RMIT UNIVERSITY

EEET2490 – Embedded System: OS and Interfacing, Semester 2024-1

Assessment 2 – Individual Assignment Report

ADDITIONAL FEATURES FOR BARE METAL OS

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

Student name: Seokyung Kim

Student ID: s3939114

Date: 2024/04/30

TABLE OF CONTENTS

I. INTRODUCTION

1-1. Overview of the Assessment

In the realm of embedded systems, the development of a bare metal operating system (OS) stands as a cornerstone for cultivating practical skills essential for real-world applications. This report delves into the EEET2490 assessment task for Semester 2024-1 at RMIT University, which challenges students to expand their embedded OS development capabilities. The assignment focuses on implementing advanced features such as a command line interpreter (CLI), ANSI code for terminal formatting, a standard printf function, and handling variable arguments1.

The journey of this assignment begins with the creation of a welcoming interface for the OS, followed by the intricate task of developing a CLI that is both functional and user-friendly. The CLI is designed to support auto-completion, command history, and a set of basic commands to facilitate user interaction. Additionally, the report discusses the enhancement of the UART driver to support various configurations, thereby enabling a more versatile communication interface.

Furthermore, the assignment encourages students to engage with common sensors through TinkerCad Circuits, providing a platform for understanding the operational principles, pin functions, and potential applications of these sensors. This hands-on experience is invaluable for comparing virtual prototyping with actual market-available devices.

This report not only presents the technical accomplishments but also reflects on the learning journey, drawing parallels between the skills developed during the assignment and those sought after in the job market for embedded software engineers. Through this introspective analysis, the report highlights the challenges, opportunities, and new insights gained, setting a path for future exploration and skill enhancement in the field of embedded systems.

1-2. Project Structure Overview

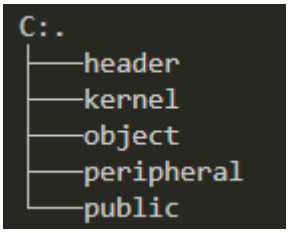

Figure 1: Project Structure

The project, named KimOS, is structured into several directories to organize its components effectively:

- **header**: Contains header files used across the project.
- **kernel**: Holds the source files (.c and .S) related to the kernel.
- **object**: This directory is designated for storing compiled object files (.o) and the final kernel image (kernel8.img).
- **peripheral**: Includes peripheral-specific source files, such as uart0.c.
- **public**: Reserved for any public-facing documents or resources related to the project.

```
CFILES = $(wildcard ./kernel/*.c)
OFILES = $(CFILES:./kernel/%.c=./object/%.o)
GCCFLAGS = -Wall -O2 -ffreestanding -nostdinc -nostdlib

all: clean uart0_build ./object/kernel8.img run

./object/boot.o: ./kernel/boot.S
    aarch64-none-elf-gcc $(GCCFLAGS) -c ./kernel/boot.S -o ./object/boot.o

uart0_build: ./peripheral/uart0.c
    aarch64-none-elf-gcc $(GCCFLAGS) -c ./peripheral/uart0.c -o ./object/uart.o

./object/%.o: ./kernel/%.c
    aarch64-none-elf-gcc $(GCCFLAGS) -c $< -o $@

./object/kernel8.img: ./object/boot.o ./object/uart.o $(OFILES)
    aarch64-none-elf-ld -nostdlib ./object/boot.o ./object/uart.o $(OFILES) -T ./kernel/link.ld -o ./object/kernel8.elf
    aarch64-none-elf-objcopy -O binary ./object/kernel8.elf ./object/kernel8.img

clean:
    del .\object\kernel8.elf .\object\*.o .\object\*.img

# Run emulation with QEMU
run:
    qemu-system-aarch64 -M raspi3 -kernel ./object/kernel8.img -serial stdio
```

Figure 2: Makefile

The Makefile orchestrates the compilation and linking process to build the final kernel image. Here's a breakdown of its main targets and functionalities:

- ./object/boot.o: Compiles the boot.S file into the boot.o object file.
- uart0_build: Compiles uart0.c into the uart.o object file.
- ./object/%.o: Compiles each .c file in the kernel directory into its corresponding .o file in the object directory.
- ./object/kernel8.img: Links all object files into a single kernel8.elf executable, which is then converted to a binary kernel image named kernel8.img.
- clean: Removes all compiled object files, the final kernel image, and any temporary files.
- run: Executes QEMU with the Raspberry Pi 3 (or Raspberry Pi 4) machine type, using the generated kernel8.img file as the kernel image and configuring the serial interface for standard I/O.

This structured approach and Makefile configuration streamline the development and build processes, ensuring the project's components are organized and compiled efficiently.

II. FEATURES FOR BARE METAL OS

1. Default Configuration of the System

KimOS, a Bare Metal Operating System, utilizes UART for its default configuration. Among various types of the PL011 UART, UART0 has been chosen to manage various operations including output printing, terminal input retrieval, and UART configuration. Default settings for baud rates, data bits, stop bits, parity, and handshaking mode have been initialized as follows:

Baud Rates

The baud rate is set to 115200, determined by the values of IBRD (26) and FBRD (3). These values are dynamically passed through the uart_init function as parameters, specifically 'ibrd' and 'fbrd'. Within the uart_init function, the 'ibrd' and 'fbrd' values are assigned to UART0_IBRD and UART0_FBRD, respectively.

```
void main()
{

    uart_init(26, 3);
    display_welcome_msg();
    display_prompt();

    while (1)···
}
```

```
UART0_IBRD = ibrd;
UART0_FBRD = fbrd;
```

Figure 3: uart_init call with baud rates parameters

Figure 4: UART0_IBRD and UART0_FBRD assignment

Data Bits

```
int DATA_BITS = 8;
```

```
UART0_LCRH = set_lcfh_val();
```

Figure 5: DATA_BITS global variable **Figure 6: set_lcfh_val call for UART0_LCRH**

The number of data bits is initially set to 8 as a global variable DATA_BITS during the first call to uart_init. Within the uart_init function, this value is configured in the UART0_LCRH register using a separate function called set_lcfh_val, which handles all LCRH-relevant configurations such as data bits, stop bits, parity, and handshaking mode. Inside set_lcfh_val, the value of UART0_LCRH varies depending on the DATA_BITS value. It sets UART0_LCRH_FEN and the appropriate number of bits, formatted as UART0_LCRH_WLEN_#BIT, where # represents 5, 6, 7, or 8.

```
// Data bits
if (DATA_BITS == 5)
{
    lcfh |= UART0_LCRH_FEN | UART0_LCRH_WLEN_5BIT;
}
else if (DATA_BITS == 6)
{
    lcfh |= UART0_LCRH_FEN | UART0_LCRH_WLEN_6BIT;
}
else if (DATA_BITS == 7)
{
    lcfh |= UART0_LCRH_FEN | UART0_LCRH_WLEN_7BIT;
}
else if (DATA_BITS == 8)
{
    lcfh |= UART0_LCRH_FEN | UART0_LCRH_WLEN_8BIT;
}
```

Figure 7: DATA_BITS re-assignment

```
#define UART0_LCRH_WLEN_5BIT (0 << 5)
#define UART0_LCRH_WLEN_6BIT (1 << 5)
#define UART0_LCRH_WLEN_7BIT (2 << 5)
#define UART0_LCRH_WLEN_8BIT (3 << 5)
#define UART0_LCRH_FEN (1 << 4)  /* FEN = enable FIFOs */
```

Figure 8: LCRH_FEN & LCRH_WLEN_#BIT

Stop Bits

```
int STOP_BITS = 2;
```

Figure 9: STOP_BITS global variable

```
// Stop bits
if (STOP_BITS == 1)
{
    lcfh &= ~UART0_LCRH_STP2;
}
else if (STOP_BITS == 2)
{
    lcfh &= ~UART0_LCRH_STP2;
    lcfh |= UART0_LCRH_STP2;
}
```

Figure 10: STOP_BITS re-assignment

The number of stop bits is initially set to 2 as a global variable STOP_BITS during the first call to uart_init. It follows a similar process as data bits, being configured within the set_lcfh_val function using the dynamic value of STOP_BITS.

Parity

```
int PARITY = 0;
```

Figure 11: PARITY global variable

```
// Parity
if (PARITY == 0) // None
{
    lcfh &= ~(UART0_LCRH_PEN | UART0_LCRH_EPS);
}
else if (PARITY == 1) // Odd
{
    lcfh |= UART0_LCRH_PEN;
    lcfh &= ~UART0_LCRH_EPS;
}
else if (PARITY == 2) // Even
{
    lcfh &= ~(UART0_LCRH_PEN | UART0_LCRH_EPS);
    lcfh |= UART0_LCRH_PEN | UART0_LCRH_EPS;
}
```

Figure 12: PARITY re-assignment

The parity is initially set to none as a global variable PARITY, where the value 0 represents none, 1 represents odd, and 2 represents even parity. This configuration occurs during the first call to uart_init. Similar to the configurations mentioned above, it is managed within the set_lcfh_val function using the dynamic value of PARITY, using UART0_LCRH_EPS and UART0_LCRH_PEN.

```
#define UART0_LCRH_EPS (1 << 2)
#define UART0_LCRH_PEN (1 << 1)
```

Figure 13: EPS & PEN

Handshaking

```
int HANDSHAKING = 0;
```

Figure 14: HANDSHAKING global variable

```
/* Enable UART0, receive, and transmit */
UART0_CR = 0x301; // enable Tx, Rx, FIFO

if (HANDSHAKING)
{
    UART0_CR |= UART0_CR_RTSEN | UART0_CR_CTSEN;
}
else
{
    UART0_CR &= ~(UART0_CR_RTSEN | UART0_CR_CTSEN);
}
```

Figure 15: HANDSHAKING re-assignment

The handshaking mode is initially set to off as a global variable HANDSHAKING, where the value 0 represents off, and 1 represents on. This configuration is established during the first call to uart_init. It is managed through the UART0_CR register, utilizing the dynamic value of HANDSHAKING to enable or disable UART0_CR_RTSEN and UART0_CR_CTSEN.

```
#define UART0_CR_CTSEN (1 << 15)
#define UART0_CR_RTSEN (1 << 14)
```

