RMIT University Vietnam

School of Science, Engineering & Technology (SSET)

EEET2490 – Embedded Systems: Operating Systems and Interfacing

Semester A – Year 2024

ASSIGNMENT 2: ADDITIONAL FEATURES FOR BARE METAL OS

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

Student Name: Nguyen Dinh Viet

Student ID: s3927291

Submission Date: April 30, 2024

"I declare that in submitting all work for this assessment I have read, understood and agree to the content and expectations of the Assessment declaration "

Table of Contents

I. Introduction

The project's objective is to create a bare-metal operating system (OS) from scratch tailored for a Raspberry Pi model. This embedded OS will include essential features like a command line interpreter (CLI), support for ANSI codes for terminal formatting, support for PL011 Universal Asynchronous Receiver-Transmitter (UART) configuration. Additionally, the report will explore some common sensors available on Tinkercad Circuit. Each sensor's operation principle, pin functions, potential applications, and example circuits demonstrated on Tinkercad will be discussed. Furthermore, a comparison with real-world devices available in the market will be provided to highlight any disparities.

II. Additional Features for Bare Metal OS

1. Prerequisites

In this project, C libraries such as **stdio.h**, **string.h** are not used. Therefore, some of the functions to process with string will be implemented manually. Those functions are declared as prototyped in **stringProcess.h** file and implemented in **stringProcess.c** file.

```
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, int n);
int strlen(const char *s);
char* strtok(char *s, const char *delim);
int strspn(const char *s, const char *accept);
int strcspn(const char *s, const char *reject);
char* strchr(const char *s, int c);
char* strcpy(char *dest, const char *src);
char* strcat(char *dest, const char *src);
void reset(char *s);
int atoi(char *s);
```

Figure 1: Function prototype to process string

Furthermore, within this project, I possess two header files: **normal.h** and **color.h**, each declaring the function prototypes for **normal.c** and **color.c**, respectively. Specifically, **normal.c** is tasked with executing the normal command, while **color.c** manages the terminal's color processing. Lastly, there is a **constant.h** file dedicated to declaring all constants utilized within this project. All those header files are grouped into **utils** folder, while its source files are stored in **source** folder.

```
void setColor(char* cmd);
void setBackgroundColor(char* color);
void setTextColor(char* color);
```

Figure 2: Function prototype to process terminal's color

```
// Function prototypes
void displayWelcomeMessage();
void displayOS();
void displayHelp(char* clibuffer, char* commands[]);
void displayBriefHelp(char* commands[]);
void displayDetailHelp(char* detail);
void clear();
void showInfo();
void deleteChar(char* str);
void deleteOneChar();
void update_baud_rate(char* buffer, int* baudrate);
void update_data_length(char* buffer, int* length);
void update_stop_bit(char* buffer, int* stop);
void update_parity_bit(char* buffer, int* parity);
void update_handshake_control(char* buffer, int* handshake);
```

Figure 3: Function prototype to process normal command

2. Welcome Message

In the Bare Metal OS, a visually striking welcome message is crafted to greet users upon successful boot-up. This captivating message is composed using ASCII art generated with the assistance of online tools. It displays the course code "EEET2490", the author's full name and student number, alongside the OS moniker "FelixOS" inspired by the author's real nickname. This personalized touch adds a unique flair to the OS experience, setting it apart from conventional boot sequences.

```
 ######## ######## ######## ########  #######  ##        #######    #####
 ##       ##       ##          ##    ##     ## ##    ##  ##     ##  ##   ##
 ##       ##       ##          ##          ## ##    ##  ##     ## ##     ##
 ######   ######   ######      ##     #######  ##    ##   ######## ##     ##
 ##       ##       ##          ##    ##        #########        ## ##     ##
 ##       ##       ##          ##    ##              ##  ##     ##  ##   ##
 ######## ######## ########    ##    #########       ##   #######    #####



 ######## ######## ##       #### ##     ##     #######   ######
 ##       ##       ##        ##   ##   ##     ##     ## ##    ##
 ##       ##       ##        ##    ## ##      ##     ## ##
 ######   ######   ##        ##     ###       ##     ##  ######
 ##       ##       ##        ##    ## ##      ##     ##       ##
 ##       ##       ##        ##   ##   ##     ##     ## ##    ##
 ##       ######## ######## #### ##     ##    #######   ######

 Developed by Nguyen Dinh Viet- s3927291
 FelixOS>
```

Figure 4: Implementation of Welcome Message

3. Command Line Interpreter (CLI)

3.1. Overview

Command Line Interpreter (CLI) is a text-based interface that facilitates the input of commands and subsequently executes those commands within the operating system [1]. In the 1960s, the Command Line Interpreter (CLI) was the sole method for interacting with computers [2]. Despite the proliferation of graphical user interfaces (GUIs) in modern times, the CLI remains indispensable. It is leveraged by software developers and system administrators for tasks such as computer configuration, software installation, and accessing features not accessible via the graphical interface [2]. In this project, **FelixOS** will implement a CLI that reads keypresses into a buffer and then executes the commands when the key “Enter” is pressed.

3.2. Implementation

3.2.1. Basic Command

In this part, the OS should support the users with some basic commands. It is noted that all the characters are case-insensitive.

a. help

There are 2 options for **help** command supported in this OS.

```
// Display help
void displayHelp(char* clibuffer, char* commands[]) {
    char* token = strtok(clibuffer, " ");
    token = strtok(NULL, " "); // Skip the first token "help"

    // If there is no command after "help", display brief help for all commands
    if (token == NULL) {
        displayBriefHelp(commands);
    }
    else {
        char helpCommand[MAX_CMD_SIZE]; // Buffer to store the command after "help"
        helpCommand[0] = '\0'; // Initialize the buffer

        while (token != NULL) {
            strcat(helpCommand, token);
            token = strtok(NULL, " ");

            if (token != NULL) {
                strcat(helpCommand, " ");
            }
        }

        // Now check if the constructed help command matches any specific command
        for (int i = 0; i < MAX_COMMAND_SIZE; i++) {
            if (strcasecmp(helpCommand, commands[i]) == 0) {
                displayDetailHelp(commandsFullInfo[i]);
                return;
            }
        }

        uart_puts("Invalid help command\n");
    }
}
```

*Figure 5: Implementation of **displayHelp** function*

The **displayHelp** function manages the help command received from the CLI. When the command is simply **help**, it presents concise information about all supported commands. Conversely, if the command is **help [command]**, the OS provides comprehensive instructions on how to utilize that particular command. It is noted that the command should be one of the supported commands by the OS, otherwise, it will show an error message.

- Result of **help** command:

```
FelixOS> help
Got command: help
For more information on a specific command, type "help [command_name]"
clear                   Clear the screen
set color               Set text color, and/or background color of the console
showinfo                Display board revision and board MAC address
baudrate                Configure the baud rate of the UART
length                  Configure the length of data bits
stop                    Configure the number of stop bits
parity                  Configure the parity bit
handshake               Configure handshake control
check                   Check the current configuration of the UART
```

*Figure 6: Result of **help** command*

- Result of **help showinfo** command:

```
FelixOS> help showinfo
Got command: help showinfo
showinfo: This command displays the revision and MAC address of the board you are currenly working with.
```

*Figure 7: Result of **help showinfo** command*

- Result of **help parity** command:

```
FelixOS> help parity
Got command: help parity
parity: This command sets the parity bit. Type "parity [parity bit]", parity bit must be 'none', 'even' or 'odd'.
```

*Figure 8: Result of **help parity** command*

b. clear

The **clear** command will clear the whole terminal and scroll down to current position of the cursor.

```
// Clear the terminal
void clear() {
    uart_puts("\033[2J\033[H");
}
```

*Figure 9: Implementation of **clear** command*

This command can be implemented simply using ANSI escape sequences. These sequences include "\033[2J" to clear the screen and "\033[H" to reset the cursor position.

- Result of **clear** command:

```
FelixOS> help
Got command: help
For more information on a specific command, type "help [command_name]"
clear                   Clear the screen
set color               Set text color, and/or background color of the console
showinfo                Display board revision and board MAC address
baudrate                Configure the baud rate of the UART
length                  Configure the length of data bits
stop                    Configure the number of stop bits
parity                  Configure the parity bit
handshake               Configure handshake control
check                   Check the current configuration of the UART
FelixOS> clear
```

*Figure 10.1: Result of **clear** command (before execution)*

```
FelixOS>
```

*Figure 10.2: Result of **clear** command (after execution)*

c. setcolor

The **setcolor** command changes the text color and background color of the CLI. There are 8 supporting colors: BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN and WHITE for both text and background.

```
// Text color
#define BLACK_TEXT "\033[30m"
#define RED_TEXT "\033[31m"
#define GREEN_TEXT "\033[32m"
#define YELLOW_TEXT "\033[33m"
#define BLUE_TEXT "\033[34m"
#define MAGENTA_TEXT "\033[35m"
#define CYAN_TEXT "\033[36m"
#define WHITE_TEXT "\033[37m"
#define RESET_TEXT "\033[0m"
```

Figure 11.1: List of ANSI Color codes for text

```
// Background color
#define BLACK_BG "\033[40m"
#define RED_BG "\033[41m"
#define GREEN_BG "\033[42m"
#define YELLOW_BG "\033[43m"
#define BLUE_BG "\033[44m"
#define MAGENTA_BG "\033[45m"
#define CYAN_BG "\033[46m"
#define WHITE_BG "\033[47m"
#define RESET_BG "\033[0m"\
```

Figure 11.2: List of ANSI Color codes for background

Implementing this feature involves using ANSI color codes to set colors for both text and background. Each command will be associated with a maximum of one specific text and a maximum of one background color.

- To set the text color, we will use the command: **setcolor -t [color]**. The figure below demonstrates an example of setting the text color to **red**.

*Figure 12: Result of **setcolor -t red** command*

- To set the background color, we will use the command: **setcolor -b [color]**. The figure below demonstrates an example of setting the background color to **green**.

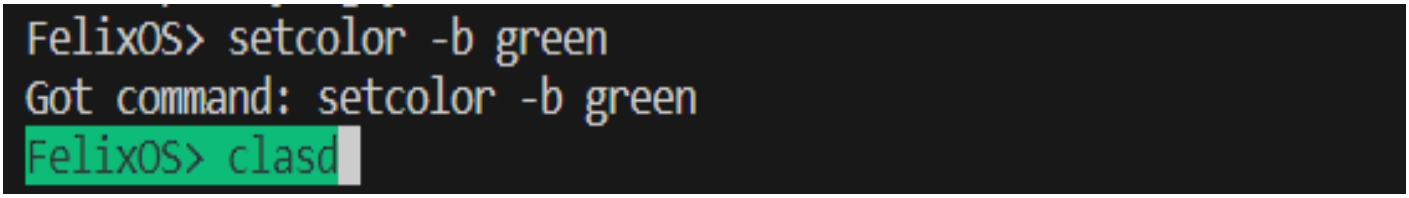

*Figure 13: Result of **setcolor -b green** command*

- It is possible to set color to both text and background in a single command. It is noted that the tags **-b** and **-t** can be in any order. The figure below shows an instance of setting the background color to **white** and text color to **red** in one command.

