler = _handle_events_call_every_100ms,
     .counter = 0},
    {.call_every_ms = 200,
     .handler = _handle_events_call_every_200ms,
     .counter = 0},
    {.call_every_ms = 500,
     .handler = _handle_events_call_every_500ms,
     .counter = 0},
    {.call_every_ms = 1000,
     .handler = _handle_events_call_every_1s,
     .counter = 0},
};
```

```
void _handle_timing_events() {
  for (int i = 0; i < sizeof(events) / sizeof(Event); i++) {
    events[i].counter++;
    if (events[i].counter >=
        events[i].call_every_ms / smallest_interval_ms + 1) {
      events[i].counter = 0;
      if (current_screen == SCREEN_GAME_PLAY) events[i].handler();
    }
  }
}
```

Figure 31. Timer events.

Basically, for each complete interval of **"smallest wait interval"** the program will increase the counters of all timer events to count how many times **"smallest wait interval"** has been passed. Timer events are defined within an array with their handler functions, when a specific time interval passed (by compare the counter with an appropriate value), the handler functions of the passed events will be called and

then the counters of these passed events will be reset to zero to start counting again in the next interval. By this way of implementation, we can time any interval we like as long as the timed intervals are multiple of **"smallest wait interval"**. With this implementation, we can also reset all the timer counters when we need to start counting again with ease. This reset functionality later will be used in one of the game commands to enhance user gaming experience. Now, we know that the OS can generate and handle events in any specific interval let deep dive into game mechanism.

Game Modelling

Piece: In Tetris there are a total of 7 different kinds of pieces with different shapes (I, O, T, S, Z, J, L). For each piece the most important information that we need to retain is its shape, current rotation angle and its center point. The piece struct will look like this:

```
typedef struct {
  Shape shape;
  Color color;
  Angle angle;
  Point center_point; // x, y position in the 2D array
} Piece;
```

Note: there are only four possible rotation angles in Tetris (0, 90, 180, 270).

```
typedef struct {
  int x;
  int y;
} Point;
```

The center point is represented by x (horizontal coordinate) and y (vertical coordinate) in the game field.

In any time in the game, there will be two pieces existing at a time: one is the dynamic piece (piece is being dropped down) and the other piece is the next piece (the next dynamic piece).

From piece's shape, angle and center point, we can calculate all the related points of a piece (each piece will have one center point and 3 related points around the center point). For example, to represent a T shape piece:

Figure 32. T shape information.

Color: To represent the current color of piece or block, there is a total of 7 colors for piece. In addition, color "clear" and "brick" are only used for block.

```
typedef enum {
  CYAN, // light blue
  YELLOW,
  PURPLE,
  GREEN,
  RED,
  BLUE,
  ORANGE,
  CLEAR,
  BRICK
} Color;
```

Game field: Game field is modelled by a 2D arrays of static blocks, readers can imagine block is just a point in the game field, but they have colors to mark if the points are occupied. Or a piece combined by many points, but after a piece is settled down its points will be converted into colored blocks inside the game fields.

```
typedef struct {
  Color color;
} Block;
```

In code base the game field is presented by 2D array:

```
Block static_game_field[GAME_FIELD_FULL_HEIGHT][GAME_FIELD_WIDHT];
```

For better visualization of the game field model, please refer to below figure:

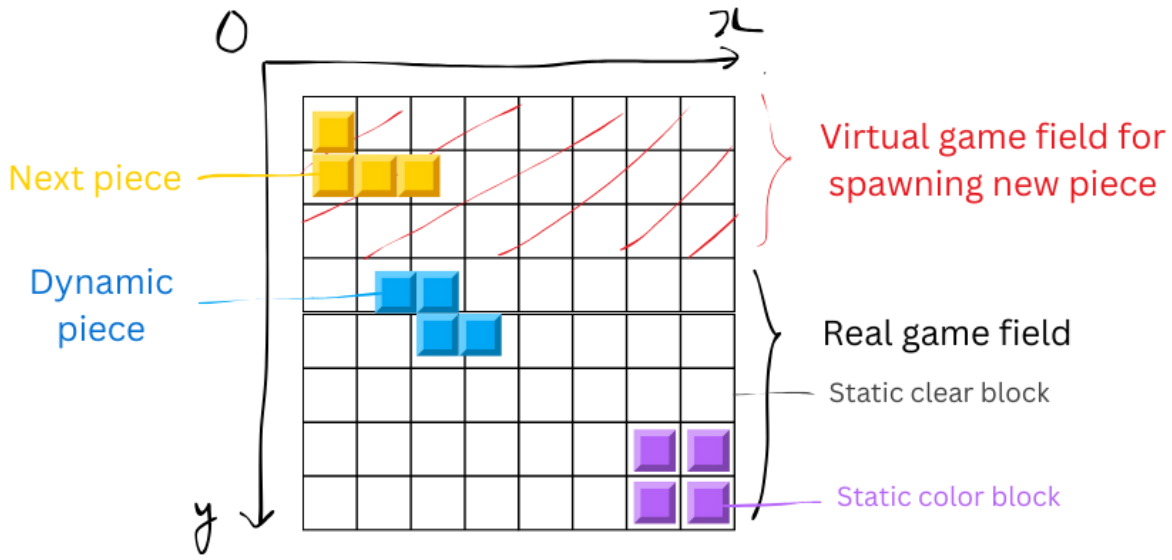

Figure 33. Game field model.

+ Virtual game field: to create drop from "sky" effect, we need the virtual game field, any colored point inside the virtual game field will not be rendered, as a result when the next piece starts to drop some points belonging to the real game field will be rendered but some points will not be.

+ Real game field: the real game field that player sees in the game play screen with width = 11, height = 21. All the colored points in this area will be rendered.

+ Static color block: at any time, there will be only one dynamic pieces, which contains 4 colored points, and when the piece is settled down by colliding with the bottom or any static-colored block, **all points belonged to the dynamic piece will be converted into static colored block with exact coordinates where it is settled down**. By this way of implementation, we can keep track of the game field state before and after a dynamic piece is settled down to mark any position in the game field that has been occupied.

+ Static clear block: indeed, clear blocks are just block with CLEAR color (the same color as game field background) to mark unoccupied positions inside the game field.

Game Algorithms

Completed rows checking: Any time a dynamic piece is settled down all the rows of the game field will be checked for its completion. A row is considered as complete when all the static blocks in that row are colored with colors for pieces (not CLEAR or BRICK color). The rows will be checked from top to the bottom, and if a row is completed it will be erased or all the rows above it will be offset down by one. For a checking, it will accumulate the number of completed rows and adjust the score based on the number following the score rule mentioned above.

Random piece spawning: Firstly, we have built a random generator by using an internal timer counter with modulus operator.

```
// Function to generate a random number within a specified range [min, max]
int rand(int min, int max) {
  wait_msec(2);
  return (SYS_TIMER_COUNTER_LOWER % (max - min + 1)) + min;
}
```

But why do we use wait_msec(2) inside a random generator? Because if we look at how we use the random generator to generate random pieces:

```
void _spawn_random_piece_to(Piece *piece) {
  int rand_angle = rand(1, 4);
  int rand_color = rand(1, 7);
  int rand_shape = rand(1, 7);
  int rand_center_point_x = rand(0, GAME_FIELD_WIDHT - 1);