Figure 16: CR_CTSEN & CR_RTSEN

2. Welcome message and Command Line Interpreter (CLI)

2-1. Welcome Message

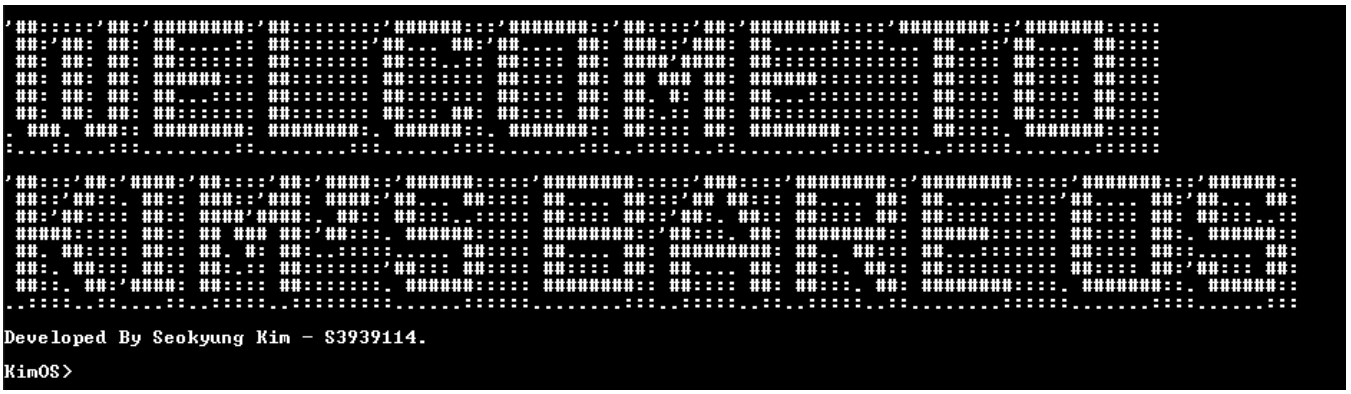

Figure 17: Welcome message

```
void display_welcome_msg()
{

    uart_puts("\n\n'##:::::'##:'########:'##::::::::'######:::'#######::'##::::'##:'########::::'########::'#######::::\n"
              " ##:'##: ##: ##......:: ##:::::::'##... ##:'##.... ##: ###::'###: ##.....:::... ##..::'##.... ##::::\n"
              " ##: ##: ##: ##::::::: ##::::::: ##::..::: ##:::: ##: ####'####: ##:::::::::::: ##:::: ##:::: ##::::\n"
              " ##: ##: ##: ######::: ##::::::: ##::::::: ##:::: ##: ## ### ##: ######:::::::: ##:::: ##:::: ##::::\n"
              " ##: ##: ##: ##...:::: ##::::::: ##::::::: ##:::: ##: ##. #: ##: ##...::::::::: ##:::: ##:::: ##::::\n"
              " ##: ##: ##: ##::::::: ##::::::: ##::: ##: ##:::: ##: ##:.:: ##: ##:::::::::::: ##:::: ##:::: ##::::\n"
              ". ###. ###:: ########: ########:. ######::. #######:: ##:::: ##: ########:::::: ##::::. #######:::::\n"
              ":...::...:::........::........:::......::::.......:::..:::::..::........:::::::..::::::.......::::::\n\n");

    uart_puts("'##::::'##:'####:'##::::'##:'####::'######::::'########::::'###::::'########::'########::::'#######::'######::\n"
              " ##:'##:. ##:: ###::'###: ####:'##... ##::::: ##.... ##::'## ##::: ##.... ##: ##.....::::'##.... ##:'##... ##:\n"
              " ##:'##::: ##:: ####'####:. ##:: ##:::..:::::: ##:::: ##:'##:. ##:: ##:::: ##: ##:::::::: ##:::: ##: ##:::..::\n"
              " #####:::: ##:: ## ### ##:'##::: ######::::::: ########::'##::. ##: ########:: ######::::: ##:::: ##:. ######::\n"
              " ##. ##::: ##:: ##. #: ##:...:::::... ##:::::: ##.... ##: #########: ##.. ##::: ##...:::::: ##:::: ##::..... ##:\n"
              " ##:. ##:: ##:: ##:.:: ##:::::'##::: ##:::::: ##:::: ##: ##.... ##: ##::. ##:: ##:::::::::: ##:::: ##:'##::: ##:\n"
              " ##::. ##:'####: ##:::: ########:. ######:::: ########:: ##:::: ##: ##:::. ##: ########::::. #######::. ######::\n"
              "..::::..::....::..:::::........:::......:::::........:::..:::::..::..:::::..::........::::::.......::::......:::\n\n");

    uart_puts("Developed By Seokyung Kim - S3939114.\n\n");
}
```

Figure 18: display_welcome_msg function

The welcome message is displayed when KimOS is successfully built from the Makefile. This is achieved by calling the function `display_welcome_msg` in the `main` function, which presents the message "WELCOME TO KIM'S BARE OS" in ASCII art format sourced from text [1]. Additionally, developer information is included: "Developed By Seokyung Kim -S3939114.".

2-2. Operating System (OS) Name


```
void display_prompt()
{
    uart_puts("KimOS> ");
}
```

Figure 19: KimOS output in terminal **Figure 20: display_prompt**

'KimOS' has been designated as the official name of the operating system (OS), appearing as the text "KimOS>" in terminals while awaiting user commands. This functionality has been implemented within the display_prompt() function.

2-3. Command Line Interpreter (CLI)

A Command Line Interpreter (CLI) is a software program that allows users to interact with a computer system or application by entering commands into a text-based interface, an interpreter of commands [2].

In KimOS, the ‘main’ function and `execute_command` function are mainly used to manage CLI commands. The main function serves as the entry point for KimOS. It continuously reads user input characters from the UART and processes them to execute commands using the `execute_command` function.

The `execute_command` function interprets the command entered by the user and executes the corresponding functionality. It matches the entered command with the supported commands in KimOS and invokes the respective functions to perform actions such as displaying help information, clearing the console, setting configurations, and handling other user-defined commands.

Figure 21: main function

Figure 22: execute_command function

3. Basic Commands

3-1. Help Commands

Figure 23: help_command function

The "help" command in KimOS is implemented through the `help_command` function. This function takes a parameter `command`, which can either be an empty character or the name of a supported command, such as "setparity".

- If the `command` parameter is an empty character, it signifies that the user wants to see a brief description of all the supported commands.
- If the `command` matches any of the supported commands, it indicates that the user wants to see a detailed description of that specific command, which is provided as a parameter to the `help_command` function.

The figures below illustrate the types of descriptions provided by the "help" command:

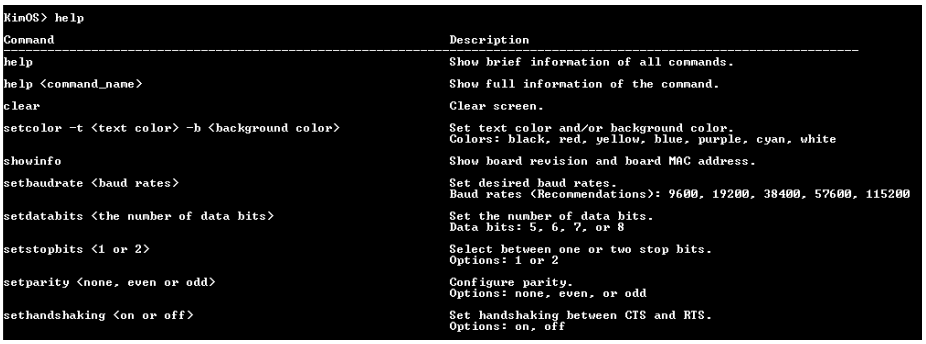

Figure 24: Brief Description of All Commands

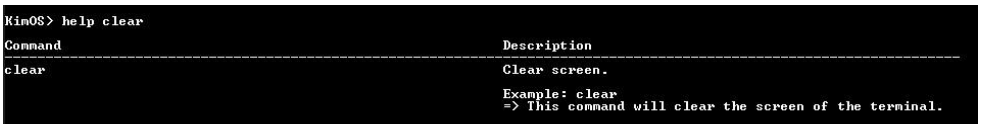

Figure 25: Detailed Description of 'clear' command

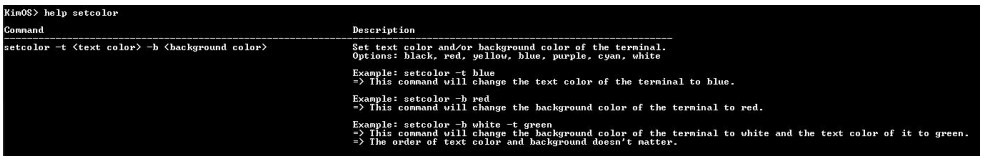

Figure 26: Detailed Description of 'setcolor' command

Figure 27: Detailed Description of 'showinfo' command

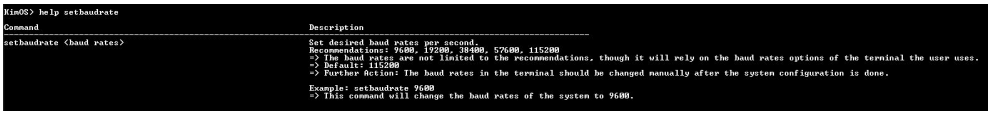

Figure 28: Detailed Description of 'setbaudrate' command

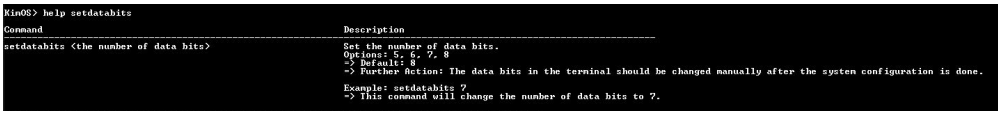

Figure 29: Detailed Description of 'setdatabits' command

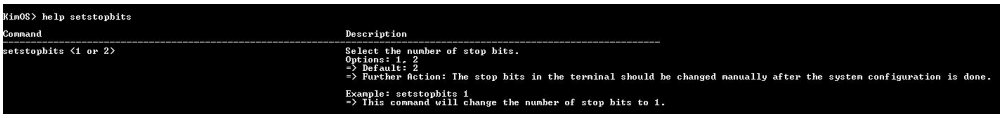

Figure 30: Detailed Description of 'setstopbits' command

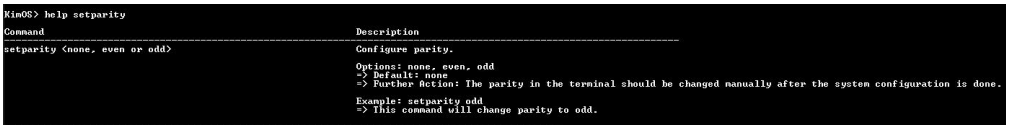

Figure 31: Detailed Description of 'setparity' command

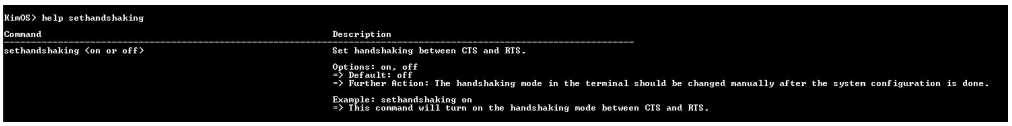

Figure 32: Detailed Description of 'sethandshaking' command

3-2. Clear Command

```
void clear_command()
{
    uart_puts("\033[2J\033[H"); // ANSI escape sequence to clear the screen
}
```

Figure 33: clear_command function

The "clear" command is facilitated by the `clear_command` function. This function sends the escape sequence "\033[2J\033[H" to the terminal, which instructs it to clear the screen and move the cursor to the top left corner.