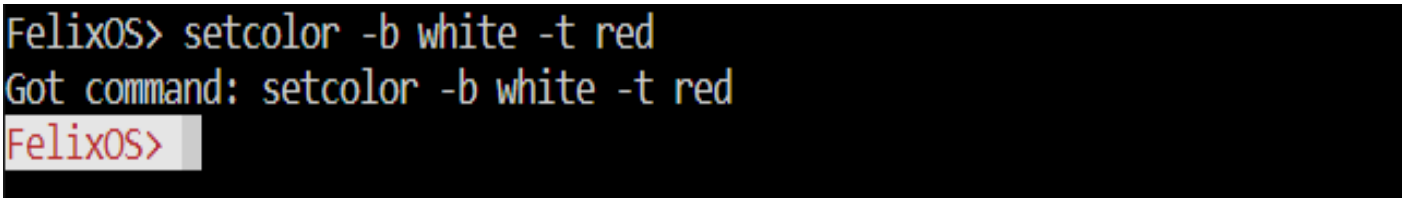

*Figure 14: Result of **set color -b white -t red** command*

Other colors aside from the list above will be considered as invalid. Consequently, the OS will print out the error message.

d. showinfo

The **showinfo** command demonstrates the board's revision, board's name and board's Media Access Control (MAC) address. This can be simply implemented using Raspberry Pi Mailbox. First, we obtain the board's revision by retrieving its hexadecimal value and comparing it to determine the version [3]. As for the MAC address, we retrieve it from the Raspberry Pi mailbox in network byte order. To present it in a more readable format, we use a function to convert it into six pairs of hexadecimal values.

- Result of **showinfo** command:

```
FelixOS> showinfo
Got command: showinfo
Board revision = 0x00A02082 (Board model: rpi-3B BCM2837 1GiB Sony UK)
Board MAC address = 52:54:00:12:34:57
```

*Figure 15: Result of **showinfo** command*

3.2.2. CLI Enhancement

a. OS name in CLI

"FelixOS" serves as the initial text displayed in the console interface, awaiting user commands. Upon pressing "Enter" to execute a command from the command line interface (CLI), the initial text reappears, maintaining its presence as a familiar backdrop for user interaction. This can be implemented by simply calling the **displayOS()** function after executing a command.

```
FelixOS> showinfo
Got command: showinfo
Board revision = 0x00A02082 (Board model: rpi-3B BCM2837 1GiB Sony UK)
Board MAC address = 52:54:00:12:34:57
FelixOS>
```

Figure 16: OS name in CLI

b. Auto-completion in CLI

Auto-completion is a vital feature for any operating system. In this project, its implementation involves creating an index that scans through the list of available commands. Whenever a user presses the TAB key, the system iterates through the command list to find matches with the current input buffer's prefix. This list of matched commands is dynamically updated as the user enters new characters. If multiple matching commands are found, the user can cycle through them by pressing TAB again. If no matching commands are found, pressing TAB will have no effect. Hence, it is noted that the auto-completion will work with the first word only.

```
// If tab is pressed, display the next matching command (only if there is a match)
if (c == '\t') {
    if (match_index > 0) {
        if (command_index == 0) {
            deleteChar(cli_buffer);
        } else {
            deleteChar(match[(command_index-1) % (match_index)]);
        }

        index = strlen(match[command_index % match_index]);
        uart_puts(match[command_index % match_index]);
        command_index++;
    }
```

Figure 17: Implementation of auto-completion

- Result of auto-completion feature: A list of figures below shows an example of auto-completion if the input buffer is **“s”** only.

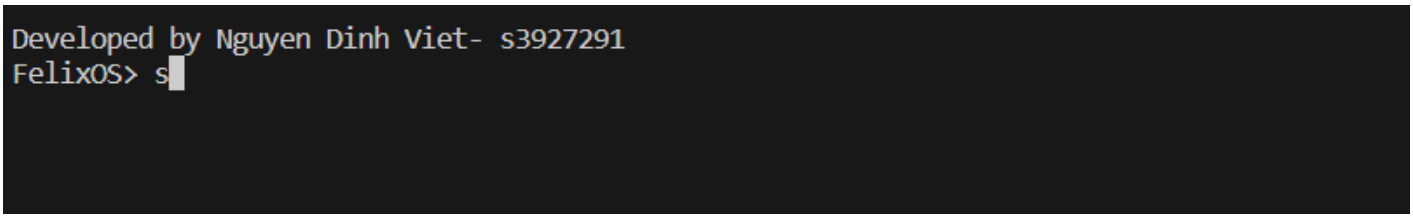

Figure 18.1: Result of auto-completion (Initial)

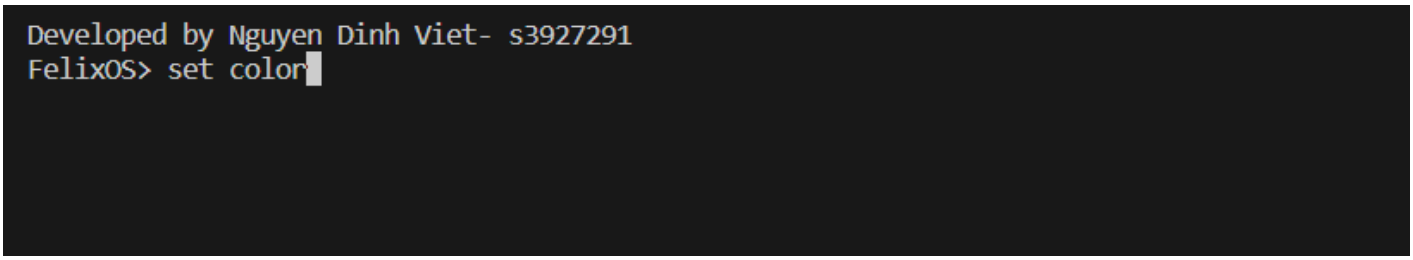

Figure 18.2: Result of auto-completion (after first TAB)

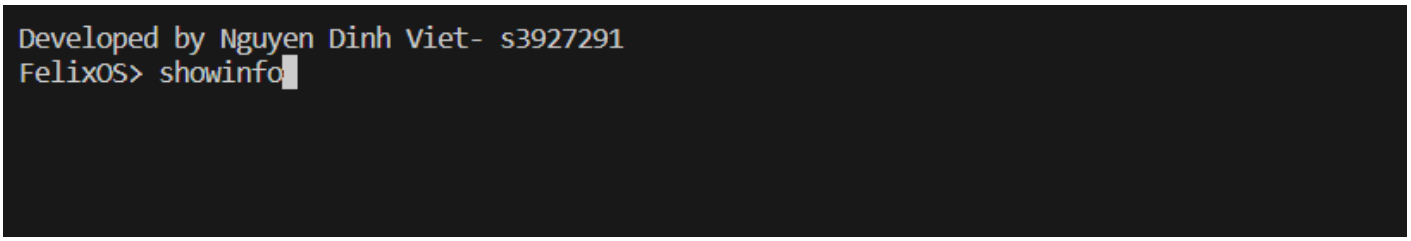

Figure 18.3: Result of auto-completion (after second TAB)

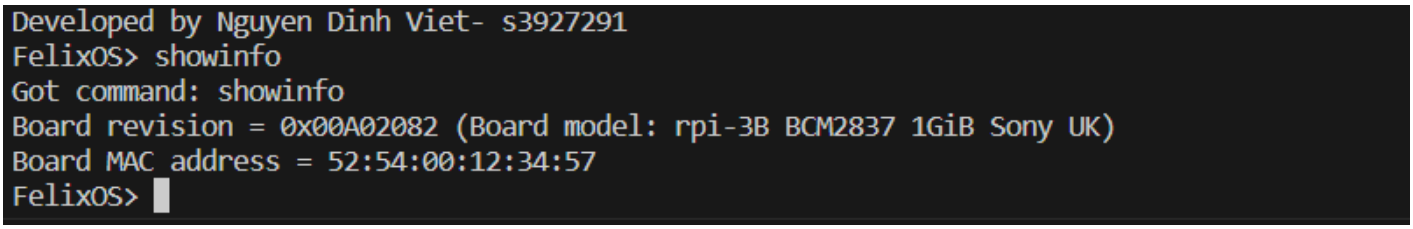

*Figure 18.4: Press Enter to execute **showinfo***

c. Command history

Command history functionality is essential in OS development, allowing users to review executed commands. Typically, users navigate command history using the UP and DOWN arrow keys. In this project, I've introduced custom keys: "_" to represent the UP arrow and "+" for the DOWN arrow. To implement this feature, I've initialized an additional array called **history** along with an index, **history_index** to track and store commands. To enhance usability, I've set a maximum capacity of 1000 commands for the history array. This ensures the OS can store up to 1000 commands. I've also defined upper and lower limits to prevent looping within the history array. When the user presses "_" or "+", the program clears the current input buffer and retrieves the command at the current history position, replacing the input. The program then displays the command in the terminal, adjusting the buffer index to the correct position to prevent errors.

- Result of command history feature: A list of figures below shows an example of how the command history works:
    - Step 1: Type and execute command **help**.

```
FelixOS> help
Got command: help
For more information on a specific command, type "help [command_name]"
clear                   Clear the screen
setcolor                Set text color, and/or background color of the console
showinfo                Display board revision and board MAC address
baudrate                Configure the baud rate of the UART
length                  Configure the length of data bits
stop                    Configure the number of stop bits
parity                  Configure the parity bit
handshake               Configure handshake control
check                   Check the current configuration of the UART
FelixOS>
```

Figure 19.1: Command history – Step 1

- Step 2: Type and execute command **showinfo**.

```
FelixOS> showinfo
Got command: showinfo
Board revision = 0x00A02082 (Board model: rpi-3B BCM2837 1GiB Sony UK)
Board MAC address = 52:54:00:12:34:57
FelixOS>
```

Figure 19.2: Command history – Step 2

- Step 3: Type and execute command **clear**.

```
FelixOS> clear
```

Figure 19.3: Command history – Step 3

- Step 4: Type some random chars. For example, type “Felix” (not execute command).

```
FelixOS> Felix
```

Figure 19.4: Command history – Step 4

- Step 5: Type ‘_’ character to show the previous command. In this case, it would display the **clear** command executed in step 3.

```
FelixOS> clear
```

Figure 19.5: Command history – Step 5

- Step 6: Type another '_' character to show the previous command. In this case, it would display the **showinfo** command executed from step 2.