  // ...
}
```

The way we call to the rand() function multiple times are the same and deterministic between each spawning function called. As a result, although, the pieces are indeed randomly spawned in different shapes but each shape will associate with deterministic color (due to the multiple consecutive deterministic call to rand()). For example: the T shape will always have ORANGE color. To eliminate this deterministic characteristic, we must use some think non-deterministic like wait_msec() function, although we declare 2ms for waiting time, but indeed the timer counter will fluctuate differently in that 2m interval because this wait_msec() is not exactly 2ms but very close to 2ms. And we just need that very small fluctuation to create a truly random generator.

When a new piece is spawned the next piece will be copied to dynamic piece, and the next piece will be the new spawned piece.

Game over checking: The game will be over if any colored block inside the virtual game field (any colored point overflows the top of the real game field).

Block rotation: Block rotation is implemented by clever matrix multiplication; the algorithm is inspired by the idea from (Said Sryheni, 2024). The algorithm

implemented in our program has a slight modification to not allow the piece to be rotated if after rotation one of the piece's points collide with a static block in game field (ensure game consistency).

```
algorithmRotateTetrisPiece(points, origin, beta):
    // INPUT
    //    points = the array of points representing the Tetris piece
    //    origin = the origin point of the shape
    //    beta = the rotation angle
    // OUTPUT
    //    rotatedPoints = the new coordinates after rotating the Tetris piece

    rotatedPoints <- { }

    for i <- 0 to points.size() - 1:
        x_0 <- points[i].x - origin.x
        y_0 <- points[i].y - origin.y

        x` <- x_0 * cos(beta) - y_0 * sin(beta)
        y` <- x_0 * sin(beta) + y_0 * cos(beta)

        x` <- x` + origin.x
        y` <- y + origin.y

        rotatePoints.add(x`, y`)

    return rotatedPoints
```

Figure 34. Pseudo code for Tetris piece rotation algorithm from (Said Sryheni, 2024).

Block moving left/right/down: Program only allows block to move left/right if it is not overflowing the game field after moving or the left/right positions not occupied by any static-colored block. When the block moved down by S or Down arrow, the event counter will be reset so that time for the next event will be reset, if we do not do this after moving the piece down by action and right after that the counter of any events is triggered then the piece will be moved again (double moving down).

Settling down checking: The piece settling down checking will be triggered every time either an event is triggered, or piece move left/right/down or piece rotation. Basically, a piece is considered as settling down when any point of piece lays down on top of any static-colored block in game field. If a piece is considered settling down, then the logic related to completed row checking, game over checking and new piece spawning will be triggered.

Waiting before settling down: There is another special case that we must tackle when we build the game. Look into the figure below:

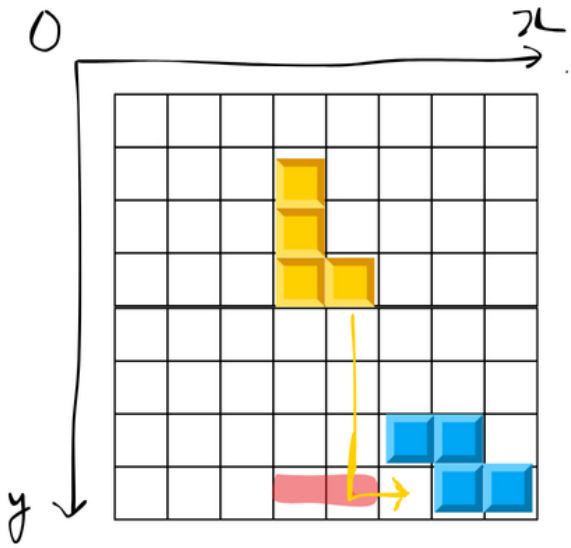

Figure 35. Waiting before settling down.

When the yellow piece reaches the bottom red position, we can consider it is settled down, as result the yellow piece can not move to desired position as show in the figure above. To overcome this, we will put a small wait interval of 200ms after the yellow piece reach to red position, the program will listen for user interaction of move piece left/right which will help the yellow piece to settle down in the desired position.

Block rendering: We already know how the game is modelled using piece, block and game field, another interesting task is how we can render the exact game state in the framebuffer. Reuse the functions to render images in the second part of the assignment. Based on that we can build functions to render a block in a position x, y with a specific color:

```
void _draw_game_point(int x, int y, Color color) {
  if (y < VIRTUAL_GAME_FIELD_OFFSET) {  // no draw virtual point
    return;
  }

  int real_point_x = x;
  int real_point_y = y - VIRTUAL_GAME_FIELD_OFFSET;

  if (real_point_y >= GAME_FIELD_HEIGHT) return;

  int physical_point_x =
```

```
    OFFSET_PHYSICAL_GAME_FIELD_X + real_point_x * BLOCK_SIZE;
  int physical_point_y =
    OFFSET_PHYSICAL_GAME_FIELD_Y + real_point_y * BLOCK_SIZE;
  if (color == CYAN) {
    drawCyanBlock(physical_point_x, physical_point_y);
  } else if (color == YELLOW) {
    drawYellowBlock(physical_point_x, physical_point_y);
  } else if (color == PURPLE) {
    drawPurpleBlock(physical_point_x, physical_point_y);
  } // ...
}
```

As we can see, the function first prevents drawing virtual game field points, after that it will convert from game field x, y to physical x, y of the framebuffer and then call appropriate functions to render block images based on their color. To map from game field x, y to physical x, y we have already known before the offset of the game field to the top left point of the physical screen when we design the UI with Canva. After that to draw each block in the mapped physical position we just need images of different colored blocks:

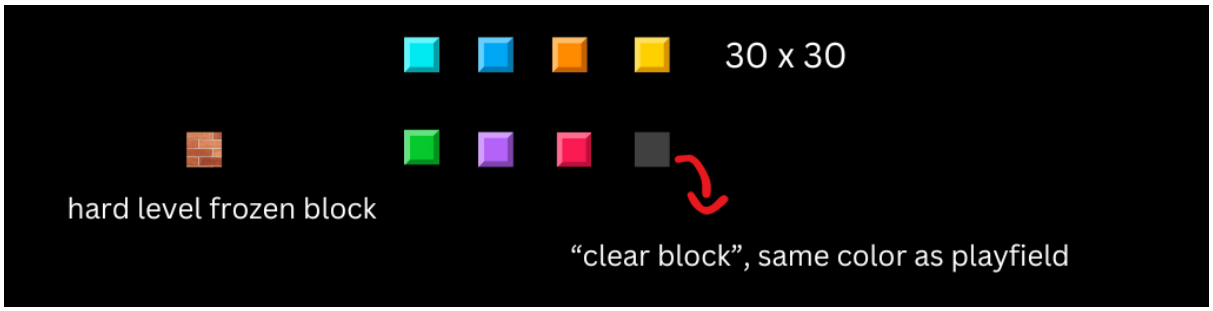

Figure 36. Different images of coloured block to render.

Game Statistics

During game play, not only ACK and NAK command is printed out, some of the game events will also be printed out the CLI for debugging.

Score changes when clear rows and new piece spawned events:

```
FIRE_OS > CLI > GAME > ACK: DOWN

- 2 rows cleared! Score +40

FIRE_OS > CLI > GAME >

New piece spawned:
+ Shape: J
+ Color: BLUE
+ Rotation: 0 degrees
+ Center point: x = 4, y = 2
```

Figure 37. Rows clear and new piece spawned statistics.

Game over events:

```
FIRE_OS > CLI > GAME > ACK: DOWN


**************************************
**************************************

GAME OVER!
Game mode: Hard
Final scores: 40
Rows clear: 4
Spawned pieces: 33
Total commands received in game (both ACK and NAK): 396
Total rotation commands: 17
Total go left commands: 88
Total go right commands: 72
Total go down commands: 208