The sequence "\033[2J\033[H" is a combination of two ANSI escape codes [3]:

- **\033[2J**: This code clears the screen or console. It clears the part of the screen from the cursor to the end of the screen.
- **\033[H**: This code moves the cursor to the top left corner of the screen or console. It's often referred to as moving the cursor to the home position.

Therefore, when used in combination, "\033[2J\033[H" clears the screen and positions the cursor at the beginning of the screen. Users can utilize this command to refresh the terminal they are currently using.

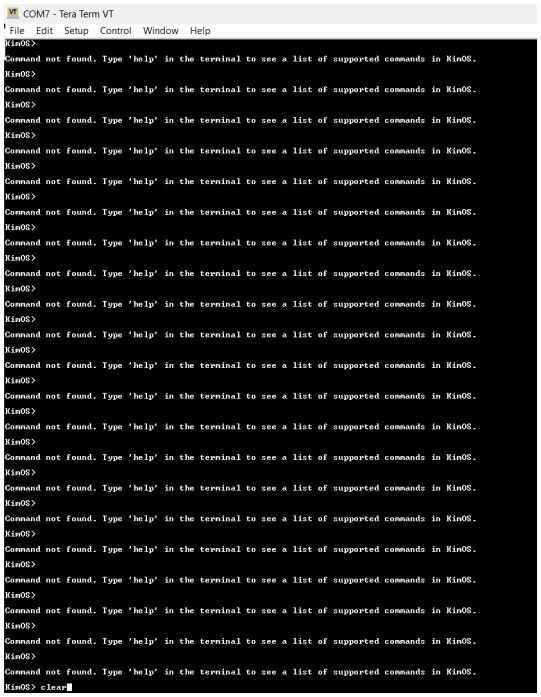

Figure 34: Before 'clear' command

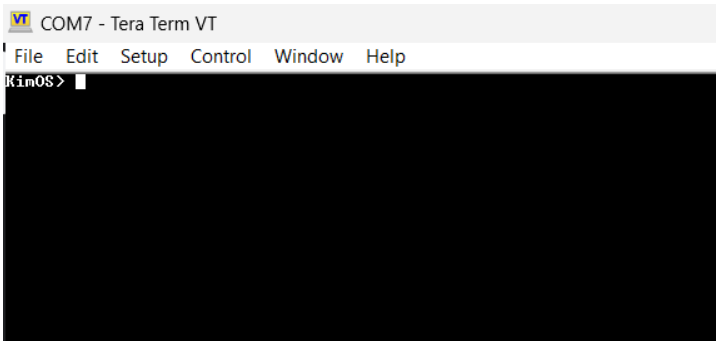

Figure 35: After 'clear' command

3-3. Setcolor Command

The "setcolor" command enables users to customize the text or background color of the terminal. In KimOS, the supported colors include black, red, yellow, blue, purple, cyan, and white, each defined using ANSI Escape Codes[4]. These codes distinguish between text and background colors. Changing the color involves invoking the `change_color` function, which prints the corresponding ANSI Escape Codes to the terminal, thus affecting the desired color change.

```
#define BLACK_TEXT "\033[30m"
#define BLACK_BACKGROUND "\033[40m"

#define RED_TEXT "\033[31m"
#define RED_BACKGROUND "\033[41m"

#define YELLOW_TEXT "\033[33m"
#define YELLOW_BACKGROUND "\033[43m"

#define BLUE_TEXT "\033[34m"
#define BLUE_BACKGROUND "\033[44m"

#define PURPLE_TEXT "\033[35m"
#define PURPLE_BACKGROUND "\033[45m"

#define CYAN_TEXT "\033[36m"
#define CYAN_BACKGROUND "\033[46m"

#define WHITE_TEXT "\033[37m"
#define WHITE_BACKGROUND "\033[47m"

int change_color(char *text, char *background);
```

Figure 36: ANSI Escape Codes for supported colors

```
int change_color(char *text, char *background)
{
    const char *colors[] = {"BLACK", "RED", "YELLOW", "BLUE", "PURPLE", "CYAN", "WHITE"};

    // Match text color
    for (int i = 0; i < sizeof(colors) / sizeof(colors[0]); i++)
    {
        if (compare_string(text, colors[i]) == 0)
        {
            switch (i)...
            break;
        }
    }

    // Match background color
    for (int i = 0; i < sizeof(colors) / sizeof(colors[0]); i++)
    {
        if (compare_string(background, colors[i]) == 0)
        {
            switch (i)...
            break;
        }
    }

    return 0;
}
```

Figure 37: change_color function

In KimOS, users can change either the text color, background color, or both simultaneously by specifying '-b' or '-t' in the command preceding the color. The order of these flags does not affect the resulting color change.

```
KimOS> setcolor -b white -t red
Color has changed!!

KimOS> setcolor -t blue
Color has changed!!

KimOS> setcolor -b yellow
Color has changed!!

KimOS> setcolor -t white -b black
Color has changed!!

KimOS>
```

Figure 38: 'setcolor' commands

3-4. Showinfo Command

```
KimOS> showinfo

---------------------------------------- BOARD INFORMATION ----------------------------------------
Board MAC Address: D8:3A:DD:3D:48:93
Board Revision: C03115

---------------------------------------------------------------------------------------------------
```

Figure 39: 'showinfo' command

The 'showinfo' command in KimOS provides users with essential information about a Raspberry Pi board, including both the MAC address and the board revision. The MAC address is formatted as ##:##:##:##:##:##,

indicating six pairs of hexadecimal digits separated by colons. On the other hand, the board revision is displayed as ######, where each '#' can denote either an alphabetical character or a single numerical digit.

To obtain the MAC address of the board in KimOS, the 'get_mac_address_info' function dispatches a request via a Raspberry Pi mailbox using specific tags for the request, MAC address, and end of the tag, all encapsulated within the buffer content denoted as mBuf in the KimOS system [6]. Similarly, the 'get_revision_info' function retrieves the board's revision by sending a request through the mailbox using tags for the request, revision, and end of the tag. Once the requests are defined, the 'mbox_call' function is invoked with the mBuf parameter containing the request information, and MBOX_CH_PROP as the channel parameter, representing a property tag, to execute the mailbox call.

```
#define MBOX_TAG_LAST 0

#define MBOX_TAG_REVISION 0x00010002
#define MBOX_TAG_MAC_ADDRESS 0x00010003
```

Figure 40: Tags for end of the tag, revision and mac address

```
// Request/Response code in Buffer content
#define MBOX_RESPONSE 0x80000000
#define MBOX_REQUEST 0
```

Figure 41: Request and Response code

```
void get_mac_address_info()
{

    // Send request for board revision
    mBuf[0] = 8 * 4;        // Length of the buffer
    mBuf[1] = MBOX_REQUEST; // Request code

    mBuf[2] = MBOX_TAG_MAC_ADDRESS; // Tag: Get board MAC address
    mBuf[3] = 8;                    // Buffer size
    mBuf[4] = 0;                    // Request/response code
    mBuf[5] = 0;                    // Value buffer

    mBuf[6] = 0;             // Tag: Get board MAC address
    mBuf[7] = MBOX_TAG_LAST; // End of tags

    // Call mailbox_call function
    if (mbox_call(ADDR(mBuf), MBOX_CH_PROP))
    {

        uart_print_mac_address(mBuf[5], mBuf[6]);
    }
    else
        (uart_puts("Board MAC address is not found."));
}
```

Figure 42: get_mac_address_info function

```
void get_revision_info()
{

    // Send request for board revision
    mBuf[0] = 8 * 4;        // Length of the buffer
    mBuf[1] = MBOX_REQUEST; // Request code

    mBuf[2] = MBOX_TAG_REVISION; // Tag: Get board tag of revision
    mBuf[3] = 8;                 // Buffer size
    mBuf[4] = 0;                 // Request/response code
    mBuf[5] = 0;                 // Value buffer

    mBuf[6] = 0;             // Tag: Get board MAC address
    mBuf[7] = MBOX_TAG_LAST; // End of tags

    // Call mailbox_call function
    if (mbox_call(ADDR(mBuf), MBOX_CH_PROP))
    {
        uart_print_revision(mBuf[5]);
    }
    else
        (uart_puts("Board Revision is not found."));
}
```

Figure 43: get_revision_info function

4. Further Development of UART Driver

4-1. Setbaudrate Command

The 'setbaudrate' command in KimOS enables the configuration of various baud rates, including common options like 9600, 19200, 38400, 57600, and 115200 bits per second. When invoked, this command prompts the user to input their desired baud rate. Subsequently, the command calculates the values of the integer and fractional parts of the baud rate divisor using the formula: baud rate divisor = UART's clock frequency / (16 * baud rate), where IBRD represents the integer part and FBRD represents the fractional part of the divisor. These calculated values are then assigned to the UART0_IBRD and UART0_FBRD registers, respectively. Finally, the new baud rate settings are passed as parameters to the uart_init function, effectively resetting the UART configuration with the updated baud rate value.

```
/* IBRD = Integer part of Baud Rate Divisor */
/*    bottom 16 bits */
/*    UART must be disabled to change */
#define UART0_IBRD (*(volatile unsigned int *)(UART0_BASE + 0x24))
/* FBRD = Fractional part of Baud Rate Divisor */
/*   bottom 5 bits */
/*   Baud rate divisor BAUDDIV = (FUARTCLK/(16 Baud rate)) */
/*    UART must be disabled to change */
#define UART0_FBRD (*(volatile unsigned int *)(UART0_BASE + 0x28))
```

Figure 44: IBRD and FBRD registers

```
// Calculate the baud rate divisor
float BAUDDIVs = UART0_CLOCK_FREQ / (16.0f * baud_rate);

int ibrd = (int)BAUDDIVs;
int fbrd = (int)(((BAUDDIVs - ibrd) * 64) + 0.5);
```

Figure 45: Baud rate calculation

After updating the baud rate value and resetting the UART configuration in KimOS, users must manually adjust the baud rate setting in their terminal to match the new configuration. This manual adjustment is necessary because the baud rate configuration on the KimOS side only affects the transmission from KimOS to the terminal. However, the baud rate setting on the terminal side determines how the terminal interprets the incoming data. If the baud rate settings on both sides do not match, communication between KimOS and the terminal will be disrupted, resulting in garbled or incorrect data being displayed on the terminal.