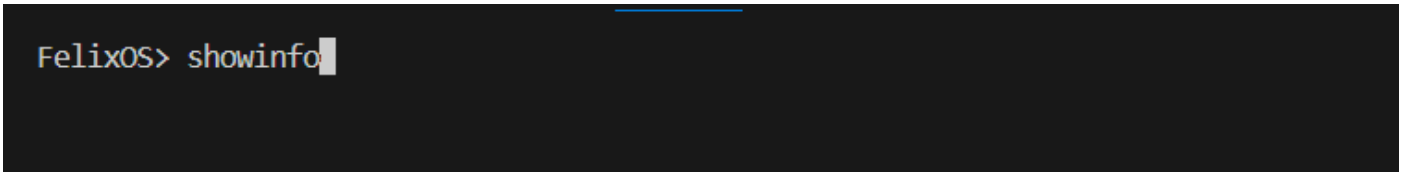

Figure 19.6: Command history – Step 6

- Step 7: Type '+' character to display the next command in the list of command history. In this case, it would display the **clear** command from step 3.

Figure 19.7: Command history – Step 7

- Step 8: Continue type '+' will display the string before using the command history. This is the lower limit of the command history. The upper limit, in this flow, is the **help** command. Upon exceeding the upper/lower limit, '_' or '+' will have no effect.

Figure 19.8: Command history – Step 8

3.2.3. UART settings

In this section, I will develop further with PL011 UART. This development includes: baud rate configuration, number of data bit (data length) configuration, number of stop bit configuration, parity bit configuration and handshake control (CTS/RTS hardware flow control).

a. baudrate

This command helps configure the baud rate of the UART. This can be implemented by setting the value of 2 registers of UART: **IBRD** and **FBRD.** We have a formula:

$$\text{Baud rate divisor} = \frac{Clock\ of\ UART}{16 \times Baud\ rate}\ [3]$$

IBRD = The integer part of the baud rate divisor [3]

FBRD = The fractional part of the baud rate divisor × 64 + 0.5.

By default, the baud rate of UART is 115200. To set up, we will use command syntax: **baudrate [baud rate]**. Supported baud rates are: 9600, 14400, 19200, 38400, 57600, 115200, 230400, 460800, 921600. Other values than those will be considered as invalid baud rate configuration, and the system will print out an error message.

```c
// Function to configure the baud rate of the UART
void configure_baud_rate(int baud_rate) {
    unsigned int integerPart = UART_CLOCK / (16 * (baud_rate));
    float actual = ((float) UART_CLOCK) / (16 * (baud_rate));
    float fractionalPart = (actual - integerPart);

    // Baud rate
    UART0_IBRD = integerPart;
    UART0_FBRD = (int)(fractionalPart * 64 + 0.5);
}
```

Figure 20: Implementation of baud rate configuration

- An example of setting up **9600** baud rate configuration:

```
FelixOS> baudrate 9600
Got command: baudrate 9600
FelixOS>
```

*Figure 21: Example of **baudrate 9600** command*

- **Check with QEMU emulation:** To check the current baud rate of the UART, we will use command **check** (mentioned in section f).

```
FelixOS> check
Got command: check
UART0_IBRD: 312
UART0_FBRD: 32
Baud rate (approx): 9600
Nunmber of data bit: 8
Number of stop bit: 1
Parity bit: None
Handshake control: CTS/RTS handshake is disabled
FelixOS>
```

Figure 22: Check the configuration of the baud rate

- **Check with real Raspberry Board:** The demo video will demonstrate how the configuration works with Raspberry Pi 3.

b. length

This command helps configure the number of data bits. This can be implemented by setting the bits **5** and **6** (**WLEN)** of register **LCRH** [3]. There are 4 configurations for the length of data bits:

Values of bits 6-5 of register LCRH	Operation
11	Set data length to 8 bits
10	Set data length to 7 bits
01	Set data length to 6 bits
00	Set data length to 5 bits

Table 1: Configuration for the length of data bits [3]

By default, the number of data bits is 8. To set up, we will use command syntax: **length [length of data bits]**. Supported numbers are: 5, 6, 7, 8. Other values than those will be considered as invalid length configuration, and the system will print out an error message.

```
// Function to change data bit length
int change_bit_length(int data_bit_length) {
    switch (data_bit_length) {
        case 5:
            return UART0_LCRH_WLEN_5BIT;
        case 6:
            return UART0_LCRH_WLEN_6BIT;
        case 7:
            return UART0_LCRH_WLEN_7BIT;
        case 8:
            return UART0_LCRH_WLEN_8BIT;
        default:
            return UART0_LCRH_WLEN_8BIT;
    }
}
```

Figure 23: Implementation of data length configuration

- An example of setting up data length to **7 bits**:

```
Developed by Nguyen Dinh Viet- s3927291
FelixOS> length 7
Got command: length 7
FelixOS>
```

*Figure 24: Example of **length 7** command*

- **Check with QEMU emulation:** To check the current number of the data bits of the UART, we will use command **check** (mentioned in section f.

```
FelixOS> check
Got command: check
UART0_IBRD: 26
UART0_FBRD: 3
Baud rate (approx): 115176
Nunmber of data bit: 7
Number of stop bit: 1
Parity bit: None
Handshake control: CTS/RTS handshake is disabled
FelixOS>
```

Figure 25: Check the configuration of the number of data bits

- **Check with real Raspberry Board:** The demo video will demonstrate how the configuration works with Raspberry Pi 3.

c. stop

This command helps configure the number of stop bits. This can be implemented by configuring bit **3** (**STP2**) of register **LCRH** [2]. When this bit is set to 1, two stop bits are transmitted at the end of the frame. Otherwise, one stop bit is selected.

```
// Function to change stop bit
int change_stop_bit(int stop_bit) {
    switch (stop_bit) {
        case 1:
            return 0;
        case 2:
            return UART0_LCRH_STP2;
        default:
            return 0;
    }
}
```

Figure 26: Implementation of stop bit configuration

By default, the number of stop bit is 1. To set up, we will use command syntax: **stop [number of stop bits]**. The number of stop bits should be 1 or 2. Any other values will be considered as invalid configuration, and the system will print out an error message.

- An example of setting up stop bit to **2**:

```
FelixOS> stop 2
Got command: stop 2
FelixOS>
```

*Figure 27: Example of **stop 2** command*

- **Check with QEMU emulation:** To check the current number of the stop bits of the UART, we will use command **check** (mentioned in section f).

```
FelixOS> check
Got command: check
UART0_IBRD: 26
UART0_FBRD: 3
Baud rate (approx): 115176
Nunmber of data bit: 8
Number of stop bit: 2
Parity bit: None
Handshake control: CTS/RTS handshake is disabled
FelixOS>
```

Figure 28: Check the configuration of the number of stop bits

- **Check with real Raspberry Board:** The demo video will demonstrate how the configuration works with Raspberry Pi 3.

d. parity

This command helps setup the parity bit configuration of the UART. This can be implemented by setting the bits **1 (PEN)** and **2** (**EPS)** of register **LCRH** [3]. There are 3 configurations for the parity bit:

Bit 1 (PEN)	Bit 2 (EPS)	Operation
0	0/1	Parity is disabled (None)
1	0	Odd parity selected (Odd)
1	1	Even parity selected (Even)

Table 2: Configuration for parity bit

```
// Function to change parity bit
int change_parity_bit(int parity_bit) {
    switch (parity_bit) {
        case 0:
            return 0;
        case 1:
            return UART0_LCRH_PEN;
        case 2:
            return UART0_LCRH_PEN | UART0_LCRH_EPS;
        default:
            return 0;
    }
}
```

Figure 29: Implementation of parity bit configuration

By default, the parity bit configuration is None. To set up, we will use command syntax: **parity [parity bit]** The parity bit should be “none”, “even” or “odd” (case-insensitive). Any other values will be considered as invalid configuration, and the system will print out an error message.

- An example of setting up parity bit configuration to **even**:

```
FelixOS> parity even
Got command: parity even
FelixOS>
```

*Figure 30: Example of **parity even** command*

- **Check with QEMU emulation:** To check the current parity bit configuration of the UART, we will use command **check** (mentioned in section f).

```
Got command: check
UART0_IBRD: 26
UART0_FBRD: 3
Baud rate (approx): 115176
Nunmber of data bit: 8
Number of stop bit: 1
Parity bit: Even
Handshake control: CTS/RTS handshake is disabled
FelixOS>
```

Figure 31: Check the configuration of the parity bit

- **Check with real Raspberry Board:** The demo video will demonstrate how the configuration works with Raspberry Pi 3.

e. handshake

This command helps configure the handshake control of UART. This can be implemented by setting the bits **14 (RTSEN)** and **15** (**CTSEN)** of register **CR** [3]. In addition, it is required to toggle the GPIO pin **16** and **17** on Raspberry PI board to enable **CTS0** and **RTS0** (Alternative function: **ALT3**) [3]. There are 2 configurations: **None** and **CTS/RTS** Control.

```
    /* Set GPIO14 and GPIO15 to be pl011 TX/RX which is ALT0    */
    r = GPFSEL1;

    if (handshake == 0) {
        r &=  ~((7 << 12) | (7 << 15) | (7 << 18) | (7 << 21)); //clear bits 23-12 (FSEL15, FSEL14, FSEL16, FSEL17)
        r |= (0b100 << 12)|(0b100 << 15);   //Set value 0b100 (select ALT0: TXD0/RXD0)
    } else {
        r &=  ~((7 << 12) | (7 << 15) | (7 << 18) | (7 << 21)); //clear bits 23-12 (FSEL15, FSEL14, FSEL16, FSEL17)
        r |= (0b100 << 12)|(0b100 << 15) | (0b111 << 18) | (0b111 << 21);   //Set value 0b100 (select ALT0: TXD0/RXD0), value 0b111
    }
    GPFSEL1 = r;

    /* enable GPIO 14, 15 */
#ifdef RPI3    // RPI3
    GPPUD = 0; // No pull up/down control
    // Toogle clock to flush GPIO setup
    r = 150;
    while (r--)
    {
        asm volatile("nop");
    }                           // waiting 150 cycles
    GPPUDCLK0 = (1 << 14)|(1 << 15)| (1<<16) | (1<<17); //enable clock for GPIO 14, 15
    r = 150;
    while (r--)
    {
        asm volatile("nop");
    }             // waiting 150 cycles
    GPPUDCLK0 = 0;        // flush GPIO setup
```

Figure 32.1: Implementation of handshake configuration (set up GPIO)

```
// Function to configure handshake control of the UART
void configure_handshaking_control(int status) {
    if (status == 1) {
        UART0_CR = 0xC301;
    } else {
        UART0_CR = 0x301;
    }
}
```

*Figure 32.2: Implementation of handshake configuration (set up register **CR**)*

By default, the handshake configuration is None. To set up, we will use command syntax: **handshake [mode].** The mode should be “on” (turn on CTS/RTS) or “off” (turn off CTS/RTS). Any other values will be considered as invalid configuration, and the system will print out an error message.