**************************************
**************************************


FIRE_OS > CLI > GAME > FIRE_OS > CLI > GAME > ACK: DOWN
```

Figure 38. Game over statistics.

Game result discussion

Implementation

- The project successfully implemented a classic Tetris game, including core features such as moving pieces left, right, and down, as well as piece rotation.
- A unique scoring system was introduced to maintain player engagement and add a strategic element to gameplay.
- Three distinct difficulty levels were integrated, offering increasing challenges to players.
- A novel "frozen" block feature was introduced, adding complexity to the gameplay compared to the traditional version.
- The game was developed using an object-oriented programming approach (struct-based in C), which enhances code maintainability and modularity.
- A non-blocking timer function with reset capability was thoroughly implemented, ensuring a smooth and responsive user experience.
- The user interface was carefully designed, with multiple team meetings dedicated to refining its visual appeal and functionality.

Testing Results

- Players can successfully navigate the game using the WASD keys or arrow keys.
- Interaction with the game, including moving and rotating pieces, is fully functional with both key sets.
- The game operates smoothly across all difficulty levels, with noticeable increases in speed at higher difficulty settings.
- In the hard difficulty mode, frozen blocks appear as expected, adding further challenge.
- The random number generator works effectively, producing random pieces with varying shapes, colors, and rotation angles.
- Upon reaching the losing state, the game displays a "game over" popup as intended.
- The score is updated accurately whenever a row is cleared, adhering strictly to the predefined scoring formula.

Limitations

- The game has undergone thorough testing, and all identified bugs have been resolved. As of now, no known bugs remain.
- The game currently lacks sound effects, which could be implemented in the future to enhance the player's experience.
- The game does not include animations, so future improvements could focus on adding animations to make the gameplay more dynamic and visually engaging.

V. CONCLUSION

This project involved the successful development of a command-line interface, image and video display functionalities, and a Tetris game application for a bare-metal operating system for the Raspberry Pi. The work necessitated a profound comprehension of low-level hardware interactions, system timers, and UART connection, compelling us to acquire and implement essential features in a restricted context without the utilization of external libraries.

Our team acquired substantial practical knowledge in embedded system development, namely in connecting with hardware components like UART and utilizing mailboxes for communication with the GPU. We refined our abilities in user interface implementation, timing functions, and non-blocking event processing, enabling us to create a responsive and interactive game. The project instructed us on the efficient management of resources inside a bare-metal system and underscored the significance of collaboration, as each member contributed to various components of the operating system.

We individually acquired the skills to develop efficient and scalable code in C, manage real-time events, and design and implement game algorithms within the limitations of the Raspberry Pi. We examined sophisticated topics such random number generation, struct-based object-oriented programming in C, and non-blocking timers to enhance performance.

This project enhanced our comprehension of embedded systems and provided practical insights into developing a fully operational system with constrained resources. The communication among team members was crucial in overcoming the problems encountered during the assignment, and the experience has endowed us

with abilities that will be beneficial in future projects and professional endeavors in embedded systems and software development.

VI. Project Presentation

Project presentation can be found at: https://youtu.be/froshQd_5R0

VII. References

[1] *GitHub*. (2014). Retrieved 9 23, 2024, from https://github.com/raspberrypi/documentation/blob/develop/documentation/asciidoc/computers/raspberry-pi/revision-codes.adoc

[2] GNU. (n.d.). *Character Escapes - Using the GNU Compiler Collection (GCC)*. Retrieved 9 22, 2024, from https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Character-Escapes.html#Character-Escapes

[3] Humby, M. T. (2016, 1 25). *Algorithms 10: Abstract data types – Stack*. (The Open University) Retrieved 9 23, 2024, from https://learn1.open.ac.uk/mod/oublog/viewpost.php?post=162582

[4] Mirek, G. (2017, 12 21). Retrieved 9 23, 2024, from https://dzone.com/articles/ring-buffer-a-data-structure-behind-disruptor

[5] Said Sryheni, G. P. (2024, 3 18). *Tetris Piece Rotation Algorithm*. Retrieved 9 23, 2024, from https://www.baeldung.com/cs/tetris-piece-rotation-algorithm

[6] Zsh. (n.d.). Retrieved 9 23, 2024, from https://wiki.archlinux.org/title/Zsh#Example