The following figures illustrate an example process of changing the baud rate from 115200 to 9600:

Figure 46: Terminal baud rates before changing baud rates

```
KimOS> setbaudrate 9600
------------------------------------ Baud Rate Setting ------------------------------------

IBRD before setting: 26
FBRD before setting: 3

IBRD after setting: 312
FBRD after setting: 32

Baud Rate has been set to 9600

Baud rate of the system has been changed. Please manually change the baud rate of your environment.

-------------------------------------------------------------------------------------------

KimOS>
```

Figure 47: After changing baud rates

Figure 48: Terminal baud rates change (Manual)

```
KimOS> setbaudrate 9600
------------------------------------ Baud Rate Setting ------------------------------------

IBRD before setting: 26
FBRD before setting: 3

IBRD after setting: 312
FBRD after setting: 32

Baud Rate has been set to 9600

Baud rate of the system has been changed. Please manually change the baud rate of your environment.

-------------------------------------------------------------------------------------------

KimOS>now I can type!!!!
```

Figure 49: After changing terminal baud rates to 9600

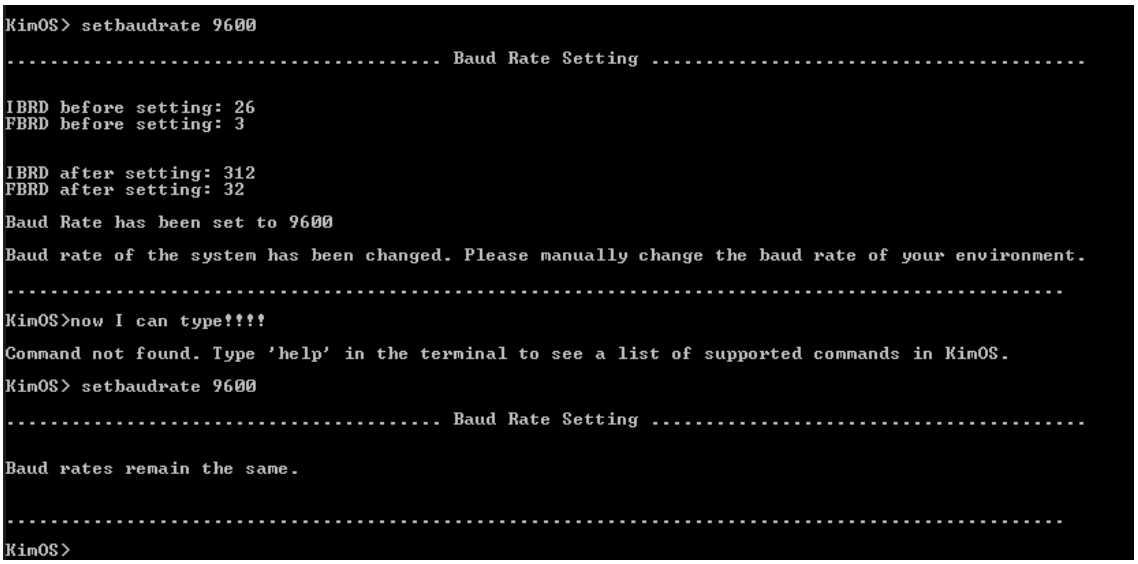

Figure 50: Handling the same baud rate input

4-2. Setdatabits Command

The 'setdatabits' command in KimOS facilitates the configuration of various data bit settings, including options for 5, 6, 7, or 8 data bits. Upon receiving the desired data bits from the user input, the command sets the DATA_BITS global variable to the specified value. As described in the 'Default Configuration of the System' section, the configuration of UART0_LCRH is managed within the uart_init function. Consequently, to apply the new DATA_BITS value, the uart configuration is reset by invoking the uart_init function with the updated data bits setting.

```
switch (data_bits)
{
case 5:
    DATA_BITS = 5;
    break;
case 6:
    DATA_BITS = 6;
    break;
case 7:
    DATA_BITS = 7;
    break;
case 8:
    DATA_BITS = 8;
    break;
default:
    // Invalid number of data bits
    uart_puts("\nInvalid number of data bits.\n");
    display_end();
    return;
}
```

Figure 51: DATA_BITS value update

After updating the data bits value and resetting the UART configuration in KimOS, users must manually adjust the data bits setting in their terminal to match the new configuration. This ensures consistency between KimOS and the terminal, as misaligned data bit settings can disrupt communication and lead to garbled or incorrect data display.

The following figures illustrate an example process of changing the data bits from 8 to 7:

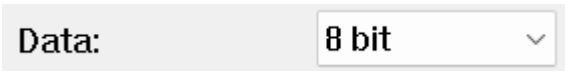

Figure 52: Terminal data bits before changing data bits

```
KimOS> setdatabits 7

------------------------------------ Data Bits Setting ------------------------------------

LDRH before setting data bits: 0x00000078
LDRH after setting data bits: 0x00000058

The number of data bits has been set to  7

The number of data bits has been changed. Please manually change the data bits of your environment.

-------------------------------------------------------------------------------------------

KimOS>
```

Figure 53: After changing data bits

Figure 54: Terminal data bits change (Manual)

```
-------------------------------------------------------------------------------------------

KimOS> setdatabits 7

------------------------------------ Data Bits Setting ------------------------------------

LDRH before setting data bits: 0x00000078
LDRH after setting data bits: 0x00000058

The number of data bits has been set to  7

The number of data bits has been changed. Please manually change the data bits of your environment.

-------------------------------------------------------------------------------------------

KimOS>       now I can type!!!
```

Figure 55: After changing terminal data bits to 7

```
KimOS> setdatabits 7

------------------------------------ Data Bits Setting ------------------------------------

LDRH before setting data bits: 0x00000078
LDRH after setting data bits: 0x00000058

The number of data bits has been set to  7

The number of data bits has been changed. Please manually change the data bits of your environment.

-------------------------------------------------------------------------------------------

KimOS>       now I can type!!!

Command not found. Type 'help' in the terminal to see a list of supported commands in KimOS.

KimOS> setdatabits 7

------------------------------------ Data Bits Setting ------------------------------------

Data bits remain the same.


-------------------------------------------------------------------------------------------

KimOS>
```

Figure 56: Handling the same data bits input

4-3. Setstopbits Command

The 'setstopbits' command in KimOS facilitates the configuration of various stop bit settings, including options for 1 or 2 stop bits. Upon receiving the desired stop bits from the user input, the command sets the STOP_BITS global variable to the specified value. As described in the 'Default Configuration of the System' section, the configuration of UART0_LCRH is managed within the uart_init function. Consequently, to apply the new STOP_BITS value, the uart configuration is reset by invoking the uart_init function with the updated stop bits setting.

```
switch (stop_bits)
{
case 1:
    STOP_BITS = 1;
    break;
case 2:
    STOP_BITS = 2;
    break;
default:
    // Invalid number of stop bits
    uart_puts("\nInvalid number of stop bits.\n");
    display_end();
    return;
}
```

Figure 57: STOP_BITS value update

After updating the stop bits value and resetting the UART configuration in KimOS, users must manually adjust the stop bits setting in their terminal to match the new configuration. This ensures consistency between KimOS and the terminal, as misaligned stop bit settings can disrupt communication and lead to garbled or incorrect data display.

The following figures illustrate an example process of changing the stop bits from 2 to 1:

Figure 58: Terminal stop bits before changing stop bits

```
KimOS> setstopbits 1
----------------------------------- Stop Bits Setting -----------------------------------

LDRH before setting stop bits: 0x00000058
LDRH after setting stop bits: 0x00000050

Stop bits have been set to  1

Stop data bits have been changed. Please manually change the stop bits of your environment.

-----------------------------------------------------------------------------------------
KimOS>
```

Figure 59: After changing stop bits

Figure 60: Terminal stop bits change (Manual)

Figure 61: After changing terminal stop bits to 1

Figure 62: Handling the same stop bits input

4-4. Setparity Command

The 'setparity' command in KimOS facilitates the configuration of various types of parity settings, including options for none, odd or even parity. Upon receiving the desired parity from the user input, the command sets the PARITY global variable to the specified value. As described in the 'Default Configuration of the System' section, the configuration of UART0_LCRH is managed within the uart_init function. Consequently, to apply the new PARITY value, the uart configuration is reset by invoking the uart_init function with the updated parity setting.

```
if (compare_string(arg, "none") == 0)
{
    PARITY = 0;
}
else if (compare_string(arg, "odd") == 0)
{
    PARITY = 1;
}
else if (compare_string(arg, "even") == 0)
{
    PARITY = 2;
}
else
{
    uart_puts("\nInvalid parity type. Please use 'none', 'even', or 'odd'.\n");
    display_end();
    return;
}
```

Figure 63: PARITY value update

After updating the parity option and resetting the UART configuration in KimOS, users must manually adjust the parity setting in their terminal to match the new configuration. This ensures consistency between KimOS and the terminal, as misaligned parity settings can disrupt communication and lead to garbled or incorrect data display.