- An example of turning on CTS/RTS control:

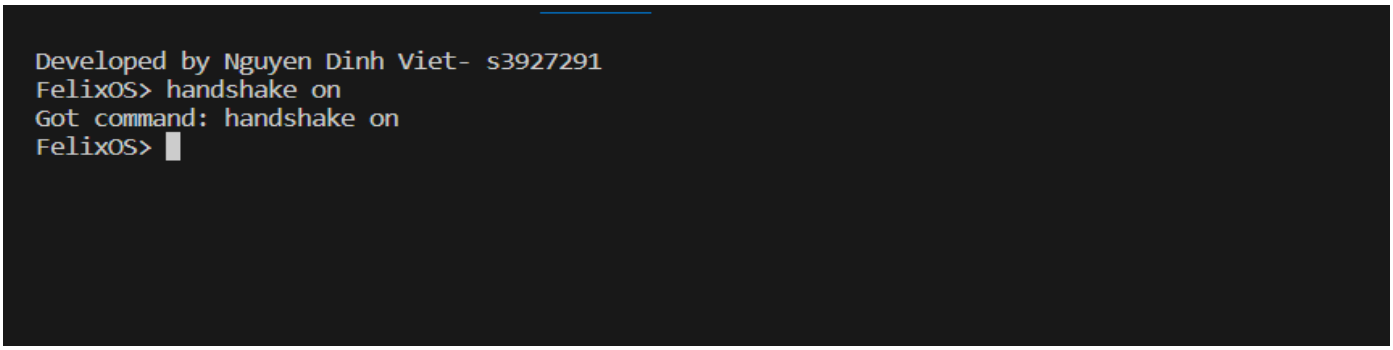

*Figure 33: Example of **handshake on** command*

- **Check with QEMU emulation:** To check the current handshake configuration of the UART, we will use command **check** (mentioned in section f).

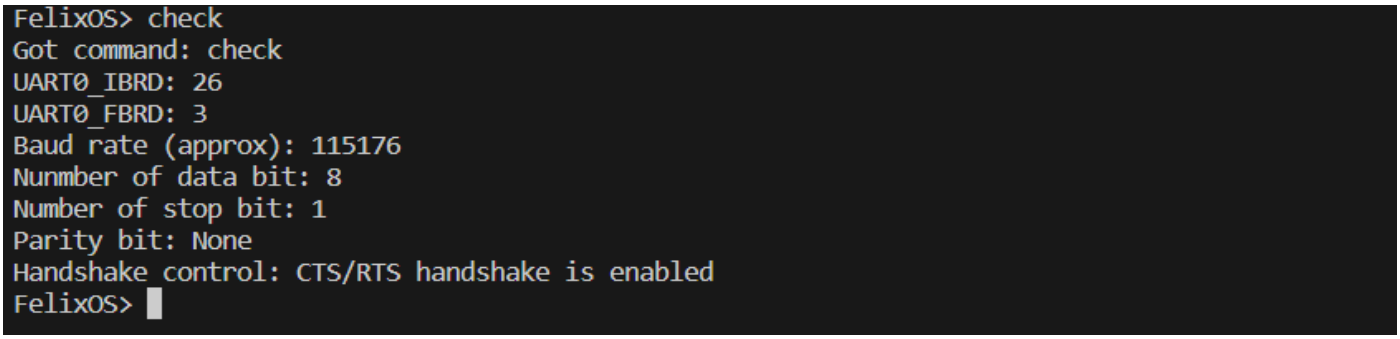

Figure 34: Check the configuration of handshake control

- **Check with real Raspberry Board:** The demo video will demonstrate how the configuration works with Raspberry Pi 3.

f. check

This command helps check the current configuration of the UART including the baud rate (**IBRD** and **FBRD** values), the number of data bits, the number of stop bits, parity bit configuration and handshake control configuration. To use this, users should type **check** and then press Enter to execute the command.

- An example of **check** command:

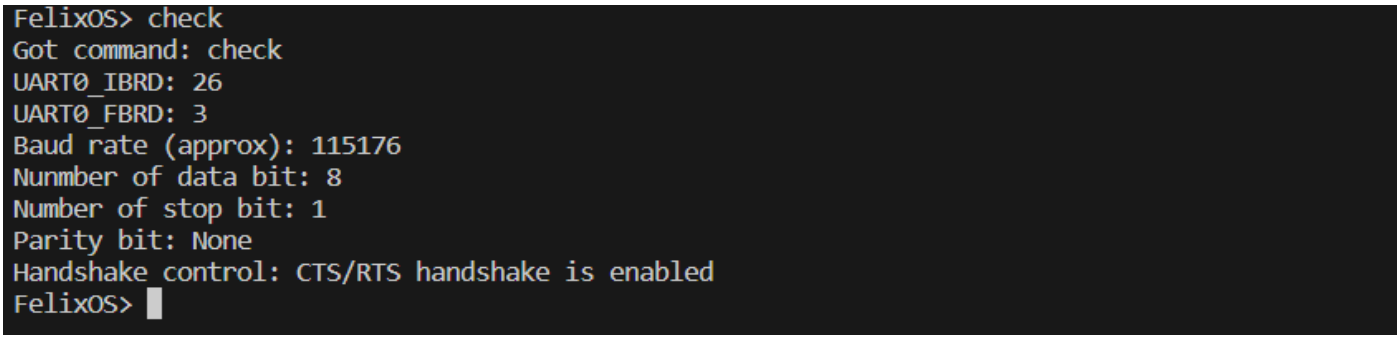

*Figure 35: Example of **check** command*

3.3. Evaluation

The table below summarizes the features in both Task 1 and 2.

Feature Group		Command/ Feature	Implementation	Testing	Issues / limitations
Basic Commands		help	complete	“help”	
				“help clear”	
		clear	complete	“clear”	The previous message was not fully shown in some cases
		setcolor	complete	“setcolor -t red”	
				“setcolor -b green”	
				“setcolor -b black -t cyan”	
		showinfo	complete	“showinfo”	
CLI enhancement		OS name in CLI	complete	Display “FelixOS” at the beginning	
		Auto-completion in CLI	complete	Press Tab for auto-completion	
				Press another Tab to switch to other auto-completion	
		Command history in CLI	complete	Press “_” to show previous command from the command history	
				Press “+” to show next command from the command history	
UART settings	**Baud rate**	baudrate	complete	“baudrate 9600”	
	Data bits	length	complete	“length 7”	
	Stop bits	stop	complete	“stop 2”	
	Parity	parity	complete	“parity even”	
	Handshaking	handshake	complete	“handshake on”	
Additional		check	complete	“check”	

Table 3: Summary of feature implementation from task 1 and 2

4. Some common sensors

In this section, we will get familiar with some of the common sensors that are available in Tinkercad Circuits.

4.1. Temperature Sensor TMP36

4.1.1. Overview

The temperature sensor TMP36 is a low voltage, precision centigrade temperature sensor [4]. It provides a voltage output that is linearly proportional to the Celsius temperature [4]. This sensor operates over a wide temperature range, typically from -40°C to 125°C [4], making it suitable for a variety of temperature sensing tasks. It's also known for its simplicity of use and low cost [4].

4.1.2. Pin configuration

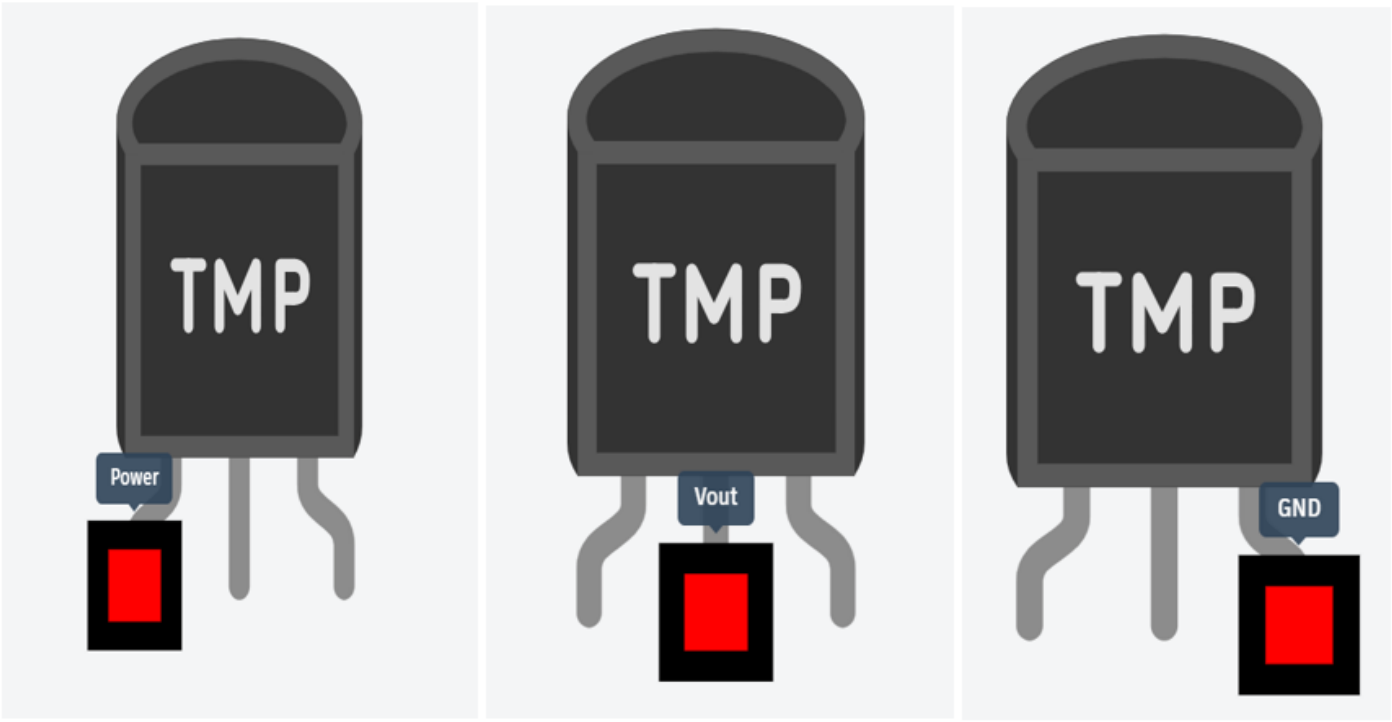

Figure 36: Temperature sensor (TMP36) pin configuration on Tinkercad [5]

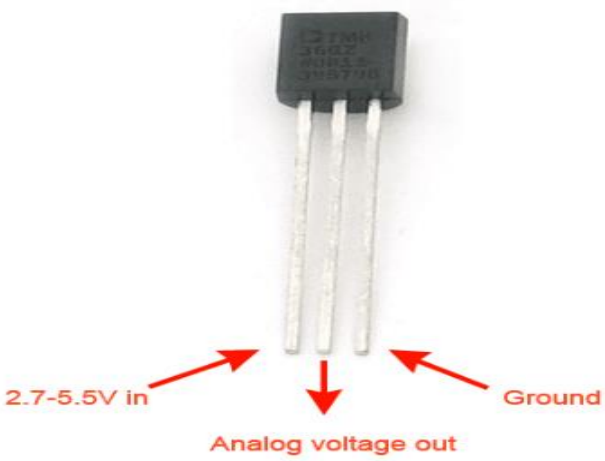

Figure 37: Temperature sensor (TMP36) pin configuration in real life [6]

The temperature sensor TMP36 typically has 3 pins: **VCC**, **VOUT** and **GND.**

- **VCC (Power):** This pin is used to supply the power to the sensor. It typically operates within a range of 2.7V to 5.5V [6], making it compatible with a wide range of microcontrollers and circuits.
- **VOUT (Analog voltage pin):** This pin provides an analog voltage output that is linearly proportional to the temperature being measured [6].
- **GND (Ground):** This pin is connected to the ground of the circuit, completing the electrical circuit and providing the reference point for voltage measurements.