The following figures illustrate an example process of changing the parity from none to odd:

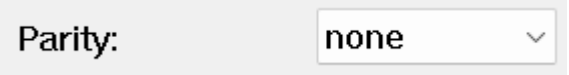

Figure 64: Terminal parity before changing parity

```
KimOS> setparity odd

------------------------------------ Parity Setting ------------------------------------

LDRH before setting parity: 0x00000050
LDRH after setting parity: 0x00000052

Parity has been set to odd

Parity has been changed. Please manually change the Parity of your environment.

----------------------------------------------------------------------------------------

Kim=Mgy
```

Figure 65: After changing parity

Figure 66: Terminal parity change (Manual)

```
KimOS> setparity odd

------------------------------------ Parity Setting ------------------------------------

LDRH before setting parity: 0x00000050
LDRH after setting parity: 0x00000052

Parity has been set to odd

Parity has been changed. Please manually change the Parity of your environment.

----------------------------------------------------------------------------------------

Kim=MgysaeBBBxsasasafdasadafsadafnow I can type normally!!
```

Figure 67: After changing terminal parity to odd

```
KimOS> setparity odd

------------------------------------ Parity Setting ------------------------------------

LDRH before setting parity: 0x00000050
LDRH after setting parity: 0x00000052

Parity has been set to odd

Parity has been changed. Please manually change the Parity of your environment.

----------------------------------------------------------------------------------------

Kim=MgysaeBBBxsasasafdasadafsadafnow I can type normally!!

Command not found. Type 'help' in the terminal to see a list of supported commands in KimOS.

KimOS> setparity odd

------------------------------------ Parity Setting ------------------------------------

Parity remain the same.


----------------------------------------------------------------------------------------

KimOS>
```

Figure 68: Handling the same parity input

4-5. Sethandshaking Command

The 'sethandshaking' command in KimOS configures handshaking settings, such as turning handshaking mode on or off. When users input their desired handshaking mode, the command updates the HANDSHAKING global variable accordingly. The actual configuration adjustment occurs within the uart_init function, where UART0_CR is managed. To implement the new handshaking setting, the uart_init function is called with the updated handshaking value. In the figure titled 'HANDSHAKING value update,' note that the 'new_CR' variable temporarily holds the changed value for user display purposes, without altering the real value assigned to UART0_CR.

```
unsigned int new_CR = UART0_CR;

if (compare_string(arg, "on") == 0)
{
    HANDSHAKING = 1;
    new_CR |= UART0_CR_RTSEN | UART0_CR_CTSEN;
}
else if (compare_string(arg, "off") == 0)
{
    HANDSHAKING = 0;
    new_CR &= ~(UART0_CR_RTSEN | UART0_CR_CTSEN);
}
else
{
    uart_puts("\nInvalid handshaking type. Please use 'on' or 'off'.\n");
    display_end();
    return;
}

if (new_CR == UART0_CR)
{
    // Same handshaking
    uart_puts("\nHandshaking remains the same.\n");
    display_end();
    return;
}
```

```
/* Enable UART0, receive, and transmit */
UART0_CR = 0x301; // enable Tx, Rx, FIFO

if (HANDSHAKING)
{
    UART0_CR |= UART0_CR_RTSEN | UART0_CR_CTSEN;
}
else
{
    UART0_CR &= ~(UART0_CR_RTSEN | UART0_CR_CTSEN);
}
```

Figure 69: HANDSHAKING value update Figure 70: UART0_CR configuration in uart_init function

After updating the handshaking option and resetting the UART configuration in KimOS, users must manually adjust the handshaking setting in their terminal to match the new configuration. This ensures consistency between KimOS and the terminal, as misaligned handshaking settings can disrupt communication and lead to garbled or incorrect data display.

The following figures illustrate an example process of changing the handshaking from off to on:

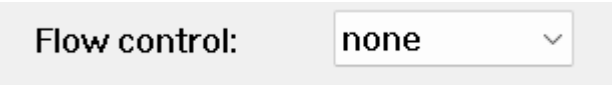

Figure 71: Terminal flow control before changing flow control

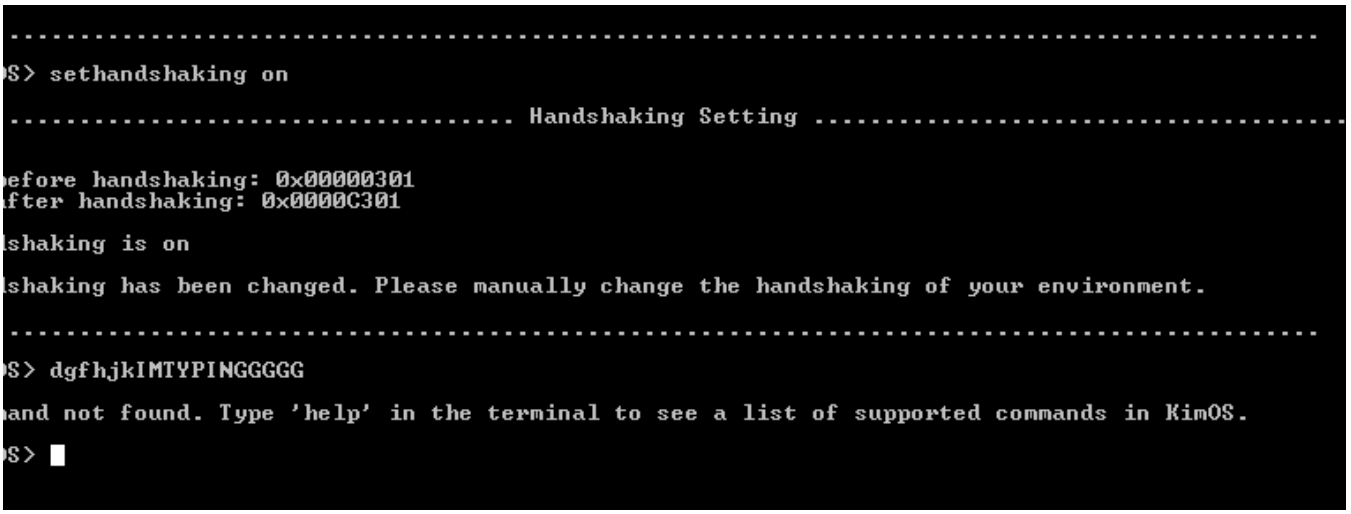

Figure 72: After changing handshaking mode

In the domain of UART (Universal Asynchronous Receiver/Transmitter) communication, the intricacies of handshaking configuration stand out in contrast to other setups. CTS (Clear To Send) serves as a flow control signal, indicating to a UART device that it can transmit data to another device. Conversely, RTS (Request To Send) acts as a flow control signal, serving as an output from a UART device.

Upon activating handshaking mode and disconnecting GPIO 16 (CTS), users encounter a delay in the appearance of their typed input within the terminal until reconnection with the board is established. This delay stems from the UART device awaiting the CTS signal before proceeding with data transmission. When GPIO 16, which is configured as the CTS line, is disconnected, the UART transmitter is essentially put on hold because it no longer receives the hardware signal indicating that the data can be sent. This is because the CTS line is used to prevent data overflow by allowing the receiving device to control the flow of data. If the receiving device's buffer is full, it will not assert the CTS line, signaling the transmitting device to wait before sending more data.

Therefore, when GPIO 16 is disconnected, the transmitter does not receive any signal on the CTS line, leading it to believe that it should not send any data, causing the typed input to not appear in the terminal. Upon reconnection of GPIO 16, the CTS line is asserted again, indicating that the transmitter can resume sending data, which results in the previously typed input 'IMTYPINGGGG' to gradually become visible on the screen.

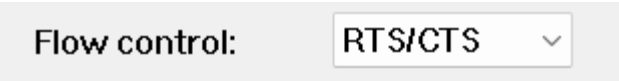

Figure 73: Terminal flow control change (Manual)

Figure 74: Handling the same handshaking mode input

5. Feature Enhancement

5-1. Command History

The history of previously entered commands is stored in a global variable named ‘command_history’, with a predefined maximum size. The user typically navigates through this history using the UP and DOWN arrow keys. However, since these special keys are not directly read by the UART, alternative keys are defined for this purpose: the '_' key serves as the UP arrow, and the '+' key serves as the DOWN arrow.

```
// Command history
char command_history[MAX_HISTORY_SIZE][MAX_COMMAND_LENGTH];
int history_count = 0;
int history_index = 0;
```

Figure 75: Command history global variables

The ‘command_history’ variable persists even after the UART is reset via the ‘uart_init’ function call. Each time the user presses the Enter key, not only does the index tracking the command history reset to the latest command, but the new command is also added to the command_history array through the add_to_history function. This ensures that the latest command is always included in the command history for subsequent access and navigation. When the user presses either the '_' or '+' key, the history corresponding to the current history index is displayed in the terminal through the ‘display_history’ function. Subsequently, the history index is updated in the respective direction based on the key pressed by the user.

```
else if (c == '_') // UP arrow key
{
    if (history_index >= 0)
    {
        display_history();
        if (history_index != 0)
            history_index--;
    }
}
else if (c == '+') // DOWN arrow key
{
    if (history_index <= history_count - 1)
    {
        display_history();
        if (history_index != history_count - 1)
            history_index++;
    }
}
```

Figure 76: ‘_’ and ‘+’ keys handling

```
void display_history()
{
    reset_command_line();

    clear_buffer();

    const char *history = command_history[history_index];

    for (int i = 0; history[i] != '\0'; i++)
    {
        command_buffer[buffer_index++] = history[i];
    }

    uart_send_string(history);
}
```

Figure 77: display_history

```c
if (c == '\n') // Enter key
{
    // Add the command to history
    add_to_history(command_buffer);

    // Null-terminate the command buffer
    command_buffer[buffer_index] = '\0';

    // Execute the command
    execute_command(command_buffer);

    // Reset indexes for next command
    buffer_index = 0;
    history_index = history_count - 1;
    auto_complete_index = 0;

    // Display prompt for next command
    display_prompt();
}
```

Figure 78: '_' and '+' keys handling

```c
void add_to_history(const char *command)
{
    // Update history count
    if (history_count >= MAX_HISTORY_SIZE)
    {
        // Shift commands in history to make space for the new command
        for (int i = MAX_HISTORY_SIZE - 1; i > 0; i--)
        {
            copy_string(command_history[i], command_history[i - 1]);
        }
        copy_string(command_history[history_count], command);
    }
    else
    {
        copy_string(command_history[history_count], command);
        history_count++;
    }
}
```

Figure 79: display_history

5-2. Auto Completion

KimOS features an auto-completion functionality for default commands, accessible by pressing the TAB key. This feature assists users in completing their current command by matching the prefix they've typed with available commands. It keeps track of the current prefix (`cur_prefix`) and the last auto-completion (`last_autocompletion`).

```c
// Command auto completion
char cur_prefix[20];
char last_autocompletion[20];

const char *command_list[] = {
    "help",
    "clear",
    "setcolor",
    "showinfo",
    "setbaudrate",
    "setdatabits",
    "setstopbits",
    "setparity",
    "sethandshaking"};

const int num_commands = sizeof(command_list) / sizeof(command_list[0]);
```

Figure 80: Auto completion global variables

```
}
else if (c == '\t') // TAB key for autocompletion
{
    int next_index = find_next_auto_completion();

    if (next_index != -1)
    {
        reset_command_line();
        clear_buffer();

        copy_string(last_autocompletion, command_list[next_index]);
        copy_string(command_buffer, command_list[next_index]);

        int length = strlen(command_buffer);
        buffer_index = length;

        uart_puts(last_autocompletion);
    }
}
```

Figure 81: TAB key handling

```
int find_next_auto_completion()
{
    int skipped_index = 0;

    if (strlen(last_autocompletion) > 0)
    {
        for (int i = 0; i < num_commands; i++)
        {
            // check index to skip
            if (compare_string(last_autocompletion, command_list[i]) == 0)
            {
                skipped_index = i + 1;
                break;
            }
        }

        // if skipped index is at the end of the command list
        if (skipped_index == num_commands)
            skipped_index = 0;
    }

    for (int i = skipped_index; i < skipped_index + num_commands; i++)
    {
        skipped_index = i % num_commands;
        if (is_prefix(cur_prefix, command_list[i]) == 0)
        {
            return skipped_index;
        }
    }

    return -1; // No auto-completion found
}
```

Figure 82: find_next_auto_completion function

When the user presses TAB without any character input, the system iterates through all commands in the `command_list` array, which contains the default commands. The auto-completion logic checks for the next available command that matches the prefix. If found, it resets the command line, clears the buffer, and copies the matched command to both `last_autocompletion` and `command_buffer`, allowing the user to continue typing or execute the command directly. This feature enhances user interaction by providing a convenient way to navigate and execute commands efficiently.

```
KimOS>
```

Figure 83: Empty command before TAB key

```
KimOS> help
```

Figure 84: Auto-completion for empty command

```
KimOS> s
```

Figure 85: Command before TAB key

```
KimOS> setcolor
```

Figure 86: First auto-completion

```
KimOS> showinfo
```

Figure 87: Second auto-completion

5-3. Handle Wrong Commands

KimOS always provides helpful guidance to users by displaying a message whenever they enter commands that are not supported. This message prompts users to use the 'help' command to see a list of supported commands, ensuring they have access to the available functionalities of KimOS.

```
KimOS> hel
Command not found. Type 'help' in the terminal to see a list of supported commands in KimOS.
```

Figure 88: Handle wrong command

5-4. Handle Deletion in Typing

KimOS offers users the ability to utilize the backspace or delete key while typing a command in the terminal. When either key is pressed, the respective character is removed from the command buffer. Additionally, the cursor moves back one position on the terminal display to visually represent the deletion triggered by the backspace (0x08) or delete (0x7F) key. This functionality is achieved by utilizing the `uart_sendc` function to send the received character (backspace or delete) to the terminal, effectively updating the display.

```
else if (c == 0x08 || c == 0x7F) // Delete or backspace key
{
    // Check if buffer_index is greater than 0 to avoid underflow
    if (buffer_index > 0)
    {
        // Move the cursor back one position
        uart_sendc(c);

        // Remove the last character from the command buffer
        buffer_index--;
    }
}
```

Figure 89: Deletion key handling

6. Summary of features implemented in both Task 1 & 2

Feature Group		Command/ Feature	Implementation	Testing (any issues/limitations)
Basic Commands		help	completed	N/A
		clear	completed	N/A
		setcolor	completed	N/A
		showinfo	completed	N/A
CLI enhancement		OS name in CLI	completed	N/A
		Auto-completion in CLI	completed	N/A
		Command history in CLI	completed	N/A
UART settings	**Baud rate**	setbaudrate	completed	N/A
	Data bits	setdatabits	completed	N/A
	Stop bits	setstopbits	completed	N/A
	Parity	setparity	completed	N/A
	Handshaking	sethandshaking	completed	N/A

Figure 90: Table of the features summary

7. Common Sensors Available in TinkerCad Circuits

The section below outlines three commonly used sensors found in TinkerCad Circuits: the Flex Sensor, the Photodiode, and the TMP36 Temperature Sensor.

7-1. What is TinkerCad?

TinkerCad is an online platform for circuit simulation and 3D design [7]. It allows users to design and simulate circuits using a wide range of electronic components without the need for physical hardware. TinkerCad provides a virtual environment where users can create, test, and iterate on their electronic designs.

7-2. Flex Sensor

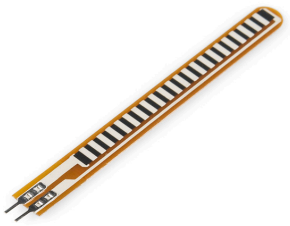

Figure 91: Flex sensor [10]

Flex sensor is a variable resistor whose resistance increases when it is bent [8]. It can be used to measure the bending angle. The flex sensor can be used with Arduino and a voltage divider circuit to scan the value of the sensor.

- **How it works**

  Flex sensors utilize the principle of resistance change in response to bending. They typically consist of a flexible substrate, such as plastic, with conductive material (often carbon) deposited on it in a serpentine pattern. When the sensor is bent, the distance between the conductive traces changes, altering the resistance of the sensor. This change in resistance can be measured to determine the degree of bending.

- **Function of pins**
  - **Signal Pin (Output)**

    This pin serves as the output of the flex sensor and is connected to the circuit where the bending measurement is required. The signal pin provides a variable voltage output based on the degree of bending of the sensor. The voltage output typically changes in a linear fashion relative to the amount of bending.

  - **Vcc (Voltage) Pin (Power)**

    This pin is connected to the positive supply voltage of the circuit (usually +5V or +3.3V). It provides the necessary power for the flex sensor to operate. The voltage supplied should be within the specified operating range of the sensor to ensure accurate and reliable measurements.

  - **Ground Pin**

The ground pin is connected to the ground (0V) reference of the circuit. It completes the electrical circuit and provides a reference point for the voltage measurements. Ensuring a solid ground connection is essential for accurate readings from the flex sensor.

- **Applications with example circuits demonstrated on TinkerCad**
  - **Arduino FSR using Flex Sensor [9]**

    In this example demonstrated in TinkerCad, the integration of a Flex Sensor with an Arduino board enables users to create projects focused on detecting and measuring force or pressure applied to the sensor.

    The Flex Sensor is connected to one of the analog input pins on the Arduino board. Analog pins on Arduino boards can measure voltage levels, which allows for the detection of varying degrees of bending or flexing in the sensor. Additionally, a resistor may be used to create a voltage divider circuit, depending on the specific requirements of the project and the sensor's resistance range.

    Once connected, the Arduino board reads the analog voltage output from the Flex Sensor. This voltage corresponds to the amount of bending or flexing experienced by the sensor. By analyzing this voltage level, users can determine the force or pressure applied to the sensor.

  - **Arduino Flex Sensor and Servo Motor [11]**

    This example in TinkerCad illustrates the utilization of both a Flex Sensor and a Servo Motor with an Arduino board. The Flex Sensor is connected to an analog input pin on the Arduino, similar to the previous example. However, in this setup, the output from the Flex Sensor is used to control the position of a Servo Motor.

    By mapping the bending angle measured by the Flex Sensor to specific servo positions, users can create projects where the servo motor's rotation is directly influenced by the degree of bending detected by the sensor. For instance, bending the Flex Sensor to different angles can cause the servo motor to rotate to corresponding positions, enabling interactive motion control applications.

    This example showcases the versatility of Flex Sensors in combination with other components, such as servo motors, to create dynamic and responsive systems. It demonstrates how sensor inputs can be translated into physical actions, opening up possibilities for projects ranging from robotic arms controlled by gestures to interactive sculptures driven by user movements.

- **Real devices available on the market**
  - **Wearable fitness trackers**

    Flex sensors integrated into wearable fitness trackers play a crucial role in monitoring various aspects of physical activity and health. These sensors are strategically placed on different body parts, such as joints, limbs, or the spine, to accurately track movement patterns during activities like running, cycling, or weightlifting.

    By measuring the degree of bending or flexing in these areas, the tracker can assess exercise intensity, provide feedback on posture and technique, and offer personalized coaching based on the wearer's movement data. This real-time feedback helps users optimize their workouts, reduce the risk of injury, and achieve their fitness goals more effectively.

    Overall, flex sensors enhance the functionality and utility of wearable fitness trackers, empowering users to monitor their physical activity, improve their exercise routines, and maintain optimal health and fitness levels.

- **Differences Between Real Devices and Example Circuits on TinkerCad**

Example circuits using flex sensors, such as those demonstrated in platforms like TinkerCad, are primarily educational tools designed to teach electronics and programming concepts. These circuits showcase the integration of flex sensors with microcontrollers, such as Arduino boards, and other electronic components like servo motors. They allow users to experiment with sensor interfacing, signal processing, and control systems in a simulated environment. The main purpose of these circuits is to provide hands-on learning experiences and encourage exploration and experimentation with sensor-based projects.

On the other hand, real devices like wearable fitness trackers utilize flex sensors for practical fitness monitoring purposes. In these devices, flex sensors are integrated into wearable accessories such as wristbands or clothing to track the movement of various body parts during physical activities. The data collected by the flex sensors is used to monitor motion patterns, measure exercise intensity, and provide feedback on posture or technique. Wearable fitness trackers enable users to track their fitness progress, analyze their performance, and make informed decisions about their workout routines.

In summary, example circuits using flex sensors focus on educational purposes and hands-on learning experiences, while real devices like wearable fitness trackers serve practical applications in fitness monitoring and performance tracking.