4.1.3. Operation

The TMP36 employs a solid-state method to gauge temperature, capitalizing on the phenomenon that the voltage drop across the base and emitter of the Diode-connected transistor declines predictably as temperature rises [7]. Through precise amplification of this voltage shift, generating an analog signal directly proportional to temperature becomes straightforward [7].

To measure the output voltage, we can simply use the formula:

$$V_{out} = 10 \times T + 500\ (mV)\ [6]$$

where V_{out} is the output voltage (mV), T is the temperature (℃).

For example, if the temperature is 50℃, then the output voltage should be: $V_{out} = 10 \times 50 + 500 = 1000\ (mV) = 1V$.

4.1.4. Example circuit

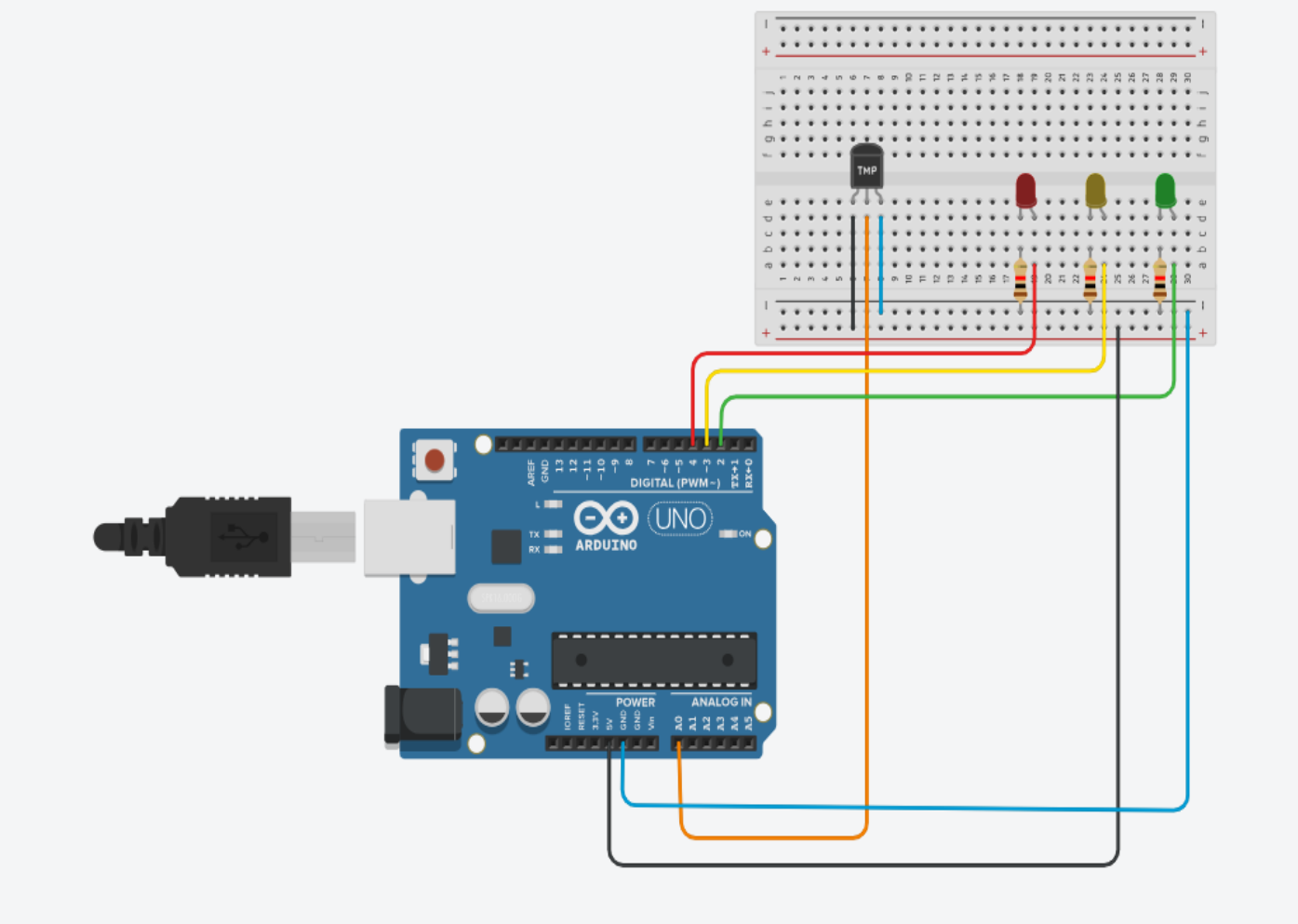

Figure 38: Example circuits of using temperature sensor TMP36 [8]

```
int baselineTemp = 0;
int celsius = 0;
int fahrenheit = 0;

void setup()
{
  pinMode(A0, INPUT);
  Serial.begin(9600);

  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
}

void loop()
{
  baselineTemp = 30;

  celsius = map(((analogRead(A0) - 20) * 3.04), 0, 1023, -40, 125);

   fahrenheit = ((celsius * 9) / 5 + 32);
  Serial.print(celsius);
  Serial.print(" C, ");
  Serial.print(fahrenheit);
  Serial.println(" F");

```

Figure 39.1: Code snippet of example circuit using temperature sensor TMP36 (part 1)

```
   if (celsius < baselineTemp) {
    digitalWrite(2, LOW);
    digitalWrite(3, LOW);
    digitalWrite(4, LOW);
  }
  if (celsius >= baselineTemp && celsius < baselineTemp + 10) {
    digitalWrite(2, LOW);
    digitalWrite(3, LOW);
    digitalWrite(4, HIGH);
  }
  if (celsius >= baselineTemp + 10 && celsius < baselineTemp + 20) {
    digitalWrite(2, LOW);
    digitalWrite(3, HIGH);
    digitalWrite(4, LOW);
  }
  if (celsius >= baselineTemp + 20 && celsius < baselineTemp + 30) {
    digitalWrite(2, HIGH);
    digitalWrite(3, LOW);
    digitalWrite(4, LOW);
  }
  if (celsius >= baselineTemp + 30) {
    digitalWrite(2, HIGH);
    digitalWrite(3, HIGH);
    digitalWrite(4, HIGH);
  }
  delay(1000);
}
```

Figure 39.2: Code snippet of example circuit using temperature sensor TMP36 (part 2)

From the example circuit:

- The left pin of the temperature sensor (Power) is connected to the 5V power pin of the Arduino.
- The right pin of the temperature sensor (Ground) is connected to the ground.
- The middle pin of the temperature sensor (VOUT) is connected to pin Analog Pin A0 of the Arduino.
- **RED** light's cathode is connected to Digital Pin 4 of the Arduino.
- **YELLOW** light's cathode is connected to Digital Pin 3 of the Arduino.
- **GREEN** light's cathode is connected to Digital Pin 2 of the Arduino.
- All lights' anodes are connected to the ground.

In this code snippet, the temperature will define which lights will be turned on.

- If the temperature is lower than $30℃$, all lights will be turned off.
- If the temperature is greater or equal to $30℃$ and less than $40℃$, **RED** light will be turned on while others are off.
- If the temperature is greater or equal to $40℃$ and less than $50℃$, **YELLOW** light will be turned on while others are turned off.
- If the temperature is greater or equal to $50℃$ and less than $60℃$, **GREEN** light will be turned on while others are turned off.
- If the temperature is greater or equal to $60℃$, all lights will be turned on.

4.1.5. Possible application

The temperature sensor TMP36 is useful in many applications. One of them is "Smart Plant Monitoring System" where The TMP36 sensor is placed near the plant to monitor the ambient temperature [9]. It continuously measures the temperature around the plant and based on the temperature readings, the system can activate cooling or heating mechanisms if the temperature deviates from the ideal range for plant growth.

4.1.6. Component in real life

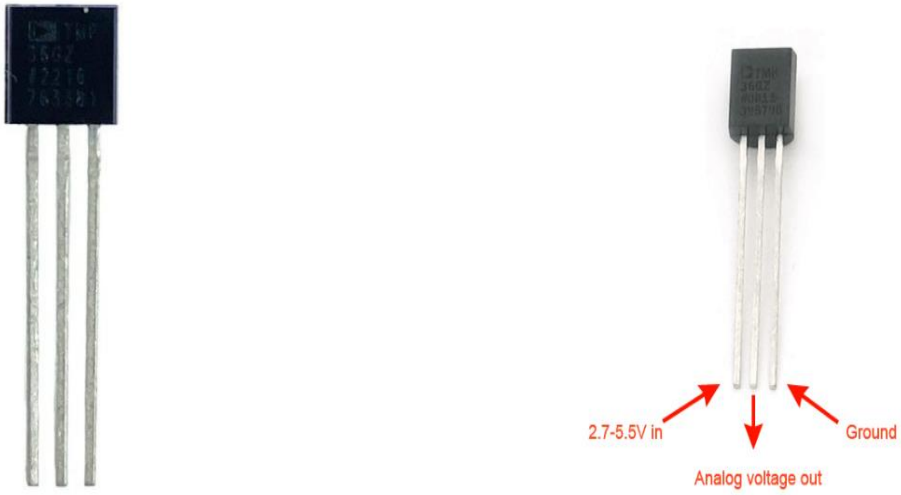

Figure 40.1 & 40.2: Temperature sensor (TMP36) in real life [6] [10]

4.2. Photodiode

4.2.1. Overview

Photodiode is a semi-conductor that converts light into electricity [11]. It operates in reverse bias mode, meaning that when light strikes the diode, it generates electron-hole pairs which create a flow of current [11].

4.2.2. Pin configuration

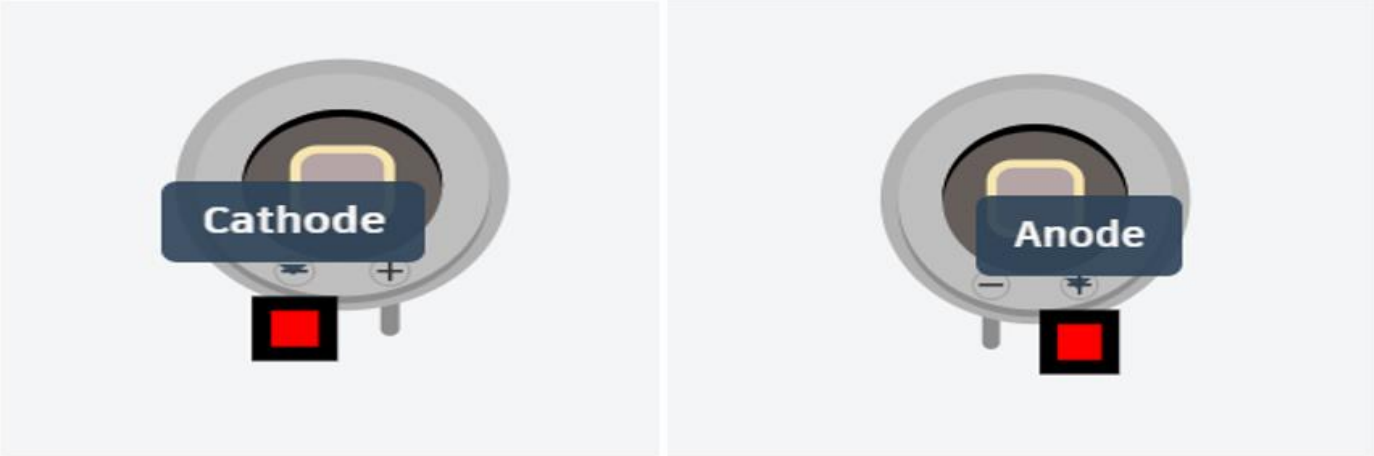

Figure 41: Photodiode pin configuration on Tinkercad [5]

The photodiode in Tinkercad has 2 pins:

- **Cathode (-):** This is the negative terminal of the photodiode. Electrons flow out of the diode through this terminal when light strikes it, creating a current [12].
- **Anode (+):** This is the positive terminal of the photodiode. Holes flow into the diode through this terminal when light strikes it, completing the circuit [12].

4.2.3. Operation

The p-n junction is a crucial component of a photodiode, consisting of a space between two layers of semiconductor material: the p-type layer, rich in holes, and the n-type layer, abundant in electrons [13]. This junction forms the depletion layer. In the absence of light, minimal current flows through the photodiode, known as "dark current," typically close to 0. Dark current diminishes with increased device sensitivity [13]. When light surpasses a threshold level, typically the bandgap, it generates electron-hole pairs within the device, increasing electrical current in the p-n junction [13].

4.2.4. Example circuit

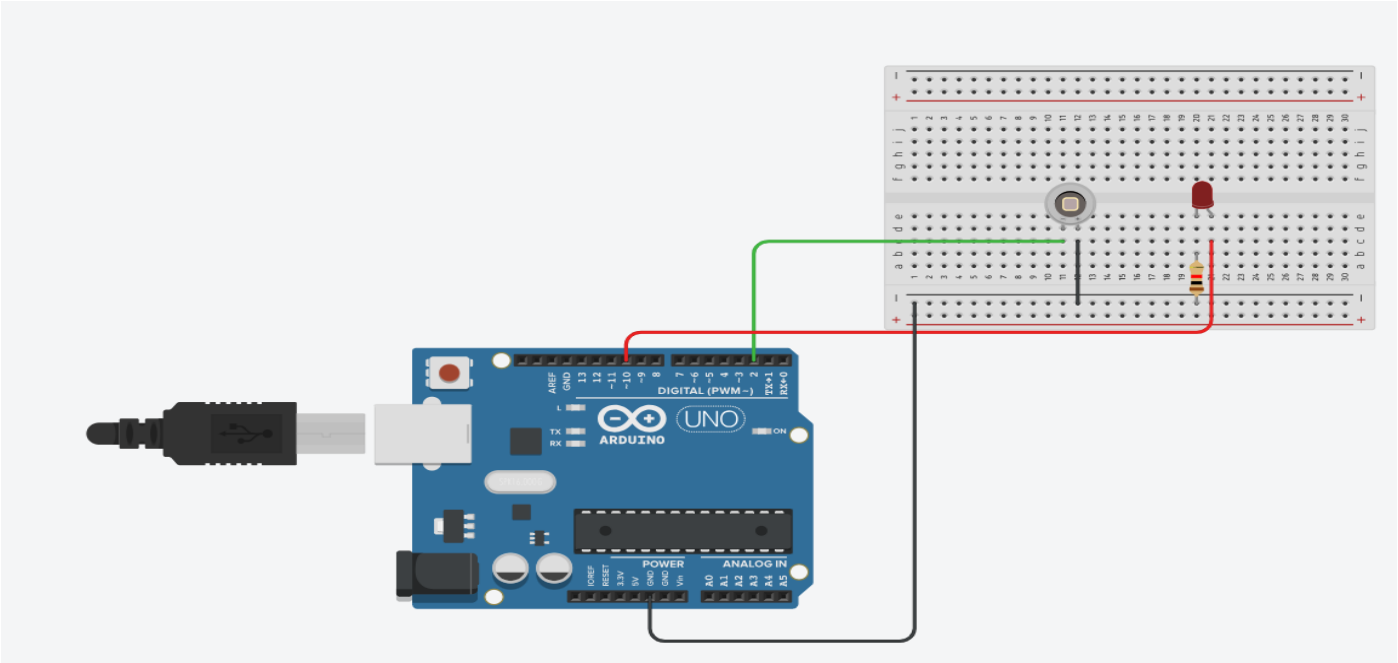

Figure 42: Example circuit using photodiode in Tinkercad

```
int ledPin = 10;
int photoDiodePin = 2;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(photoDiodePin, INPUT_PULLUP);
}

void loop()
{
  if (digitalRead(photoDiodePin) == LOW)
  {
    digitalWrite(ledPin, HIGH);
  }
  else
  {
    digitalWrite(ledPin, LOW);
  }
}
```

Figure 43: Code snippet of example circuit using photodiode

From the example circuit:

- The left pin of the photodiode, which is the cathode (-), is connected to Digital Pin 2 of the Arduino.
- The right pin of the photodiode, which is the anode (+), is connected to the ground.
- **RED** light's cathode is connected to the ground.
- **RED** light's anode is connected to Digital Pin 10 of the Arduino.

In this code snippet:

- Define LED's pin as digital pin 10.
- Define photodiode pin as digital pin 2.
- Turn on the LED (set to HIGH) if the photodiode pin is HIGH and turn off the LED (set to LOW) if the photodiode pin is LOW.

Result: if there is a light to the photodiode, the LED light will be turned on.

4.2.5. Possible application

Photodiodes find utility across various domains, including in light meters for photography. In this context, they play a crucial role in gauging light intensity. By transforming incident light into electrical current, photodiodes facilitate the determination of optimal exposure settings for capturing finely balanced photographs. Their function involves detecting the scene's brightness, aiding cameras in dynamically adjusting parameters such as aperture and shutter speed [14]. This adaptive mechanism prevents issues like underexposure or overexposure, thus ensuring precise exposure even amidst fluctuating lighting conditions, ultimately contributing to the production of high-quality images [14].

4.2.6. Component in real life

Figure 44: Photodiode in real life [15][16]

4.3. Force Sensor

4.3.1. Overview

A force sensor, also known as a force transducer or load cell, is a device designed to measure the force applied to it [17]. It converts the force into an electrical signal that can be measured and interpreted by various instruments or systems [17]. Force sensors come in various shapes and sizes, and they can measure different types of forces such as tension, compression, or shear forces [17]. These sensors find applications in various fields such as industrial automation, robotics, automotive, aerospace, healthcare, and research.

4.3.2. Pin configuration

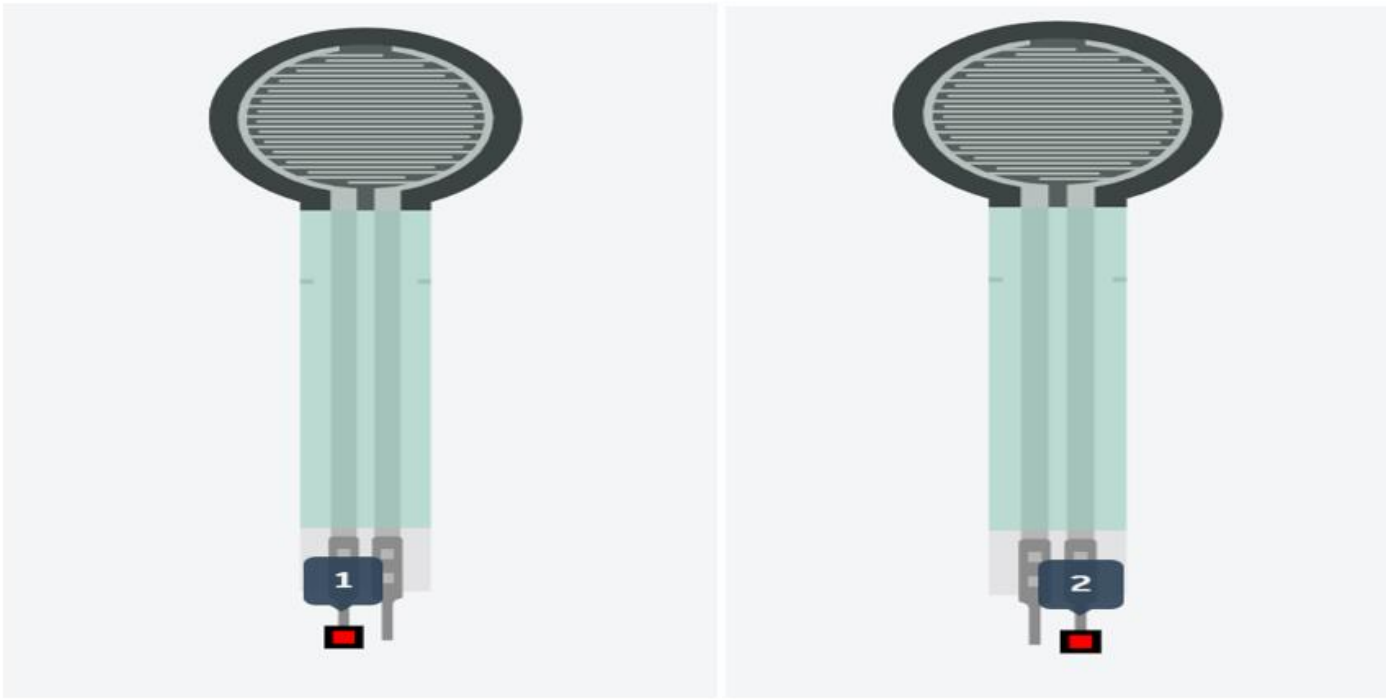

Figure 45: Force sensor's pin configuration on Tinkercad [5]

The force sensor features two terminals, but they lack polarization, indicating the absence of distinct positive or negative terminals [18].