7-3. Photodiode

Figure 92: Photodiode [19]

Photodiode is a semiconductor device that converts light into an electrical current. The current is generated when photons are absorbed in the photodiode. Photodiodes may contain optical filters, built-in lenses, and may have large or small surface areas. Photodiodes usually have a slower response time as their surface area increases. The common, traditional solar cell used to generate electric solar power is a large area photodiode [12].

- **How it works**

  A photodiode is a semiconductor device that generates a flow of current when exposed to light. When photons (light particles) strike the photodiode's semiconductor material, they create electron-hole pairs, generating a photocurrent proportional to the intensity of the incident light.

- **Function of pins**
  - **Anode (Positive) Pin**

    The anode pin of the photodiode is connected to the positive terminal of the circuit. When exposed to light, the photodiode generates a photocurrent that flows from the anode to the cathode. It serves as the output terminal for the generated current.

  - **Cathode (Negative) Pin**

    This pin is connected to the negative terminal of the circuit or ground. It completes the electrical circuit and provides a path for the photocurrent generated by the photodiode. The voltage across the photodiode is typically measured between the anode and cathode terminals.

- **Applications with example circuits demonstrated on TinkerCad**
  - **Light Sensor (Photoresistor) With Arduino [13]**

In this TinkerCad example, a Light Sensor (Photoresistor) is interfaced with an Arduino board to create a light sensing application. The photoresistor, also known as a light-dependent resistor (LDR), exhibits a change in resistance based on the intensity of incident light. When connected to an analog input pin on the Arduino, the photoresistor's resistance can be measured, allowing the Arduino to detect changes in light levels.

The circuit typically consists of the photoresistor connected in series with a fixed resistor, forming a voltage divider circuit. The analog output voltage from the voltage divider is then read by the Arduino's analog input pin. As the ambient light level changes, the resistance of the photoresistor adjusts accordingly, resulting in a corresponding variation in the analog voltage output. By monitoring this voltage, the Arduino can determine the intensity of the surrounding light.

Applications for this setup include ambient light sensing, automatic lighting control, and light intensity measurement. For instance, the Arduino could be programmed to turn on a light when the ambient light level falls below a certain threshold or to adjust the brightness of an LED based on the detected light intensity.

- **IR Sensor using photodiode [14]**

  This TinkerCad example demonstrates the utilization of a photodiode as an infrared (IR) sensor with an Arduino board. Infrared sensors, such as photodiodes, detect infrared radiation emitted by objects and are commonly used in proximity sensing, object detection, and remote control applications.

  In this circuit, the photodiode is connected to one of the analog input pins on the Arduino. When exposed to infrared radiation, the photodiode generates a photocurrent, which can be measured by the Arduino. The intensity of the detected infrared radiation depends on factors such as the distance and reflectivity of nearby objects.

  Applications for this setup include proximity sensing, object detection, and remote control. For example, the Arduino could be programmed to detect the presence of an object in front of the IR sensor and trigger a corresponding action, such as activating a motor or emitting an audible alert. Additionally, the IR sensor could be used in conjunction with an IR transmitter to create a remote control system for electronic devices.

- **Real devices available on the market**
  - **Smartphone automatic brightness control**

    In smartphones, photodiodes are integrated into the device's hardware to measure the ambient light level in the surrounding environment. By detecting the intensity of light, photodiodes provide feedback to the smartphone's software, which then adjusts the brightness of the screen accordingly. This automatic brightness adjustment feature ensures that the screen's brightness matches the ambient lighting conditions, providing optimal visibility for the user while minimizing power consumption. By dynamically adjusting the screen brightness, smartphones equipped with photodiodes can optimize battery life and enhance user comfort, especially in varying lighting environments such as indoors and outdoors. Overall, the usage of photodiodes in smartphone automatic brightness control systems exemplifies their importance in enhancing user experience and energy efficiency in modern electronic devices.

- **Differences Between Real Devices and Example Circuits on TinkerCad**

Example circuits utilizing photodiodes, such as those demonstrated in TinkerCad, primarily serve educational purposes and experimental learning. These circuits showcase the integration of photodiodes with microcontrollers like Arduino boards and other electronic components to demonstrate principles of light sensing, analog signal

processing, and control system design. The main focus of these circuits is to provide hands-on experience and practical understanding of photodiode-based applications in electronics and programming projects.

In contrast, real devices like smartphone automatic brightness control systems utilize photodiodes for practical purposes in consumer electronics. In smartphones, photodiodes are integrated into the device's hardware to measure ambient light levels accurately. The data collected by the photodiodes is then used to automatically adjust the brightness of the screen to match the surrounding lighting conditions. This automatic brightness adjustment feature enhances user experience by ensuring optimal screen visibility while conserving battery life. The usage of photodiodes in real devices demonstrates their practical application in enhancing functionality and energy efficiency in consumer electronics.

In summary, example circuits with photodiodes focus on educational exploration and experimentation, while real devices like smartphone automatic brightness control systems leverage photodiodes for practical applications in enhancing user experience and energy efficiency.

7-4. Temperature Sensor TMP36

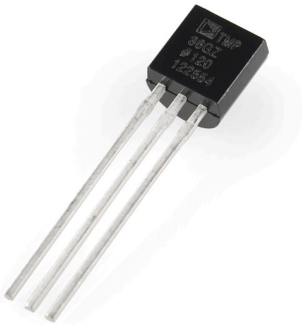

Figure 93: Temperature Sensor TMP36 [18]

TMP36 Temperature Sensor is a low voltage, precision centigrade temperature sensor. It provides a voltage output that is linearly proportional to the Celsius (centigrade) temperature. The TMP36 does not require any external calibration to provide typical accuracies of ±1°C at +25°C and ±2°C over the −40°C to +125°C temperature range [15]. The TMP36 device has three pins: GND, VCC, and Vout. The Vout pin gives the output voltage that is linearly proportional to the temperature [16].

- **How it works**

  The TMP36 is a type of analog temperature sensor based on the principle of voltage output proportional to temperature. It contains a temperature-sensitive voltage output that varies linearly with changes in temperature. The output voltage can be converted into temperature readings using simple linear calibration.

- **Function of pins**
  - **Vcc (Voltage) Pin**

    The Vcc pin is connected to the positive supply voltage of the circuit. It provides the required power (typically between +2.7V to +5.5V) for the temperature sensor to operate. The voltage supplied should be within the specified operating range to ensure accurate temperature measurements.

  - **Ground Pin**

This pin is connected to the ground (0V) reference of the circuit, completing the electrical circuit. A solid ground connection is crucial for accurate temperature readings and stable sensor operation.

- **Output Pin**

  The output pin provides an analog voltage signal proportional to the temperature being measured. The voltage output changes linearly with temperature, typically with a scale factor of 10 mV/°C. This pin is connected to the analog input of a microcontroller or other measuring device for temperature monitoring and control.

- **Applications with example circuits demonstrated on TinkerCad**
  - **TMP36 Temperature Sensor With Arduino [17]**

    In TinkerCad, the TMP36 temperature sensor is commonly interfaced with an Arduino board to create temperature sensing applications. The TMP36 sensor is a low-cost, easy-to-use analog temperature sensor that provides accurate temperature readings over a wide range. When connected to an analog input pin on the Arduino, the TMP36 sensor outputs a voltage proportional to the temperature it detects. The Arduino then converts this analog voltage into temperature readings using the built-in analog-to-digital converter (ADC).

    The TMP36 sensor typically consists of three pins: Vcc, Vout, and GND. The Vcc pin is connected to the positive supply voltage (usually +5V) of the Arduino, while the GND pin is connected to ground (0V). The Vout pin outputs an analog voltage signal that varies linearly with temperature. By reading this voltage signal using one of the Arduino's analog input pins, the temperature can be accurately measured.

    Applications for the TMP36 temperature sensor with Arduino include ambient temperature monitoring, temperature-controlled systems, and environmental monitoring. For example, the Arduino can be programmed to monitor room temperature and activate a cooling fan or heater based on predefined temperature thresholds. Additionally, the TMP36 sensor can be used in conjunction with other sensors to create more complex systems, such as weather stations or temperature logging devices.

- **Real devices available on the market**
  - **Home thermostat**

    In home thermostats, the TMP36 temperature sensor plays a crucial role in maintaining indoor comfort by accurately measuring room temperature. Integrated into the thermostat's hardware, the TMP36 sensor continuously monitors the ambient temperature of the room. This temperature reading serves as a key input for the thermostat's control system, allowing it to regulate the operation of the heating, ventilation, and air conditioning (HVAC) systems.

    The TMP36 sensor provides reliable and precise temperature measurements over a wide range, making it ideal for residential heating and cooling applications. By accurately detecting changes in room temperature, the thermostat can adjust the HVAC settings to achieve and maintain the desired indoor temperature level set by the user.

    In operation, the thermostat's control algorithm compares the current room temperature measured by the TMP36 sensor with the target temperature set by the user. If the measured temperature deviates from the desired setpoint, the thermostat activates the HVAC system accordingly. For example, if the room temperature falls below the setpoint during cold weather, the thermostat signals the heating system to turn on and raise the temperature. Conversely, if the room temperature exceeds the setpoint on a hot day, the thermostat triggers the air conditioning system to cool the room.

By continuously monitoring and adjusting the HVAC systems based on temperature readings from the TMP36 sensor, home thermostats ensure optimal comfort and energy efficiency. This proactive temperature control helps to maintain a comfortable indoor environment while minimizing energy consumption and utility costs.

- **Differences Between Real Devices and Example Circuits on TinkerCad**

In Arduino projects, such as those demonstrated on TinkerCad, the TMP36 sensor is commonly used as an ambient temperature sensor interfaced with an Arduino board. The sensor outputs an analog voltage signal proportional to the temperature it detects, which is then converted into temperature readings by the Arduino's analog-to-digital converter (ADC). Arduino applications with the TMP36 sensor often involve ambient temperature monitoring, temperature-controlled systems, and environmental monitoring. For instance, the Arduino can be programmed to monitor room temperature and activate cooling or heating systems based on predefined temperature thresholds, or it can be integrated into weather stations or temperature logging devices.

On the other hand, in home thermostats, the TMP36 sensor is integrated into the thermostat's hardware to maintain indoor comfort by accurately measuring room temperature. The sensor continuously monitors the ambient temperature, serving as a key input for the thermostat's control system. Unlike Arduino projects where the TMP36 sensor is directly connected to an Arduino board, in home thermostats, the sensor is part of a larger HVAC control system that regulates heating, ventilation, and air conditioning (HVAC) systems. The thermostat's control algorithm compares the current room temperature measured by the TMP36 sensor with the user-defined setpoint temperature. Based on this comparison, the thermostat activates the HVAC system to either heat or cool the room to maintain the desired temperature level set by the user.