4.3.3. Operation

The functionality of a Force-sensing resistor relies on the concept of “Contact Resistance”. These resistors incorporate a conductive polymer film that exhibits predictable changes in resistance when pressure is applied. Comprising sub-micrometer-sized conductive and non-conductive particles arranged within a matrix, this film undergoes resistance alteration as pressure prompts the micro-sized particles to contact the sensor electrodes. The degree of resistance changes correlates with the magnitude of the applied force, providing a measure of force exertion [19].

4.3.4. Example circuit

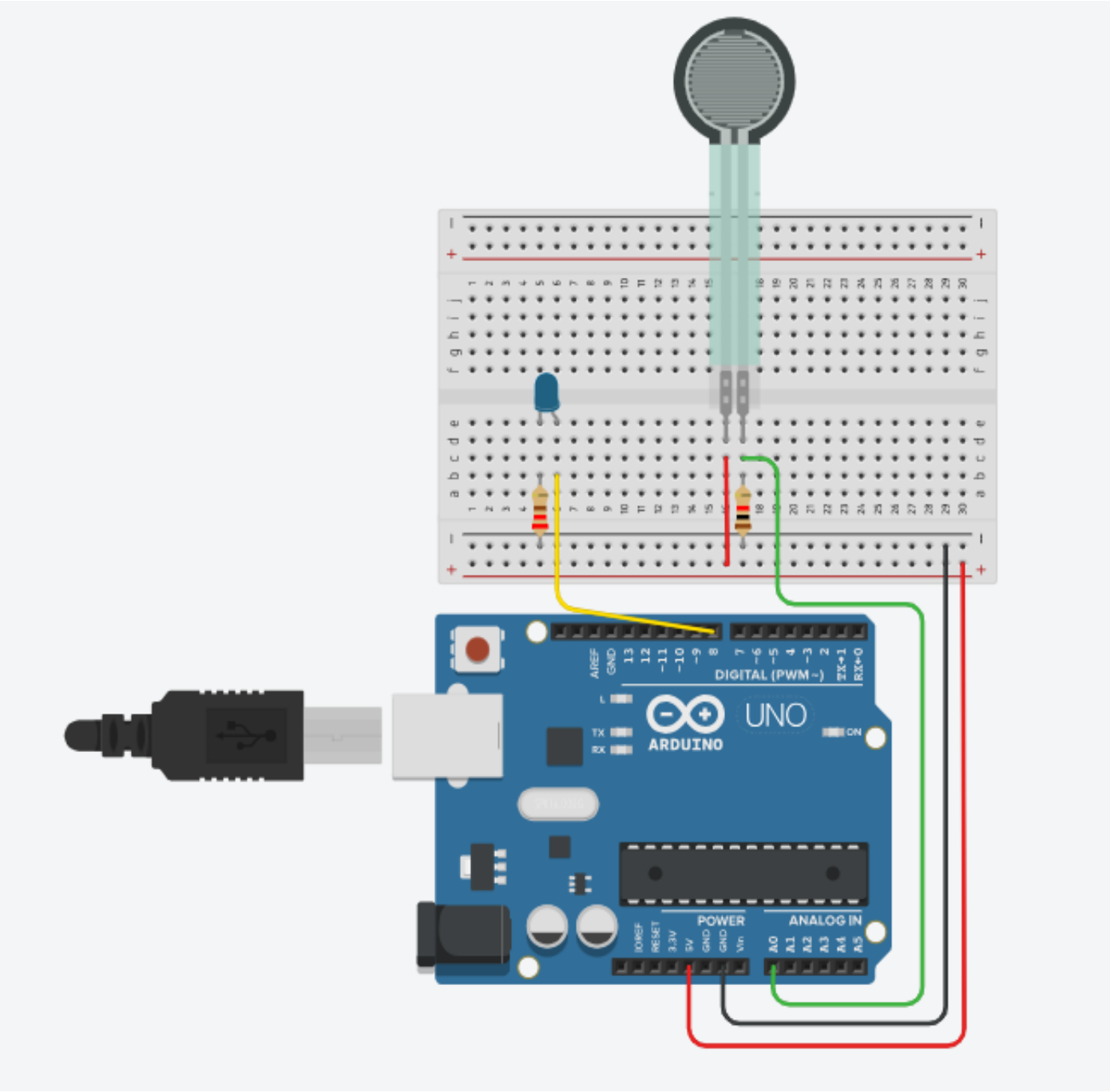

Figure 46: Example circuit using force sensor in Tinkercad

```
int sensor = 0;
void setup()
{
  pinMode(A0, INPUT);
  pinMode(8, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  sensor = analogRead(A0);
  if (sensor > 300) {
    digitalWrite(8, HIGH);
  } else {
    digitalWrite(8, LOW);
  }
  Serial.print("sensor = ");
  Serial.println(sensor);
  delay(100); // Wait for 100 millisecond(s)
}
```

Figure 47: Code snippet of example circuit using force sensor

From the example circuit:

- The left pin of the force sensor is connected to the 5V power pin of the Arduino.
- The right pin of the force sensor is connected to Analog Pin A0 of the Arduino. It is also connected to the ground.
- **BLUE** light's cathode is connected to the ground.
- **BLUE** light's anode is connected to Digital Pin 8 of the Arduino.

In this code snippet:

- Check the sensor configuration. If the sensor value is greater than 300, set the pin 8 to HIGH (turn on the **BLUE** LED). Otherwise, set it to LOW (turn off the LED).

Result: It is proven from Tinkercad simulation that if the force is greater than 3.66N, the LED will be turned on. Otherwise, the LED will be turned off.

4.3.5. Possible application

Force sensors find numerous practical applications in various fields. One notable example is their utilization as a trigger mechanism for home alert systems. For instance, they can be employed to activate a chime or buzzer inside a residence, effectively alerting occupants to the presence of a visitor at the door. Additionally, these sensors can be integrated into security setups to detect unauthorized entry attempts, enhancing home safety.

4.3.6. Component in real life

Figure 48: Force sensor in real life [20]

In the marketplace, there are many types of force sensors.

III. Reflection

1. Greater insights into Embedded System Engineering in real life

An embedded system, a specialized computing system crafted to fulfill specific functions within a larger system or product, is experiencing a surge in demand as a profession, offering abundant job prospects [21]. In this section, we'll delve into various job opportunities and their descriptions, spanning not only Vietnam but also global contexts.

1.1. Vietnam

1.1.1. Job description

Ampere is an international semi-conductor manufacturing company, and it has its branch in Ho Chi Minh City, Vietnam. The company is recruiting for an embedded software engineer to design, develop, and execute test sequence for ARM64 processor and industry bus interface such as I2C/SPI/UART/SD/SRAM/Flash, SATA, PCIe, DDR4/5, USB, Ethernet 1G/10G, etc. and develop applications and scripts for device diagnostic, performance tuning, and performance analysis. This role involves Engaging in all testing phases for specific IPs across multiple devices on various interfaces such as I2C, USB, Ethernet, SATA, PCIe, and DDR or Developing applications/scripts to validate functionality on ARM64 processors and boards [22].

1.1.2. Requirements

a. Academic requirements [22]

Bachelor's degree / Master Degree of Science in Computer Engineering/Electronic Engineering or equivalent.

b. Skill requirements [22]

- More than 2 years working in C, Java, Python, Linux Bash Shell.
- Proficiency in Ethernet and TCP/IP protocols.

- Familiarity with standard interfaces like I2C, SPI, UART, SD, SRAM, Flash, SATA, PCI, DDR4/5, USB, and Ethernet (1G/10G).
- Strong debugging and problem-solving skills.
- Excellent written and verbal communication skills in English and Vietnamese.

c. Extra requirements [22]

- Understanding of CPU-based architectures (8051/PIC/MCU).
- Experience in Microcontroller/Embedded System Programming.
- Knowledge of Embedded Linux Programming.
- Familiarity with handling logic analyzer equipment.

1.2. Worldwide - Singapore

As one of the most developing countries in the world, Singapore offers a great opportunity for Embedded System Engineering.

1.2.1. Job description

Dyson, an international company renowned for its electric domestic appliances like vacuum cleaners, fans, lighting products, and more, is seeking an embedded system engineer for its Singapore branch. The role involves developing real-time embedded software for consumer products on various silicon platforms, with a focus on ARM-based systems [23]. Responsibilities include reviewing system requirements, conducting feasibility studies, and contributing to software architecture and design [23]. Additionally, the position entails low-level design, coding, and testing, along with establishing continuous integration environments [23]. The candidate will also be tasked with identifying and addressing technical risks, driving process enhancements, and promoting best practices within the software team and across the broader engineering domain [23].

1.2.2. Requirements

a. Academic requirements [23]

Bachelor's degree in Electrical/Electronics/Telecommunication/Computer Engineering or equivalent.

b. Skill requirements [23]

- Minimum 1 year of software development experience encompassing requirement analysis, architecture, design, coding, and testing within embedded systems.
- Proficiency in C/C++, UML, and Python.
- Experience in embedded real-time software engineering, including familiarity with FreeRTOS.
- Strong knowledge of MCU/MPU architecture and common peripherals.
- Practical experience configuring and using sensor devices over interfaces such as I2C, SPI, and UART with DMA.
- Familiarity with on and off-target test harnesses like Ceedling and Behave.
- Understanding of software development life cycles, particularly Agile methodologies, and basic electronics hardware interfacing with software systems.
- Knowledge of embedded development tools, debuggers, emulators, analyzers, oscilloscopes, etc., as well as configuration management tools and concepts.
- Familiarity with the Atlassian suite (JIRA, Confluence, Bitbucket) is advantageous.
- Excellent written and verbal communication skills in English.

c. Extra requirements [23]

- Proficiency in Bluetooth Low Energy (BLE) protocols, including BLE 5.0.
- Ability to detect and resolve issues stemming from the interoperability of wireless protocols like BLE, Bluetooth Classic, and True Wireless Stereo (TWS).