In summary, while both Arduino projects and home thermostats use the TMP36 temperature sensor to measure ambient temperature, their implementations and purposes differ significantly. Arduino projects focus on experimental and educational applications, allowing users to create temperature sensing systems for various purposes. In contrast, home thermostats utilize the TMP36 sensor as part of a sophisticated HVAC control system to maintain indoor comfort and energy efficiency in residential environments.

III. Reflection & Conclusion

1. Short conclusion on the final results

- **CLI Implementation**: The Command Line Interpreter was successfully implemented with features like auto-completion, command history, and essential commands such as help, clear, and setcolor. This allowed for efficient interaction with the OS.
- **Welcome Message**: A custom ASCII art welcome message was effectively integrated into the OS boot sequence, enhancing the user experience upon startup.
- **ANSI Terminal Formatting**: The ANSI code for terminal formatting was incorporated, enabling text and background color customization in the console.
- **UART Driver Development**: The UART driver was further developed to support various configurations, including baud rate, data bits, stop bits, parity, and handshaking control, through the CLI.
- **Sensor Integration**: At least three different types of sensors were explored, with their functionality, pin functions, and applications demonstrated using TinkerCad Circuits, providing a practical understanding of common sensors in embedded systems.

These results demonstrate a comprehensive enhancement of the embedded OS's capabilities and user interface, aligning with the objectives of the EEET2490 assessment.

2. Individual Reflection

2-1. Challenges and Opportunities during the Bare Metal OS Development

Throughout the process of developing a Bare Metal Operating System, I encountered a range of challenges and opportunities that significantly enhanced my understanding of embedded systems. The tasks, which included implementing a Command Line Interpreter (CLI), handling ANSI codes for terminal formatting, and developing UART drivers, pushed me to apply theoretical knowledge in a practical setting.

Challenges and Opportunities

- **Low-Level Development:** This assignment required building an operating system's Command Line Interpreter (CLI) from the ground up. Every aspect, from the Makefile to basic C library functions like 'strlen' and 'atoi', had to be meticulously crafted, showcasing the intricate process of developing fundamental OS components.
- **Complex Problem-Solving:** The task of creating a CLI from scratch presented a significant challenge. It demanded a comprehensive understanding of the underlying mechanisms of operating systems, particularly how they process user commands and manage execution flows. This exercise was a testament to the intricate art of problem-solving in the realm of embedded systems.
- **Utilization of C Pointers**: Implementing features such as handling the command_buffer underscored the pivotal role of C pointers. Particularly during the implementation of the TAB key feature, leveraging pointers proved to be both crucial and challenging, showcasing the importance of mastering this fundamental concept for low-level development tasks.
- **Practical Application**: The opportunity to use tools like the ASCII art converter and TinkerCad Circuits provided a hands-on experience that bridged the gap between theory and real-world application.

2-2. New Learnings and Relevant Skills in Embedded Software Engineer Job descriptions

New Learnings

- **Embedded OS Insights**: I gained practical experience with embedded OS features, which aligns with the skills sought in job descriptions for Embedded Software Engineers.
- **UART Configuration**: I acquired a deeper understanding of UART (Universal Asynchronous Receiver/Transmitter) configuration, including parameters such as baud rate, RTS/CTS (Request To Send/Clear To Send), data bits, stop bits, and parity. This knowledge is invaluable for working with serial communication interfaces in embedded systems, allowing me to optimize data transmission and reception.
- **ANSI Codes Mastery**: Learning to use ANSI codes for terminal formatting was a new and valuable skill that will be beneficial in future projects.

Relevant Skills to the ones in job descriptions

After delving into the job market for Embedded Software Engineers in Singapore via LinkedIn [20], I gained practical insights into the industry's expectations and requirements. By focusing on job descriptions from prominent companies like Dyson [21], Continental [22], and NodeFlair [23], I identified key skills and competencies sought after in candidates. These insights provide a valuable framework for applying the concepts I've learned in this assessment to real-world scenarios.

Key Skills Identified:

- **C/C++ Programming**: Proficiency in C/C++ with practical experience in developing embedded systems, aligning with job descriptions' requirements for programming expertise.
- **Embedded System Development**: Skills in real-time operating systems (RTOS), MCU/MPU architecture, and peripherals, crucial for roles involving embedded system design and development.
- **Programming Expertise**: Competence in ANSI/Embedded C/C++ programming.
- Sensor Configuration: Experience in configuring and utilizing sensor devices over I2C, SPI, UART with DMA.
- **Tools and Technologies**: Familiarity with embedded development tools, configuration management tools (JIRA, Confluence, Bitbucket), and continuous integration/testing tools, suitable for software development and testing in embedded environments.

Overall, this assessment has provided me with valuable experience in embedded system design and development, particularly in ANSI, Embedded C, and configuring and utilizing sensor devices over UART.

Skills for Further Exploration

Upon exploring the real-world landscape of embedded software engineering roles via job market searches, I've identified areas where further learning is needed to bridge the gap and advance as an embedded software engineer.

- **Experience with Other Communication Protocols**: Expand knowledge and experience in configuring and utilizing sensor devices over communication protocols such as I2C and SPI, in addition to UART.
- **Familiarity with Additional Tool**s: Gain proficiency in using a wider range of embedded development tools and configuration management tools, including JIRA, Confluence, and Bitbucket, to enhance efficiency and collaboration in software development projects.

IV. References

[1]“Convert Text to ASCII Art,” onlinetools.com. https://onlinetools.com/ascii/convert-text-to-ascii-art

[2]M. Lynch, “What Is a Command Line Interpreter?,” *The Tech Edvocate*, Jun. 23, 2023. https://dev.thetechedvocate.org/what-is-a-command-line-interpreter/ (accessed Apr. 28, 2024).

[3]“What does printf("\033[H\033[J") do in C?,” *Stack Overflow*. https://stackoverflow.com/questions/55672661/what-does-printf-033h-033j-do-in-c (accessed Apr. 28, 2024).

[4]“Terminal Colors | Chris Yeh,” *chrisyeh96.github.io*. https://chrisyeh96.github.io/2020/03/28/terminal-colors.html

[5]“Checking Your Raspberry Pi Revision Number & Board Version,” Raspberry Pi Spy, Sep. 05, 2012. https://www.raspberrypi-spy.co.uk/2012/09/checking-your-raspberry-pi-board-version/

[6]“Mailbox property interface,” *GitHub*. https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface

[7]“Learn how to use Tinkercad,” Tinkercad. https://www.tinkercad.com/learn/circuits

[8]“All About Flex Sensor in 8 Min | Basics | Working | Arduino | Interfacing | TinkerCAD Simulations,” *www.youtube.com*. https://www.youtube.com/watch?v=OZwxh_u1geE (accessed Apr. 29, 2024).

[9]“Circuit design Arduino FSR using Flex Sensor,” *Tinkercad*. https://www.tinkercad.com/things/2Qywui0wvlq-arduino-fsr-using-flex-sensor (accessed Apr. 29, 2024).

[10]“Flex Sensor 2.2" - SEN-10264 - SparkFun Electronics,” *Sparkfun.com*, Dec. 05, 2015. https://www.sparkfun.com/products/10264

[11]“Circuit design Arduino Flex Sensor and Servo Motor,” *Tinkercad*. https://www.tinkercad.com/things/9mQytHqlzpY- (accessed Apr. 29, 2024).

[12]“Circuit design Interfacing with photodiode using Arduino,” *Tinkercad*. https://www.tinkercad.com/things/7w5sghr1RGA-interfacing-with-photodiode-using-arduino (accessed Apr. 29, 2024).

[13]“Light Sensor (Photoresistor) With Arduino in Tinkercad,” *Tinkercad*. https://www.tinkercad.com/projects/Light-Sensor-Photoresistor-Arduino-Tinkercad (accessed Apr. 29, 2024).

[14]“Circuit design IR Sensor using photodiode,” *Tinkercad*. https://www.tinkercad.com/things/ct6UJXnDUVh-ir-sensor-using-photodiode (accessed Apr. 29, 2024).

[15]L. A, “TMP36 Temperature Sensor Arduino Tutorial (2 Examples),” *Makerguides.com*, Oct. 25, 2020. https://www.makerguides.com/tmp36-arduino-tutorial/

[16]“Arduino - TMP36 Temperature Sensor | Arduino Tutorial,” *Arduino Getting Started*. https://arduinogetstarted.com/tutorials/arduino-tmp36-temperature-sensor

[17]“Circuit design TMP36 Temperature Sensor With Arduino,” *Tinkercad*. https://www.tinkercad.com/things/7KlkHaxx20c-tmp36-temperature-sensor-with-arduino

[18]“Temperature Sensor - TMP36 - SEN-10988 - SparkFun Electronics,” *www.sparkfun.com*. https://www.sparkfun.com/products/10988

[19]rashikagupta1985, “What Is Photodiode? Its Operations And Applications,” Jun. 14, 2023. https://quick-learn.in/photodiode-and-its-operation/ (accessed Apr. 29, 2024).

[20]“4,000+ Embedded Software Engineer jobs in United States,” *www.linkedin.com*, Apr. 26, 2024. https://www.linkedin.com/jobs/embedded-software-engineer-jobs/?currentJobId=3855212854&originalSubdomain=sg (accessed Apr. 30, 2024).

[21]“Dyson hiring Embedded Software Engineer in Singapore, Singapore | LinkedIn,” *sg.linkedin.com*, Mar. 29, 2024. https://www.linkedin.com/jobs/view/3797410027 (accessed Apr. 30, 2024).

[22]“Continental hiring Embedded Software Engineer in Singapore, Singapore | LinkedIn,” *sg.linkedin.com*, Mar. 29, 2024. https://www.linkedin.com/jobs/view/3902058621 (accessed Apr. 30, 2024).

[23]“NodeFlair - Tech Salaries, Jobs & more hiring Embedded Software Engineer in Singapore, Singapore | LinkedIn,” *sg.linkedin.com*, Mar. 29, 2024. https://www.linkedin.com/jobs/view/3881608813 (accessed Apr. 30, 2024).