- Experience in optimizing BLE/Bluetooth Classic product behavior to ensure interoperability and achieve high data throughput across a variety of Dyson products.
- Exposure to at least one of the following wireless technologies: Wi-Fi (802.11), NFC, ZigBee, 802.15.4, 3GPP, MQTT, Apple HomeKit.
- Knowledge of software security, LabVIEW, and Linux-based embedded systems.

1.3. Conclusion

In today's global landscape, the demand for Embedded System Engineering professionals is on the rise, not only in Vietnam but also worldwide. Typically, candidates seeking employment in this field are expected to demonstrate proficiency in programming languages such as C/C++ and Python, alongside familiarity with essential hardware components like I2C, UART, SPI, USB, Ethernet, and more. Moreover, adept debugging and problem-solving skills are highly valued attributes. Effective written and verbal communication skills are also crucial, as they facilitate seamless collaboration within teams, ensuring the quality and success of the project. Additionally, the required years of working experience and any extra qualifications may vary depending on many factors such as the company's size, the project's scope, or specific industry requirements.

2. Individual Reflection

2.1. Learning outcomes

Throughout this project, I've acquired a wealth of new knowledge and skills. In the initial phase, I delved into the intricacies of Bare OS, learning how to construct it and its pivotal features. Additionally, I honed my abilities in implementing a Command Line Interpreter (CLI) and utilizing ANSI Codes for terminal formatting. Moving on to the second stage, I gained valuable experience working with PL011 UART, mastering tasks such as baud rate configuration, bit length adjustments, and handshake control. This hands-on experience with real hardware, particularly the Raspberry Pi 3 board, provided me with a solid foundation in C programming. The third phase of the project exposed me to various common sensors available in Tinkercad. I familiarized myself with their functionalities, explored circuit connections, and learned how to integrate them into real-life applications. This understanding of sensor technology is vital for any aspiring embedded system engineer. In the final part of the project, I delved into the practical aspects of Embedded System Engineering in real-world scenarios, gaining insights into job qualifications and descriptions. The knowledge and skills acquired throughout this project have equipped me with valuable assets for my future career in embedded systems.

2.2. Challenges

Besides the precious knowledge and skill obtained throughout the project, I faced many challenges during the project development.

2.2.1. Phase 1: Basic command & CLI enhancement

While implementing basic commands proved to be a straightforward task completed within a mere two days, enhancing the Command Line Interpreter (CLI) with features like auto-completion and command history presented a more complex challenge. Due to the absence of library support, processing strings became a daunting endeavor, leading to several bugs such as malfunctioning characters and inadvertent deletion of vital information. Addressing these issues consumed an additional four days of effort. Despite these obstacles, I persevered and successfully fulfilled all requirements for enhancing the CLI. Moreover, grappling with these challenges provided valuable experience in logic and string processing within the C programming language.

2.2.2. Phase 2: UART configuration

During the second phase, my assignment involved configuring the UART settings. This task wasn't particularly challenging, as the PL011 UART configurations were thoroughly documented in the Raspberry Pi 3 & 4 Peripheral document. Implementing these configurations only required half a day of work.

2.2.3. Phase 3: Hardware configuration on real Raspberry Pi 3 board

This phase proved to be the most challenging aspect of the entire project. Despite the relatively straightforward pin configuration on the Raspberry Pi 3, simulating the setup with a real board using Tera Term presented numerous difficulties. One major issue arose when the mailbox on the real board failed to function correctly, rendering it unable to return the board revision and MAC address. Interestingly, these functions worked flawlessly during QEMU emulation. After several days of troubleshooting efforts, along with guidance from my lecturer, I eventually identified and rectified the issues. This experience was invaluable in enhancing my proficiency in working with hardware devices.

2.2.4. Phase 4: Some common sensors

During this phase, I encountered some challenges. Initially, grasping the inner workings of the sensors, utilizing them effectively in Tinkercad circuits, and envisioning their practical applications proved to be daunting tasks. However, after dedicating two days to thorough research online and engaging in practical exercises on Tinkercad, I managed to gain a deeper understanding of how the sensors operate and identified several potential real-world applications for them.

2.3. Improvements

While this project has provided invaluable experience, skills, and knowledge, it's clear that learning is an ongoing process. In my view, there's a need for further focus on improving my string processing abilities, particularly within the realm of C language, as this remains a weak area for me. Additionally, broadening my familiarity with hardware devices beyond just Raspberry Pi 3 or 4 to include platforms like Arduino Uno and ARM boards would be beneficial. Lastly, enhancing my proficiency in reading hardware documentation is essential, as I have encountered challenges in understanding register configurations at times. Strengthening these skills will not only aid in my studies in the "Embedded System: OS and Interfacing" course but will also be advantageous for my future career as an embedded system engineer.

IV. Conclusion

In conclusion, the objective of this project is to construct and enhance a bare-metal OS specifically designed for a Raspberry Pi model. Key features include the implementation of CLI, support for ANSI codes for terminal formatting, and configuration support for PL011 UART. Additionally, the project involves exploring the usage, functionality, and real-world applications of common sensors such as temperature sensors, force sensors, and photodiodes using Tinkercad. This project serves as an excellent opportunity for me to enrich my understanding and proficiency in embedded systems, thereby paving the way for my future career endeavors.

V. References

[1] M. Rouse, “Command Line Interface,” Techopedia, Oct. 03, 2023. https://www.techopedia.com/definition/3337/command-line-interface-cli (accessed Apr. 23, 2024).

[2] “What is CLI,” W3schools.com, 2024. https://www.w3schools.com/whatis/whatis_cli.asp (accessed Apr. 23, 2024).

[3] “BCM2837 ARM Peripherals.” Available: https://cs140e.sergio.bz/docs/BCM2837-ARM-Peripherals.pdf

[4] “TMP36 Temperature Sensor,” Adafruit Learning System, Jul. 29, 2012. https://learn.adafruit.com/tmp36-temperature-sensor/overview (accessed Apr. 25, 2024).

[5] “Circuits on Tinkercad - Tinkercad,” Tinkercad, 2024. https://www.tinkercad.com/circuits (accessed Apr. 25, 2024).

[6] “Temperature Sensor - TMP36 - SEN-10988 - SparkFun Electronics,” Sparkfun.com, Jan. 08, 2015. https://www.sparkfun.com/products/10988#:~:text=The%20TMP36%20is%20a%20low,proportional%20to%20the%20Celsius%20temperature. (accessed Apr. 25, 2024).

[7] LME Editorial Staff, “Interfacing TMP36 Temperature Sensor with Arduino,” Last Minute Engineers, Aug. 24, 2021. https://lastminuteengineers.com/tmp36-temperature-sensor-arduino-tutorial/ (accessed Apr. 25, 2024).

[8] “Circuit design TMP36 Temperature Sensor With Arduino - Tinkercad,” Tinkercad, 2024. https://www.tinkercad.com/things/7KlkHaxx20c-tmp36-temperature-sensor-with-arduino (accessed Apr. 26, 2024).

[9] Robocraze, “Smart Plant Monitoring System,” Robocraze, Jun. 02, 2023. https://robocraze.com/blogs/post/smart-plant-monitoring-system (accessed Apr. 26, 2024).

[10] “TMP36 Temperature Sensor,” Cytron Technologies Malaysia, 2024. https://my.cytron.io/p-tmp36-temperature-sensor (accessed Apr. 26, 2024).

[11] R. Teja, “What is a Photodiode? Working, V-I Characteristics, Applications,” ElectronicsHub USA, Apr. 04, 2024. https://www.electronicshub.org/photodiode-working-characteristics-applications/ (accessed Apr. 26, 2024).

[12] “PHOTODIODE BASICS – Wavelength Electronics,” Teamwavelength.com, Jan. 13, 2020. https://www.teamwavelength.com/photodiode-basics/ (accessed Apr. 26, 2024).

[13] “How Do Photodiodes Work? | RS,” Rs-online.com, Jun. 16, 2023. https://uk.rs-online.com/web/content/discovery/ideas-and-advice/how-do-photodiodes-work (accessed Apr. 26, 2024).

[14] F. Soldevila, E. Salvador-Balaguer, P. Clemente, E. Tajahuerce, and J. Lancis, “High-resolution adaptive imaging with a single photodiode,” Scientific reports, vol. 5, no. 1, Sep. 2015, doi: https://doi.org/10.1038/srep14300.

[15] “Hamamatsu, S1223-01 Full Spectrum Si Photodiode, Through Hole TO-5,” Rs-online.com, 2020. https://au.rs-online.com/web/p/photodiodes/4155722 (accessed Apr. 26, 2024).

[16] Arduino Forum, “How to use a 3 pin photodiode,” Arduino Forum, Apr. 06, 2018. https://forum.arduino.cc/t/how-to-use-a-3-pin-photodiode/518698 (accessed Apr. 26, 2024).

[17] “Force Sensors: Types, Uses, Features and Benefits,” Iqsdirectory.com, 2024. https://www.iqsdirectory.com/articles/load-cell/force-sensors.html (accessed Apr. 27, 2024).

[18] “FSR400 - Force Sensor,” Components101, 2018. https://components101.com/sensors/fsr400-force-sensor (accessed Apr. 27, 2024).

[19] T. Agarwal, “Force Sensor - Working Principle and It’s Applications,” ElProCus - Electronic Projects for Engineering Students, Dec. 25, 2019. https://www.elprocus.com/force-sensor-working-principle-and-applicationa/ (accessed Apr. 27, 2024).

[20] “Force Sensitive Resistor Sensor FSR 1.4 (Round - 14mm Sensing Diameter) Weight Sensor,” Future Electronics Egypt , 2024. https://store.fut-electronics.com/products/force-sensitive-resistor-square (accessed Apr. 27, 2024).

[21] “LinkedIn,” Linkedin.com, 2024. https://www.linkedin.com/pulse/do-embedded-systems-have-good-career-scope-cranes-varsity/ (accessed Apr. 28, 2024).

[22] “LinkedIn,” Linkedin.com, 2024. https://www.linkedin.com/jobs/search/?currentJobId=3797105144&geoId=104195383&keywords=embedded%20software%20engineer&location=Vietnam&origin=JOB_SEARCH_PAGE_SEARCH_BUTTON&originalSubdomain=sg&refresh=true (accessed Apr. 28, 2024).

[23] “LinkedIn,” Linkedin.com, 2024. https://www.linkedin.com/jobs/search/?currentJobId=3797410027&geoId=102454443&keywords=embedded%20software%20engineer&location=Singapore&origin=JOB_SEARCH_PAGE_LOCATION_AUTOCOMPLETE&originalSubdomain=sg&refresh=true (accessed Apr. 28, 2024).